Semantics-Based Cache-Side-Channel Quantification in Cryptographic Implementations

Doctoral thesis submitted in fulfillment of the requirements
for the degree of Doctor of Engineering (Dr.-Ing.)
at the Department of Computer Science
of the Technical University of Darmstadt

by Alexandra Weber

First Assessor: Prof. Dr.-Ing. Heiko Mantel
Second Assessor: Prof. Pasquale Malacaria, Ph.D.

Date of Submission: March 9, 2022
Date of Defense: April 21, 2022

D 17
Darmstadt, 2022

Weber, Alexandra:
Semantics-Based Cache-Side-Channel Quantification in Cryptographic Implementations
Darmstadt, Technische Universität Darmstadt
Date of the viva voce: April 21, 2022

This document was published in 2022 by tuprints, an e-publishing service of the Technical University of Darmstadt (http://tuprints.ulb.tu-darmstadt.de).

URN: urn:nbn:de:tuda-tuprints-212088
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/21208

Abstract

Performance has been and will continue to be a key criterion in the development of computer systems for a long time. To speed up Central Processing Units (CPUs), micro-architectural components like, e.g., caches and instruction pipelines have been developed. While caches are indispensable from a performance perspective, they also introduce a security risk. If the interaction of a software implementation with a cache differs depending on the data processed by the software, an attacker who observes this interaction can deduce information about the processed data. If the dependence is unintentional, it is called a cache side channel. Cache side channels have been exploited to recover entire secret keys from numerous cryptographic implementations.

There are ways to mitigate the leakage of secret information like, e.g., crypto keys through cache side channels. However, such mitigations come at the cost of performance loss, because they cancel out the performance benefits of caching either selectively or completely. That is, there is a security-performance trade-off that is inherent in the mitigation of cache-side-channel leakage. This security-performance trade-off can only be navigated in an informed fashion if reliable quantitative information on the cache-side-channel security of an implementation is available. Quantitative security guarantees can be computed based on program analyses. However, the existing analyses either do not consider caches, do not provide quantitative guarantees across all side-channel output values, or are only applicable to a limited range of crypto implementations.

In this thesis, we propose a suite of program analyses that can provide quantitative security guarantees in the form of reliable upper bounds on the cache-side-channel leakage of a variety of real-world cryptographic implementations. Technically, our program analyses are based on a combination of information theory and abstract interpretation. The distinguishing feature of each analysis is the underlying abstraction of the execution environment and program semantics.

Our first program analysis is based on an abstraction that captures the state of a CPU with a regular Arithmetic Logic Unit (ALU) during the execution of x86 instructions. In particular, our abstraction captures two status flags that are used, e.g., during the execution of different AES implementations. Our analysis is capable of computing quantitative cache-side-channel security guarantees for off-the-shelf AES implementations from multiple popular libraries. In a comparative study, we clarify the security impact of design choices in these implementations. For instance, we find that the number and size of lookup tables used for just the last transformation round of AES already has a significant impact on the guarantees for the entire implementation.

Our second program analysis is based on an abstraction that captures the execution of additional x86 instructions, including instructions that process larger operands. This abstraction can be used to quantify the leakage of crypto implementations that are based on large parameters. For instance, the lattice-based signature scheme ring-TESLA has a maximum key size of 49 152 bit. With our analysis, we successfully computed leakage bounds for the implementation of ring-TESLA. These bounds lead to the detection of multiple vulnerabilities that might be exploited to break the entire signature scheme. As a result, mitigations were integrated into the implementations of ring-TESLA and qTESLA, before the latter was submitted to the NIST PQC standardization.

Our third program analysis is based on an abstraction that captures the state of a CPU with an ALU and a Floating-Point Unit. It can be used to compute leakage bounds for crypto implementations that rely on floating-point instructions, e.g., to compute probabilities. The software used in Quantum Key Distribution (QKD), e.g., heavily relies on probabilities to perform error correction. With our analysis, we computed leakage bounds for a QKD implementation and detected a vulnerability that might leak the entire secret key. We proposed a mitigation and verified its effectiveness using our analysis. In the new version of the implementation, which is used at the TU Darmstadt Department of Physics, our mitigation is already integrated.

Finally, we broaden the scope to side channels that arise from the combination of caching and instruction pipelining. Such side channels are exploited, e.g., by the Spectre-PHT attack. The fourth program analysis in our suite is, to our knowledge, the first ever program analysis that computes reliable quantitative security guarantees with respect to such side channels.

Abstract in German

Performanz spielt schon seit langer Zeit eine wichtige Rolle in der Entwicklung von Computersystemen. Auf Mikroarchitekturebene werden verschiedene Komponenten eingesetzt, um die Ausführungsgeschwindigkeit zu erhöhen. Insbesondere Caches sind aus modernen Computerarchitekturen nicht mehr wegzudenken. Gleichzeitig bergen Caches allerdings ein Sicherheitsrisiko. Hängt die Cachenutzung einer Softwareimplementierung von den verarbeiteten Daten ab, dann kann ein Beobachter aus der Cachenutzung Rückschlüsse auf diese Daten ziehen. Im Falle einer unbeabsichtigten Abhängigkeit handelt es sich um einen sogenannten Cacheseitenkanal. Solche Kanäle werden oft ausgenutzt, um geheime Schlüssel aus Kryptoimplementierungen zu extrahieren.

Es gibt verschiedene Gegenmaßnahmen, um Cacheseitenkanäle zu schließen oder zu reduzieren. Allerdings werden dadurch die Vorteile von Caches ganz oder teilweise aufgehoben, sodass sich die Performanz verschlechtert. Es muss also zwischen Sicherheit und Performanz abgewogen werden. Um informiert über Seitenkanalgegenmaßnahmen entscheiden zu können, werden quantitative Informationen über das Ausmaß des jeweiligen Cacheseitenkanals benötigt. Solche quantitativen Sicherheitsgarantien können grundsätzlich mithilfe von Programmanalysen hergeleitet werden. Die bisherigen Analysen sind allerdings entweder nicht quantitativ, berücksichtigen keine Caches, oder sie sind nicht für ein breites Spektrum realer Kryptoimplementierungen anwendbar.

Diese Dissertation präsentiert eine Toolsuite aus Programmanalysen, die verlässliche quantitative Sicherheitsgarantien in Form von oberen Schranken für die durch Cacheseitenkanäle preisgegebenen Informationen in verschiedenen Kryptoimplementierungen bestimmen können. Die Analysen basieren auf Informationstheorie und Abstract Interpretation. Sie setzen dabei verschiedene neuartige abstrakte Repräsentationen von Programmausführungen ein.

Die erste Programmanalyse basiert auf einer Abstraktion eines x86-Prozessors mit Arithmetisch-logischer Einheit (ALU). Diese Abstraktion berücksichtigt zwei Statusflags, die beispielsweise bei der Ausführung von AES-Implementierungen verschiedener Kryptobibliotheken verwendet werden. Dadurch kann die Analyse quantitative Sicherheitsgarantien für solche Implementierungen bestimmen. Mithilfe der Analyse klären wir auch, welchen Einfluss Designentscheidungen in AES-Implementierungen auf deren Cacheseitenkanalsicherheit haben. So kann beispielsweise die Nutzung verschiedener Lookuptabellen allein in der letzten AES-Transformationsrunde die Sicherheitsgarantien für die Gesamtimplementierung bereits signifikant beeinflussen.

Die zweite Programmanalyse basiert auf einer Abstraktion, die Instruktionen mit größeren Operanden berücksichtigt. Sie kann quantitative Sicherheitsgarantien auch für Implementierungen mit großen Parametern berechnen. In der Implementierung des ring-TESLA-Signaturverfahrens, dessen maximale Schlüsselgröße 49 152 bit beträgt, konnten durch unsere Analyse mehrere Cacheseitenkanäle gefunden werden. Entsprechende Gegenmaßnahmen wurden in die Implementierungen von ring-TESLA und dem Nachfolgeverfahren qTESLA integriert, bevor letzteres zum Standardisierungsprozess für Postquantenkryptographie des NIST eingereicht wurde.

Die dritte Programmanalyse basiert auf einer Abstraktion eines x86-Prozessors mit ALU und Gleitkommaeinheit. Sie kann die Cacheseitenkanalsicherheit von Implementierungen quantifizieren, die Gleitkommainstruktionen z. B. zur Berechnung von Wahrscheinlichkeiten nutzen. Ein Beispiel sind Implementierungen zum Quantenschlüsselaustausch, die Gleitkommainstruktionen für ein Fehlerkorrekturverfahren nutzen. In einer solchen Implementierung konnten wir mit unserer Analyse einen Cacheseitenkanal finden, der zur Preisgabe des gesamten geheimen Schlüssels führen könnte. Um den Kanal zu schließen, entwickelten wir eine Gegenmaßnahme, die mittlerweile in einer neuen Version der Implementierung am Fachbereich Physik der TU Darmstadt genutzt wird.

Abschließend untersuchen wir Seitenkanäle, die aus der Kombination von Cache und Pipeline entstehen und bei Angriffen wie Spectre-PHT ausgenutzt werden. Unsere vierte Programmanalyse kann erstmals verlässliche quantitative Sicherheitsgarantien für solche Kanäle berechnen.

Acknowledgments

First and foremost, I would like to thank my supervisor Heiko Mantel. Already during my undergraduate studies, his inspiring courses got me interested in the area of program analysis. He gave me the chance to get a taste of the research in this area during my Bachelor and Master thesis, as a student research assistant, and as a guest at meetings of the priority program Reliably Secure Software Systems. Throughout my PhD studies, his feedback, guidance, and encouragement were extremely valuable and I am very grateful for the many inspiring research discussions and for his continuous support of both, my research and my personal development as a researcher.

I would like to thank Pasquale Malacaria for serving on my PhD committee despite his very busy schedule. I am also very grateful to him for inviting me to a research visit at Queen Mary University of London and for the very inspiring discussions about side-channel quantification.

Beyond my PhD committee, I would like to thank Boris Köpf for the fruitful collaboration, the inspiring research discussions, and for introducing me to the details of the CacheAudit framework.

I am also very thankful for the fruitful collaborations with Gernot Alber, Johannes Buchmann, Thomas Schneider, Thomas Walther, Nina Bindel, Florian Dewald, Juliane Krämer, Oleg Nikiforov, Alexander Sauer, Johannes Schickel, Friedrich Weber, Christian Weinert, and Tim Weißmantel.

The chair MAIS and the collaborative research center CROSSING at TU Darmstadt provided a great working environment throughout my PhD research, which I am very grateful for. I would also like to thank the Research School of Computer and Information Security at the Norwegian University of Science and Technology for giving me the opportunity to present my research as a lecturer at their summer school. I am also very grateful for the chance to attend the very inspiring Marktoberdorf summer school on the Verification and Synthesis of Correct and Secure Systems.

I would like to thank all of my colleagues at MAIS and at CROSSING for many interesting and motivating discussions. In particular, I would like to thank Richard Grewe, Görkem Kılınç, Matthias Perner, Johannes Schickel, David Schneider, Artem Starostin, and Tim Weißmantel. Additionally, I would like to thank Tim Weißmantel in his former role as a student assistant for his support with the implementation of the program-analysis tool CacheAudit-FPU.

A special thanks also goes to Maximilian Parr and Niels Weber for proofreading this thesis.

Last, but not least, I would like to thank my family, Sabine Weber, Andreas Weber, Katharina Weber, and Niels Weber, and my friends for their continuous support and encouragement.

Publications

We have published excerpts of this thesis as follows:

Preliminary versions of the program analysis and the case study described in Chapter 3 were presented in [89]. In the preliminary version, the abstract domain underlying the program analysis was not formalized. Furthermore, the tool support for the analysis was described only at a high level and is described in more detail in Chapter 3.

Earlier versions of the program analysis and the case study described in Chapter 4 were presented in [23]. The earlier version of the case study consisted of cache-side-channel aspects and cryptographic aspects. Our Chapter 4 describes the cache-side-channel aspects, while Bindel describes the cryptographic aspects in her dissertation [21].

Preliminary versions of the program analysis and case study from Chapter 5 were described in [132]. The preliminary formalization of the abstract domain covered fewer aspects of the cache than the domain in Chapter 5. Moreover, the case study in the preliminary version covered only one attacker model, while Chapter 5 covers four attacker models.

A publication about the program analysis Spectrescope and the corresponding evaluation from Chapter 6 is currently under submission to a major conference.

Additional research during my PhD studies, which has not been included in this thesis, concerns the quantification of software-based energy side channels with distinguishing experiments [87], the verification of the absence of timing side channels in AVR assembly programs [40], and the guidance and verification of circuit compilation for cache-side-channel mitigation using quantitative program analysis [86]. The relation of the quantitative cache-side-channel analysis from [86] to the analyses in this thesis is discussed in more detail in Chapter 7.

Contents

Chapter 1

Introduction

Performance has been and will continue to be a key criterion in the development of computer systems for a long time. The goal to optimize performance plays an important role for both, the development of software implementations and the development of hardware platforms. In software implementations, e.g., conditional branches are used to perform computation steps only in cases that require them and lookup tables are used to store precomputed values for efficient retrieval at runtime. In hardware platforms, e.g., caches are used to store frequently-accessed memory entries closer to the Central Processing Unit (CPU) where they can be accessed more quickly, and pipelines are used to parallelize the processing of multiple instructions.

While optimizations to software and hardware can significantly improve the performance of a computer system, they can also significantly affect the security of the system. In particular, they might give rise to side channels. A side channel is a channel through which an implementation that is executed on a computer system communicates unintentionally.

Consider a software implementation that is executed on a computer system and processes some information. If a property of the execution depends on the information that is processed, the implementation communicates through a side channel. The property that depends on the information is also called the side-channel output. Examples of properties that can function as side-channel outputs include the execution's running time [71], the execution's power consumption [73], or the interaction with a shared cache during the execution [103]. If the side-channel output depends on a secret that is processed by the implementation, e.g., a cryptographic key, and if an attacker observes the side-channel output, e.g., by measuring the running time, information about the secret is leaked to the attacker through the side channel. The code section of the implementation that causes the dependence of the side-channel output on the secret is called a side-channel vulnerability. An attack in which an attacker observes the output of a side channel and uses it to deduce information about the secret is called a side-channel attack.

As an example, consider an implementation that checks whether a password entered by a user is correct. Assume that the implementation compares the user's candidate password to the actual password by iterating through all characters and reporting failure as soon as a mismatch between the passwords is detected. Consider an attacker who guesses a candidate password and measures the time taken to check whether the candidate password is correct. The attacker keeps modifying the first character of the candidate password and measuring the time taken by the implementation for checking the candidate password. When the attacker guesses the first character correctly, the implementation will perform an additional comparison operation on the second character of the candidate password. This will increase the running time of the password-checking implementation. When the attacker observes a longer running time, he deduces that his guess for the first character is correct. He proceeds by guessing each character of the password with the same technique until he is in possession of the complete password. This is an example of a timing-side-channel attack, because it exploits the dependence of the implementation's running time on secret information.

Side channels work across different layers of abstraction in the program execution. For instance, the execution time of a program depends on the architecture and micro-architecture of the computer system on which the program is executed. The side-channel leakage in the above example can be spotted already at a comparatively coarse level of abstraction by counting the number of instructions that are executed. However, there are also side channels that arise from low-level micro-architectural features. In this thesis, we focus on cache side channels, a prominent class of side channels on the micro-architectural level that arises from the use of CPU caches.

Cache-side-channel leakage occurs if the addresses of the memory entries that are loaded into a CPU cache during an execution depend on secret information. This can happen, e.g., if an implementation accesses an array at indices that depend on a secret. Cache-side-channel leakage can be exploited in multiple ways. For instance, an attacker who controls a spy process that is executed on the same system as the program under attack, e.g., due to co-location on cloud machines [117], can infer information about the cached addresses based on the time required for his own memory accesses [102] or the memory accesses of the program under attack [135].

Cache side channels have been exploited to recover secret keys from numerous cryptographic implementations. Examples range from implementations of symmetric crypto like, e.g., BouncyCastle AES [80] and OpenSSL AES [16, 1, 102, 59, 66, 65, 25, 114], to implementations of classical asymmetric crypto like, e.g., GnuPG RSA [135], Libgcrypt RSA [17, 25], and OpenSSL RSA [107, 136], to implementations of lattice-based crypto like, e.g., the reference implementation of BLISS [54] and the implementation of BLISS-B in the strongSwan VPN suite [109]. Moreover, cache side channels are an important building block in recent attacks that exploit side channels that arise from the combination of multiple micro-architectural features, including, e.g., [84, 72].

As shown by the numerous successful key recoveries, cache side channels pose a serious security risk. At the same time, caches are indispensable for performance reasons. That is, there is a trade-off between security and performance that is inherent in the mitigation of cache-side-channel leakage. Quantitative security guarantees for implementations allow one to navigate this trade-off in an informed way. Quantitative guarantees can be used to evaluate whether a partial side-channel mitigation that preserves the desired level of performance also achieves the desired level of security. Furthermore, quantitative guarantees can be used to compare multiple alternative implementations, e.g., of a cryptographic algorithm, with respect to the level of security that they provide against cache-side-channel leakage.

The overall contribution of this thesis is a suite of program analyses that can provide quantitative security guarantees in the form of reliable upper bounds on the cache-side-channel leakage of a variety of real-world cryptographic implementations, ranging from different implementations of the Advanced Encryption Standard (AES), to an implementation of lattice-based cryptography, to an implementation of Quantum Key Distribution (QKD). In addition, we propose the first program analysis that can provide upper leakage bounds on side channels that arise from the combination of caching and instruction pipelining. Our analysis suite can be used to verify the absence of leakage in an implementation in case a zero-leakage guarantee is derivable. If leakage is present in an implementation, our analysis suite can be used to quantify this leakage in order to better understand the remaining level of security and its relation to implementation details.

Prior program analyses that provide quantitative leakage bounds either do not consider cache side channels (e.g., [112, 110, 85]), do not provide upper bounds across side-channel output values (e.g., [30, 11]), or are based on abstractions that are not suitable to analyze the range of crypto implementations that we consider in this thesis (e.g. [45, 44]). Other program analyses that consider cache side channels are only qualitative (e.g., [129, 26, 128]), i.e., can only verify the absence of leakage but not quantify how much leakage is present. For side channels that arise from the combination of caching and instruction pipelining, all prior analyses, including [28, 57, 13, 27, 58, 126], are only qualitative.

With our analysis suite, we derive insights about the impact of design choices in AES implementations on the derivable cache-side-channel leakage bounds. For instance, we find that the leakage bounds for lookup-table-based AES implementations are heavily influenced by the

technique used to implement the last transformation round of AES. Furthermore, we analyze implementations of a lattice-based signature scheme and of quantum-key distribution. In both cases, we identify a significant amount of leakage. More concretely, the leakage might be used to break the signature scheme and might reveal the entire key established using QKD, respectively. We investigate mitigation techniques and verify their effectiveness against the detected leakage. In both cases, our mitigations have been integrated into the implementations by the developers before the publication of this thesis.

1.1 Approach in this Thesis

Our program-analysis suite consists of four analyses. The first three analyses incrementally lift the conceptual and technical restrictions that hinder the application of the prior state-of-the-art analysis [45] for cache-side-channel quantification to different types of real-world crypto implementations. The fourth analysis augments the scope to side channels that arise from a cache combined with an instruction pipeline with branch prediction and out-of-order execution.

Like the prior analysis from [45], our analyses are based on a combination of abstract interpretation [34] and information theory [36]. The former is used to overapproximate the set of observations that a side-channel attacker might make. The latter is used to compute an upper bound on the number of leaked bits based on the overapproximated set of observations. In our analyses, we improve on the abstract interpretation to overapproximate attacker observations.

Abstract interpretation is a static program-analysis technique that makes analyses across the possible executions of a program feasible by abstracting from details of the executions that are not relevant. An analysis is defined by an abstract domain and an abstract semantics. The abstract domain defines how snapshots of the system state during a program execution are modeled abstractly. The abstract domain underlying [45], e.g., captures the possible values of each CPU register and memory entry by a set of possible values. The abstract semantics models the possible modifications to the system state during a program execution at the level of the abstract domain.

The four program analyses in our suite are based on abstract domains and abstract semantics that improve the state of the art in multiple dimensions. The first three analyses are motivated by different types of cryptographic implementations that require more powerful abstractions for a precise cache-side-channel analysis. With the fourth analysis, we explore how to capture the combination of caching and instruction pipelining in the abstract domain and abstract semantics.

- Our first program analysis is motivated by a range of AES implementations. We define an abstract domain and abstract semantics that capture the effect of each execution step on two important CPU status flags. The resulting program analysis is applicable across multiple off-the-shelf AES implementations.
- Our second program analysis is motivated by lattice-based cryptography, where the parameters (e.g., the keys) are much larger than for AES. We augment the abstract semantics to capture multiple additional x86 instructions, including instructions for handling bigger operand sizes. The resulting program analysis is applicable to the implementation of the lattice-based signature scheme ring-TESLA [3].
- Our third program analysis is motivated by Quantum Key Distribution, where computations are based on probabilities encoded as floating-point numbers. We augment the abstract domain to capture the Floating-Point Unit (FPU) that processes floating-point computations. The resulting program analysis is applicable to software-based parts of the implementation of the BB84 protocol for QKD used at the Department of Physics at TU Darmstadt [100].
- For our fourth program analysis, we develop an abstract domain and semantics that capture both, the cache and the CPU instruction pipeline. The resulting program analysis quantifies the leakage of assembly-level programs to attacks that exploit the combination of caching and pipelining (a prominent example of such an attack is, e.g., Spectre-PHT [72]).

We define the abstract semantics underlying our analyses in a way that they overapproximate

program executions reliably. As a reference point, we use the formalization of the concrete semantics by Degenbaev [38] for regular x86 instructions. For x87 instructions, we rely on the specifications in Intel's Software Developer's Manual [62]. In Appendix B, we define a formal concrete semantics for an architecture with an instruction pipeline and cache. This concrete semantics is the reference point for the abstract semantics underlying our fourth program analysis.

We evaluate each of our program analyses in a case study. The target implementations in our case studies belong to the classes of implementations that motivated the respective analyses.

Overall, our case studies cover a broad spectrum of cryptographic primitives, ranging from encryption (AES), to digital signatures (ring-TESLA), to key exchange (QKD). For all these primitives, our case studies focus on algorithms that are relevant even for post-quantum cryptography.

Our case studies not only serve as evaluation for our augmented program analyses, but also had a direct impact on the security of cryptographic implementations in practice. Based on our analysis results for the ring-TELSA implementation, the implementation was adapted to fix multiple cache-side-channel vulnerabilities. This improved implementation was the basis for the implementation of the successor signature scheme qTESLA [22], which advanced to Round 2 of the Post-Quantum Cryptography (PQC) standardization of the U.S. National Institute of Standards and Technology (NIST) [4]. Based on our analysis results for the QKD implementation, we detected and mitigated a cache-side-channel vulnerability that might leak the entire secret key during the key exchange. Our mitigation was integrated into the new version of the implementation that is used for the development of QKD setups at the Department of Physics at TU Darmstadt.

1.2 Contributions of this Thesis

The overall contribution of this thesis is a suite of quantitative program analyses. Figure 1.1 visualizes the structure of each program analysis and the corresponding evaluation. We discuss the individual contributions related to each program analysis in detail in the following.

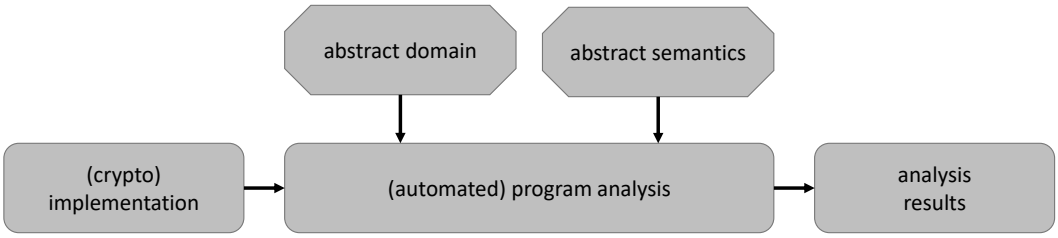

Figure 1.1: Structure of Program Analysis and Evaluation

Contributions Related to the First Program Analysis The contributions related to the first program analysis in our suite are an abstract domain, an abstract semantics, automatic tool support for the analysis, and a set of quantitative security guarantees.

The abstract domain, called $\overline{\mathcal{D}}_{32}$, captures the possible snapshots of the state of an x86 CPU during the execution of an x86 binary. The distinguishing feature of our domain is that it captures the possible states of the CPU status flags SF (sign flag) and OF (overflow flag) in relation to the states of architectural and micro-architectural components, including the cache. The prior state-of-the art domain [45] for the quantification of cache side channels on x86 CPUs does not take these CPU flags into account. For the analysis of off-the-shelf AES implementations, however, it is crucial to capture these status flags, because in multiple AES implementations conditional branching decisions depend on these flags. Since the flags influence the control flow, capturing their state at a suitable level of granularity is especially critical for the precision of the

program analysis. Our abstract domain represents the possible states at a level of granularity that allows for the computation of meaningful leakage bounds across AES implementations from multiple cryptographic libraries, including, e.g., OpenSSL and mbedTLS, with and without side-channel countermeasures in place. Furthermore, our domain is not limited to the analysis of AES implementations but can be reused also for analyzing implementations of other types of cryptography. In Chapter 4, we reuse the domain $\overline{\mathcal{D}}_{32}$ as a building block for our second analysis.

The abstract semantics, called $\mathtt{upd}_{\overline{\mathcal{D}}_{32}}$, captures the semantics of a large subset of the x86 assembly language with respect to the abstract domain $\overline{\mathcal{D}}_{32}$. It overapproximates the possible changes to the system state caused by the execution of x86 instructions. The semantics covers all instructions that occur in the x86 binaries corresponding to off-the-shelf AES implementations from the crypto libraries OpenSSL [101], mbedTLS [9], NaCl [18], Nettle [94], and LibTomCrypt [79]. In particular, the semantics captures the effects of all these x86 instructions on the status flags SF and OF. A key difficulty in the development of the semantics was to model the effects of each instruction on the flags at a suitable level of granularity, such that all effects that can occur in an execution are overapproximated but such that the approximation is precise enough to enable the computation of meaningful leakage bounds. Our abstract semantics is a reliable overapproximation and, at the same time, allows us to compute meaningful leakage bounds across the above-mentioned AES implementations. For instance, our semantics is precise enough to verify that preloading and bitslicing reduce the leakage of the AES implementations to zero.

Our program analysis is automated in the analysis tool CacheAudit 0.2b. We performed both, the conceptual refinement for the new abstraction as well as the actual implementation.

The quantitative security guarantees that we provide are upper bounds on the cache-side-channel leakage with respect to four different attacker models and seven different cache sizes. The guarantees cover AES implementations from the libraries OpenSSL, mbedTLS, Nettle, and LibTomCrypt. Moreover, they cover variants of these implementations that are hardened with the preloading countermeasure, as well as the bitsliced AES implementation from the library NaCl.

We evaluate our program analysis based on the leakage bounds that it computes for the AES implementations. More concretely, we use the analysis to perform a comparative study across different implementation techniques for AES. For lookup-table-based implementations, we clarify, e.g., the role of the tables used to implement the last round of AES. Interestingly, we find that the number and size of the tables used for just this one round already has a significant impact on the security guarantees for the entire implementation. Out of the techniques that we considered, the most beneficial one was to reuse the tables from the main AES rounds and to mask out the effect of the transformation that is skipped in the last AES round. We also investigate the effect of the attack surface, the cache size, and the use of countermeasures on the security guarantees.

Contributions Related to the Second Program Analysis The contributions related to the second program analysis in our suite are an abstract semantics, automatic tool support for the analysis, and a set of quantitative security guarantees.

The abstract semantics, called $\mathtt{upd}'_{\overline{\mathcal{D}}_{32}}$, overapproximates the semantics of x86 assembly instructions, reusing the abstract domain from our first program analysis. The distinguishing feature of $\mathtt{upd}'_{\overline{\mathcal{D}}_{32}}$ is that it captures the semantics of multiple previously unsupported x86 instructions, including instructions that are used to handle larger operands. The semantics can, therefore, be used to analyze crypto implementations with large parameters. For instance, the lattice-based signature scheme ring-TESLA [3] has a maximum key size of 49 152 bit, which is 192 times larger than the maximum key size of AES (256 bit). Our semantics overapproximates the semantics of all instructions used by our target ring-TESLA implementation reliably. At the same time, it allows for the computation of leakage bounds that are precise enough to support the detection of multiple vulnerabilities in the ring-TESLA implementation and to verify the effectiveness of mitigations against these vulnerabilities.

Our program analysis is automated in the tool CacheAudit 0.2c. As for the first analysis, we

performed the conceptual refinement for the new abstraction and the actual implementation.

The quantitative security guarantees based on our analysis are cache-side-channel leakage bounds with respect to four attacker models and a cache configuration that corresponds to the Level 1 cache of the Intel Skylake architecture [62]. The bounds cover the overall ring-TESLA signature generation, as well as four individual functions in which vulnerabilities were detected.

We evaluate our program analysis based on the leakage bounds for the ring-TESLA implementation. As usual in this field, we use a slightly simplified variant of the original implementation to avoid features that would unnecessarily complicate the analysis. In our case, the simplifications are twofold: We analyze the implementation without the integrated random-number generator and we eliminate infinite loops. Our evaluation leads to the detection of multiple vulnerabilities. These vulnerabilities were detected in collaboration with the cryptographers who developed ring-TESLA. Our focus was on the cache-side-channel perspective. The cryptographic impact of the vulnerabilities was assessed by Bindel, who came to the conclusion that the detected leakage might be exploited to break the entire signature scheme using a so-called learning-the-parallelepiped attack [21]. In the development of mitigations for the vulnerabilities, our focus was on removing the side-channel leakage, while Bindel focused on preserving the cryptographic functionality. With our program analysis, we were able to verify that the mitigations eliminate the leakage effectively. Subsequently, the mitigations were adopted in a new version of the ring-TESLA reference implementation, as well as the reference implementation of the successor scheme qTESLA [22], which advanced to Round 2 of the NIST PQC standardization [4].

Contributions Related to the Third Program Analysis The contributions related to the third program analysis in our suite are an abstract domain, an abstract semantics, automatic tool support for the analysis, and a set of quantitative security guarantees.

The abstract domain, called $\overline{\mathcal{D}}_{64}$, captures the possible snapshots of an x86 CPU during the execution of an x86/87 binary. That is, the domain captures not only the micro-architectural components that execute the core x86 instruction set, but also the components required to execute the set of x87 floating-point instructions. While the previous abstract domains, including the domain from [45] and our domain $\overline{\mathcal{D}}_{32}$ captured only the Arithmetic Logic Unit (ALU) of an x86 CPU, the domain $\overline{\mathcal{D}}_{64}$ captures the ALU, as well as the FPU, which is a second execution unit that also maintains a separate set of status flags, a separate stack of registers, and tags that track the validity of the register values [63]. A key difficulty in the development of $\overline{\mathcal{D}}_{64}$ was to capture these additional components at a level of abstraction that allows for an efficient, yet precise computation of leakage bounds. With the new domain, it becomes possible to analyze implementations that inherently rely on floating-point computations. This is, e.g., the case for implementations of QKD software. QKD software operates on probability values in order to recover the most likely values of physical measurements in the presence of imperfect measurement equipment. We were able to analyze a QKD implementation and to obtain precise leakage bounds with a program analysis that is based on our abstract domain $\overline{\mathcal{D}}_{64}$.

Our analysis is automated in the tool CacheAudit-FPU. We performed the conceptual refinement, while the actual implementation in this case was supported by a Master student.

The quantitative security guarantees that we provide based on our analysis are cache-side-channel leakage bounds for implementations of multiple steps that are required for a quantum key distribution and that are performed in software. While the security of QKD is based on the transmission of physical particles, e.g., photons, into which a secret bitstring is encoded, software plays a critical role for turning the transmitted bitstring into a functioning symmetric cryptographic key. In particular, the software performs error correction to eliminate, e.g., measurement errors and the software performs privacy amplification to counter potential leakage during the transmission of the physical particles. We provide cache-side-channel leakage bounds for implementations of the encoding step for error correction, the decoding step for error correction, and the privacy-amplification step from the QKD software used at the Department of Physics at

TU Darmstadt [100]. As in the previous case, our leakage bounds are with respect to four different cache-side-channel attacker models and with respect to a cache configuration that reflects the Level 1 cache of the Intel Skylake architecture [62]. The leakage bounds lead to the detection of a vulnerability in the implementation of the encoding step. Moreover, the bounds were precise enough to verify the security of a hardened version of the implementation.

We evaluate our program analysis based on the leakage bounds for the QKD implementation. As in the ring-TESLA case, we follow the usual practice of simplifying the original implementation to avoid unnecessary analysis complexity. In case of the QKD implementation, our simplifications reduce the complexity in four aspects. More concretely, our simplifications avoid object orientation, dynamic memory allocation, potentially infinite loops, and global variables. In our evaluation, we detected a vulnerability in the encoding implementation of the QKD software from [100]. This vulnerability might leak the entire secret key through a cache side channel. That is, an attacker might be able to break the QKD without even attacking the particle transmission and without access to a quantum computer. The vulnerability does not only occur in the software from [100], but also in the original implementation of the LDPC error-correcting code from [97], which has also been forked by many others [52]. We propose a mitigation for this vulnerability and verify its effectiveness with our program analysis. We also discuss how the security guarantees that we obtain for the hardened QKD software relate to the traditional QKD security guarantees that focus on attackers with quantum computers. Our mitigation has been adopted by the physicists who maintain the software and has been integrated in a new version of the software, which is now used for the further development of QKD setups at the TU Darmstadt Physics Department.

Contributions Related to the Fourth Program Analysis The contributions related to the fourth program analysis in our suite are an abstract domain, an abstract semantics, and a set of quantitative security guarantees.

The abstract domain, called $\overline{Cos}$, captures the possible snapshots of the state of a pipelined out-of-order CPU during a program execution. In particular, the domain captures the possible states of an instruction pipeline that supports branch prediction with a static always-not-taken prediction policy (like the Null Prediction supported by the BOOM RISC-V architecture [29]) and out-of-order execution based on implicit register renaming using reservation stations [125]. Furthermore, the domain tracks the relation between the possible pipeline states and the possible states of a fully-associative Level 1 data cache. To our knowledge, $\overline{Cos}$ is the first abstract domain for side-channel quantification that captures the combination of caching and pipelining, i.e., the combination from which side channels exploited in attacks like, e.g., Spectre-PHT [72] arise. The main challenge in the development of this domain was to find an abstraction based on which a program analysis remains feasible while computing precise leakage bounds at the same time. We show that our domain outperforms a candidate alternative in terms of both, precision and analysis complexity already for a very small example program.

The abstract semantics, called upd^{α}, captures the semantics of the language pASM with respect to the abstract domain $\overline{Cos}$. As usual for a formal semantics that models a new type of side channel, the underlying language is simplified. More concretely, the language pASM, which we define in this thesis, is much simpler than x86 assembly, but supports the basic assembly instructions that are relevant for the analysis of side channels that arise from caching and pipelining. The semantics is formalized rigorously and reliably overapproximates the changes to architectural and micro-architectural components, including the cache, pipeline stages, reservation stations, and reorder buffer, in each clock cycle during the execution of a pASM program.

The resulting program analysis is called Spectrescope and is defined in this thesis. Since the analysis is a very recent contribution, its automation is planned, but not yet completed.

The quantitative security guarantees that we compute with Spectrescope are leakage bounds with respect to side channels that arise from the combination of caching and pipelining. We compute such bounds for pASM programs that capture two excerpts from the Linux kernel. One

excerpt corresponds to a known vulnerability that occurs in kernel version 4.16.8 [138]. The second excerpt corresponds to the mitigation of this vulnerability that has been deployed in the Linux kernel starting from version 4.16.9. In addition, we provide leakage bounds with respect to a range of simple *p*ASM programs that showcase the benefits of Spectrescope for the evaluation of partial mitigations and the prioritization of mitigation efforts.

Overall, Spectrescope is the first program analysis that can provide quantitative security guarantees with respect to cache side channels that arise from the combination of caching and pipelining. All prior program analyses for such side channels, including [28, 13, 126, 57, 58, 27], are purely qualitative. We evaluate Spectrescope based on our leakage bounds computed with the analysis. In the excerpts from the Linux kernel, Spectrescope is able to quantify the leakage through the known vulnerability and to verify the effectiveness of the corresponding mitigation. For a simple example program based on the original Spectre code snippet from [72], Spectrescope verifies that a partial mitigation reduces the side-channel leakage by 87.5% at comparatively low performance cost. The partial mitigation is based on a bit-mask and incurs only a small overhead, because it does not prevent speculative execution.

1.3 Structure of this Thesis and Contributions per Chapter

In Chapter 2 of this thesis, we introduce the preliminaries for and the notation used in this thesis. In particular, we introduce the attacker models and the basic notions (from information theory and abstract interpretation) underlying our analyses, as well as the background on the cryptographic implementations that we analyze.

The original contributions of this thesis are described in Chapter 3-6. Each of these chapters describes one analysis from our program-analysis suite. The elements discussed in each chapter are visualized in Figure 1.2.

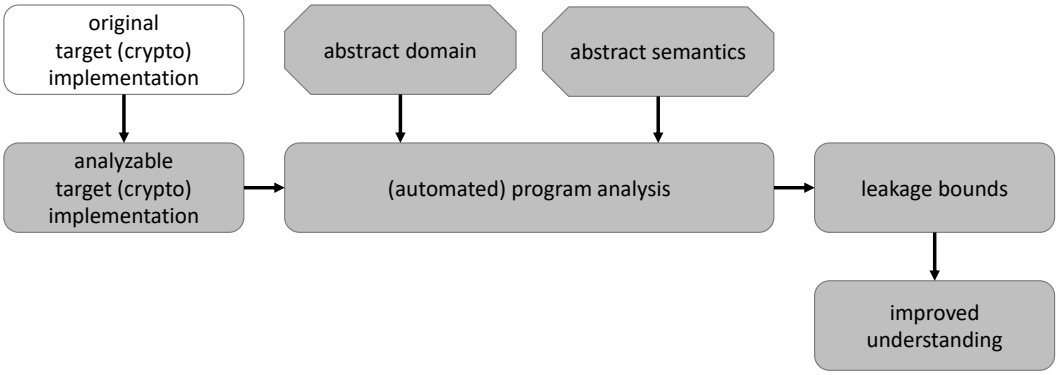

Figure 1.2: Refined Structure of Program Analysis and Evaluation

Each chapter begins with a description of the target implementations that are analyzed in the respective chapter. In Chapter 3, these are the AES implementations from the different crypto libraries. In Chapter 4, this is the simplified ring-TESLA implementation. In Chapter 5, these are the simplified implementations of the different QKD steps. In Chapter 6, these are *p*ASM programs corresponding to excerpts from the Linux kernel. Subsequently, each chapter presents the program analysis, including the respective abstract domain and abstract semantics. The automatic tool support (where applicable) is discussed next, followed by a description of the

analysis setup for the evaluation of the analysis on the target implementations. The resulting leakage bounds and the improved understanding that arises from our evaluation is discussed in the end of each chapter. The detailed structure of this discussion differs across chapters depending on the nature of the evaluation targets and the findings in the evaluation.

In the end of this thesis, we discuss related work in Chapter 7 before concluding in Chapter 8. Appendix A contains the raw leakage bounds that are visualized in the figures and tables of Chapter 3. Appendix B contains the detailed formalization of the concrete execution model for *p*ASM programs, from which the program analysis in Chapter 6 abstracts.

Figure 1.3 visualizes in detail how Chapter 3-6 refine the elements from Figure 1.2. We summarize the mapping of individual contributions to chapters below.

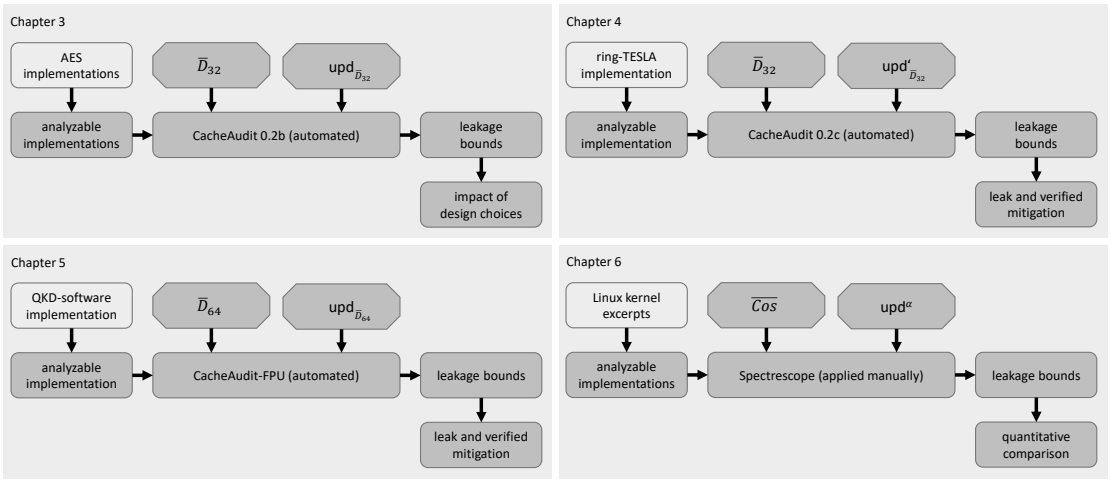

Figure 1.3: Instantiated Structures of Program Analyses and Evaluations

Chapter 3: Analysis across AES Implementations

This chapter describes the following contributions:

- the abstract domain $\overline{\mathcal{D}}_{32}$ that captures the possible snapshots of the architectural and micro-architectural components involved in the execution of x86 binaries, including the status flags SF and OF,
- the abstract semantics $\mathtt{upd}_{\overline{\mathcal{D}}_{32}}$ for x86 instructions with respect to $\overline{\mathcal{D}}_{32}$,
- the automation of the program analysis based on $\overline{\mathcal{D}}_{32}$ and $\mathtt{upd}_{\overline{\mathcal{D}}_{32}}$ in CacheAudit 0.2b, and
- quantitative cache-side-channel security guarantees across different AES implementations, different attacker models, different cache sizes, and different side-channel countermeasures.

The chapter also describes the insights gained about the effect of the implementation technique used for the last transformation round in AES implementations and about the effects of other factors like, e.g., the attack surface and cache size. The raw leakage bounds underlying the presentation and discussion in this section are provided in Appendix A.

Chapter 4: Analysis of ring-TESLA Implementation

This chapter describes the following contributions:

- the abstract semantics $\mathtt{upd}'_{\overline{\mathcal{D}}_{32}}$ that captures the semantics of x86 instructions, including instructions for handling larger operands, with respect to $\overline{\mathcal{D}}_{32}$,
- the automation of the program analysis based on $\overline{\mathcal{D}}_{32}$ and $\mathtt{upd}'_{\overline{\mathcal{D}}_{32}}$ in CacheAudit 0.2c, and

- quantitative cache-side-channel security guarantees for the ring-TESLA signature generation and the four individual functions in which vulnerabilities were detected.

The chapter also describes the side-channel leakage detected in the ring-TESLA implementation based on our leakage bounds and the verification of the mitigations for this leakage.

Chapter 5: Analysis of QKD-Step Implementations

This chapter describes the following contributions:

- the abstract domain $\overline{\mathcal{D}}_{64}$ that captures the possible snapshots of an architecture with ALU and FPU execution units during the execution of x86/87 binaries,
- the automation of the program analysis based on $\overline{\mathcal{D}}_{64}$ in CacheAudit-FPU, and
- quantitative cache-side-channel security guarantees for the decoding, encoding, and privacy-amplification implementations of the QKD software from [100].

The chapter also describes the vulnerability that we discovered in the QKD implementation, the mitigation that we developed, the verification of this mitigation, and the lifting of the security guarantees to overall QKD solutions.

Chapter 6: Analysis with Instruction Pipelining

This chapter describes the following contributions:

- the abstract domain $\overline{Cos}$ that captures the possible snapshots of a CPU that features caching and instruction pipelining with branch prediction and out-of-order execution,
- the abstract semantics upd^{α} for the assembly language pASM with respect to $\overline{Cos}$,
- the program analysis Spectrescope based on $\overline{Cos}$ and upd^{α}, and
- quantitative cache-side-channel security guarantees with respect to a known vulnerability and corresponding mitigation based on excerpts from the Linux kernel.

The chapter also describes how Spectrescope supports the evaluation of partial mitigations and the prioritization of mitigation efforts at the example of simple assembly programs. The basic execution model for pASM programs that underlies our analysis is described in Appendix B.

Chapter 2

Preliminaries and Notation

We capture cache-side-channel leakage based on four attacker models. The general approach underlying our program analyses is based on concepts from information theory and abstract interpretation. The crypto implementations that we target implement AES, the lattice-based signature scheme ring-TESLA, and the software parts of the BB84 protocol for QKD.

Notation and Conventions Throughout this thesis, we denote the power set of a set X by $\mathcal{P}(X)$. For the set of all sequences that consist of elements from a set X, we write $X^\star$. For the set of all partial functions from a set X to a set Y, we write $X \rightharpoonup Y$. We write $\mathbb{N}$ for the set of all natural numbers without zero and $\mathbb{N}_0$ for the set of all natural numbers including zero. We write $\mathbb{Z}$ for the set of all integers and $\mathbb{B}$ for the set of all Boolean values.

We write $dom(f)$ and $rng(f)$ for the domain and the range of a function f, respectively. If f is a partial function, we write $f(x)\uparrow$ if $x \notin dom(f)$, and $f(x)\downarrow$ if $x \in dom(f)$.

We use the notation $\langle x_1, \ldots, x_n \rangle$ for a sequence with the elements $x_1, \ldots, x_n$. We write $|\tau|$ for the length of a sequence τ and $\tau[i]$ for the i-th element of a sequence τ. For the concatenation of a sequence τ with a sequence τ', we write $\tau \bullet \tau'$.

We write KiB for Kibibytes, i.e., binary Kilobytes where $1\,\text{KiB} = 1024\,\text{B}$. When we report leakage bounds for an implementation, we round them to one decimal place and truncate them to the maximum leakage that is possible, i.e., the amount of secret information that the respective implementation processes.

2.1 Preliminaries on Attacker Models

The main focus of this thesis are cache side channels, which we capture using four existing attacker models. We also build on one of these attacker models to capture attacks through side channels that arise from the combination of caching with pipelining.

2.1.1 Caches

Caches are small memories that are located close to the CPU. They store selected entries from the main memory so that the CPU can access them more quickly. This speeds up the execution of programs with memory accesses that adhere to the principle of locality, i.e., the execution of programs that are likely to access the same or a nearby memory address in close succession.

The main memory is split into so-called memory blocks of a fixed size. When an address from a memory block is accessed, this block is loaded into the cache. The cache is partitioned into so-called cache lines whose size corresponds to the size of a memory block. Each memory block that gets loaded into the cache gets loaded into one such cache line.

In a set-associative cache, the cache lines are organized in equally-sized groups of cache lines, called cache sets. If each cache set contains k cache lines, the cache is called k-way set associative. Each memory block can be cached in all cache lines of one cache set. This cache set is determined based on the address of the memory block in the main memory. A fully-associative cache is a special case in which the entire cache consists of only one cache set, i.e., each memory block can be cached in any cache line of the cache.

When the CPU tries to read a value from the memory, the address of the corresponding memory block is determined by dropping the least-significant bits of the value's address (these bits define the offset within the memory block at which the value is stored). Based on the address of the memory block, the cache set to which the block belongs is determined. To this end, the least-significant bits of the memory-block address are used. It is then checked whether this cache set already contains the desired memory block. A situation in which the memory block is available in the cache set is called a cache hit. In case of a cache hit, the CPU reads the data directly from the cache. This is comparatively quick, because no access to the main memory is required.

A situation in which the memory block is not available in the cache set is called a cache miss. In case of a cache miss, the CPU reads the desired data from the main memory, which is comparatively slow. To prepare for future accesses to the same memory block (which, according to the principle of locality, are likely to occur), the memory block is then added to the cache set.

When the CPU performs a write access to a memory block, a quick reuse of the block is less likely than in case of a read access. This is reflected in caches by a so-called no-write-allocate policy. Caches that implement a no-write-allocate policy do not load blocks into the cache based on write accesses. Caches with write-allocate policy also load blocks based on write accesses.

To handle the case in which there is no room in the cache for a new block that shall be cached, caches implement so-called replacement policies. Such policies define which memory block to drop from the cache to make room for the new block. A replacement policy is usually based on some heuristic which block is least likely to be needed in the near future, e.g., the block that has been in the cache for the longest time (FIFO) or the block that has been least recently used (LRU).

Contemporary hardware usually uses not just one cache but a hierarchy of multiple caches, starting from the smallest Level 1 (L1) cache, which is located very close to the CPU core and has very low latency. Higher level caches are increasingly bigger and slower because they are located farther away from the CPU core. In multiprocessor systems, the L1 cache is usually exclusive to one CPU core and higher levels of cache are shared between increasingly many CPU cores. The last-level cache (LLC) is usually shared among all CPU cores.

2.1.2 Cache-Side-Channel Attacker Models

Cache side channels are based on the difference between the time required to process a cache hit and the time required to process a cache miss. They can be exploited using three different types of attacks: time-based, trace-based, or access-based.

A time-based cache-side-channel attack like, e.g., [16], extracts information from the running time of the program under attack. Such an attack exploits that this running time depends on the number of cache hits and cache misses that occur during the execution of the program. If the memory blocks that the CPU accesses during the execution depend on a secret, an attacker might be able to deduce information about this secret from the number of cache hits and cache misses and, hence, from the execution time.

We call the trace of cache hits, cache misses, and steps without cache accesses that occur during a program execution the cache trace of the execution. The attacker model *time* captures a time-based cache-side-channel attack on a program execution in terms of the execution's cache trace. Each cache hit from the cache trace is considered to take *HIT* units of time, each cache miss is considered to take *MISS* units of time, and each other step is considered to take *NONE* units of time. The sum of time units across all elements in the cache trace is the attacker observation under *time*. Unless stated otherwise, we use $HIT = 3$, $MISS = 20$, and $NONE = 1$.

An example of a practical attack that is captured by this model is an attack in which the attacker collects timing measurements of program executions in an offline phase and an online phase. In the offline phase, the attacker runs the program under attack on a local copy of the system under attack. He runs the program on multiple possible secret values and measures the time taken by each of the program runs. If public values are also involved in the computation, he collects timing measurements across the possible public values so that he obtains a set of timing measurements that characterize each possible secret value. In the online phase of the attack, the attacker collects timing measurements on the actual system under attack while it is processing the actual secret information. He compares these measurements to the measurements from the offline phase to deduce information about the secret value that is used on the system under attack. The techniques used to deduce such information often involve a series of distinguishing attacks, in which the attacker learns the secret in a bit-wise or Byte-wise fashion by considering timing measurements across selected sets of public values only.

A trace-based cache-side-channel attack like, e.g., [1] extracts information from the cache trace of an execution of the program under attack. Under the attacker model *trace*, an attacker observes the entire cache trace of the execution of the program under attack. This model captures all trace-based attacks, independently of how the attacker obtains the cache trace. In a practical attack, the attacker might obtain the trace by measuring the power consumption during the program execution (cache hits and cache misses can be distinguished based on power consumption, because the activation of additional memory components consumes additional power) or by monitoring pseudo files in which the operating system stores statistics about cache usage.

In an access-based cache-side-channel attack, the attacker controls a spy process that runs on the same system as the program under attack. The spy process shares a cache with the program under attack and interacts with this cache either synchronously (i.e., before and after the execution of the program under attack) or asynchronously (i.e., interleaved with the program under attack). In this thesis, we consider only synchronous access-based attacks.

A synchronous access-based attacker extracts information from the state of the shared cache after a run of the program under attack. If the memory blocks that the program under attack loads into the cache depend on a secret, e.g., because the program accesses an array at a secret-dependent index, the attacker can deduce information about the secret from the cache state. The resolution at which the attacker can observe the cache state depends on whether his spy process shares the accessed memory blocks with the program under attack. If the programs share the same memory blocks, e.g., because they make use of a common cryptographic library, the attacker can observe the exact addresses of the memory blocks in the cache. If the programs do not share memory blocks, the attacker can still observe how many blocks are cached in which cache set.

The attacker model *acc* captures synchronous access-based cache-side-channel attackers who share the same memory blocks with the program under attack. Under this model, an attacker observes the cache line in which each memory block is cached after a run of the program under attack. The attacker model *accd* captures synchronous access-based attackers who do not share memory blocks with the program under attack. Under this model, an attacker observes for each cache set the number of memory blocks that it contains after a run of the program under attack.

The models *acc* and *accd* capture access-based attackers independently of how they place the spy process on the system under attack and independently of how they infer the cache state. The former can be achieved by an attacker, e.g., using techniques to facilitate co-residency on the same cloud machine as the program under attack [117]. The latter can be achieved by measurement techniques that bring the cache into a controlled state before the run of the program under attack and then measure the amount of time that it takes to access individual memory blocks during or after the program run. Numerous measurement techniques exist, including, e.g., Prime+Probe [102], Evict+Time [102], Flush+Reload [135], and Reload+Refresh [25].

Note that, the four attacker models *acc*, *accd*, *trace*, and *time* were originally defined in [45].

2.1.3 Cache-Side-Channel Attacker Model for Systems with Pipelining

For our analysis that takes into account side channels that arise from the combination of caching and instruction pipelining, we capture attackers also based on the attacker model *acc* from Section 2.1.2. That is, we consider attackers who can make high-resolution, synchronous, access-based observations in the same way as for a sequential in-order execution. Only the possible program executions after which they make these observations are different.

Access-based attacks, as captured by *acc*, have been used in practice to exploit side channels that arise from caching and pipelining. As an example, consider the code snippet in Figure 2.1. The code snippet operates on a parameter `x`, two public arrays `array1` and `array2`, and the variable `array1_size` that contains the size of `array1`. When Line 1 is executed, the CPU retrieves the value of `array1_size` and checks whether `x` is within bounds of `array1`. When Line 2 is executed, the CPU first retrieves the entry at index `x` of `array1`. We call this entry `k`. Then, the CPU retrieves an entry from `array2`, using `k*4096` as the index. That is, the memory block that is accessed in this step depends on the value of `k`. Due to the multiplication with `4096`, the memory block is uniquely determined by `k` on a system with block size of 4 KiB or less. If `array2` is not cached prior to the execution of the code snippet, the accessed memory block is loaded into the cache upon the cache miss during the execution of Line 2. That is, the memory block from `array2` that is contained in the cache after the execution is uniquely determined by the value of `k`. The code snippet leaks the value of `k` through a cache side channel that could be exploited in a synchronous access-based attack (as captured by the attacker model *acc*).

```
1  if (x < array1_size)
2    y = array2[array1[x] * 4096];
```

Figure 2.1: Spectre-PHT Code Snippet from [72]

Under the assumption of sequential in-order execution, the value of `k` is a public value from `array1` in any possible execution of the code snippet. The reason is that the guard in Line 1 ensures that Line 2 is only executed if `x` is within bounds of `array1`. That is, the code snippet does not leak secret information if executed sequentially and in-order.

On a system that features an instruction pipeline with branch prediction and out-of-order execution, there are possible executions of the code snippet that leak secret information. More concretely, executions are possible in which `x` is out-of-bounds with respect to `array1`, but in which Line 2 is executed speculatively because the guard in Line 1 is predicted to evaluate to `true`. Since `x` is out-of-bounds with respect to `array1`, the access to `array1[x]` retrieves a memory entry outside of `array1`. That is, the value of `k` is a secret value from a private memory entry. When the CPU accesses `array2` speculatively, it loads a memory block into the cache that depends on this private memory entry. This block remains in the cache even after the actual value for the guard in Line 1 is available. The reason is that the speculative execution is only partially reverted. The control flow is reverted to the correct branch (skipping Line 2) and the state of some micro-architectural components, e.g., the state of the registers, is reverted to remove the speculatively calculated values. However, the state of the cache is not reverted. An access-based cache-side-channel attack can be mounted to retrieve the secret value from the cache even after the speculative execution has ended. This example is the essence of the Spectre-PHT attack [72].

In addition to the cache-side-channel attack, a Spectre-PHT attack comprises two optimizations. To control which information is leaked, the attacker triggers an execution of the code snippet on a parameter `x` of his choice. To increase the chance that Line 2 will be speculatively executed and the leak will occur, he trains the branch predictor in advance by triggering executions on values of `x` that are within bounds of `array1`. Both of these capabilities are not part of *acc*, but they are captured implicitly in our program analysis.

2.2 Preliminaries on Side-Channel Quantification

Our program analyses are based on concepts from information theory, the program-analysis technique abstract interpretation, and an existing approach for combining both. The novelty of our analyses lies in the abstractions used.

2.2.1 Information-Theoretic Leakage Bounds

The leakage of a program can be measured using information-theoretic concepts as follows.

First, the program is modeled by an information-theoretic channel. Let $\mathcal{X}$ be the set of possible inputs to the program and $\mathcal{O}$ be the set of possible side-channel output values of the program, e.g., the possible final states of the cache. Let $Ch = (\mathcal{X}, \mathcal{O}, \mathcal{C})$ be a channel with the channel matrix $\mathcal{C} : \mathcal{X} \times \mathcal{O} \to [0, 1]$. An entry $\mathcal{C}(x, o)$ of the channel matrix captures the probability that the side-channel output is o if the program input is x. Let the channel Ch be discrete, i.e., $\mathcal{X}$ and $\mathcal{O}$ are sets of distinct symbols, memoryless, i.e., the side-channel output depends only on the current program input and not on prior inputs, and deterministic, i.e., $\forall x \in \mathcal{X}.\ \forall o \in \mathcal{O}.\ \mathcal{C}(x, o) \in \{0, 1\}$.

Then, the choice of the program input and the occurrence of a side-channel output are modeled by random variables. Let $\pi : \mathcal{X} \to [0, 1]$ be the probability distribution according to which the program input is selected. The event corresponding to the choice of the program input is captured by the random variable X_π with range $\mathcal{X}$ and probability mass function $p_{X_\pi} = \pi$. The event corresponding to the occurrence of a side-channel output is captured by the random variable O_π with range $\mathcal{O}$ and probability mass function $p_{O_\pi}(o) = \Sigma_{x \in \mathcal{X}} \mathcal{C}(x, o) \cdot \pi(x)$, because the probability of each side-channel output value depends on the input choice and the channel matrix.

Consider an attacker who has one try to guess the program input. His initial uncertainty about the program input, i.e., his uncertainty before making any side-channel observation, is captured by the min-entropy of the random variable X_π as follows [123]:

$$H_\infty(X_\pi) = -\log_2 \max_{x \in \mathcal{X}} \pi(x).$$

His remaining uncertainty about the program input after observing a side-channel output is captured by the conditional min-entropy, which is defined as follows for the deterministic case:

$$H_\infty(X_\pi | O_\pi) = -\log_2 \Sigma_{o \in \mathcal{O}} \max_{x \in \mathcal{X}} \mathcal{C}(x, o) \cdot \pi(x).$$

The side-channel leakage of the program can be measured by the decrease of the attacker's uncertainty that is caused by a side-channel observation, i.e., by

$$L_\infty(\pi, Ch) = H_\infty(X_\pi) - H_\infty(X_\pi | O_\pi).$$

That is, the leakage depends on the prior distribution π for the choice of the program input. The maximum leakage across all prior distributions is called the channel capacity:

$$ML_\infty(Ch) = \max_\pi L_\infty(\pi, Ch).$$

From [78, Theorem 1], it follows that

$$ML_\infty(Ch) \leq \log_2 |\{o \in \mathcal{O} \mid \Sigma_{x \in \mathcal{X}} \mathcal{C}(x, o) > 0\}|.$$

That is, the maximum leakage through the side channel is bounded by the logarithm of the number of side-channel output values that occur with non-zero probability, i.e., that might be observed by an attacker. Throughout this thesis, we use this fact to compute leakage bounds as bounds on the logarithm of the number of possible attacker observations through the side channel.

Remark 2.1. In the approach described above, the leakage bound refers to leakage about the program input from the set $\mathcal{X}$. Technically, this means that when applying the approach to a set $\mathcal{X}$ that covers all input parameters, all input parameters are considered secret information.

However, this does not mean that the approach is only applicable in scenarios where all input parameters are secret. To handle public parameters, one defines a family of channels $Ch_i = (\mathcal{X}, \mathcal{O}, \mathcal{C}_i)$ for each possible combination i of values for the public parameters. One then computes a leakage bound for each channel in the family as described before. The maximum of these leakage bounds is the overall leakage bound with respect to the secret parameter [45].

Remark 2.2. In addition to the distinction between public and secret parameters, one might want a more fine-grained distinction between different secret parameters that are not equally valuable. This point has already been made by Alvim, Scedrov, and Schneider [6]. In the context of crypto, a fine-grained distinction might, e.g., lead to even more informative bounds when applied to distinguish between the secret key and plaintext in the analysis of symmetric encryption.

Such a fine-grained distinction is, to date, out of scope for the existing approach described before. However, incorporating it is an interesting direction for future work (see Section 8.2).

2.2.2 Overapproximating Bounds with Abstract Interpretation

Computing the possible side-channel observations directly on a detailed model of execution is possible in principle, but often infeasible in practice for non-trivial programs. Using abstract interpretation [34], it is possible to overapproximate the set of reachable execution states and the set of the corresponding reachable side-channel observations. By performing a reachability analysis on an abstract representation of all program executions, the computation becomes feasible.

To perform abstract interpretation, one first defines a concrete domain $\mathcal{D}$, which captures the possible states of program executions, and a concrete semantics

$$upd_{\mathcal{D}} : \mathcal{D} \times \mathcal{I} \to \mathcal{D},$$

which captures the effect of each instruction from the set $\mathcal{I}$ on the execution state.

Next, one defines an abstract domain $\overline{\mathcal{D}}$, which captures the possible execution states on a more abstract level. For instance, if the concrete domain captures the value stored in each register by a number, the abstract domain might capture the value stored in each register by a set of numbers. This is called a set abstraction. An abstraction function

$$\alpha : \mathcal{P}(\mathcal{D}) \to \overline{\mathcal{D}}$$

defines how a set of concrete states is represented in the abstract domain. For a set abstraction, such an abstraction function might transform the number stored in each register of each concrete state into a singleton set that contains exactly this number and combine the sets for the same register across different concrete states using the set-union operation. In the reverse direction, a concretization function

$$\gamma : \overline{\mathcal{D}} \to \mathcal{P}(\mathcal{D})$$

maps each abstract state to the set of concrete states that it represents.

The changes that the execution of each instruction from the set $\mathcal{I}$ causes with respect to an abstract execution state from the domain $\overline{\mathcal{D}}$ is defined by an abstract semantics

$$upd_{\overline{\mathcal{D}}} : \overline{\mathcal{D}} \times \mathcal{I} \to \overline{\mathcal{D}}.$$

The abstract semantics $upd_{\overline{\mathcal{D}}}$ is sound with respect to the concrete semantics $upd_{\mathcal{D}}$ if

$$\forall D \in \mathcal{P}(\mathcal{D}).\ \forall i \in \mathcal{I}.\ \gamma(upd_{\overline{\mathcal{D}}}(\alpha(D), i)) \supseteq \{upd_{\mathcal{D}}(d, i) \mid d \in D\}$$

holds, i.e., if the abstract semantics overapproximates the set of program-execution states that are reachable according to the concrete semantics.

We use abstract interpretation to compute a superset $\overline{O} \supseteq \{o \in \mathcal{O} \mid \Sigma_{x \in \mathcal{X}} \mathcal{C}(x, o) > 0\}$ of the reachable side-channel output values that the attacker can observe. Since the logarithm is a monotonic function, $\log_2 |\overline{O}|$ is an upper bound on the side-channel leakage. This approach of combining abstract interpretation and information theory for side-channel quantification was pioneered in [77] and is also used in program analyses like, e.g., [45, 44].

2.2.3 CacheAudit Framework

The CacheAudit framework is a code base for quantitative cache-side-channel analysis that is implemented in OCaml. The original analysis tools implemented in the framework are called CacheAudit 0.1 [43] and CacheAudit 0.2 [45]. The input of program analyses in the framework are x86 binaries that are written in a well-defined subset of the x86 assembly language. The output of such program analyses are upper bounds on the cache-side-channel leakage of the binaries with respect the four attacker models *acc*, *accd*, *trace*, and *time* described in Section 2.1.2.

These leakage bounds are computed by an abstract interpretation of the x86 binary. Based on the reachable abstract states, the possible abstract attacker observations are determined. For each attacker model, the number of distinct concrete attacker observations that are captured by the reachable abstract observations is counted. Finally, the logarithm of this number for each attacker model is returned as an upper bound on the leakage to the respective attacker model.

The abstract domain and abstract semantics used by different program analyses in the CacheAudit framework and the resulting restrictions on the analyzable x86 binaries differ.

2.3 Preliminaries on Cryptographic Implementations

We consider multiple existing cryptographic implementations in this thesis, ranging from implementations of the Advanced Encryption Standard, to an implementation of the ring-TESLA scheme, to an implementation of Quantum Key Distribution.

2.3.1 Advanced Encryption Standard

AES is a symmetric-key block cipher that was proposed by Daemen and Rijmen under the name Rijndael [37] and standardized by the U.S. National Institute of Standards and Technology [96].

AES will likely remain secure even against attackers with quantum computers, because while quantum computers can be used to break computational hardness assumptions, they cannot provide an exponential speedup for search algorithms [108]. Unlike public-key cryptography, AES is not based on computational hardness assumptions, but on a substitution-permutation network.

AES encrypts a message using a secret key by applying multiple rounds of transformations. To this end, the message is split into blocks of size 128 bit and transferred into a 4×4 table representation where each table entry corresponds to one Byte of the message block.

The message block is then transformed using the secret key. There are three possible key sizes: 128 bit (with 10 transformation rounds), 192 bit (with 12 rounds), and 256 bit (with 14 rounds). Based on the key size, the secret key is expanded into so-called round keys that are used in the respective transformation rounds. Before the first transformation round, the first round key is added to the message block using bit-wise xor, resulting in an intermediate state of the 4×4 table.

The first transformation round then consists of four steps. The first step is called SubBytes. It substitutes the bits in each Byte of the intermediate state using a predefined substitution table called S-Box. The second step, called ShiftRows, transforms each row of the intermediate state by shifting the entries to the left within the row. The index by which the entries are shifted is determined by the row index and the key size. For a key size of 128 bit, the first row (with index 0) is not shifted at all, the second row (with index 1) is shifted by $C1 = 1$ entries, the third row (with index 2) is shifted by $C2 = 2$ entries, and the third row (with index 3) is shifted by $C3 = 3$ entries. The third step, called MixColumns, transforms each column of the intermediate state. To this end, each column is multiplied by a fixed matrix of constant values. Finally, the fourth step AddRoundKey is performed, in which the next round key is added to the intermediate state.

The subsequent rounds follow the same pattern as the first round, with the exception of the last round. In the last round, the third step (MixColumns) is skipped. The resulting state of the 4×4 table after all transformation rounds is the AES-encrypted ciphertext.

Traditionally, AES encryption is implemented using lookup tables. This approach was suggested in the original Rijndael proposal [37] in order to optimize the performance of AES implementations. The key idea is to store precomputed results for the transformations SubBytes, ShiftRows, and MixColumns in lookup tables, so-called T-tables, and to retrieve the results that are needed in each round by accessing the tables. There are four T-Tables (one for each row of the intermediate state), each consisting of 256 entries (one for each possible Byte to be transformed), where each entry consists of four Bytes. To compute the result of one transformation round for one column of the intermediate state, one looks up the values from the T-tables at the index corresponding to the value of the Byte to be transformed. The results of the lookups are added up using xor and the AddRoundKey transformation is performed by adding the round key in addition. The computation of a round output e with lookup tables is visualized in Figure 2.2 for round input a (the intermediate state of the 4×4 table), round key k, and S-Box S. The colors highlight how the individual round transformations are reflected in the lookup tables: Each table entry is based on an S-Box entry, multiplied by a constant value from the MixColumns matrix, i.e., encodes the SubBytes and MixColumns transformations. The ShiftRows transformation is encoded into the choice of the round-input Byte that is used for each table lookup. Finally, the round key is added using a regular xor operation.

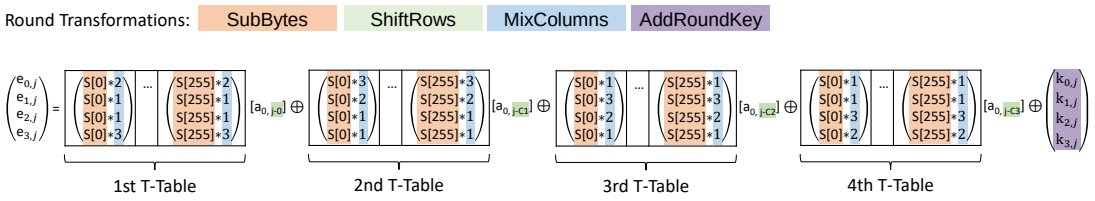

Figure 2.2: Use of Lookup Tables in AES Implementations

The last round of AES is a special case because the MixColumns transformation is skipped. The lookup tables cannot be directly reused from the other AES rounds. Daemen and Rijmen propose two options for implementation techniques to resolve this. The first option is to reuse the T-Tables, but to mask out the effects of the MixColumns transformation. The second option is to compute the last round directly using the S-Box and performing the remaining transformations without lookup tables. Alternatively, it is also possible to precompute a separate set of lookup tables for the last AES round that omits the MixColumns transformation.

All of these implementation techniques can be found in existing cryptographic libraries: OpenSSL (v. 1.0.1t) [101] provides an AES implementation that uses masked T-Tables. The libraries mbedTLS (v. 2.2.1-gpl) [9] and Nettle (v. 3.2) [94] provide AES implementations that compute the last round based on the S-Box. LibTomCrypt (v. 1.17) [79] provides an AES implementation that uses a separate set of lookup tables for the last round.

Note that, the lookups in the T-Tables happen at indices that depend on the previous intermediate state, which in turn depends on the secret key and message. This holds for the main transformation rounds, as well as for the above-mentioned approaches to implementing the last round. Even if the last round is computed without T-Tables, the lookups to the S-Box for the SubBytes step still happen at secret-dependent indices. That is, during a run of a T-table-based AES implementation, the memory is accessed at locations that indirectly depend on the secret AES key. The memory entries are loaded into the cache so that the contents of the cache also depends on the secret information. This can be exploited using access-based, trace-based, and time-based cache-side-channel attacks.

2.3.2 ring-TESLA

The scheme ring-TESLA [3] is a post-quantum digital signature scheme. Digital signature schemes are cryptographic primitives that allow one to certify and verify the authenticity of messages.

The security of traditional digital signature schemes often relies on computational hardness assumptions. Consider, for instance, RSA [119]. The RSA signature of a message m is $\sigma = m^d \mod N$, where $N = p \cdot q$ is the product of two prime numbers and where the secret key d and the public key e satisfy $e \cdot d \equiv 1 \mod ((p-1) \cdot (q-1))$ [69]. To obtain the secret key e (e.g., for forging a signature), an attacker would have to know $(p-1) \cdot (q-1)$, for which he would have to determine the prime factors p and q of N. The security of RSA signatures is based on the assumption that computing these prime factors is computationally infeasible.[1]

If an attacker has access to a quantum computer, he can efficiently solve some mathematical problems that were previously considered infeasible. This includes, e.g., the computation of prime factors, which can be solved efficiently using Shor's algorithms on a quantum computer [122]. That is, RSA signatures are not secure against attackers who have access to quantum computers. Such attackers could compute the secret key and use it to forge signatures.

To obtain signature schemes that are secure against attackers with quantum computers, post-quantum cryptography relies on mathematical problems that are likely to be computationally infeasible even in the presence of quantum computers. A promising such problem is the problem of finding the shortest vector in a mathematical object called lattice. This problem is called Shortest-Vector Problem (SVP). Lattice-based cryptography relies on the computational infeasibility of mathematical problems on lattices like, e.g., the SVP.

The lattice-based signature scheme ring-TESLA [3] relies on the infeasibility of solving the Ring-Learning-With-Errors (R-LWE) problem, which can be reduced to the SVP on a specific class of lattices [82]. More concretely, ring-TESLA is based on an instance of the R-LWE problem for the ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$, where $n = 2^k$ for some $k \in \mathbb{N}$. That is, R_q is a ring of polynomials with coefficients in $\mathbb{Z}_q$ (i.e., the ring of integers modulo a prime number q), where addition and multiplication are performed modulo $(x^n + 1)$, such that all polynomials have degree less than n.

Consider the polynomials $a_1, \ldots, a_m \in R_q$ and the so-called error polynomials $e_1, \ldots, e_m \in R_q$ for some $m \in \mathbb{N}$. The decisional variant of this R-LWE instance is to distinguish whether the polynomials $b_1, \ldots, b_m \in R_q$ were sampled uniformly random from R_q or whether they were constructed by computing $b_i = s \cdot a_i + e_i$ for some $s \in R_q$.

In ring-TESLA, the parameters are instantiated to $k = 9$, $n = 2^9 = 512$, and $m = 2$. The parameter q is either instantiated to 8 399 873 or to 39 960 577. The keys for the signature scheme ring-TESLA consist of five polynomials a_1, a_2, s, e_1, and e_2. The secret key is the triple (s, e_1, e_2). The public key is the pair (b_1, b_2) with $b_1 = a_1 \cdot s + e_1$ and $b_2 = a_2 \cdot s + e_2$, where a_1 and a_2 are also known publicly. If an attacker would be able to forge a signature, he would also be able to solve the R-LWE problem. He could distinguish a valid public key (b_1, b_2) from a random pair of polynomials, because the former yields a valid signature and the latter most likely does not.

The signature computation in ring-TESLA for a message m is depicted in Figure 2.3. A random polynomial y is sampled and multiplied with the public key polynomials a_1 and a_2, respectively. A hash function and an encoding function are subsequently applied to the tuple of the two resulting polynomials $a_1 \cdot y$, $a_2 \cdot y$, and the message m. The vector c resulting from the hashing and encoding is multiplied by the secret-key polynomial s and the random polynomial y is added to obtain $z = y + s \cdot c$. The signature consists of z and c and is only valid if the coefficients of z and the coefficients of two auxiliary polynomials $a_1 \cdot y - e_1 \cdot c$ and $a_2 \cdot y - e_2 \cdot c$ are within permitted ranges. If the signature is not valid, it is discarded and a new signature is computed.

We consider the integer-variant of the reference implementation of ring-TESLA [23]. The size of the secret key (s, e_1, e_2) in this implementation can be up to 49 152 bit (3 polynomials, each consisting of up to 512 coefficients that are stored as 32 bit integers). The signature generation in

[1] Note that, in practice, RSA signatures are not secure in the textbook variant described here but only in certain variants using padding techniques due to multiple known attacks [69].

the implementation closely follows the specification of the scheme. It uses rejection sampling and restarts the signature generation when invalid coefficients are encountered during the process.

Figure 2.4 shows the core of the function `crypto_sign` from the ring-TESLA integer implementation. It generates a signature for a message `m` based on the public polynomials `a1` and `a2` and the secret polynomials `e1`, `e2`, and `s`. Figure 2.5 provides an overview of the computation. The dark gray boxes correspond to inputs and outputs of the signature generation. The light gray boxes correspond to intermediate values computed during the signature generation and the white ellipses are function applications.

The signature generation generates candidate signatures and checks whether they are valid according to criteria implemented in the functions `test_w` (see Lines 9 and 12 in Figure 2.4) and `test_rejection` (see Line 15 in Figure 2.4). If the generated candidate signature is invalid, the signature generation is restarted. While the secret polynomials `e1`, `e2`, and `s` have to be kept secret in any case, the intermediate values `c` and `pos_list` only need to remain secret if the generated candidate signature is invalid and has to be discarded [23].

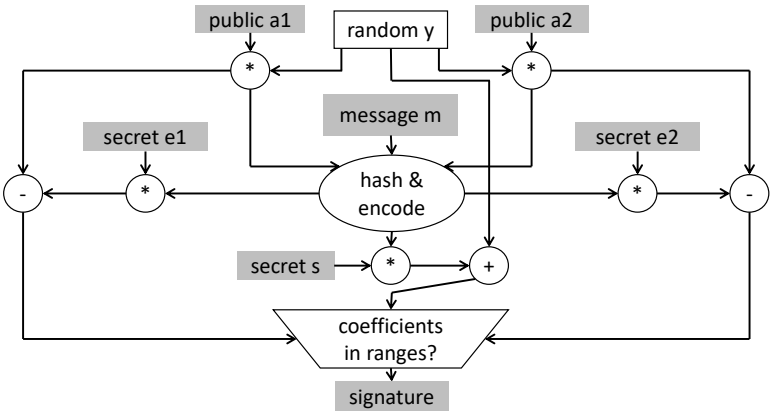

Figure 2.3: Overview of Signature Generation in ring-TESLA

```
1  while(1) {
2   sample_y(vec_y);
3   poly_mul_fixed(vec_v1, vec_y, poly_a1);
4   poly_mul_fixed(vec_v2, vec_y, poly_a2);
5   random_oracle(c, vec_v1, vec_v2, m, mlen);
6   generate_c(pos_list, c);
7   computeEc(E1c, sk+sizeof(int)*PARAM_N, pos_list);
8   poly_sub(vec_v1,vec_v1, E1c);
9   if (test_w(vec_v1) != 0){ continue; }
10  computeEc(E2c, sk+sizeof(int)*PARAM_N*2, pos_list);
11  poly_sub(vec_v2,vec_v2, E2c);
12  if (test_w(vec_v2) != 0){ continue; }
13  computeEc(Sc, sk, pos_list);
14  poly_add(vec_y, vec_y, Sc);
15  if (test_rejection(vec_y) != 0){ continue; }
16  for(i=0; i<mlen; i++){ sm[i]=m[i]; }
17  *smlen = CRYPTO_BYTES + mlen;
18  compress_sig(sm+mlen, c, vec_y);
19  return 0; }
```

Figure 2.4: Core of the ring-TESLA Function `crypto_sign`

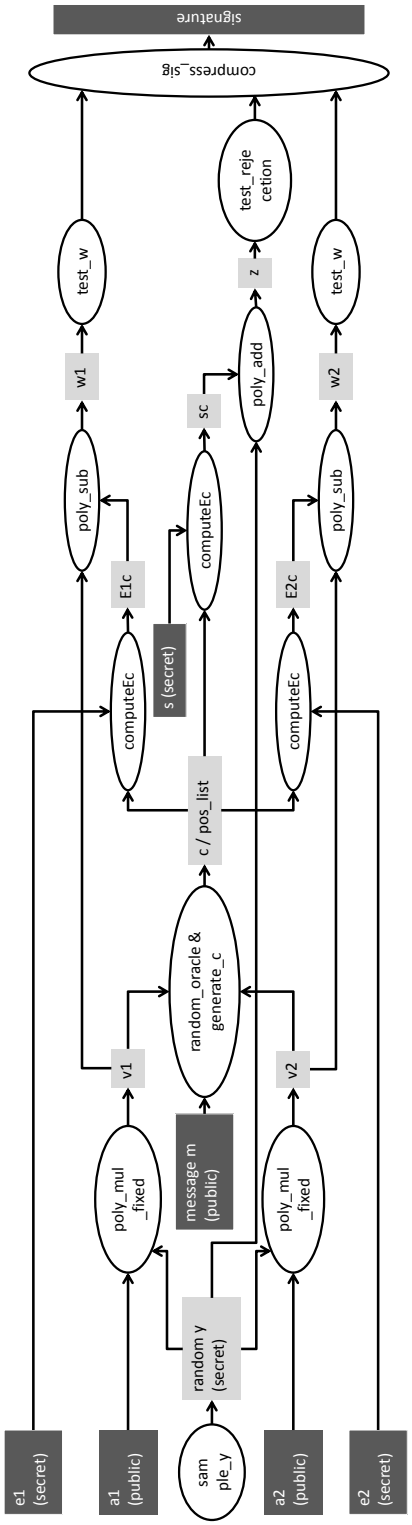

Figure 2.5: Data Flows in the ring-TESLA implementation

2.3.3 Quantum Key Distribution

Secret key cryptography like, e.g., AES relies on a shared secret key. This key should be known to the two parties that would like to communicate encryptedly, but should not be known to any third party. We call the two communicating parties Alice and Bob. In-person meetings between Alice and Bob are not always possible. Therefore, shared keys are often established remotely.

Traditionally, remote key establishment is done using key agreement protocols that exploit the computational hardness of mathematical problems. For instance, the Diffie-Hellman Key Exchange Protocol [41] is based on the computational hardness of computing discrete logarithms over cyclic groups. In Diffie-Hellman Key Exchange, Alice and Bob agree on a cyclic group G and a generating element $g \in G$. Alice and Bob pick random numbers $a \in \mathbb{N}$ and $b \in \mathbb{N}$, respectively. They compute g^a and g^b, respectively, and send these values to each other. To obtain a shared key g^{ab}, Alice computes $(g^b)^a = g^{ab}$ and Bob computes $(g^a)^b = g^{ab}$. A man-in-the-middle attacker who intercepts g^a and g^b would have to compute the discrete logarithm $\log_g(g^a)$ or $\log_g(g^b)$ to obtain the key g^{ab}. The security of Diffie-Hellman Key Exchange relies on the assumption that the computation of discrete logarithms is computationally infeasible using classical computers.[2]

As already mentioned in Section 2.3.2, such hardness assumptions are threatened by quantum computers. Quantum computers encode information as quantum bits (qbits) instead of bits. While bits can have the value 0 or 1, qbits can have values that are two-dimensional unit vectors [99]. Bits are physically encoded using different levels of voltage (low voltage for 0 and high voltage for 1). Qbits can be encoded using different quantum-physical states of particles, e.g., the polarization of photons with respect to a polarization base. A polarization base consists of two base vectors with respect to which the amplitude of the photon's polarization is measured. The linear combinations of the base vectors then correspond to the possible values of the qbit that the photon encodes.

That is, qbits can capture more information than traditional bits. Quantum algorithms solve mathematical problems while making use of the capability to encode information as qbits [99]. For instance, Shor proposed quantum algorithms that can be used to factor numbers and to compute discrete logarithms efficiently [122]. Thus, the hardness assumption underlying the security of Diffie-Hellman Key Exchange does not hold in the presence of quantum computers.

The goal of Quantum Key Distribution is to establish a shared secret key between Alice and Bob in the presence of an attacker who has access to a quantum computer, i.e., a post-quantum attacker. In the following, we refer to the post-quantum attacker as Eve.

A QKD solution consists of two phases: The transmission of a bitstring from Alice to Bob trough a quantum channel and the computation of a shared key from this bitstring through software-based postprocessing. The transmission through the quantum channel is based on qbits. For instance, the first protocol for QKD, BB84 [14], uses polarized photons to represent qbits. The subsequent software-based postprocessing consists of four steps. These postprocessing steps involve computations on Alice's and Bob's traditional computers, as well as communication between Alice and Bob over a traditional electromagnetic channel.

The first step, the transmission of polarized photons from Alice to Bob through the quantum channel, is referred to as *raw-key exchange*. The resulting bitstrings are referred to as raw keys. We denote Alice's raw key by $b_{\mathrm{r}}^{\mathrm{A}}$ and Bob's raw key by $b_{\mathrm{r}}^{\mathrm{B}}$. Alice's raw key might differ from Bob's raw key due to errors caused by inconsistent polarization bases, errors caused by the interference of Eve, and errors caused by imperfect measurement equipment. These errors are corrected in the following four steps of software-based postprocessing. Moreover, the postprocessing detects whether Eve obtained information about the secret key and either counters the leakage by privacy amplification or, if there is too much leakage, aborts and restarts the key distribution.

In the software-based postprocessing, Alice and Bob first remove the errors caused by inconsistent polarization bases. This step is referred to as *key sifting* and results in shorter bitstrings, referred to as sifted keys. We denote Alice's and Bob's sifted keys at this stage by $b_{\mathrm{si}}^{\mathrm{A}}$ and $b_{\mathrm{si}}^{\mathrm{B}}$.

[2] Note that, in practice, Diffie-Hellman Key Exchange is not secure in the textbook variant described here but only in more complex variants due to known man-in-the-middle attacks [69].

Next, Alice and Bob estimate the remaining error rate between their sifted keys, i.e., the error rate due to interference by Eve and due to imperfect measurement equipment. We denote the actual error rate between the sifted keys by err_{true}. The step in which Alice and Bob estimate this error rate is called *parameter estimation*. It results in an estimated error rate err_{est} and in shorter versions of the sifted keys on Alice's and Bob's side. These shorter bitstrings are also referred to as sifted keys. To distinguish them from the initial sifted keys, we denote them by b_{s}^{A} and b_{s}^{B}, respectively. A high estimated error rate err_{est} indicates leakage to Eve. If err_{est} exceeds a predefined threshold, the QKD is aborted. Otherwise, the postprocessing continues.

To obtain identical bitstrings on both sides, Alice and Bob perform the so-called *error correction* that removes measurement errors and errors due to interference by Eve. The resulting bitstring is called the error-corrected key and we denote it by $x_{\text{ec}}^{\text{A}}$and x_{ec}^{B}, respectively, where $x_{\text{ec}}^{\text{A}} = x_{\text{ec}}^{\text{B}}$.

Finally, Alice and Bob perform the so-called *privacy amplification*, which counters leakage to Eve during the previous steps of QKD. We denote the resulting so-called privacy-amplified key by x_{pa}^{A} and x_{pa}^{B}, respectively, where $x_{\text{pa}}^{\text{A}} = x_{\text{pa}}^{\text{B}}$. This key is the final result of the QKD and can be used to encrypt subsequent communication between Alice and Bob symmetrically.

The postprocessing phase, including key sifting, parameter estimation, error correction, and privacy amplification, happens based on software implementations. These implementations perform a sequence of operations on bitstrings in order to ensure that Alice and Bob obtain equal bitstrings that are not known to Eve. Since the postprocessing involves error correction, QKD implementations have to deal with probabilities. This distinguishes QKD implementations from implementations of, e.g., block ciphers like AES or public-key schemes like ring-TESLA.

In the following, we describe the individual steps of QKD and the corresponding traditional security guarantees for QKD in more detail.

Raw-Key Exchange

The BB84 protocol uses polarized photons to encode qbits for the raw-key exchange. The polarization of a photon is relative to a polarization base of two orthogonal vectors. In the BB84 protocol, two such bases are used: rectilinear $\oplus$ and diagonal $\otimes$. The rectilinear polarization base consists of the two vectors $\rightarrow$ and $\uparrow$. The diagonal polarization base consists of the vectors $\nearrow$ and $\nwarrow$. The polarization of a photon relative to base $\oplus$ is a linear combination, written as

$$\alpha \left|\uparrow\right\rangle + \beta \left|\rightarrow\right\rangle,$$

where α and β are the polarization amplitudes along the respective base vectors. Based on the polarization relative to the base $\oplus$, the corresponding polarization relative to the base $\otimes$ can be determined as

$$\gamma \left|\nearrow\right\rangle + \delta \left|\nwarrow\right\rangle \text{ with } \alpha = \sqrt{\frac{1}{2}} \cdot (\gamma + \delta) \text{ and } \beta = \sqrt{\frac{1}{2}} \cdot (\gamma - \delta).$$

To perform the raw key exchange, Alice first selects her raw key b_{r}^{A}, a random bitstring based on which the shared key shall be generated. She then polarizes photons to create qbits that represent this bitstring. For each photon, Alice randomly selects one of the polarization bases $\oplus$ and $\otimes$. If the selected base is $\oplus$, Alice encodes a 1 as the polarization $1 \left|\uparrow\right\rangle + 0 \left|\rightarrow\right\rangle$ (short: $\uparrow$) and a 0 as the polarization $0 \left|\uparrow\right\rangle + 1 \left|\rightarrow\right\rangle$ (short: $\rightarrow$). If the selected base is $\otimes$, Alice encodes a 1 as the polarization $1 \left|\nearrow\right\rangle + 0 \left|\nwarrow\right\rangle$ (short: $\nearrow$) and a 0 as the polarization $0 \left|\nearrow\right\rangle + 1 \left|\nwarrow\right\rangle$ (short: $\nwarrow$). Alice then transmits her polarized photons to Bob, e.g., through free space or fiber cables.

Bob receives the photons and measures their polarization, e.g., using avalanche photo diodes as in [48]. For each photon, Bob chooses randomly with respect to which of the two polarization bases he measures. Then he converts the measured polarizations back into a bitstring. This bitstring is Bob's raw key b_{r}^{B}. That Bob chooses the bases randomly instead of receiving them from Alice is crucial for the detection of attacks that happens in the parameter-estimation stage.

In an idealized setting, if Bob chooses the same polarization base that Alice used to polarize the photon, he receives the same bit that Alice wanted to transmit. If Bob chooses a different polarization base than Alice, he has a 50% chance to measure a 1 and a 50% chance to measure a 0. That is, he has a 50% chance to receive the correct bit.

Example 2.3.1. Alice selects the random bitstring $b_{\mathrm{r}}^{\mathrm{A}} = 101000$ on which the key establishment shall be based. For the polarization of the corresponding photons, she selects the sequence of polarization bases $\oplus$, $\otimes$, $\oplus$, $\oplus$, $\otimes$, $\oplus$. Then Alice polarizes her photons with the sequence of polarizations $\uparrow$, $\nwarrow$, $\uparrow$, $\rightarrow$, $\nwarrow$, $\rightarrow$ as shown in the following table.

Bit to encode	1	0	1	0	0	0
Selected base	$\oplus$	$\otimes$	$\oplus$	$\oplus$	$\otimes$	$\oplus$
Resulting polarization	$\uparrow$	$\nwarrow$	$\uparrow$	$\rightarrow$	$\nwarrow$	$\rightarrow$

To measure the polarization of the photons received from Alice, Bob chooses the sequence of polarization bases $\oplus$, $\oplus$, $\oplus$, $\otimes$, $\otimes$, $\otimes$. Then he might measure the sequence of polarizations $\uparrow$, $\uparrow$, $\uparrow$, $\nwarrow$, $\nwarrow$, $\nearrow$, which would correspond to the bitstring $b_{\mathrm{r}}^{\mathrm{B}} = 111001$ as shown in the following table.

Selected base	$\oplus$	$\oplus$	$\oplus$	$\otimes$	$\otimes$	$\otimes$
Measured polarization	$\uparrow$	$\uparrow$	$\uparrow$	$\nwarrow$	$\nwarrow$	$\nearrow$
Decoded bit	1	1	1	0	0	1

$\Diamond$

In reality, measurement errors might occur that cause Bob to receive an erroneous bit even if he measures with the correct polarization base. Moreover, Eve might interfere with the quantum-channel transmission and introduce additional errors. We refer to the error rate resulting from measurement errors and Eve's interference as err_{true}.

The Conventional Attacker Model in QKD

Traditionally, security guarantees for QKD solutions consider an attacker Eve who has access to a quantum computer and can intercept and actively interfere with transmissions on both, the quantum channel and the traditional channel.

On the quantum channel, Eve can intercept the photons sent by Alice. She can perform arbitrary measurements on intercepted photons with any existing or hypothetical measurement equipment. She can also create arbitrarily polarized single or entangled photons and send them to Bob. In doing so, Eve has to comply with the laws of quantum physics. In particular, the uncertainty principle of quantum physics states that measuring the polarization of a photon in one polarization base disturbs the polarization in the other polarization base [49].

More concretely, if Eve intercepts a photon and measures the polarization, she has to choose between the polarization bases $\oplus$ and $\otimes$ for her measurement. With a probability of 50% she chooses the correct base used by Alice. In this case, she can measure the polarization and send a correct copy of the intercepted photon to Bob. If she guesses the wrong polarization base, she measures a random polarization and can only guess which polarization to forward to Bob. On average, Eve introduces errors in 25% of the photons she intercepted. This increase in the error rate is used to detect attacks in the parameter-estimation step of QKD. Note that, the detection is only possible because the correct polarization bases are kept secret until the raw key exchange is completed. If Eve knew the bases, she could avoid introducing errors and being detected.

The photons available to Eve after intercepting transmissions on the quantum channel might include entangled photons, which cannot be represented in the form $\alpha\,|\uparrow\rangle + \beta\,|\rightarrow\rangle$. An alternative way to represent the polarization of a set of photons is a so-called density matrix ρ_E [99].

Example 2.3.2. Consider two non-entangled photons with polarization states

$$\alpha_1 \left|\uparrow\right\rangle + \beta_1 \left|\rightarrow\right\rangle \text{ and } \alpha_2 \left|\uparrow\right\rangle + \beta_2 \left|\rightarrow\right\rangle.$$

We can represent this pair of photons by the density matrix

$$\rho_E = \begin{pmatrix} \alpha_1\alpha_2\alpha_1\alpha_2 & \alpha_1\alpha_2\alpha_1\beta_2 & \alpha_1\alpha_2\beta_1\alpha_2 & \alpha_1\alpha_2\beta_1\beta_2 \\ \alpha_1\beta_2\alpha_1\alpha_2 & \alpha_1\beta_2\alpha_1\beta_2 & \alpha_1\beta_2\beta_1\alpha_2 & \alpha_1\beta_2\beta_1\beta_2 \\ \beta_1\alpha_2\alpha_1\alpha_2 & \beta_1\alpha_2\alpha_1\beta_2 & \beta_1\alpha_2\beta_1\alpha_2 & \beta_1\alpha_2\beta_1\beta_2 \\ \beta_1\beta_2\alpha_1\alpha_2 & \beta_1\beta_2\alpha_1\beta_2 & \beta_1\beta_2\beta_1\alpha_2 & \beta_1\beta_2\beta_1\beta_2 \end{pmatrix}.$$

◊

On the traditional channel used in the remaining steps of the QKD, Eve can also intercept and send messages. However, she cannot impersonate others. By intercepting all transmissions on this channel, Eve can obtain the sequence of correct polarization bases (from the key-sifting step), a sample section removed from Alice's and Bob's sifted keys (from the parameter-estimation step), and parity bits for Alice's remaining sifted key (from the error-correction step).

The set E models the information available to Eve. It covers ρ_E, the polarization bases, the sample sections of the sifted keys, the parity bits, and any public information. That is, the conventional attacker model captures attackers who have access to quantum computers and arbitrary measurement equipment but who cannot impersonate others on the traditional channel.

Key Sifting and Parameter Estimation

In the key-sifting step, Bob sends Alice the polarization bases that he used in the raw-key exchange and Alice replies which bases were correct [92]. For each mismatch in the polarization bases, Alice and Bob discard the respective bits from the raw keys $b_{\mathrm{r}}^{\mathrm{A}}$ and $b_{\mathrm{r}}^{\mathrm{B}}$ to obtain the sifted keys $b_{\mathrm{si}}^{\mathrm{A}}$ and $b_{\mathrm{si}}^{\mathrm{B}}$. On average, 50% of the raw keys are discarded in this step.

Example 2.3.3. In the transmission from Example 2.3.1, the second, fourth and sixth polarization base mismatch. Alice and Bob would discard the corresponding bits from their raw keys $1, 0, 1, 0, 0, 0$ and $1, 1, 1, 0, 0, 1$. They would obtain the sifted key $1, 1, 0$ on both sides. ◊

In the above example, $b_{\mathrm{si}}^{\mathrm{A}} = b_{\mathrm{si}}^{\mathrm{B}}$, i.e., the error rate err_{true} between the sifted keys $b_{\mathrm{si}}^{\mathrm{A}}$ and $b_{\mathrm{si}}^{\mathrm{B}}$ is zero. In practice, error rates are usually non-zero due to imperfect measurement equipment and, in case of an attack, due to errors introduced by Eve.

In the parameter-estimation step, Alice and Bob compute an estimation err_{est} of err_{true}. To this end, they cut out a randomly selected set of bits from both of their sifted keys $b_{\mathrm{si}}^{\mathrm{A}}$ and $b_{\mathrm{si}}^{\mathrm{B}}$ [92]. They obtain the shorter sifted keys $b_{\mathrm{s}}^{\mathrm{A}}$ and $b_{\mathrm{s}}^{\mathrm{B}}$, respectively, and a sequence of the selected sample bits on each side. They compare these sample bits and use the resulting error rate as err_{est}.

If err_{est} is non-zero, this might indicate an attack. However, a few percent of measurement errors are usually due to imperfect measurement equipment [92] and Alice and Bob cannot distinguish between measurement errors and errors introduced by Eve. Therefore, the countering of attacks happens in two stages. In the parameter-estimation step, Alice and Bob check whether err_{est} exceeds a predefined threshold that indicates an attack. If this is the case, they abort the key exchange and discard the sifted keys. If err_{est} is below the predefined threshold, Alice and Bob proceed with the shortened sifted keys $b_{\mathrm{s}}^{\mathrm{A}}$ and $b_{\mathrm{s}}^{\mathrm{B}}$. Any attacks that are overlooked in this step will be countered in the privacy-amplification step at a later stage during the postprocessing.

In addition to countering attacks, the error rate err_{est} is also used as a basis for removing any remaining errors from $b_{\mathrm{s}}^{\mathrm{A}}$ and $b_{\mathrm{s}}^{\mathrm{B}}$ in the subsequent postprocessing step.

Error Correction

Alice and Bob use an error-correcting code to remove the errors from b_s^A and b_s^B. Suitable codes include Low-Density Parity Check (LDPC) [51] and Polar Codes [67]. The QKD implementation that we analyze in this thesis is based on binary (n, k) LDPC codes.

In a binary (n, k) LDPC code, a k-bit message is encoded into an n-bit codeword by appending $m = n - k$ parity bits. The parity bits are defined by a $k \times n$ parity matrix H that contains a low but fixed number of 1s. A codeword is valid, i.e., the message and parity bits match if

$$H \cdot c \mod 2 = \vec{0}.$$

First, Alice and Bob fix the parameters k, n, and m, as well as the parity matrix H, based on the estimated error rate err_{est}. In theory, all errors can be corrected if the number of parity bits is high enough such that $m > sl$, where

$$sl = k \cdot \left(err_{\text{true}} \cdot \log_2 \left(\frac{1}{err_{\text{true}}} \right) + (1 - err_{\text{true}}) \cdot \log_2 \left(\frac{1}{1 - err_{\text{true}}} \right) \right) \text{ [106].}$$

In practice, slightly more parity bits (e.g., $m \approx 1.25 \cdot sl$) are used to account for noise and sub-optimal codes [106]. The parameters k, n, m, and H are public.

Next, Alice and Bob split their sifted keys into blocks of length k. For each block, Alice performs an encoding step in which she computes the parity bits for her sifted-key block. Alice sends the parity bits to Bob, who performs a decoding step in which he computes the most likely permutation of his sifted-key block that matches the parity bits.

To compute the parity bits, Alice derives a generator matrix G from the parity matrix H: She first brings H into the form $H' = [A \mid I_m]$, where $[A \mid I_m]$ is a matrix consisting of an $m \times k$ matrix A on the left and the $m \times m$ identity matrix I_m on the right. Second, she transposes A to A^T and composes it with the $k \times k$ identity matrix I_k to obtain

$$G = [I_k \mid A^T].$$

She then writes her sifted-key block as a row vector $block_s^A$ and computes

$$[block_s^A \mid parity_s^A] = block_s^A \cdot G \mod 2,$$

which is the codeword consisting of the sifted-key block and the parity bits $parity_s^A$. Then she sends $parity_s^A$ to Bob over the electromagnetic channel.

Bob receives the parity bits and computes the most likely permutation $block_s^{B'}$ resulting from flipping bits in his sifted-key block $block_s^B$ such that

$$H \cdot [block_s^{B'} \mid parity_s^A] \mod 2 = \vec{0},$$

where $[block_s^{B'} \mid parity_s^A]$ is a column vector consisting of the permuted sifted-key block computed by Bob and the parity bits received from Alice.

On Alice's side, the resulting error-corrected key is identical to her sifted key, i.e., $x_{ec}^A = b_s^A$. Bob obtains x_{ec}^B by concatenating the error-corrected blocks $block_s^{B'}$ across all sifted-key blocks. If err_{est} overapproximates err_{true}, the error-corrected keys are identical, i.e., $x_{ec}^A = x_{ec}^B$.

The shared bitstring $x_{ec}^A = x_{ec}^B$ is, however, not a shared secret key yet. Recall that the parameter-estimation step only counters attacks that exceed a predefined threshold. Moreover, additional information is leaked to Eve since she can intercept the parity bits exchanged during the error-correction step. To reduce the information that Eve has about the error-corrected key, Alice and Bob perform the final step of QKD: the privacy amplification.

Privacy Amplification

In the privacy-amplification step, Alice and Bob reduce the correlation between information available to Eve and the error-corrected key. They turn the error-corrected key into a shorter key, called the privacy-amplified key. To this end, they use a 2-universal family of hash functions [15].

A hash function is a function that maps inputs to fixed-size outputs. If two distinct inputs are mapped to the same output, this is called a collision. A 2-universal family of hash functions is defined as a family of hash functions such that for any pair of inputs, the probability for a collision in a randomly selected function from the family is low. More concretely, the probability for a collision has to be less than or equal to the probability that two random values collide [50].

The QKD implementation that we analyze in this thesis uses a specific family of hash functions, namely multiplications with Toeplitz matrices. Toeplitz matrices are defined as matrices in which each descending diagonal consists of equal values.

In a privacy amplification with Toeplitz matrices, Alice and Bob first fix a length l for the privacy-amplified key blocks and a public $l \times k$ Toeplitz matrix T.

Next, both Alice and Bob compute the product

$$T \cdot block_{\mathrm{ec}} \mod 2$$

for each k-bit block $block_{\mathrm{ec}}$ of their respective error-corrected keys $x^{\mathrm{A}}_{\mathrm{ec}}$ and $x^{\mathrm{B}}_{\mathrm{ec}}$, written as column vectors. They concatenate the resulting privacy-amplified-key blocks to obtain their respective privacy-amplified keys $x^{\mathrm{A}}_{\mathrm{pa}}$ and $x^{\mathrm{B}}_{\mathrm{pa}}$. Since $x^{\mathrm{A}}_{\mathrm{ec}} = x^{\mathrm{B}}_{\mathrm{ec}}$, it follows that $x^{\mathrm{A}}_{\mathrm{pa}} = x^{\mathrm{B}}_{\mathrm{pa}}$.

The privacy-amplified key $x^{\mathrm{A}}_{\mathrm{pa}} = x^{\mathrm{B}}_{\mathrm{pa}}$ is the overall result of the QKD and can be used as the secret key for the symmetric encryption of the subsequent communication between Alice and Bob.

The security of this key against Eve depends on the length l of the privacy-amplified key blocks. Assume that Eve obtains at most r bits of information about each k-bit sifted-key block during the raw-key exchange. During the postprocessing phase, Eve learns at most m additional bits about each sifted-key block, namely the parity bits exchanged in the error-correction step. She does not learn additional bits from the key sifting (the polarization bases she intercepts cannot be used to cover attacks at this stage), parameter estimation (the sample bits that Eve might intercept are discarded), and privacy amplification (no information is sent). Roughly speaking, Eve's remaining uncertainty about each k-bit error-corrected key block is $k - r - m$. The smaller the length l of the privacy-amplified key blocks is compared to $k - r - m$, the more secure is the resulting privacy-amplified key. Formally, the security guarantees for QKD are based on a notion of security for the privacy-amplified key with respect to the conventional QKD attacker model.

Conventional Security Guarantees for QKD

Eve's remaining uncertainty about a sifted-key block is captured formally using the information-theoretic concept of smooth min-entropy [116]. As introduced in Section 2.2.1, min-entropy is a notion of an attacker's uncertainty about a secret, given the probability distribution of the secret. More concretely, min-entropy captures the success probability of an attacker who has one try to guess the secret and guesses the most likely value [123]. Smooth min-entropy is a variant of min-entropy that is parametric in a smoothness parameter ϵ.

For a given probability distribution, smooth min-entropy corresponds to min-entropy with respect to the worst-case distribution that is ϵ-close to the given distribution. The smooth min-entropy $H^{\epsilon}_{\infty}(\rho_{x^{\mathrm{A}}_{\mathrm{ec}}E}|E)$ captures the uncertainty of Eve with smoothness-parameter ϵ, given the set E of information available to Eve. The density matrix $\rho_{x^{\mathrm{A}}_{\mathrm{ec}}E}$ captures the correlation between the polarization states available to Eve and the error-corrected key $x^{\mathrm{A}}_{\mathrm{ec}}$. Given a fixed E, $\rho_{x^{\mathrm{A}}_{\mathrm{ec}}E}$ corresponds to the probability distribution of the error-corrected key from the perspective of an attacker who makes optimal measurements on the available polarization states in E.

The desired security of the privacy-amplified key is also formalized in terms of density matrices. More concretely, the security is captured as the L_1-distance $d(\rho_{f(x^{\mathrm{A}}_{\mathrm{ec}})Ef}|Ef)$ between

the distribution of the privacy-amplified key from the attacker's perspective and a uniform random distribution [116]. Here, $\rho_{f(x_{\text{ec}}^{\text{A}})Ef}$ captures the correlation between the privacy-amplified key $f(x_{\text{ec}}^{\text{A}})$ and the attacker's information E, given a hash function f. Again, for fixed E and f, $\rho_{f(x_{\text{ec}}^{\text{A}})Ef}$ corresponds to the probability distribution of the privacy-amplified key from the perspective of an optimal attacker. The conventional security guarantees for QKD relate the desired level of security to the attacker uncertainty required to achieve this level of security. Formally,

$$d(\rho_{f(x_{\text{ec}}^{\text{A}})Ef}|Ef) \leq 2 \cdot \epsilon + 2^{-\frac{1}{2}(H_{\infty}^{\epsilon}(\rho_{x_{\text{ec}}^{\text{A}}E}|E)-l)} \text{ [116, } p.\text{ 85]}.$$

That is, to obtain a privacy-amplified key that is close to a random value from the attacker's perspective, the length l should be selected small enough compared to the attacker's uncertainty $H_{\infty}^{\epsilon}(\rho_{x_{\text{ec}}^{\text{A}}E}|E)$ (the smaller l, the closer to random is the distribution of x_{ec}^{A} for the attacker).

Chapter 3

Cache-Side-Channel Quantification across AES Implementations

3.1 Introduction

AES is one of the most popular classical crypto algorithms. It is a block cipher that can be used to encrypt messages and it has been approved by the U.S. National Institute for Standards and Technology for the protection of sensitive information by the U.S. Government [96]. Since it is not based on hardness assumptions, AES is likely not threatened by quantum computers [108].

As described in Section 2.3.1, AES encryption happens in multiple rounds. Each round transforms each block of the secret message with multiple transformations, based on a round key that is generated from the secret AES key. The developers of AES, Daemen and Rijmen [37], suggested an implementation technique to efficiently compute the round transformations by storing precomputed results in so-called T-Tables. AES implementations that follow this approach are available in multiple crypto libraries, e.g., OpenSSL (v. 1.0.1t) [101] and mbedTLS (v. 2.2.1-gpl) [9].

For the last round of AES, the transformations differ from the main rounds. Hence, the T-Tables cannot be reused directly and a design choice has to be made in the implementation of this round. Existing AES implementations resolve this design choice in different ways: OpenSSL 1.0.1t masks the T-Tables, LibTomCrypt 1.17 [79] uses a second set of lookup tables, and mbedTLS 2.2.1 and Nettle 3.2 [94] directly use the S-Box to compute the last round of AES.

Since the T-Tables and S-Box are accessed at indices that depend on the secret key and message, the table-based implementation approach might lead to cache-side-channel vulnerabilities. While the threat of cache-side-channel attacks on table-based AES implementations is known [16, 102, 59], the influence of the last-round implementation has not been in the focus so far.

In this chapter, we provide a quantitative study of the cache-side-channel leakage across AES implementations with different last-round implementations and AES implementations with different side-channel countermeasures in place. Our study clarifies the influence of the last-round implementation on quantitative security guarantees with respect to cache-side-channel leakage across different cache configurations. For instance, we find that the masking of the original T-tables in the last round, as implemented in OpenSSL, leads to the best security guarantees with respect to access-based attackers. Furthermore, the security guarantees with respect to access-based attackers stabilize with increasing cache size across all three implementation techniques, as soon as the cache size allows for an injective mapping from the memory blocks of the lookup tables to cache sets. This insight is, e.g., helpful for the navigation of the security-performance

trade-off in AES implementations that shall be deployed across systems with different cache sizes.

To make such a comparative study possible, we develop an abstract domain and abstract semantics for quantifying cache-side-channel leakage following the approach described in Section 2.2. Unlike the prior state-of-the-art abstract interpretation for cache-side-channel quantification [45], ours is applicable across multiple target AES implementations. In particular, our abstract domain captures the CPU flags SF (sign flag) and OF (overflow flag), which occur in the x86 code for the AES implementations. Finding a suitable level of granularity at which to capture these flags is critical for the precision of security guarantees, because the flags determine the control flow of the AES implementations. Our abstract semantics captures the effect of x86 instructions on these flags at a granularity that allows us to compute meaningful leakage bounds across the AES implementations. Another aspect that distinguishes our semantics from the state of the art is that it captures 33 previously unsupported x86 instructions.

We implement our abstract domain and semantics in the CacheAudit framework to create tool support for the automatic application of our analysis to the target implementations.

In Section 3.2, we describe the AES implementations that we target. In Section 3.3, we describe our abstract domain and semantics for analyzing the AES implementations. In Section 3.4, we describe the implementation of our abstract domain and semantics in the CacheAudit framework. We describe our analysis setup in Section 3.5. In Section 3.6, we describe our quantitative analysis of the cache-side-channel security of an AES implementation from the library mbedTLS. In Section 3.7, we compare the cache-side-channel security across the lookup-table-based implementations from LibTomCrypt, mbedTLS, Nettle, and OpenSSL. Finally, we quantify and compare the effectiveness of different cache-side-channel countermeasures in Section 3.8.

3.2 Target AES Implementations

We analyze x86 binaries of AES implementations from the four crypto libraries LibTomCrypt, mbedTLS, Nettle, and OpenSSL. To this end, we use wrapper functions that call two functions from each library: One function that performs the round-key generation for AES (e.g., `mbedtls_aes_setkey_enc` from mbedTLS) and one function that performs the encryption of one message block (e.g., `mbedtls_aes_encrypt` from mbedTLS). The functions that we analyze from each library are listed in Table 3.1.

We configure all AES implementations to use a message size and key size of 128 bit, respectively. This key size complies with the U.S. NIST's recommendation to use at least 128 bit security

library	configuration	analyzed functions
LibTomCrypt 1.17	`ENCRYPT_ONLY`, `LTC_NO_ASM`, `ARGTYPE`	`rijndael_enc_setup`, `rijndael_enc_ecb_encrypt` (`aes.c`)
mbedTLS 2.2.1	`MBEDTLS_AES_ROM_TABLES`, removed `MBEDTLS_PADLOCK_C`	`mbedtls_aes_setkey_enc`, `mbedtls_aes_encrypt` (`aes.c`)
Nettle 3.2	default	`aes128_set_encrypt_key` (`aes128-set-encrypt-key.c`), `aes128_encrypt` (`aes-encrypt.c`)
OpenSSL 1.0.1t	default	`private_AES_set_encrypt_key`, `AES_encrypt` (`aes_core.c`)

Table 3.1: Analyzed AES Implementations

strength to protect sensitive data in unclassified applications beyond the year 2031 [12, Section 5.6.2]. We compile the wrappers for the AES implementations using `gcc 4.8.4` with the flags `-m32 -fno-stack-protector`.

In the following, we refer to the four target binaries that result from this compilation by LibTomCrypt AES, mbedTLS AES, Nettle AES, and OpenSSL AES.

3.3 Program Analysis for AES Implementations

Our program analysis uses abstract interpretation to compute the logarithm of the number of possible attacker observations as an upper bounds on min-entropy leakage. The underlying abstract domain and abstract semantics support the analysis of AES implementations without any modifications to the off-the-shelf code (see the target implementations in Section 3.2).

We define the following abstract domain for our abstract interpretation:

$$\begin{aligned}\overline{\mathcal{D}}_{32} =& (\{CF, ZF, SF, OF\} \to \mathbb{B}) \rightharpoonup \\ & ((\mathcal{R}_{32} \cup \mathcal{M}_{32} \to \mathcal{P}(\mathcal{V}_{32})) \times (\mathcal{M}_{32} \to \mathcal{P}(\mathcal{C}_{\text{pos}})) \times \mathcal{P}(\{h, m, n\})^\star)\end{aligned}$$

Each element of the abstract domain represents a set of possible snapshots of the CPU state during the execution of a program.

The symbols CF, ZF, SF, and OF capture four status flags that occur in x86 CPUs: the carry flag, the zero flag, the sign flag, and the overflow flag. The CPU sets these flags based on the results of arithmetic or comparison instructions. The CPU reads these flags when executing a conditional jump instruction. More concretely, the flags are used to determine whether the execution performs a jump or whether the execution proceeds along the regular control flow without jumping. Each flag in the CPU stores one bit of information. This is modeled by a mapping from flags to Boolean values in our abstract domain.

The sets $\mathcal{R}_{32}$ and $\mathcal{M}_{32}$ model the CPU registers and memory addresses. The set $\mathcal{V}_{32}$ models the possible values that might be stored in registers or memory entries. In our abstract domain, the possible values of each register and the memory entry at each memory address are captured by a mapping from registers and memory addresses to sets of values.

The set $\mathcal{C}_{\text{pos}}$ models the possible positions (i.e., all cache lines) at which a memory entry might be cached within its cache set. We use the special symbol $\bot \in \mathcal{C}_{\text{pos}}$ to capture that a memory entry is not cached. The possible snapshots of the cache during a program execution are modeled in our abstract domain by a mapping from memory addresses to cache lines.

The symbols h, m, and n model the occurrence of a cache hit, cache miss, and no memory access, respectively. Each element of the abstract domain captures the possible cache traces of a program execution that might have occurred up to the point at which the execution snapshot that is represented by the element is taken. The possible cache traces are captured by a sequence of sets. Each set in the sequence captures the possible cache interactions during one step (i.e., the execution of one x86 instruction) of the program execution.

Overall, our abstract domain uses set abstractions to represent the possible values of registers and memory, the possible cache states, and the possible cache traces encountered. For the CPU flags, the abstract domain preserves more precise information. More concretely, it preserves the relation between possible values of the flags and possible states of registers, memory, and cache. To this end, it maps each possible flag combination to the abstract states of registers, memory, and cache that are possible under this flag combination.

The key novelty of this abstract domain is that it covers all CPU flags that are required for the analysis of our target AES implementations. In particular, the prior domain from [45] covers neither the sign flag nor the overflow flag.

Jump instructions that branch on the sign flag and overflow flag occur in multiple AES implementations: `Jl` (opcode 0x0F8C) occurs in the binary of LibTomCrypt AES decryption and

`Jnle` (opcode 0F8F) occurs in the binary of mbedTLS AES. `Jl` is an instruction that triggers a jump to a target address if the sign flag and the overflow flag have non-equal values [63]. `Jnle` triggers a jump if the zero flag is not set and the sign and overflow flag have equal values. To support conditional jump instructions like `Jl` and `Jnle` without capturing the sign and overflow flags in the abstract domain, one would have to assume that always both branches can be taken. This would lead to an imprecise overapproximation of the possible executions of a binary and, hence, to unnecessarily high overapproximations of the binary's leakage.

Let $\mathcal{I}_{\text{CA}}$ be the set of x86 instructions supported by the analysis in [45]. We define the abstract semantics for our program analysis as a function

$$\texttt{upd}_{\overline{\mathcal{D}}_{32}} : \overline{\mathcal{D}}_{32} \times \mathcal{I}_{32} \to \overline{\mathcal{D}}_{32} \qquad \text{for} \qquad \mathcal{I}_{32} = \mathcal{I}_{\text{CA}} \cup \mathcal{I}_{\text{AES}}.$$

The instructions contained in $\mathcal{I}_{\text{AES}}$ are listed by their opcodes in Table 3.2.

Consider, for example, the instruction 0FA4 (`Shld`). This instruction performs a left-shift of a destination operand (register or memory entry), given a pattern of bits to shift in from the right (provided in a register) and an offset (provided as an immediate) [63]. The CPU sets the flags depending on the result, e.g., the carry flag to the last bit that is shifted out of the destination.

We define the abstract semantics of this instruction using an auxiliary function

$$\begin{aligned}\texttt{aux-upd}_{\overline{\mathcal{D}}_{32}} : ((\{CF, ZF, SF, OF\} \to \mathbb{B}) \times (\mathcal{R}_{32} \cup \mathcal{M}_{32} \to \mathcal{P}(\mathcal{V}_{32})))& \\ \to ((\{CF, ZF, SF, OF\} \to \mathbb{B}) \to (\mathcal{R}_{32} \cup \mathcal{M}_{32} \to \mathcal{P}(\mathcal{V}_{32}))),&\end{aligned}$$

that maps a concrete flag state and an abstract state of registers and memory to a mapping from flag states to abstract register and memory states. The abstract semantics takes an abstract state $\overline{d} \in \overline{\mathcal{D}}_{32}$ and applies this auxiliary function to each possible concrete flag state and the abstract register and memory states to which the flag state is mapped in $\overline{d}$. Then, the abstract semantics joins the resulting updated auxiliary states into one abstract state by applying the set union operator to the sets of possible values that can occur for each register or memory entry under the same flag state across the updated auxiliary states. Furthermore, the abstract semantics combines the resulting states with the abstract cache state and abstract cache trace that result from an access to the destination operand, in case the operand is stored in memory.

The function $\texttt{aux-upd}_{\overline{\mathcal{D}}_{32}}$ defines the possible updates to flags, registers and memory. To this end, it maps each flag combination that might result from the update to the join of the possible abstract register and memory states that might result from the left-shift based on any combination of values from the given sets of operand values. Thereby, the function overapproximates the concrete semantics of the `Shld` instruction precisely with respect to the abstract domain $\overline{\mathcal{D}}_{32}$.

Overall, our abstract semantics differs from the prior semantics in [45] by covering 33 additional x86 instructions and by modeling the semantics of all covered instructions with respect to the sign and overflow flags. This abstract semantics allows us to analyze our target AES implementations from LibTomCrypt, mbedTLS, Nettle, and OpenSSL.

Type	New Instructions
Arithmetic	2D (Sub), 18 (Sbb), 19 (Sbb), 11 (Adc), F7/6 (Div), 3C (Cmp)
Logic	08 (Or), 30 (Xor), 84 (Test), A9 (Test), F6/0 (Test)
Bitstring	0FA4 (Shld), 0FA5 (Shld), 0FAC (Shrd), 0FAD (Shrd)
Stack	07 (Pop)
Jump	7C, 0F8C, 7D, 0F8D, 70, 0F80, 71, 0F81, 78, 0F88, 79, 0F89, 7E, 0F8E, 7F, 0F8F (all Jcc)
Move	0F48 (Cmovs)

Table 3.2: x86 Instructions Contained in the Set $\mathcal{I}_{\text{AES}}$

3.4 Tool Support for the Analysis

We automate the analysis described in Section 3.3 by an implementation in the CacheAudit framework. The resulting tool, called CacheAudit 0.2b, takes an x86 binary that consists of instructions from the set $\mathcal{I}_{32}$ as input. It returns four cache-side-channel leakage bounds for the binary, which are computed as the logarithms of the numbers of possible concrete attacker observations under the attacker models *acc*, *accd*, *time*, and *trace*.

The analysis workflow of CacheAudit 0.2b is visualized in the sequence diagrams in Figure 3.1 and Figure 3.2. The parts that are highlighted in gray in the diagrams show where the novel aspects of our abstract semantics are implemented. For the remaining analysis workflow, CacheAudit 0.2b reuses existing code from the CacheAudit framework.

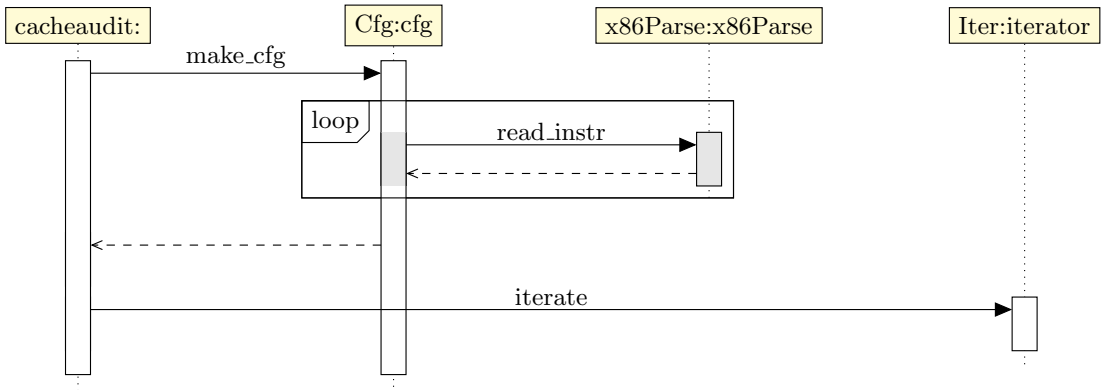

Figure 3.1: Sequence Diagram of the Overall Analysis Workflow

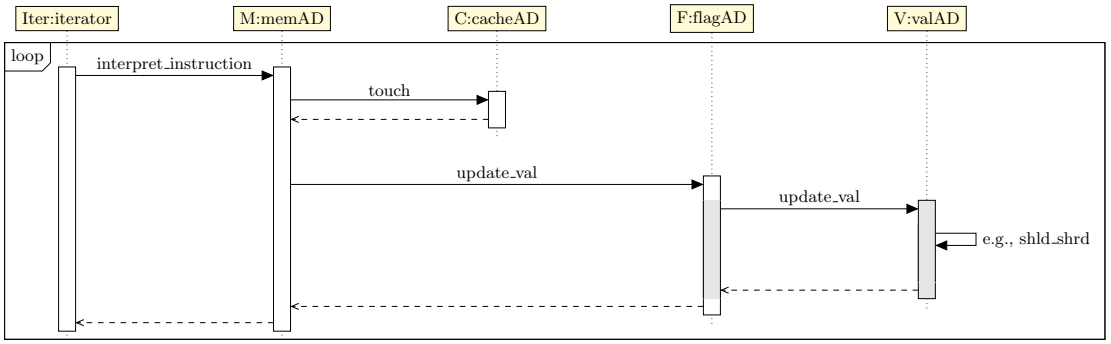

Figure 3.2: Sequence Diagram of the Abstract Semantics Computation

Figure 3.1 visualizes the key parts of the overall analysis workflow. Given an x86 binary, the analysis tool first constructs a control-flow graph. To this end, it parses the individual instructions from the x86 binary in the function `read_instr` of the module `x86Parse`. In our CacheAudit 0.2b, this function is able to parse all instructions from the set $\mathcal{I}_{32}$, i.e., it can parse the newly supported instructions from the set $\mathcal{I}_{\text{AES}}$ in addition to the previously supported instructions from the set $\mathcal{I}_{\text{CA}}$. Based on the resulting control-flow graph, the analysis tool iteratively applies the abstract semantics, using the function `iterate` of the module `Iterator`.

Figure 3.2 visualizes how the changes to an abstract state caused by an individual instruction are computed based on the abstract semantics. The iterator calls the function `interpret_instruction` of the module `memAD`. This function then performs two steps: Firstly, it updates the abstract cache state based on any memory accesses that the interpreted instruction

might perform given the possible operands captured by the preceding abstract state. To this end, `interpret_instruction` calls the function `touch` of the module `cacheAD` for each possible memory access. Secondly, `interpret_instruction` computes the possible updates to registers and memory by calling the function `update_val` of the module `flagAD`.

The function `touch` of the module `cacheAD`, which is called in the first step, computes the abstract cache state based on one memory access. To this end, it calls the auxiliary function `one_touch` of the module `cacheAD` for each possible position of the accessed memory entry in the cache. The function `one_touch`, computes an abstract cache state that captures the changes caused by an access to the respective memory entry if it is cached at the respective position. If the position is $\perp$, i.e., in case of a cache miss, the accessed entry is loaded to the first cache line and all other memory blocks in the same cache set are moved one cache line ahead. If the position is not $\perp$, i.e., in case of a cache hit, the positions of all memory entries are updated according to the replacement policy. The function `touch` joins the abstract cache states resulting from each call of `one_touch` in order to overapproximate all possible changes to the cache.

The function `update_val` of the module `flagAD`, which is called in the second step, takes care of splitting the abstract state based on the possible concrete flag combinations, calling the function `update_val` of the module `valAD` for each possible flag combination, and joining the resulting abstract values. The function `update_val` of the module `valAD` corresponds to the auxiliary function $\texttt{aux-upd}_{\overline{\mathcal{D}}_{32}}$ from our abstract semantics. For instance, if the instruction to interpret is `Shld`, `update_val` calls the function `shld_shrd` of the module `valAD`, in which we implemented the abstract semantics of `Shld` for a flag combination, a set of possible values for the source operand, and a set of possible values for the destination operand.

Finally, the resulting abstract state is computed based on the updated components (abstract cache, registers, and memory), before moving on to the next instruction from the target binary.

Note that, in addition to the implementation of the parsing and abstract semantics for the instructions from $\mathcal{I}_{\text{AES}}$, CacheAudit 0.2b also features an updated implementation of the abstract semantics for the instructions from the set $\mathcal{I}_{\text{CA}}$, which captures the semantics of these instructions with respect to the previously unsupported sign and overflow flags.

Based on the overall more powerful abstraction underlying CacheAudit 0.2b, the binaries of all four target off-the-shelf AES implementations can now be analyzed with the tool automatically.

CacheAudit 0.2b is available open source under `http://www.mais.informatik.tu-darmstadt.de/assets/tools/CacheAudit-ESSoS17.zip`.

3.5 Analysis Setup

Like all analyses in the CacheAudit framework, CacheAudit 0.2b assumes a system with one single level of cache and is parametric in the configuration of this cache.

For our study, we focus on a four-way set-associative data cache with 64 B line size, a configuration used, e.g., for the Level 2 cache in the Intel Skylake micro-architecture [62, Table 2-4]. We focus on the FIFO replacement strategy and vary the cache size from 2 KiB to 128 KiB.

We apply the analysis tool to each of the target binaries to obtain upper bounds on their leakage through cache side channels with respect to the four models *acc*, *accd*, *time*, and *trace*.

3.6 Analysis of mbedTLS AES

In this section, we focus on the AES implementation from mbedTLS. We first study the leakage of mbedTLS AES to attackers under *acc* and then discuss the effect of other attacker models.

3.6.1 Leakage of mbedTLS AES under *acc*

Table 3.3 shows the leakage bounds that we obtained with CacheAudit 0.2b on mbedTLS AES with respect to attackers under the model *acc*. We computed bounds across varying cache sizes from 128 KiB down to 2 KiB. All bounds in this chapter are truncated to the maximum leakage of 256 bit (128 bit message and 128 bit key).

Cache Size [KiB]	128	64	32	16	8	4	2
Leakage [bit]	69.0	69.6	69.6	71.2	91.8	114.5	92.6

Table 3.3: Leakage Bounds for mbedTLS under *acc*

Leakage bound for 128 KiB Cache Size For cache size 128 KiB, the leakage bound for mbedTLS AES is 69 bit. That is, at most 27% of the secret bits in the message and AES key are leaked to an attacker under *acc* for a cache size of 128 KiB.

To understand why the leakage bound is 69 bit, recall that it depends on the secret key and message which lookup-table entries are accessed in which order. However, an attacker will not be able to observe memory accesses at the granularity of individual lookup-table entries. Memory is accessed at the granularity of memory blocks, which have the same size as cache lines.

The mbedTLS implementation of AES uses 4 KiB of lookup tables for the main AES rounds as suggested by Daemen and Rijmen [37]. For the last round of AES, the implementation computes the results on the fly, using only the 0.25 KiB S-Box for table lookups. This amounts to 4.25 KiB of lookup tables in total. Given a cache-line size of 64 B, the lookup tables would fit into $4.25 \cdot 1024\,\text{B}/64\,\text{B} = 68$ cache-line-sized memory blocks. Most likely, the lookup tables will not be aligned to the boundaries of the memory blocks perfectly, because other, smaller variables are also stored in the memory. In this case, there are 69 memory blocks that contain elements of the lookup tables. It depends on the secret AES key and message, which of these 69 memory blocks are accessed in which order. That is, it depends on the secrets, which of the blocks are contained in the final cache state in which order. How much information about the secret is revealed by the order of blocks in the cache depends on the cache configuration.

In a set-associative cache, the cache set in which a memory block is stored is determined by the least-significant bits of the block's address in memory. That is, consecutive memory blocks will be stored in consecutive cache sets. This way, nearby memory blocks, which are likely to be used together, do not compete for the same space in the cache. In a 128 KiB 4-way set associative cache, there are $128 \cdot 1024\,\text{B}/64\,\text{B}/4 = 512$ cache sets. That is, each of the 69 memory blocks that belong to the lookup tables has its own cache set.

Since each lookup-table memory block is cached in its own cache set, an attacker under *acc* can learn only which memory blocks have been accessed and not in which order they have been accessed. From the 69 blocks of lookup-tables, he can thus learn at most 69 bit of information.

Comparison across Cache Sizes The leakage bounds for mbedTLS AES differ across the cache sizes, going up with decreasing cache size, peaking at 114.5 bit for cache size 4 KiB, and then going down again. This can, again, be explained by how much information about the order of memory accesses is revealed by the final cache state.

If each lookup-table block is cached in a separate cache set, only 69 bit of information can be learned. Once the cache gets smaller and multiple lookup-table blocks start sharing the same cache set, an attacker under *acc* can also observe the order between these memory blocks. From this, he obtains more information about the order of memory accesses, i.e., the bound increases.

Once the cache size falls below $69 \cdot 64\,\text{B}/1024 = 4.3125\,\text{KiB}$, however, not all 69 memory blocks holding the lookup tables fit into the cache at the same time. On a system with such a small

cache, if a run of mbedTLS AES accesses all 69 memory blocks, some of these blocks will be evicted during the run and not be present in the final cache state. The smaller the cache size, the more memory blocks will be evicted. That is, more information will be hidden from the final cache state and from the attacker under *acc*. The potential for leakage decreases.

In our leakage bounds, a peak can be observed at 4 KiB cache size, because this is the closest cache size to 4.3125 KiB that we considered.

Implications in Practice We have presented upper bounds on the leakage, i.e., security guarantees with respect to cache-side-channel attackers under *acc* for mbedTLS AES. Our leakage bounds cover 7 exemplary cache sizes. However, our interpretation of these leakage bounds gives rise to a more general insight. If the cache size is big enough to allow for an injective mapping from memory blocks to cache sets, the leakage bounds will be stable when the size increases further, because the granularity of information visible to an attacker under *acc* remains stable.

This insight is helpful in multiple ways when using the security guarantees to navigate the security-performance trade-off, e.g., when deploying or building on a table-based AES implementation. First, if the implementation shall be deployed across systems with different cache size, the number of cache sizes for which leakage bounds need to be computed can be reduced based on the size of the lookup tables. For mbedTLS, e.g., it would suffice to consider caches with 70 or fewer cache sets. Second, the impact of increasing cache sizes in future hardware generations can be taken into account efficiently when reasoning about the security-performance trade-off. Leakage bounds only need to be computed proactively for bigger cache sizes until the stabilization point (where the mapping from memory blocks to cache sets becomes injective) is reached.

3.6.2 Comparison across Attacker Models

A crucial factor in the computation of security guarantees is the attack surface that one assumes. In this section, we explore the effect of the cache-side-channel attack surface on security guarantees for mbedTLS in the form of leakage bounds.

Comparison across *acc* and *accd* Figure 3.3 visualizes the leakage bounds that we obtained with CacheAudit 0.2b for mbedTLS AES across multiple cache sizes and attacker models. Each curve corresponds to the leakage bounds under one attacker model, e.g., the curve labeled -▲- corresponds to the leakage bounds for attackers under *acc* that we discussed in the previous section. On the y-axis, the figure shows the leakage bounds in bit and on the x-axis, it shows the cache sizes in KiB. Each point of the curve corresponds to one leakage bound that we computed. We connected the individual points by dashed lines to improve readability.

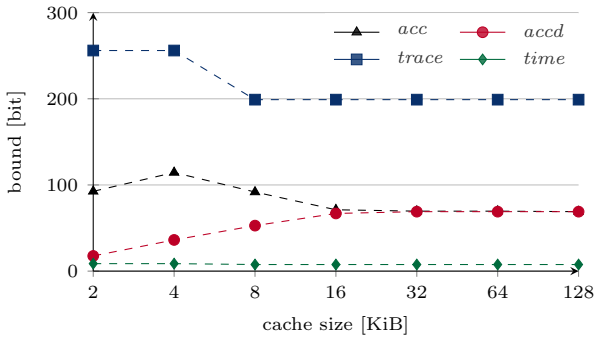

Figure 3.3: Leakage Bounds for mbedTLS AES Encryption (Raw Bounds see Appendix A)

The curve labeled -●- corresponds to the attacker model *accd*. Note that, the leakage bounds increase with increasing cache size until they reach 69 bit, where they stabilize. After stabilization, the bounds for *accd* are identical to the bounds for the attacker model *acc*.

Attackers under *accd* are less powerful than attackers under *acc*. They cannot observe the exact memory blocks in the final cache state, but only the amount of memory blocks cached in each cache set. For small caches, where many lookup-table blocks compete for the same cache set, the attacker can derive only little information from an observation that he makes. There are many combinations of AES key and message that are indistinguishable based on his observation. The larger the cache becomes, the more the lookup-table blocks are spread across the cache sets. The attacker can learn more and more information. The peak is reached once the mapping from memory blocks to cache sets is injective, because he can infer from the fill-degree of each cache set exactly which memory blocks have been accessed during the AES encryption. At this point, the *accd* attacker learns the same information as an attacker under the more powerful model *acc*.

Implications in Practice The stabilization of the leakage bounds for *accd*, together with our interpretation, indicates that security guarantees for *accd* will transfer to newer architectures with larger caches. Moreover, the leakage bounds for *acc* and *accd* are equal starting from a cache size where the mapping from memory blocks to cache sets is injective. That is, for systems with large caches, reducing the attack surface from *acc* to *accd* (e.g., by avoiding the use of shared libraries between victim and attacker) will not lead to an improvement of the security guarantees.

Comparison with *time* and *trace* The curve labeled -♦- corresponds to the attacker model *time*. The leakage bounds for *time* are consistently much lower than the leakage bounds for *acc*. The bounds lie between 8.7 bit (for 2 KiB cache) and 7.7 bit (for 128 KiB cache), i.e., between 3.4% and 3% of the secret bits in the message and key. For 128 KiB cache size, e.g., the bounds for *time* are about 61 bit lower than the bounds for *acc* and *accd* and, hence, much better guarantees.

The fourth curve in Figure 3.3 is labeled -■- and belongs to the attacker model *trace*. Throughout the cache sizes, the leakage bounds with respect to *trace* are very high. They guarantee almost no security at all against this attacker model. For a cache of 128 KiB, up to 199 bit about the secret key and message might potentially be leaked. For small caches of 4 KiB, even the entire key and message might potentially be leaked.

Implications in Practice While reducing the attack surface from *acc* to *accd* does not lead to better security guarantees for sufficiently big cache sizes, there are two factors that have a significant impact on the security guarantees. If the attack surface includes the attacker model *trace*, it seems very worthwhile to reduce the surface, e.g., by ensuring that the attacker has no physical access to the system and no software-based access to performance counters or power-consumption measurements. If trace-based attacks are ruled out, reducing the attack surface further to *time* would be very worthwhile from the perspective of security guarantees. If attacks under *acc* and *accd* can be ruled out (e.g., avoiding the presence of attacker-controlled spy processes by installing protection against Trojans or by moving out of a public cloud system), the security guarantees improve significantly.

Overall, the insights from our study of the cache-side-channel leakage of mbedTLS AES already support (1) the transfer of the corresponding security guarantees across hardware generations and (2) the prioritization of efforts to limit the cache-side-channel attack surface on mbedTLS AES. In the following section, we expand our study to other AES implementations to investigate the influence of implementation details on cache-side-channel security guarantees.

3.7 Comparison across Implementations

Recall from Section 2.3.1 that existing lookup-table-based AES implementations differ in the way how the last transformation round is implemented. The AES implementation from Nettle that we consider computes the last round on the fly, like the mbedTLS implementation of AES that we considered in the last section. That is, like mbedTLS AES, Nettle AES uses 4.25 KiB of lookup tables. The OpenSSL implementation of AES that we consider reuses the lookup tables from the first nine AES rounds in the last round and masks out the effect of the MixColumns step. Thus, OpenSSL AES uses only 4 KiB of lookup tables. The LibTomCrypt implementation of AES that we consider uses a dedicated set of lookup tables with precomputed results for the last round of AES. That is, LibTomCrypt AES uses 8 KiB of lookup tables in total.

In this section we explore how these details of implementing AES based on lookup tables influence the leakage bounds on cache side channels.

3.7.1 Influence of Implementations for 128 KiB Caches

Based on our insights from the analysis of mbedTLS AES, we can predict the leakage bounds with respect to the attacker models *acc* and *accd* and a 128 KiB cache. More concretely, we expect that the leakage bounds across the AES implementations will correspond to the number of memory blocks needed to store the lookup tables.

To store the 8 KiB of LibTomCrypt-AES lookup tables, $8 \cdot 1024\,\mathrm{B}/64\,\mathrm{B} + 1 = 129$ memory blocks are needed if the tables are not aligned to the beginning of a fresh memory block. Thus, we expect a leakage bound of 129 bit. For Nettle AES the same number of memory blocks as for mbedTLS (69 blocks) is needed, because the implementations use the same amount of lookup tables. Hence, we expect a leakage bound of 69 bit for Nettle AES. For OpenSSL AES, only $4 \cdot 1024\,\mathrm{B}/64\,\mathrm{B} + 1 = 65$ memory blocks are needed, so that we expect the bound to be 65 bit.

To check our hypotheses, we compute the actual leakage bounds for the three additional AES implementations with CacheAudit 0.2b. The resulting bounds are shown in Table 3.4

	LibTomCrypt AES	mbedTLS AES	Nettle AES	OpenSSL AES
accd	129 bit	69 bit	69 bit	64 bit
acc	129 bit	69 bit	69 bit	64 bit

Table 3.4: *acc* and *accd* Leakage Bounds for 128 KiB Cache Size

For LibTomCrypt and Nettle AES, the bounds are exactly equal to the bounds we expected. For OpenSSL AES, the leakage bound is 1 bit lower than expected, which is likely due to a lucky alignment of the lookup tables to memory blocks.

Our results suggest that using fewer lookup tables leads to better security guarantees with respect to attackers under the models *acc* and *accd*. The best among the options that we considered seems to be OpenSSL, which uses only 4 KiB of lookup tables.

Osvik, Shamir, and Tromer [102] have made a similar observation already for the main rounds of AES. They discuss the use of, e.g., one 1 KiB lookup table or one 2 KiB lookup table for the main rounds as a countermeasure against access-based attacks, stating that smaller lookup tables reduce the leakage of one program run. Our leakage bounds match their observation. We obtain better security guarantees if the overall size of lookup tables used is smaller.

Moreover, the lookup table sizes in the implementations that we analyzed differ only for the last AES round. That is, already the choice of tables for the last round has a significant impact on the resulting security guarantees. Reusing the lookup tables from the main AES rounds in the last round (as done in OpenSSL AES) leads to the best guarantees in our case study. This reuse incurs a performance overhead compared to the use of additional tables, because the table

lookups need to be postprocessed in order to undo the MixColumns transformation. However, this overhead only affects one out of the ten (for 128 bit keys) AES rounds. This combination of the significant improvement of the security guarantees and the limited performance overhead makes the reuse of tables an attractive option with respect to the security-performance trade-off.

With respect to the attacker models *time* and *trace*, we expect the leakage bounds to be higher for LibTomCrypt and OpenSSL AES and lower for mbedTLS and Nettle AES. The reason for our hypothesis is the size of the lookup-table entries used in the last AES round across the implementations. While LibTomCrypt and OpenSSL AES both use T-Tables with 32 bit entries, mbedTLS and Nettle AES use the S-Box with 8 bit entries. As described by Page [104], cache hits and misses reveal less information about the index of a table access if more table entries are located in the same memory block. This was confirmed by Tiri, Aciiçmez, Neve, and Andersen [124] in a practical evaluation of time-based attacks across two variants of OpenSSL AES.

Table 3.5 shows the leakage bounds computed using CacheAudit 0.2b for the attacker models *time* and *trace* for a cache size of 128 KiB across the AES implementations. Surprisingly, our expectation of lower leakage bounds for mbedTLS and Nettle AES and higher leakage bounds for LibTomCrypt and OpenSSL AES is not confirmed. The leakage bounds for *time* are identical across all implementations and the leakage bounds for *trace* are almost identical across the implementations. They are even slightly lower for LibTomCrypt and OpenSSL AES.

	LibTomCrypt AES	mbedTLS AES	Nettle AES	OpenSSL AES
time	7.7 bit	7.7 bit	7.7 bit	7.7 bit
trace	198 bit	199 bit	199 bit	196 bit

Table 3.5: *time* and *trace* Leakage Bounds for 128 KiB Cache Size

The similarity in the leakage bounds across the implementations is probably due to an imprecision in the computation of the bounds. The leakage bounds are upper bounds but not necessarily exact. The actual leakage might be lower than the bounds that we obtained.

Even given the potential imprecision, the leakage bounds for the attacker model *time* are still very low and, hence, very useful. In particular, they are between 87% and 94% lower than the *acc*/*accd* leakage bounds across the AES implementations. That is, reducing the attack surface of an AES implementation to attackers under *time* will greatly improve the security guarantees. For the reduced attack surface of attackers under *time*, we obtain high security guarantees (guarantees for low leakage) consistently across all four AES implementations.

3.7.2 Influence of Implementations across Cache Sizes

Figure 3.4a shows the *time* leakage bounds with respect to different cache sizes across the four AES implementations. Each curve corresponds to one AES implementation. The x-axis corresponds to the different cache sizes and the y-axis corresponds to the leakage bounds. We observe that the leakage bounds are very similar across the four implementations. Moreover, the leakage bounds for each implementation stabilize starting from a certain cache size. For LibTomCrypt AES, the *time* bounds stabilize starting from 16 KiB cache size. For all other implementations, the bounds stabilize starting from 8 KiB cache size.

Figure 3.4b shows the *trace* leakage bounds across the four AES implementations. To improve readability, we only show the range from 190 bit to 260 bit leakage on the y-axis. While the *trace* bounds are significantly higher than the *time* bounds from the previous figure, the development across the cache sizes is very similar between *trace* and *time*. The *trace* bounds stabilize at 16 KiB cache size for LibTomCrypt AES and at 8 KiB cache size for the other implementations.

For both, *time* and *trace*, the stabilization of leakage bounds occurs as soon as all lookup tables fit into the cache. We explain this based on the *trace* attacker model. For the *time* attacker

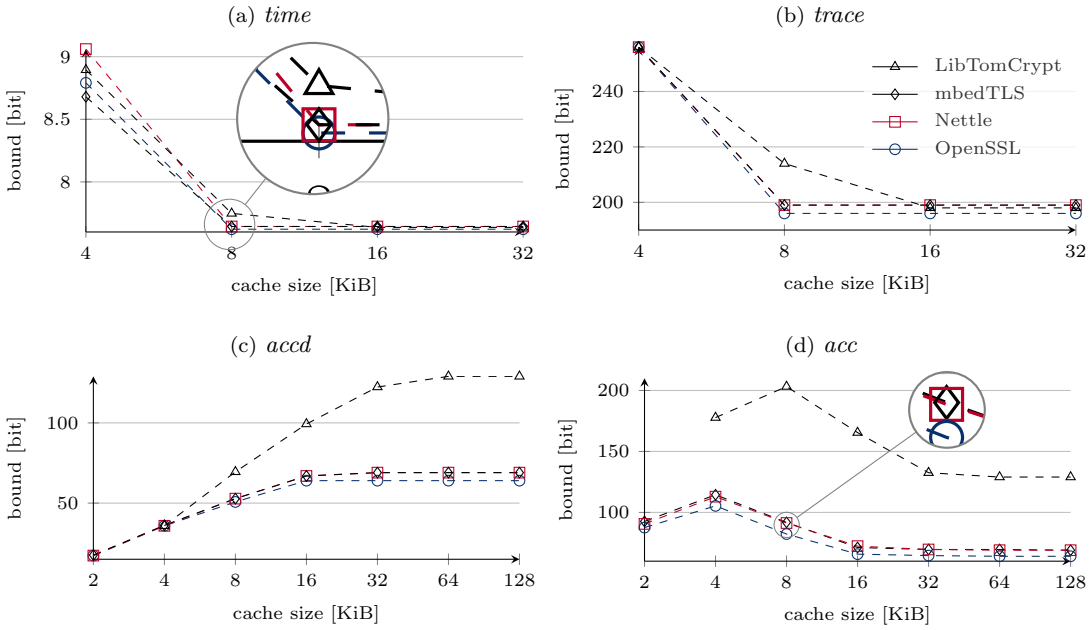

Figure 3.4: Leakage Bounds across Attacker Models (Raw Bounds see Appendix A)

model, the leakage bounds are drastically lower (only about 5% as high as the *trace* leakage bounds), but the effect of the cache size is the same as for *trace*.

The *trace* leakage bounds stabilize to roughly 200 bit for all implementations. The AES implementations perform about 200 accesses to the lookup tables during one encryption. When all lookup tables fit into the cache, roughly each lookup-table access corresponds to one bit of potential leakage (leaking whether the accessed table entry is already cached or not). If the lookup tables do not fit into the cache, the leakage might be higher because accesses to other variables might leak additional information. If an attacker observes a cache miss for an access to such a variable, he might conclude that this variable was evicted from the cache because a lookup-table entry was accessed that maps to the same cache set. Thereby he might learn additional information about the secret-dependent indices at which the lookup tables are accessed.

Figures 3.4c and 3.4d show the development of *accd* and *acc* leakage bounds, respectively, across different cache sizes for the four AES implementations.[3]

Under both attacker models, *accd* and *acc*, the leakage bounds stabilize at some cache size not only for mbedTLS AES (as described in Section 3.6), but also for all other implementations. As it was already the case for mbedTLS AES, the stabilization point is the cache size at which an injective mapping from memory blocks to cache sets becomes possible. The smaller the lookup tables, the earlier the leakage bounds reach the stabilization point.

Across all four attacker models, our results show that using lookup tables with smaller overall size leads to stable leakage bounds and, hence, stable security guarantees for more cache sizes and, hence, more systems. That is, reusing the existing lookup tables in the last AES round not only improves the security guarantees against the access-based attacker models but also improves the stability of the leakage bounds across all attacker models.

[3]For LibTomCrypt AES on cache size 2 KiB, the analysis ran out of memory.

3.8 Comparison across Countermeasures

While we obtain comparatively low leakage bounds for the AES implementations if the attack surface is reduced to time-based attackers under the model *time*, the leakage bounds are still non-zero, i.e., the AES key would have to be replaced regularly. In this section, we study the effectiveness of implementation-level countermeasures against cache-side-channel leakage to find out whether these can reduce the leakage bounds to zero. We focus on the countermeasures preloading (in combination with cache locking) and bitslicing.

3.8.1 Effectiveness of the Preloading Countermeasure

The preloading countermeasure loads the AES lookup tables into the cache before starting the encryption. The goal of preloading is to ensure that the cache positions of all table entries and the durations of all table accesses are independent of secret information.

To achieve this goal, preloading needs to be applied in a careful way. The countermeasure might become ineffective if preloaded table entries are evicted from the cache by other processes [74, 75]. To avoid this, preloading can be combined with cache locking [91]. Cache locking locks the memory entries in place in the cache to ensure that they do not get evicted.

We study the effectiveness of preloading (under the assumption that preloaded entries do not get evicted by other processes) across our four target AES implementations. To this end, we manually add a loop that accesses each lookup-table entry in the beginning of each implementation and compute leakage bounds for each implementation with CacheAudit 0.2b.

Table 3.6 visualizes our analysis results. Each row corresponds to one AES implementation with preloading and each column corresponds to one cache size. A cell contains a checkmark if and only if we obtained a leakage bound of 0 bit for the corresponding combination of AES implementation and cache size. That is, the cells with checkmarks indicate the cases in which preloading effectively removed all potential for cache-side-channel leakage. For all combinations of AES implementation and cache size, we obtained 0 bit leakage bounds either for all attacker models or for none. That is, the table applies for all four attacker models.

Cache Size [KiB]	4	8	16	32	64	128
LibTomCrypt			✓	✓	✓	✓
mbedTLS		✓	✓	✓	✓	✓
Nettle		✓	✓	✓	✓	✓
OpenSSL		✓	✓	✓	✓	✓

Table 3.6: Preloading Effectiveness for *acc*/*accd*/*trace*/*time* (Raw Bounds see Appendix A)

The cache sizes for which preloading is effective differ across the AES implementations. While for the AES implementations from Nettle, OpenSSL, and mbedTLS preloading is effective starting from 8 KiB cache size, for LibTomCrypt AES, preloading is only effective starting from 16 KiB.

The results suggest that the effectiveness of preloading is related to the amount of lookup tables used in an AES implementation. Our interpretation is that for small caches, some lookup-table entries get evicted by the AES process itself, reintroducing some cache-side-channel leakage. For caches that are large enough to hold all lookup tables and additional variables of an AES implementation, preloading should be effective. This is the case for caches that can hold 8 KiB of lookup tables and some additional variables in the case of LibTomCrypt, which explains why we see the first zero-leakage bounds for 16 KiB cache size. For the other implementations, the lookup tables plus other variables are slightly larger than 4 KiB. Hence, we see the first zero-leakage bounds for cache size 8 KiB.

Overall, our results suggest that (1) preloading (with cache locking) is an effective countermeasure against attackers under *acc*, *accd*, *time*, and *trace* for sufficiently large caches and that (2) using fewer lookup tables can be an advantage because it decreases the minimum cache size required for the countermeasure to be effective. Note that, fewer lookup tables also have the advantage that the performance penalty introduced by preloading is lower.

3.8.2 Effectiveness of the Bitslicing Countermeasure

Bitslicing is an implementation technique that was developed for implementations of the Data Encryption Standard (DES) by Biham [20]. The technique avoids secret-dependent memory accesses by construction and can, hence, be used as a countermeasure against cache-side-channel leakage. To this end, all round transformations are computed on the fly using software emulations of basic logic gates [20]. This countermeasure requires a complete reimplementation of AES and cannot simply be added on top of existing AES implementations. Luckily, a bitsliced AES implementation is available in the cryptographic library NaCl [68, 18].

To evaluate the effectiveness of bitslicing against cache-side-channel leakage, we compute leakage bounds with CacheAudit 0.2b using the same setup as for the lookup-table-based implementations. The target binary is generated from a wrapper that calls the functions `crypto_stream_beforenm` (from the class `beforenm.c`) and `crypto_stream_xor_afternm` (from the class `xor_afternm.c`) from NaCl version 20110221 using the same compilation process as for the lookup-table-based AES implementations described above.

For the NaCl implementation of AES, we obtain the leakage bound 0 bit across all four attacker models (*acc*, *accd*, *trace*, *time*) and all six cache sizes (4, 8, 16, 32, 64, and 128 KiB). That is, the implementation is secure against cache-side-channel attacks in all 24 scenarios covered by our analysis setup. This confirms that the bitslicing countermeasure is effective even for small caches.

While the security guarantees for existing lookup-table-based AES implementations can be improved by reducing the attack surface and adding preloading, switching to bitsliced implementations would lead to the broadest improvement. For situations in which it is feasible to use a bitsliced implementation (e.g., where the NaCl library can be used or where enough time and expertise for creating a customized bitsliced implementation is available), it seems worthwhile to choose bitslicing over a lookup-table-based implementation.

3.9 Summary

In this chapter, we studied the cache-side-channel leakage across multiple AES implementations using upper leakage bounds. To this end, we developed the abstract domain $\overline{\mathcal{D}}_{32}$ and the abstract semantics $\texttt{upd}_{\overline{\mathcal{D}}_{32}}$ that capture the CPU flags SF and OF at a high level of precision. More concretely, they capture the relation between these flags and the possible states of the memory, registers, and cache. Based on $\overline{\mathcal{D}}_{32}$, we performed a successful comparative study across AES implementations. However, the domain is not limited to the analysis of AES implementations. It is reused, e.g., in our second program analysis that is described in Chapter 4.

With our program analysis and its implementation in the tool CacheAudit 0.2b, we successfully computed security guarantees in the form of quantitative cache-side-channel leakage bounds across the attacker models *acc*, *accd*, *trace*, and *time* and across cache sizes ranging from 4 KiB to 128 KiB for OpenSSL AES, mbedTLS AES, NaCl AES, Nettle AES, and LibTomCrypt AES.

Our results clarify that the implementation technique used to implement the last round of AES has a significant impact on the security guarantees for the entire AES implementation. The technique used in OpenSSL AES, i.e., the reuse of the lookup tables from the main AES rounds combined with masking out the effects of the MixColumns transformation, is the most beneficial in our evaluation. It leads to the lowest leakage bounds throughout. The technique used in LibTomCrypt AES, i.e., the use of a separate 4 KiB set of lookup tables, leads to the worst

security guarantees. The leakage bounds for LibTomCrypt AES with respect to access-based attackers are, e.g., roughly twice as high as the bounds for OpenSSL AES.

We also derived multiple other practical insights from our study. For access-based attackers under the models *acc* and *accd*, e.g., the leakage bounds are stable for cache sizes that are large enough to allow an injective mapping from lookup-table memory blocks to cache sets. That is, the security guarantees given by the leakage bounds transfer to larger caches (e.g., of future hardware generations). For the attacker model *time*, the leakage bounds that we obtained were consistently low across all AES implementations. That is, reducing the attack surface of a system running AES to attackers who can only measure the execution time is a worthwhile system-level countermeasure. On the implementation level, we analyzed the effectiveness of the countermeasures preloading (with cache locking) and bitslicing. For preloading, which can be added on top of existing lookup-table-based implementations, we obtained zero-leakage bounds for all cache sizes that are big enough to hold all lookup tables. For bitslicing, which requires a re-implementation of AES, we even obtained zero-leakage bounds across all cache sizes. Thus, both can be suitable countermeasures, depending on the situation in which they are deployed.

Chapter 4

Cache-Side-Channel Quantification for the ring-TESLA Implementation

4.1 Introduction

The lattice-based signature scheme ring-TESLA was designed to produce digital signatures that are secure even with respect to post-quantum attackers (i.e., attackers with quantum computers). The focus during the development of ring-TESLA has been on the post-quantum security of the scheme. Cache side channels were not considered initially. However, cache-side-channel attacks pose a serious threat to implementations of lattice-based cryptography. For instance, an implementation of the lattice-based signature scheme BLISS [46] was attacked using a cache-side-channel attack that extracted the entire secret key in less than two minutes [54].

In this chapter, we quantify the cache-side-channel leakage of the ring-TESLA implementation by program analysis. The ring-TESLA implementation is significantly more complex than the AES implementations that we analyzed in Chapter 3. The computations are more involved and the parameters are much bigger. For instance, the secret key can be up to 49 152 bit long.

The program analysis in this chapter is based on an abstract semantics that is sufficiently powerful to analyze a slightly simplified version of the ring-TESLA implementation (with a well-defined correspondence to the original implementation). In particular, the abstract semantics captures multiple instructions that process longer operands (e.g., multiplication with two destination registers and conversion from double-word to quad-word operands).

We analyzed the ring-TESLA implementation with our program analysis to obtain upper bounds on its cache-side-channel leakage. In collaboration with the cryptographers behind ring-TESLA, we detected multiple vulnerabilities in the implementation. We developed mitigations and assessed their effectiveness together with the cryptographers. In this chapter, we focus on the assessment of the vulnerabilities and the mitigations from the cache-side-channel perspective. The assessment from the cryptographic perspective is described in detail by Bindel in [21].

With our mitigations, the cryptographers were able to avoid cache-side-channel leakage in an updated version of the ring-TESLA implementation and the reference implementation of the successor scheme qTESLA [22]. The scheme qTESLA, with its hardened reference implementation, advanced to Round 2 of the Post-Quantum Cryptography Standardization of the U.S. NIST [4].

In Section 4.2, we describe our target implementation and its correspondence to the original ring-TESLA implementation. In Sections 4.3 and 4.4, we describe our extended abstract semantics and its implementation in the analysis tool. In Section 4.5, we describe the setup of our analysis.

In Sections 4.6 and 4.7, we describe the detected vulnerabilities and their mitigations from the perspective of cache-side-channel security.

4.2 Target ring-TESLA Implementation

Our target implementation is based on the function `crypto_sign` from the file `sign.c` of the ring-TESLA integer implementation described in Section 2.3.2. Our target implementation uses custom variants of the functions `sample_y` and `generate_c` and a simplification with respect to the number of tries performed to generate a valid signature. We discuss the simplifications in detail below. The goal behind the simplifications is to eliminate potentially infinite loops. Our target implementation does not include the generation of random numbers. We assume that the ring-TESLA implementation is used with a secure implementation of random-number generation.

The custom implementation of the function `sample_y` is shown at the top of Figure 4.1. It computes an array `mat_y`, where each entry corresponds to the result of subtracting the value `PARAM_B` from the value that is stored at address `0x4`. The original implementation of `sample_y` (shown at the bottom of Figure 4.1) computes an array `mat_y`, where each entry corresponds to the result of subtracting the value `PARAM_B` from a random value. The original implementation uses rejection sampling to restrict the possible values in `mat_y`.

Our program analysis overapproximates the value stored at address `0x4` by the set of all possible 32 bit values. Therefore, the analysis of the custom variant of `sample_y` overapproximates all possible results of the original `sample_y`. The cache-side-channel leakage of the original variant is not overapproximated, because the custom variant does not account for additional memory accesses performed during the rejection sampling. It was manually verified that the original variant of `sample_y` does not leak secret information through cache side channels [23, 21].

Figure 4.2 shows our custom implementation of `generate_c` on the left-hand side. It computes a representation of the hashed vector `c` in the form of an array called `pos_list`. It allows for duplicates in the resulting array. The original implementation of `generate_c`, shown on the right-hand side of Figure 4.2 avoids such duplicates by rejection sampling (highlighted in gray).

Since our custom implementation of `generate_c` allows more values of `pos_list` than the original implementation, applying our program analysis to the custom variant will overapproximate the possible results of the original variant. The cache usage of the original variant is not overapproximated, because dropping the rejection sampling eliminates memory accesses that depend on the sampled random values. But the modification is necessary to obtain an analyzable target implementation. In theory, the number of loop iterations in the original implementation

```
1  void sample_y(poly mat_y){ int i;
2   for (i=0; i < PARAM_N; ++i){
3    mat_y[i] = *(int *)(0x4) - PARAM_B; }}
```

```
1  void sample_y(poly mat_y){ int32_t val;
2   unsigned char buf[3*PARAM_N+68]; int pos=0, i=0;
3   fastrandombytes(buf,3*PARAM_N+68);              // random-number generation
4   do{
5    if(pos == 3*PARAM_N+66){
6     fastrandombytes(buf,3*PARAM_N+68); pos = 0;} // random-number generation
7    val  = (*(int32_t *)(buf+pos)) & 0x7fffff;
8    if(val < 0x7fffff)
9     mat_y[i++] = val-PARAM_B;  // overapproximated by custom implementation
10   pos+=3;}
11  while(i< PARAM_N);}
```

Figure 4.1: Implementation of `sample_y` - Custom (top) and Original (bottom)

```
1  void generate_c(uint32_t *pos_list,
       unsigned char *c_bin){
2   int32_t c[PARAM_N];
3   int cnt = 0; int pos; [...]
4   crypto_stream(r, R_LENGTH, nonce,
        c_bin);
5   for(i=0; i<PARAM_N; i++){
6     c[i] = 0;}
7   i=0;
8   while(i<PARAM_W){
9    pos = 0;
10   pos = (r[cnt]<<8) | (r[cnt+1]);
11   pos &= PARAM_N-1; cnt += 2;
12
13   pos_list[i] = pos;
14
15   i++; cnt++; }}
```

```
1  void generate_c(uint32_t *pos_list,
       unsigned char *c_bin){
2   int32_t c[PARAM_N];
3   int cnt = 0; int pos; [...]
4   crypto_stream(r, R_LENGTH, nonce,
        c_bin);
5   for(i=0; i<PARAM_N; i++){
6     c[i] = 0;}
7   i=0;
8   while(i<PARAM_W){
9    pos = 0;
10   pos = (r[cnt]<<8) | (r[cnt+1]);
11   pos &= PARAM_N-1; cnt += 2;
12   if (c[pos] == 0) {
13     pos_list[i] = pos;
14     c[pos]=1;
15     i++; cnt++; }}}
```

Figure 4.2: Implementation of `generate_c` - Custom (left) and Original (right)

might be infinite (with very low probability). Our variant always performs a finite number of iterations and can therefore be analyzed statically. We take this deviation from the original implementation into account in the interpretation of our analysis results.

Finally, the signature generation as a whole creates signatures in a loop until one signature is valid. To eliminate the potentially infinite loop in the function `crypto_sign`, we fix the number of iterations in our target implementation. More concretely, we fix the number to two iterations so that the effects that occur when a signature is rejected are taken into account.

Overall, the target for our analysis is an x86 binary that executes the function `crypto_sign` from our custom ring-TESLA implementation with parameters selected by the ring-TESLA authors Bindel, Buchmann, and Krämer [23]. We compile with `gcc 4.8.4` and the flags `-static -m32 -fno-stack-protector`.

4.3 Program Analysis for ring-TESLA

Like the program analysis in Chapter 3, the program analysis that we use to quantify the leakage of the ring-TESLA implementation is based on the abstract domain

$$\begin{aligned}\overline{\mathcal{D}}_{32} =& (\{CF, ZF, SF, OF\} \to \mathbb{B}) \rightharpoonup \\ & ((\mathcal{R}_{32} \cup \mathcal{M}_{32} \to \mathcal{P}(\mathcal{V}_{32})) \times (\mathcal{M}_{32} \to \mathcal{P}(\mathcal{C}_{\text{pos}})) \times \mathcal{P}(\{h, m, n\})^\star).\end{aligned}$$

That is, the domain preserves the relation between the CPU flags CF, ZF, SF, and OF and the abstract states of registers, memory, and cache. The possible values of registers from the set $\mathcal{R}_{32}$ and memory entries from the set $\mathcal{M}_{32}$ are represented using a set abstraction of the concrete 32 bit values from the set $\mathcal{V}_{32}$. The possible positions of each memory entry in the cache are represented using a set abstraction of the concrete positions from the set $\mathcal{C}_{\text{pos}}$. Finally, the possible cache traces are represented by sets of possible traces that contain cache hits (symbol h), cache misses (symbol m), and non-memory-access steps (symbol n).

The novelty of our program analysis in this chapter lies in the abstract semantics. The abstract semantics $\mathtt{upd}_{\overline{\mathcal{D}}_{32}}$ from Chapter 3 cannot be applied to our target ring-TESLA implementation, because it does not cover all required x86 instructions.

The new abstract semantics is a function

$$\mathtt{upd}'_{\overline{\mathcal{D}}_{32}} : \overline{\mathcal{D}}_{32} \times \mathcal{I}'_{32} \to \overline{\mathcal{D}}_{32} \qquad \text{where} \qquad \mathcal{I}'_{32} = \mathcal{I}_{32} \cup \mathcal{I}_{\text{TESLA}}.$$

That is, our abstract semantics captures not only the x86 instructions from the set $\mathcal{I}_{32}$, but also the additional instructions from the set $\mathcal{I}_{\text{TESLA}}$. The newly supported set $\mathcal{I}_{\text{TESLA}}$ contains, for instance, multiple instructions related to the handling of larger operand values (e.g., `Cdq` and variants of `Imul`), which are required due to the large parameters of ring-TESLA. The set $\mathcal{I}_{\text{TESLA}}$ also contains instructions that copy the values of flags without branching on them (`Setl` and `Setg`), which are used in our cache-side-channel countermeasures for ring-TESLA to accumulate the results of multiple rejection tests.

All instructions that are contained in $\mathcal{I}_{\text{TESLA}}$ are listed in Table 4.1. We describe the abstract semantics $\texttt{upd}'_{\overline{\mathcal{D}}_{32}}$ at the example of the instruction `Bsr` in the following.

Example 4.3.1. The instruction `Bsr` (Bit scan reverse) operates on a destination register `dst` and a source register or memory location `src`. It stores the index of the most significant set bit of the source operand from `src` in the destination register `dst` [63]. If `src` contains at least one set bit, the zero flag is cleared. If `src` contains no set bit, `dst` is undefined and the zero flag is set. All other flags are undefined in both cases.

Our abstract transfer function for `Bsr` is based on the principle of the best abstract transformer [35]. That is, we concretize the abstract state, compute the effect of `Bsr` for each possible combination of operands, and then abstract from the possible results again using a set abstraction.

We compute the effect of `Bsr` using a divide-and-conquer approach based on the formalization of the concrete x86 semantics by Degenbaev [38]. This approach works as follows. In case `src` $= 0$, the zero flag is set to 0 and all other parts of the state remain unchanged. In case `src` $\neq 0$, `src` is zero-extended to 64 bit and the number of leading zeros is computed by checking recursively whether the first half of the remaining bits in `src` contains a 1. Once the number of leading zeroes is known, the index of the most significant set bit is computed as 64 minus the number of leading zeros. The resulting value of `dst` is the index of the most significant set bit. The zero flag is set to 0. The values of `src` and the values of all flags except the zero flag remain unchanged with respect to their initial values. ◊

In addition to `Bsr`, our abstract semantics covers nine other x86 instructions (see the full list of added instructions in Table 4.1). In defining the abstract transfer functions for these instructions, we followed the same approach as for `Bsr`, using the x86 formalization by Degenbaev [38] as the reference point for the concrete semantics of the x86 instructions.

4.4 Tool Support for the Analysis

We automate the program analysis from Section 4.3 by implementing it in the CacheAudit framework and call the resulting analysis tool CacheAudit 0.2c. The sequence diagram in Figure 4.3 visualizes the implementation of the abstract semantics in CacheAudit 0.2c at the example of the instruction `Bsr` (the novel part of the implementation is highlighted in gray).

The overall structure of the implementation is the same as in Chapter 3: The iterator iterates over the instructions from the target binary. For each instruction, it calls the function

Type	Opcodes (and mnemonics) of additional instructions
Arithmetic	13 (`Adc`), 1B (`Sbb`), 6B (`Imul`), F7/3 (`Neg`), F7/4 (`Mul`), F7/5 (`Imul`)
Bitstring	0FBD (`Bsr`), 99 (`Cdq`)
Move	0F9C (`Setl`), 0F9F (`Setg`)

Table 4.1: x86 Instructions Contained in the Set $\mathcal{I}_{\text{TESLA}}$.

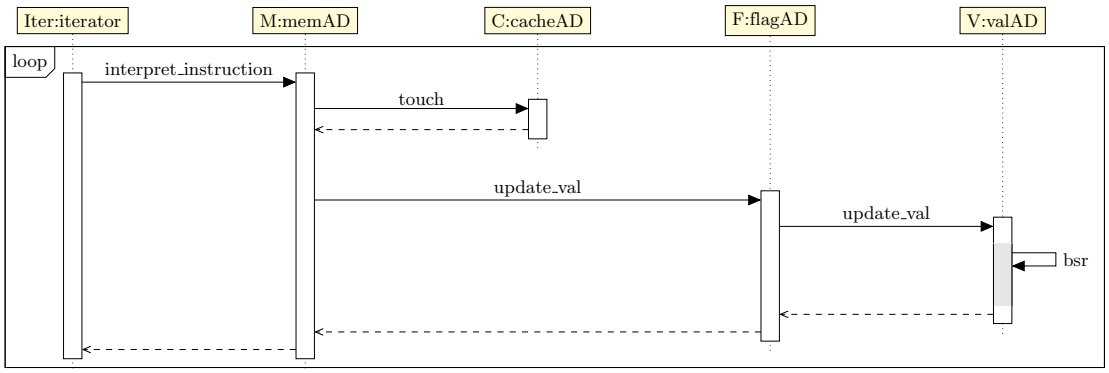

Figure 4.3: Sequence Diagram of the Abstract Semantics for `Bsr`

`interpret_instruction`, which in turn calls the functions `touch`, which updates the abstract cache state, and `update_val`, which updates the abstract register and memory state.

Unlike CacheAudit 0.2b from Chapter 3, CacheAudit 0.2c features a more powerful variant of `update_val`, which supports all instructions from the set $\mathcal{I}'_{32} = \mathcal{I}_{32} \cup \mathcal{I}_{\text{TESLA}}$. To this end, CacheAudit 0.2c features multiple functions that capture the semantics of the new instructions from $\mathcal{I}_{\text{TESLA}}$. For instance, our abstract semantics for `Bsr` is implemented by the function `bsr` in the module `valAD`, as highlighted in Figure 4.3.

The function `bsr` operates on a partial abstract state, namely on the concrete values of the CPU flags and the abstract states of the operands `dst` and `scr`. That is, as inputs it receives Boolean values for each flag, one set of potential values for `src`, and one set of potential values for `dst`. It returns a partial abstract state consisting of a map from potential status flag combinations to the corresponding sets of possible values for `dst` and `src` that can occur in combination with the respective flag combinations. The implementation abstracts from the possible concrete results by composing a result map. This map maps the flag combination in which the zero flag is set to 1 to sets of all possible resulting values of `dst` and `src` for which this flag combination can occur (i.e., the abstract value of `src` is the singleton set $\{0\}$). Analogously, the result map maps the flag combination in which the zero flag is set to 0 to the possible resulting values of `dst` and `src` in all other cases (i.e., the abstract value of `dst` is the set containing the indices of the most significant set bits in all possible values of `src`).

The abstract semantics for the other instructions from the set $\mathcal{I}_{\text{TESLA}}$ is also implemented in the module `valAD`. The parsing of the new instructions is implemented in the module `x86Parse` of CacheAudit 0.2c. CacheAudit 0.2c is available open source under `http://www.mais.informatik.tu-darmstadt.de/assets/tools/CacheAudit-FPS2017.zip`.

4.5 Analysis Setup

For our analysis, we fix the following cache configuration: We consider a 32 KiB 8-way set-associative data cache with 64 B line size. This configuration is used, e.g., in the Level 1 cache of the Skylake architecture [62, Table 2-4]. As the replacement strategy we fix LRU. We apply CacheAudit 0.2c to our target binary with respect to this cache configuration.

4.6 Detected Vulnerabilities

The leakage bounds that we obtain with CacheAudit 0.2c for our target ring-TESLA implementation across the four attacker models *acc*, *accd*, *trace*, and *time* are listed in Table 4.2.

Attacker model	*acc*	*accd*	*trace*	*time*
Leakage bound	12.9 bit	2.6 bit	51.6 bit	9.5 bit

Table 4.2: Upper bounds on the Leakage of the Signature Generation

The leakage bounds are non-zero with respect to all four attacker models. That is, there is potential cache-side-channel leakage in the target implementation.

Indeed, leakage was localized in four places through manual analysis of the ring-TESLA integer implementation [23, 21]. More concretely, actual leakage was localized in the functions `test_w`, `generate_c`, `computeEc`, and `test_rejection`. We describe the leakage in these functions from the perspective of cache-side-channel security in the following subsections and briefly summarize the impact from the cryptographic perspective that was determined by Bindel [23, 21].

4.6.1 Leakage in the Function `generate_c`

The function `generate_c` is called in Line 6 of the function `crypto_sign`, shown in Figure 2.4. Its implementation is shown in Figure 4.2. We consider the original implementation (right-hand side) here, because our customization eliminates possible cache usages as described in Section 4.2.

The function `generate_c` samples polynomial coefficients, which are stored in the variable `pos` one after the other and collected in the array `pos_list`. From a cryptographic perspective, the coefficients stored in `pos` have to be kept secret until it is clear that the signature computed in the current attempt will be valid. If the signature has to be discarded because a coefficient of the signature polynomials is too large, an attacker who knows the index of the violating coefficient and the value of `pos_list` for multiple such violations might break the signature scheme [23].

In Line 12 of Figure 4.2, the function accesses the array `c` at positions depending on the secret coefficients stored in the variable `pos`. If a cache with no-write-allocate policy is used, the array `c` will not be loaded into the cache upon initialization. The elements of `c` that correspond to secret coefficients will be loaded into the cache when Line 12 is executed. As the contents of the cache thus depends on the secret coefficients, an access-based attacker under model *acc* might learn the list `pos_list` of secret coefficients from his observation of the final cache state.

From a cryptographic perspective [23], this leakage alone is not a concern. However, the leakage becomes a serious concern in combination with the leakage in the functions `test_rejection` and `test_w` that we describe below.

4.6.2 Leakage in the Function `computeEc`

The implementation of the function `computeEc` is shown in Figure 4.4. As in the case of `generate_c`, the coefficients stored in the variable `pos` have to be kept secret in case the signature computed in the current attempt is invalid.

```
1  static void computeEc(poly Ec, const unsigned char *sk, const
2  uint32_t pos_list[PARAM_W]) {
3   int i,j, pos, * e; e = (int*)sk;
4   for(i=0;i<PARAM_N;i++){ Ec[i] = 0;}
5   for(i=0;i<PARAM_W;i++){
6    pos = pos_list[i];
7    for(j=0;j<pos;j++){ Ec[j] += e[j+PARAM_N - pos];}
8    for(j=pos;j<PARAM_N;j++){ Ec[j] -= e[j-pos];} } }
```

Figure 4.4: Implementation of `computeEc`

Like `generate_c`, the function `computeEc` contains memory accesses at positions that depend on `pos`. This time, the array `e` is accessed at positions that depend on `pos` in the two loops in Lines 7 and 8. All entries of `e` are eventually accessed for each value of `pos` because the loop in Line 7 covers the second part of `e` from index `pos` to index `PARAM_N` and the loop in Line 8 covers the first part of `e` from index `0` up to `PARAM_N - pos`. However, the split of the array `e` across the loops and, hence, the order in which the entries of `e` are accessed depends on `pos`.

Consider a cache-side-channel attacker under the model *trace*, i.e., an attacker who can observe the cache trace during an execution of `computeEc`. We show that there are cases in which secret information is leaked to such an attacker by the memory accesses to entries of the array `e`.

Each of the entries of `e` is of type integer, i.e., 32 bit long. Hence, 16 entries can be stored in one 64 B memory block and cached together in one 64 B cache line. Assume that the alignment of `e` is such that one memory block starts with `e[PARAM_N-16]` and ends with `e[PARAM_N-1]`.

Let the secret value of `pos`, e.g., be 14. Then the first loop in Line 7 accesses the memory entries `e[PARAM_N-14]`, ..., `e[PARAM_N-1]`. If a cache with no-write-allocate policy is used, the memory block containing the entries `e[PARAM_N-14]`, ..., `e[PARAM_N-1]` accessed by this first loop is not cached before the loop is reached. When the loop is executed, the attacker will observe a cache miss on the access to `e[PARAM_N-14]` and cache hits on the remaining accesses during the first loop because the memory block will be loaded into the cache upon the first miss.

When the second loop in Line 8 is reached, it will access the memory entries `e[0]`, ..., `e[PARAM_N-15]`. For the memory accesses up to `e[PARAM_N-17]`, the attacker will observe one cache miss followed by 15 cache hits for each new memory block that is accessed. However, he will observe two more cache hits for the final accesses to `e[PARAM_N-16]` and `e[PARAM_N-15]` because their memory block has already been cached by the first loop. From the distribution of the cache hits corresponding to this memory block, the attacker might learn the secret value of `pos`. By combining the values of `pos` from each iteration of the second loop in `computeEc`, the attacker might obtain the entire `pos_list`.

Like for `generate_c`, the leakage in `computeEc` is only a concern from a cryptographic point of view in combination with the leakage in the functions `test_rejection` and `test_w` [23].

4.6.3 Leakage in the Function `test_rejection`

The function `test_rejection` is shown in Figure 4.5. It is called in Line 15 of the signature-generation function from Figure 2.4. The function `test_rejection` operates on the secret array `poly_z`. For each coefficient in `poly_z`, the function checks whether it is within an allowed range (Line 3 of Figure 4.5). Once the first coefficient that is out of range is encountered, the check is aborted. That is, the amount of memory accesses during the function and, hence, the length of the cache trace depends on the index of the first coefficient that violates the allowed range.

That is, a cache-side-channel attacker under model *trace* might learn the index of the first invalid coefficient in the signature polynomial `poly_z`. Recall from Sections 4.6.1 and 4.6.2 that a cache-side-channel attacker might also learn the value of `pos_list`.

From the cryptographic perspective [23], the combination of these leaks might enable an attacker to break the scheme using a learning-the-parallelepiped attack [47, 98].

4.6.4 Leakage in the Function `test_w`

The function `test_w` is called in Lines 9 and 12 of the signature generation in Figure 2.4. The implementation of `test_w` is shown in Figure 4.6. Like `test_rejection`, the function `test_w` iterates through a polynomial (in this case the signature polynomial `poly_w`) to check the coefficients. The check is aborted upon the first coefficient that violates the conditions for a valid signature (see Line 11 and note that `left` is computed from the current coefficient).

Again, a cache-side-channel attacker under *trace* might learn the index of the invalid coefficient and combine it with information learned from `generate_c` or `computeEc` to break the scheme [23].

```
1  static int test_rejection(poly poly_z){ int i;
2   for(i=0; i<PARAM_N; i++){
3    if(poly_z[i]<-(PARAM_B-PARAM_U)||poly_z[i]>(PARAM_B-PARAM_U)){
4     return 1;}}
5   return 0; }
```

Figure 4.5: Implementation of `test_rejection`

```
1  static int test_w(poly poly_w){ int i; int64_t left, right, val;
2   for(i=0; i<PARAM_N; i++){
3    val = (int64_t) poly_w[i];
4    val = val % PARAM_Q;
5    if (val < 0){ val = val + PARAM_Q;}
6    left = val;
7    left = left % (1<<(PARAM_D));
8    left -= (1<<PARAM_D)/2;
9    left++;
10   right = (1<<(PARAM_D-1))-PARAM_REJECTION;
11   if (abs(left) > right){ return -1; } }
12  return 0; }
```

Figure 4.6: Implementation of `test_w`

4.7 Mitigation of the Vulnerabilities

In this section, we verify a successful hardening of the ring-TESLA implementation against the vulnerabilities described in Section 4.6. Since the leakage in the functions `generate_c` and `computeEc` is only a concern if there is complementary leakage in the functions `test_rejection` and `test_w`, it suffices to address the leakage in the functions `test_rejection` and `test_w`.

As a reference point for evaluating the effectiveness of the hardening, we apply CacheAudit 0.2c to compute leakage bounds for the functions `test_rejection` and `test_w` without cache-side-channel mitigations. The bounds are listed in Table 4.3 (truncated to the maximum size 49 152 bit of the secret key). The leakage bounds are non-zero with respect to all four attacker models.

	acc	*accd*	*trace*	*time*
`test_w`	31 bit	31 bit	49 152 bit	19.3 bit
`test_rejection`	31 bit	31 bit	10.1 bit	10.1 bit

Table 4.3: Leakage Bounds for Functions without Mitigation

4.7.1 Hardening of `test_rejection` and `test_w`

Figure 4.7 shows a modified implementation of the function `test_rejection`. Instead of deciding separately for each coefficient whether to abort the signature generation, the modified implementation iterates through all coefficients before deciding whether to abort. The information whether at least one coefficient has violated the allowed range is collected in the auxiliary variable `res` in Lines 4 and 5 (highlighted in gray). Finally, the overall result is returned.

This mitigation ensures that the function always accesses all entries of `poly_z` so that the interaction with the cache does not depend on secret information. The functionality of the original implementation is preserved. The original implementation returns 1 if at least one coefficient of `poly_z` violated the constraints on the allowed range and 0 otherwise. The modified implementation returns the disjunction of the violations across all coefficients, which is 1 if and only if at least one coefficient violates the constraints.

The hardened implementation of `test_w` is shown in Figure 4.8. Again, the cache-side-channel mitigation is highlighted in gray. Instead of adding `PARAM_Q` only to those coefficients `val`

```
1  int test_rejection(poly poly_z) {
2   int i; int res; res = 0;
3   for(i=0; i<PARAM_N; i++){
4     res |= (poly_z[i] < -(PARAM_B-PARAM_U));
5     res |= (poly_z[i] > (PARAM_B-PARAM_U));   }
6   return res; }
```

Figure 4.7: Hardened Implementation of `test_rejection`

```
1  int test_w(poly poly_w) { [...]
2   for(i=0; i<PARAM_N; i++) {
3    val = poly_w[i]; val = val % PARAM_Q;
4    val += (((unsigned int)val & 0x80000000) >> 31)*PARAM_Q;
5    left = val; left = left % (1<<(PARAM_D));
6    left -= (1<<PARAM_D)/2; left++;
7    right = (1<<(PARAM_D-1))-PARAM_REJECTION;
8    res |= (abs(left) - right > 0); }
9   return -res; }
```

Figure 4.8: Hardened Implementation of `test_w`

that are smaller than zero, the modified implementation adds `PARAM_Q` to all coefficients (see Line 4). To preserve the original functionality, it masks the value `PARAM_Q` by the first bit of an unsigned representation of `val`. Since the first bit of an unsigned integer indicates the sign in two's-complement notation, the masked `PARAM_Q` will be zero for positive values of `val` and remain `PARAM_Q` for negative values of `val`. The technique to replace a branch by a masked assignment is inspired by the conditional-assignment countermeasure against timing side channels [95].

Moreover, the modified implementation does not abort upon the first violation of the conditions by one coefficient `val` and the corresponding value `left` derived from `val`. Like the modified `test_rejection`, it collects the disjunction of violations across all coefficients in an auxiliary variable `res` (see Line 8). Since the original implementation returns -1 if any coefficient is invalid and 0 otherwise, the modified implementation returns `-res` to preserve the functionality.

Note that, the accumulation of the check results in the hardened variants of `test_rejection` and `test_w` leads to the use of the instructions `Setl` and `Setg` in the corresponding x86 binaries. Since our abstract semantics captures the behavior of both instructions, CacheAudit 0.2c is applicable also to the binaries of the hardened functions.

4.7.2 Analysis of the Hardened Functions

We apply CacheAudit 0.2c to implementations of `test_rejection` and `test_w` with the above mitigations in place. The leakage bounds resulting from this analysis are shown in Table 4.4.

	acc	*accd*	*trace*	*time*
`test_w`	0 bit	0 bit	0 bit	0 bit
`test_rejection`	0 bit	0 bit	0 bit	0 bit

Table 4.4: Leakage Bounds for Functions with Mitigation

The upper bounds on the cache-side-channel leakage of the two functions are 0 bit across all four attacker models. That is, there is no more leakage of secret information to attackers under these models. In particular, there is no more leakage about the index of the coefficient from the polynomials `poly_z` and `poly_w`, respectively, that made a candidate signature invalid.

This means that any information about `pos_list` leaked in `generate_c` and `computeEc` cannot be exploited. A ring-TESLA implementation with the mitigations in place is hardened against attacks with respect to the models *acc*, *accd*, *trace*, and *time*.

Note that, the mitigations introduce a performance penalty in cases where a signature has to be rejected. When `test_rejection` or `test_w` encounters a polynomial coefficient that violates the allowed range, the remaining coefficients are still checked before rejecting the signature. Since rejection means that the entire signature computation needs to be restarted, this performance penalty is likely dominated by the overhead for the recomputation. In addition, the parameters for the signature scheme were chosen in a way that limits the probability of rejections [3]. Therefore, the mitigations that remove the detected leakage completely at the cost of performance overhead in case of rejections are a sensible choice with respect to the security-performance trade-off here.

Based on our results, the mitigations have already been deployed in the reference implementations of ring-TESLA and of the follow-up scheme qTESLA [22] that was a Round-2 candidate in the NIST post-quantum standardization [4].

4.8 Summary

In this chapter, we investigated the security of an implementation of the lattice-based signature scheme ring-TESLA [3] against cache side channels. To this end, we developed a program analysis that is able to compute upper bounds on the cache-side-channel leakage of the ring-TESLA binary. The novel aspect of our analysis is an abstract semantics that captures the behavior of multiple previously unsupported x86 instructions, including multiple instructions required for the handling of large operands like, e.g., `Cqd` and variants of `Imul`. Our abstract semantics is suitable for the analysis of the ring-TESLA implementation, which is based on very large operands. The maximum key size in ring-TESLA is 192 times larger than the maximum key size of AES.

Based on our program analysis and its automation in the tool CacheAudit 0.2c, we analyzed an x86 binary of the ring-TESLA implementation that contains no infinite loops but has a well-defined correspondence to the original implementation.

Based on the leakage bounds from this case study, we described the cache-side-channel vulnerabilities that we detected together with the ring-TESLA developers in the four functions `generate_c`, `computeEc`, `test_rejection`, and `test_w`. Since these vulnerabilities in combination might be exploited to break the signature scheme [21], we developed mitigations for the leakage in `test_rejection` and `test_w` together with the ring-TESLA developers. We analyzed the mitigations and showed that they effectively remove the detected leakage in the two functions. With the leakage in these two functions removed, the above-mentioned exploit is no longer possible. The mitigations have already been adopted to harden the implementations of ring-TESLA and its successor qTESLA [22] before its submission to the NIST PQC Standardization [4].

Chapter 5

Cache-Side-Channel Quantification for QKD Software

5.1 Introduction

Quantum Key Distribution is a technique for key establishment that is secure even against attackers who have access to quantum computers. Multiple initiatives for the deployment of QKD are ongoing. The project QuNET [39], in which the German government is planning to invest 165 M€, aims at a QKD network among governmental agencies in Germany. The project Quapital [61] plans to connect cities across Europe through a QKD network. A QKD network that connects multiple Chinese cities, including Beijing and Shanghai, was already established in 2018 and is currently in a trial period [137].

The security of QKD is based on the laws of quantum physics instead of computational hardness assumptions. As described in Section 2.3.3, the key idea is to transmit a bitstring between two parties over a so-called quantum channel, on which qbits are transmitted using the states of physical particles, e.g., photons. The physical states used to encode information are selected in a way that allows one to detect eavesdropping on the transmission with high probability due to the uncertainty principle of quantum physics. Based on the bitstring transmitted over the quantum channel, the two parties then compute a shared secret key in a postprocessing phase. In this postprocessing phase, the parties exchange bits over a traditional channel (using different voltage levels to encode 1 and 0) and perform computations on their traditional computers.

The security properties that are usually investigated for QKD solutions (see Section 2.3.3) take into account attackers who have access to quantum computers and who can act as men in the middle on both, the quantum channel and the traditional channel. The properties do not take into account cache-side-channel attacks that target computations on traditional computers. In practice, however, it is common to use regular PCs for the postprocessing in QKD setups like, e.g., [120, 121]. That is, regular cache-side-channel attacks might be mounted on the postprocessing software and endanger the key. Furthermore, QKD might be deployed in the cloud [32, 93] or on mobile devices [127], which would further increase the potential for cache-side-channel attacks.

In this chapter, we focus on the security of QKD against cache-side-channel attackers. We perform a case study based on an implementation of the BB84 protocol for QKD [100]. In this case study, we analyze the threat that cache side channels pose to this implementation. During our study, we detect a vulnerability in the implementation that might leak the entire secret key to a cache-side-channel attacker. We propose a side-channel mitigation for hardening the implementation against this vulnerability and derive security guarantees for the hardened QKD solution resulting from our mitigation. We also lift these security guarantees to an overall QKD solution. Our mitigation has already been deployed in a new version of the QKD software

from [100] at the Department of Physics at TU Darmstadt.

Our detection of the vulnerability and our security guarantees are based on cache-side-channel leakage bounds for the target QKD implementation. Existing program analyses that compute such leakage bounds, including [45, 44] and our analyses from Chapter 3 and Chapter 4, are not applicable here. The reason is that QKD software, unlike the implementations of AES and ring-TESLA that we considered so far, heavily relies on the computation of probabilities, i.e., floating-point numbers. Instructions that handle floating-point numbers are executed by dedicated micro-architectural components that are not captured by the abstract domains underlying prior analyses. The computations on probabilities and, hence, on floating-point values, are inherent in the QKD protocol and cannot be avoided. Therefore, we develop a new, more powerful abstract domain, which captures both, a CPU with two execution units: a regular ALU and an FPU that executes floating-point instructions. The domain captures the possible states of the FPU, including the FPU status flags, FPU stack, and corresponding tags, at a granularity that allows for an efficient yet precise computation of leakage bounds. For instance, the possible values of FPU status flags are captured in relation to the possible values stored in other components, e.g., the registers on the FPU stack. Moreover, the domain tracks the relation between components accessed by the ALU, components accessed by the FPU, and components accessed by both.

We automate our program analysis in a tool that can compute cache-side-channel leakage bounds for programs that use floating-point computations and, in particular, for our target QKD implementation. Our case study for the QKD implementation from [100] is based on this tool.

In Section 5.2, we describe our target implementation. In Section 5.3, we describe our abstract domain that models a CPU with an FPU. In Section 5.4, we describe the analysis setup that we use. In Section 5.5, we describe the vulnerability that we detected in the QKD implementation and how we mitigated it. We provide security guarantees for a hardened QKD solution based on our mitigation in Section 5.6.

5.2 Target QKD Implementation

In this chapter, we aim at security guarantees that are meaningful for the existing implementation of QKD postprocessing software from [100]. To this end, we develop a target implementation that (1) is sufficiently close to a selected core of [100] to allow us to transfer our analysis results between the implementations, and (2) avoids unnecessary complexity in the form of object orientation, dynamic memory allocation, potentially infinite loops, and global variables. In this section we describe our target implementation and its relation to the original implementation.

5.2.1 Selection of QKD Steps

Out of the four QKD postprocessing steps, key sifting, parameter estimation, error correction, and privacy amplification, we focus on the steps from which we expect interesting insights with respect to cache side channels. We select these steps as follows.

The error-correction and privacy-amplification steps of QKD process secret bitstrings, namely the sifted key and error-corrected key. The computations performed on the secret bitstrings involve, e.g., matrix multiplications. Therefore, the implementations of error correction and privacy amplification are both, highly security critical and at high risk of cache-side-channel leakage. In our analysis, we focus on the implementation of these two steps. Since the error-correction step differs between Alice and Bob (encoding and decoding), we consider implementations for both sides. The privacy-amplification step is identical for Alice and Bob.

We do not consider the two remaining postprocessing steps. The key-sifting step discards bits from the raw key, depending only on the polarization bases. The polarization bases are not secret anymore after the raw key exchange is completed and the step therefore has a lower risk for cache-side-channel leakage. In the parameter-estimation step, sample bits from the sifted key

are selected and exchanged. The selection of the bits does not depend on secret information and the sample bits themselves are not secret anymore after the parameter-estimation step because they are discarded from the sifted key. Like for the key-sifting step, there is only a lower risk for cache-side-channel leakage here.

All in all, our target QKD implementation consists of three components: encoding, decoding, and privacy amplification. Each component is connected to the respective implementation from the original QKD software [100] in a well-defined way.

5.2.2 Target Components

We explain the components of our target implementation at the level of C code for improved readability. For our analysis, we consider the x86 binaries resulting from the C code through compilation with `gcc 7.4.0` and the flags `-m32 -fno-stack-protector`.

Encoding For the encoding side (Alice) of the error-correction step, our target implementation is shown in Figure 5.1. The implementation encodes a sifted-key block `sblk` of length `K` $= 4$ into a codeword block `cblk` of length `N` $= 7$ by appending `M` $= 3$ parity bits. To this end, the implementation multiplies the sifted-key block by an LDPC generator matrix `G`.

```
1  #define M 3
2  #define K 4
3  #define N M+K
4  void mod2dense_set(char m[N][1],int row,int col,int value){
5   m[row][col] = value;}
6  int mod2dense_get(char m[M][1],int row,int col){
7   return (int) m[row][col]; }
8  void mod2dense_multiply(char m1[M][K],char m2[K][1],char r[M][1]){
9   for (int i=0;i<M;i++){ r[i][0]=0; }
10  for(int k=0;k<K;k++){
11   if (m2[k][0]==1){
12    for(int m=0;m<M;m++){ r[m][0]^=m1[m][k]; }}}}
13
14 void dense_encode(char sblk[K],char cblk[N],char u[K][1],char v[M][1]){
15  int cols[N]={4,5,6,0,1,2,3};
16  char G[M][K]; int j;
17  for(j=M;j<N;j++){
18   cblk[cols[j]] = sblk[j-M]; }
19  for(j=M;j<N;j++){
20   mod2dense_set(u,j-M,0,sblk[j-M]);}
21  mod2dense_multiply(G,u,v);
22  for(j=0;j<M;j++){
23   cblk[cols[j]]=mod2dense_get(v,j,0);}}
24
25 void main(){
26  char sblk[K]; char cblk[N];
27  char u[N][1]; char v[M][1];
28  dense_encode(sblk,cblk,u,v);}
```

Figure 5.1: Target Encoding Implementation

The functionality is implemented by the function `dense_encode`. It receives as parameters the sifted-key block `sblk`, a location for storing the codeword block `cblk`, and locations `u` and `v` for storing temporary information (see Line 14). The function then first copies the sifted-key block into the first four entries of the codeword block in Line 18. Second, it stores a temporary copy of `sblk` in the matrix `u` (Line 20) and calls the function `mod2dense_multiply` to multiply `G` and `u` and store the result in the matrix `v` (Line 21).

This target implementation is a simplified version of the encoding implementation in the function `ldpc_encoder::dense_encode` from the file `enc.cpp` of [100]. More concretely, it differs

from the original implementation in two aspects:

1. While [100] stores matrices and vectors in dynamically allocated pointer structures, our target implementation uses fixed-size arrays.
2. While the generator matrix `G` is a global variable in [100], it is a local variable in our target implementation.

Both simplifications aim at reducing the amount of complex features that a cache-side-channel analysis of the implementation would have to support. The simplifications do not influence the functionality of the implementation, as we confirmed in a sanity check with a Hamming(7,4) code.

Moreover, the simplifications only affect the allocation and scope of variables. They do not alter the control flow or the memory accesses of the implementation. That is, they have no influence on the possible executions and their cache usage. Thus, upper bounds on the cache-side-channel leakage of the simplified implementation will be transferable to the original implementation.

Decoding For the decoding side (Bob) of the error-correction step, our target implementation is shown in Figure 5.2. The implementation uses the parity matrix `H_val` to compute the most likely permutation of Bob's sifted-key block that matches the parity bits received from Alice. It stores the result in the vector `dblk`. Bob's sifted-key block and the received parity bits are represented by the vector `lratio`, which contains initial likelihood ratios for the values of each sifted-key-block bit and parity bit. The likelihood ratios are updated iteratively during the computation. Intermediate values are stored in `H_lr` and `H_pr` and the final probabilities for each bit of the sifted-key block are stored in `bprb`. The function `initprp` initializes the likelihood ratios. The function `iterprp` updates them iteratively. The function `iterprp` also handles overflows in the intermediate probability values (see Lines 30 and 37).

Both functions are simplified versions of the original functions `ldpc_decoder::initprp` and `ldpc_decoder::iterprp` from the file `dec.cpp` of [100]. The simplifications are as follows:

1. As in the encoding case, our target implementation uses fixed-size arrays where [100] uses dynamically allocated pointer structures.
2. While the original implementation is parametric in the parity matrix, our target implementation uses a fixed parity matrix.
3. Our target implementation performs only one iteration of the function `iterprp`, while this function is usually iterated multiple times.
4. While the original implementation skips the handling of overflows in probability values if no overflow occurred, our target implementation performs overflow handling on dummy values in this case (see Lines 30 and 37 in Figure 5.2).

As in the encoding case, we performed a sanity check of the functionality of our target implementation (using 100 iterations of `iterprp`) with a Hamming(7,4) code.

Our goals for the simplifications were twofold: avoiding false positives in the analysis of the implementation (fixed parity matrix, modified overflow handling) and reducing the set of language features that our program analysis has to support (fixed-size arrays, only one iteration of `iterprp`). Note that, the decoding step inherently depends on the computation of probabilities (i.e., floating-point values), which are stored, e.g., in the variables `H_lr`, `H_pr`, and `lratio`. It is therefore not possible to simplify further and eliminate the use of floating-point instructions.

The use of fixed-size arrays alters the possible cache usage because the original implementation from [100] iterates through the parity matrix along the pointers stored in the data structure for the matrix. Our target implementation iterates through all matrix entries linearly. That is, the memory accesses in this simplified implementation no more depend on the entries of the parity matrix. This influence on the cache usage, however, does not influence the cache-side-channel leakage. The reason is that the parity matrix is public. It is not a security concern if the attacker can derive information about this matrix from the cache usage.

The use of a fixed parity matrix reduces the set of possible executions and, hence, of possible cache usages. The parity matrix determines which sifted-key bits are accessed in the computation

```
#include "math.h"
#define M 3
#define K 4
#define N M+K
void initprp (char H_val[M][N], double H_lr[M][N], double H_pr[M][N], double
    lratio[N], char dblk[N], double bprb[N]){
 int e; int j;
 for (j = 0; j<N; j++){
  for (e = 0; e < M;  e++){
   if (H_val[e][j] == 1){
    H_pr[e][j] = lratio[j]; H_lr[e][j] = 1.0;}}
  if (bprb) bprb[j] = 1 - 1/(1+lratio[j]);
  dblk[j] = lratio[j]>=1;}}

void iterprp(char H_val[M][N], double H_lr[M][N], double H_pr[M][N], double
    lratio[N], char dblk[N], double bprb[N]){
 double pr, dl, t; int e, i, j;
 double temp; double temp2; double dummy;
 for (i = 0; i<M; i++){ dl = 1;
  for (e = 0; e < N; e++){
   if (H_val[i][e] == 1){
    H_lr[i][e] = dl; dl *= 2/(1+H_pr[i][e])-1;}}
  dl = 1;
  for (e = N-1; e >= 0; e--){
   if (H_val[i][e] == 1){
    t = H_lr[i][e]*dl; H_lr[i][e] = (1-t)/(1+t);
    dl *= 2/(1+H_pr[i][e]) - 1;}}}
 for (j = 0; j<N; j++){ pr = lratio[j];
  for (e = 0; e < M; e++){
   if (H_val[e][j] == 1){
    H_pr[e][j] = pr; pr *= H_lr[e][j];}}
  if (isnan(pr)){ pr = 1;}
   else{dummy=1;__asm__("nop":::);}
  if (bprb) bprb[j] = 1 - 1/(1+pr);
  dblk[j] = pr>=1; pr = 1;
  for (e = M-1; e >= 0; e--){
   if (H_val[e][j] == 1){
    H_pr[e][j] *= pr; temp = H_pr[e][j];
    if (isnan(temp)){temp = 1;}
    else{temp2=1;__asm__("nop":::);}
    H_pr[e][j] = temp; pr *= H_lr[e][j];}}}}

void main(){
 char H_val[M][N] = {{1,1,0,1,1,0,0}, {1,0,1,1,0,1,0}, {0,1,1,1,0,0,1}};
 double H_lr[M][N]; double H_pr[M][N];
 double lratio[N]; char dblk[N]; double bprb[N];
 initprp(H_val, H_lr, H_pr, lratio, dblk, bprb);
 iterprp(H_val, H_lr, H_pr, lratio, dblk, bprb);}
```

Figure 5.2: Target Decoding Implementation

of which parity bit. That is, fixing the matrix eliminates dependencies of the cache usage on the parity matrix. Again, this does not influence the leakage since the parity matrix is public.

The two remaining simplifications might in theory influence not only the cache usage, but also the cache-side-channel leakage of the implementation. We will discuss why this is not the case in practice when we interpret our leakage bounds in Section 5.6.

Privacy Amplification For the privacy-amplification step, our target implementation is shown in Figure 5.3. The privacy amplification happens in the function `calcPAKey`. The function receives as parameters the error-corrected key block `key` of length `KEYLENGTH` = 4 and the target length of the privacy-amplified key, which we set to `PAKEYLENGTH` = 2. The function then multiplies the error-corrected key block with the Toeplitz matrix `toepMat` (Line 14) and returns the result as the privacy-amplified key block `paKey`.

```
1  #include "string.h"
2  #include "stdbool.h"
3  #define KEYLENGTH 4
4  #define PAKEYLENGTH 2
5  void calcPAKey(bool* key, int paLen){
6   int toepMatLen=KEYLENGTH+paLen-1;
7   char paKey[paLen]; char toepMat[toepMatLen];
8   for(int i=0;i<toepMatLen;++i){
9    toepMat[i]= *((int *)0x4) & 1;}
10  for(int i=0;i<paLen;i++){
11   paKey[i]=0;
12   for(int j=0;j<KEYLENGTH;j++){
13    int id=i-j+KEYLENGTH-1;
14    paKey[i]+=toepMat[id]*key[j];
15    paKey[i]=paKey[i]%2;}}}
16
17 void main(){
18  bool key[KEYLENGTH];
19  calcPAKey(key, PAKEYLENGTH);}
```

Figure 5.3: Target Privacy-Amplification Implementation

Our target implementation is a simplification of the function `qkdtools::PrivAmp::calcPAKey` from the file `PrivAmp.cpp` of [100]. It differs from [100] in three aspects:

1. While the original implementation from [100] uses variable-sized arrays, our target implementation uses fixed-size arrays.
2. While the original implementation is object-oriented and stores information in class members, our target implementation stores this information in local variables and function parameters.
3. While the original implementation is parametric in the Toeplitz matrix, our target implementation initializes each entry of the Toeplitz matrix to the least-significant bit of the value stored at the uninitialized address 0x4 (see Line 9).

For the first two simplifications, our goal was to reduce the amount of language features that our program analysis has to support. We tested that they do not affect the privacy-amplification functionality. The third simplification allows us to analyze the implementation independently of a concrete Toeplitz matrix because it makes explicit that the matrix is uninitialized.

All three simplifications (the fixed size of variables, changed scope of variables, and the explicit marking of the Toeplitz matrix as uninitialized) do not influence the control flow or memory accesses during the privacy amplification. That is, the cache usage of the original implementation is preserved and leakage bounds can be transferred from our target implementation to [100].

Overall, we now have target implementations for all three of our three target QKD postprocessing steps: encoding, decoding, and privacy amplification.

5.3 Program Analysis for QKD Software

Like our program analyses in the previous chapters, the analysis in this chapter uses abstract interpretation to overapproximate the reachable attacker observations and computes the logarithm of this overapproximated number as the leakage bound. The key novelty of the analysis in this chapter is the underlying abstract domain, which captures the possible execution snapshots on systems that feature a CPU with both, an ALU and an FPU.

The Intel x86 instruction set contains a subset of instructions for floating-point computations. The instructions in this subset are referred to as x87 instructions [63]. In addition to the x87 instructions, several extensions to the instruction set have been defined, including, e.g., the Multi Media Extension (MMX), the Streaming SIMD Extensions (SSE), and the Advanced Vector Extensions (AVX). We focus on x87 instructions, because these are the instructions that occur in our target implementation. We model the set of x86 instructions, including regular instructions as well as the x87 instructions by the set $\mathcal{I}_{64}$, where $\mathcal{I}_{64} \supset \mathcal{I}'_{32}$.

While regular x86 instructions are processed by the regular Arithmetic Logic Unit (ALU) of the CPU, x87 instructions are processed by a specialized execution unit, namely the FPU. Separate FPU co-processors are available for some micro-controllers, but usually nowadays the FPU is integrated directly on the CPU chip and shares the same L1 data cache with the ALU. This is, e.g., the case in the BOOM RISC-V micro-architecture [29]. In our abstract domain, we capture the possible states of a CPU with on-chip FPU and one L1 data cache.

For processing regular x86 instructions, the CPU operates on a set of 32 bit registers and on Byte-addressable 32 bit memory entries. It also operates on a special register, called EFLAGS register, in which it maintains six 1 bit status flags used to implement conditional control flow [63]: Carry Flag (CF), Parity Flag (PF), Auxiliary Carry Flag (AF), Zero Flag (ZF), Sign Flag (SF), and Overflow Flag (OF), which are set based on the results of arithmetic operations and comparisons. We model the set of all CPU flags by the set $\mathcal{F}_{\text{CPU}} = \{CF, PF, AF, ZF, SF, OF\}$ and the set of all possible 32 bit values that can be stored in registers or memory entries by $\mathcal{V}_{32}$.

The FPU shares the memory entries with the ALU, but has its own set of eight FPU registers. The FPU registers are arranged in the so-called FPU stack and are 80 bit wide (1 bit sign, 15 bit exponent, and 64 bit mantissa). They are each accompanied by a tag that indicates the status of their contents ("empty", "zero", "valid", or "special") [63]. The FPU also maintains four status flags of its own, which are called condition-code flags: C0, C1, C2, and C3.

Let $\mathcal{R}_{80}$ be the set of all FPU registers and let $\mathcal{T}_{\text{FPU}} = \{valid, empty\}$ be the set of FPU-register tags. Here, the symbol *empty* models the tag "empty" and the symbol *valid* captures all remaining tags. Let $\mathcal{F}_{\text{FPU}} = \{C0, C1, C2, C3\}$ be the set of all FPU flags. Furthermore, let $\mathcal{V}_{80}$ be the set of all possible 80 bit floating-point values.

We model the L1 cache that is located on the chip alongside the ALU and FPU as follows. By the set $\mathcal{C}_{\text{pos}}$, we model the set of all cache lines in the cache, i.e., all positions at which a memory entry could be cached. The set $\mathcal{C}_{\text{pos}}$ also includes a special symbol to model the position "uncached", i.e., that a memory entry is not located at any position in the cache.

Concrete Domain We first introduce a concrete domain that captures the possible states of the CPU, FPU, and cache precisely with respect to the model described above. Our concrete domain is called $\mathcal{D}_{64}$ and is defined as follows.

$$\begin{aligned}\mathcal{D}_{64} = &(\mathcal{F}_{\text{CPU}} \to \mathbb{B}) \times (\mathcal{F}_{\text{FPU}} \to \mathbb{B}) \times \\ &(\mathcal{R}_{32} \to \mathcal{V}_{32}) \times (\mathcal{R}_{80} \to \mathcal{T}_{\text{FPU}} \times \mathcal{V}_{80}) \times \\ &(\mathcal{M}_{32} \to \mathcal{V}_{32}) \times (\mathcal{M}_{32} \to \mathcal{C}_{\text{pos}}) \times \{h, m, n\}^\star\,.\end{aligned}$$

Each concrete execution state $d \in \mathcal{D}_{64}$ is a 7-tuple of six functions and one list. The state of the CPU is captured by three functions: A function of type $\mathcal{F}_{\text{CPU}} \to \mathbb{B}$ assigns each CPU flag to its Boolean value. Two functions of types $\mathcal{R}_{32} \to \mathcal{V}_{32}$ and $\mathcal{M}_{32} \to \mathcal{V}_{32}$ map each register

and memory entry to their current value. The state of the FPU is captured by two additional functions: A function of type $\mathcal{F}_{\text{FPU}} \to \mathbb{B}$ maps each FPU flag to its current value and a function of type $\mathcal{R}_{80} \to \mathcal{T}_{\text{FPU}} \times \mathcal{V}_{80}$ maps each FPU register to its current tag and value. Finally, the state of the cache is captured by a function of type $\mathcal{M}_{32} \to \mathcal{C}_{\text{pos}}$, which maps each memory entry to its current position in the cache or to "uncached", and by a list of type $\{h, m, n\}^\star$, which captures the trace of cache hits (h), misses (m), and steps without memory access (n) encountered.

The concrete semantics of regular x86 and x87 instructions can then be modeled by a function

$$\texttt{upd}_{\mathcal{D}_{64}} : \mathcal{D}_{64} \times \mathcal{I}_{64} \to \mathcal{D}_{64},$$

which maps each instruction $i \in \mathcal{I}_{64}$ and concrete state $d \in \mathcal{D}_{64}$ to an updated state $\texttt{upd}_{\mathcal{D}_{64}}(d, i)$.

Abstract Domain The cache-side-channel leakage of an x86/87 binary depends on how the cache usage differs across the possible executions of the binary with all possible inputs. Computing the set of all executions that are reachable with respect to the concrete domain is usually infeasible for complex binaries. Therefore, we define an abstract domain $\overline{\mathcal{D}}_{64}$. Each abstract state $\overline{d} \in \overline{\mathcal{D}}_{64}$ represents a set of concrete states from $\mathcal{D}_{64}$. The abstract domain is defined as follows.

$$\begin{aligned}\overline{\mathcal{D}}_{64} = {} & ((\mathcal{F}_{\text{CPU}} \to \mathbb{B}) \times (\mathcal{F}_{\text{FPU}} \to \mathbb{B})) \rightharpoonup \\ & ((\mathcal{R}_{32} \to \mathcal{P}(\mathcal{V}_{32})) \times (\mathcal{R}_{80} \to \mathcal{P}(\mathcal{T}_{\text{FPU}}) \times \mathcal{P}(\mathcal{V}_{80})) \times \\ & (\mathcal{M}_{32} \to \mathcal{P}(\mathcal{V}_{32})) \times (\mathcal{M}_{32} \to \mathcal{P}(\mathcal{C}_{\text{pos}})) \times \mathcal{P}(\{h, m, n\})^\star).\end{aligned}$$

This abstract domain captures the values of the CPU flags using a function of type $\mathcal{F}_{\text{CPU}} \to \mathbb{B}$ like the concrete domain. Unlike the concrete domain, $\overline{\mathcal{D}}_{64}$ abstracts from the concrete values of CPU registers and memory locations. These are captured by functions of types $\mathcal{R}_{32} \to \mathcal{P}(\mathcal{V}_{32})$ and $\mathcal{M}_{32} \to \mathcal{P}(\mathcal{V}_{32})$, which assign a set of possible values to each register and memory location.

The FPU state is captured in the abstract domain analogously to the CPU state: The FPU flags are captured by a function of type $\mathcal{F}_{\text{FPU}} \to \mathbb{B}$ that does not abstract from the possible values of each flag. The possible values of the FPU registers and tags are represented using a set abstraction, i.e., a function of type $\mathcal{R}_{80} \to \mathcal{P}(\mathcal{T}_{\text{FPU}}) \times \mathcal{P}(\mathcal{V}_{80})$. Finally, the cache state is represented by a function of type $\mathcal{M}_{32} \to \mathcal{P}(\mathcal{C}_{\text{pos}})$ and by a set of lists of type $\mathcal{P}(\{h, m, n\})^\star$.

Overall, the abstract domain $\overline{\mathcal{D}}_{64}$ is a component-wise set abstraction from the concrete domain $\mathcal{D}_{64}$, with the exception of the CPU and FPU flags. The flags are represented by their concrete values and the abstract domain is defined as a mapping from the state of the flags to the state of the remaining components. That is, an abstract state $\overline{d}$ keeps track of separate abstract states for each possible combination of flag values. Since both, the CPU flags and the FPU flags are used to implement conditional control flow, this design choice is crucial for the precision of the abstract domain. Distinguishing between the possible execution states across different flag values allows us to distinguish between the possible execution states across different control flows.

The abstract semantics of x86 and x87 instructions can then be defined by a function

$$\texttt{upd}_{\overline{\mathcal{D}}_{64}} : \overline{\mathcal{D}}_{64} \times \mathcal{I}_{64} \to \overline{\mathcal{D}}_{64},$$

where $\texttt{upd}_{\overline{\mathcal{D}}_{64}}(\overline{d}, i)$ overapproximates the effect of instruction $i \in \mathcal{I}_{64}$ on the concrete states represented by the abstract state $\overline{d} \in \overline{\mathcal{D}}_{64}$.

Our program analysis applies the abstract semantics to an initial abstract state $\overline{d} \in \overline{\mathcal{D}}_{64}$ and the instructions from the x86/87 binary to be analyzed. We choose the initial abstract state $\overline{d}$ for our analysis as follows: We represent fixed initial values using singleton sets and values that might vary across executions (e.g., the secret input about whose leakage we are concerned) using sets that contain all possible values. By repeatedly applying the function $\text{upd}_{\overline{\mathcal{D}}_{64}}$ starting from this initial abstract state, the analysis computes the reachable abstract states.

Let $Obs^{\overline{\mathcal{D}}_{64}}$ be the attacker observations that are possible based on the concrete states represented by the reachable abstract states. Our program analysis returns $\log_2 |Obs^{\overline{\mathcal{D}}_{64}}|$, which bounds the leakage of the x86/87 binary (see Section 2.2.1).

5.4 Analysis Setup

We automated the program analysis described in Section 5.3 with a suitable abstract semantics in the analysis tool CacheAudit-FPU [132]. The overall analysis workflow is similar to the workflows in the analysis tools from Chapter 3 and Chapter 4, but augmented by an implementation of the FPU model alongside the ALU model in the individual modules. The detailed implementation of the analysis in the CacheAudit framework was supported by a Master student.

We apply the analysis tool with respect to the following cache configuration. As in Chapter 4, we focus on a 32 KiB 8-way set-associative data cache with 64 B line size as used, e.g., in the L1 cache of the Skylake architecture [62, Table 2-4], and LRU replacement.

5.5 Vulnerability in the QKD Software

With our program analysis, we detect a vulnerability with respect to the *trace* attacker model in our target implementation. The vulnerability occurs in the encoding component and also affects the original QKD implementation from [100]. In the following, we describe the vulnerability and a solution for hardening the QKD implementation against the vulnerability. We will discuss the effects of other cache-side-channel attacker models in Section 5.6.3.

5.5.1 Detection and Assessment of the Vulnerability

For the encoding component of our target implementation, we obtain the upper bound

$$4\,\text{bit}$$

on the cache-side-channel leakage to *trace*. As our encoding target implementation (see Figure 5.1) computes on a 4 bit block of the sifted key, a leakage of 4 bit would reveal the entire block.

Recall from Section 5.2 that our target implementation differs from the original encoding implementation in [100] only in the scope and allocation time of variables. Since these changes do not impact the control flow or the memory accesses of the implementation, the possible cache traces are the same across both implementations. That is, the leakage of the original implementation about a 4 bit sifted-key block is also bounded by 4 bit.

An inspection of the original encoding implementation from [100] shows that the implementation might, indeed, reveal the entire sifted key. Consider the excerpt from the original encoding implementation in Figure 5.4. The function `mod2dense_multiply` receives Alice's secret sifted-key block as parameter `m2` and the public generator matrix as parameter `m1`. It computes the result matrix `r` by computing the values for each column.

The function iterates through all columns of the result matrix (Line 2 in Figure 5.4). For each column, it iterates through all bits of the sifted-key block (Line 3). It checks whether the respective bit of the sifted-key block is set (Line 4). If the bit is set, the corresponding row of the generator matrix contributes to the result of the matrix multiplication. In these cases, the function adds the corresponding row to the intermediate result (Lines 5 and 6).

Overall, the function `mod2dense_multiply` accesses exactly those rows of the generator matrix `m1` that correspond to set bits in the sifted-key block. A cache-side-channel attacker might exploit this as follows: He records the cache trace of the encoding execution on the secret sifted key (e.g., based on power-consumption measurements [1]). He identifies the parts of the trace that correspond to each iteration of the loop in Line 3, e.g., based on counting the non-memory-access steps between the iterations. The attacker then records in which iterations the sifted-key bit that is processed has the value 1. This is possible, because for iterations that process sifted-key bits with value 1, the trace contains more cache accesses than for iterations that process sifted-key bits with value 0 (Line 4). The attacker concatenates the recovered bits to obtain the sifted-key block. That is, an attacker under the model *trace* might deduce one of Alice's sifted-key blocks by

```
1  void mod2dense_multiply(mod2dense *m1,mod2dense *m2,mod2dense *r){ [...]
2   for(j=0;j<mod2dense_cols(r);j++){
3    for(i=0;i<mod2dense_rows(m2);i++){
4      if(mod2dense_get(m2,i,j)){
5       for(k=0;k<r->n_words;k++){
6        r->col[j][k]^=m1->col[i][k];}}}}
7
8  void dense_encode(char *sblk,char *cblk,mod2dense *u,mod2dense *v){ int j;
9   for(j=M;j<N;j++){
10   cblk[cols[j]]=sblk[j-M];}
11  for(j=M;j<N;j++){
12   mod2dense_set(u,j-M,0,sblk[j-M]);}
13  mod2dense_multiply(G,u,v);
14  for(j=0;j<M;j++){
15   cblk[cols[j]]=mod2dense_get(v,j,0);}}
```

Figure 5.4: Excerpt from the QKD Implementation [100]: Encoding

analyzing portions of the cache trace. If he repeats this process for each of the sifted-key blocks, he might obtain Alice's entire sifted key.

Recall from Section 2.3.3 that Alice's sifted key is identical to the error-corrected key. That is, the attacker might obtain the entire error-corrected key. Recall also from Section 2.3.3 that the Toeplitz matrix used to compute the privacy-amplified key from the error-corrected key is public. That is, an attacker who knows the error-corrected key can simply multiply it with the public Toeplitz matrix to obtain the privacy-amplified key, which is the final result of the QKD. Overall, an attacker under *trace* might, hence, obtain the entire shared secret key established with the QKD. That is, the QKD can be broken without even attacking any of the communication between Alice and Bob.

Since the vulnerability in [100] might reveal the entire secret key that results from the QKD, it is a very serious concern for QKD solutions that use the implementation from [100]. Moreover, the vulnerability is not limited only to the implementation from [100]. The implementation from [100] reuses the encoding function from the LDPC implementation by Radford Neal [97] who rediscovered LDPC codes together with MacKay in 1996 [83]. The LDPC implementation by Neal also contains the vulnerability described above and has been forked by many others [52].

5.5.2 Mitigation of the Vulnerability

We propose to mitigate the vulnerability described in Section 5.5.1 using a modified version of the function `mod2dense_multiply`, shown in Figure 5.5.

```
1  void mod2dense_multiply(char m1[M][K], char m2[K][1], char r[M][1]){
2   for (int m=0; m<M; m++){
3    r[m][0] = '0';
4    for (int k=0; k<K; k++){
5     r[m][0]^= m1[m][k] & m2[k][0];}
6    r[m][0] = r[m][0] % 2;}}
```

Figure 5.5: Hardening of the Encoding Function

Like the original implementation, the modified version computes each column of the result matrix `r`. For each column of `r`, the modified implementation iterates through all rows of the generator matrix `m1`, independently of the sifted-key bit to be multiplied by the respective row. That is, the amount of memory entries accessed by an execution of the modified multiplication function is independent of the bits in the secret sifted-key block.

When accessing each element of the respective row of the generator matrix, the modified function masks the element by the corresponding bit of the sifted key. This technique is inspired by the program transformation conditional assignment [95] and ensures that the correct functionality of the matrix multiplication is preserved by the modified implementation.

We replaced the function `mod2dense_multiply` in our simplified encoding implementation from Figure 5.1 and tested the functionality of the resulting implementation using a Hamming(7,4) code. By integrating our mitigation, we obtained a hardened version of our target implementation.

To apply the mitigation to the original implementation from [100], the matrices `m1`, `m2`, and `r` in Figure 5.5 need to be changed back to pointer structures. Afterwards, the function `mod2dense_multiply` can be plugged directly into the original encoding implementation from Figure 5.4. Our mitigation has already been deployed at the Department of Physics at TU Darmstadt in a new version of the QKD software from [100].

5.6 Security of the Hardened QKD Solution

With our program analysis, the effectiveness of our mitigations with respect to cache-side-channel security can be verified. We also discuss how our quantitative cache-side-channel security guarantees for the software component of QKD relate to the overall security of QKD solutions.

5.6.1 Guarantees for the Hardened QKD Software

With our program analysis, we compute leakage bounds for the hardened encoding component, as well as for the two remaining components of our target implementation.

Hardened Encoding For the encoding implementation from Figure 5.1, hardened with the mitigation from Figure 5.5, our analysis returns the bound

$$0\,\text{bit}$$

on the cache-side-channel leakage to attackers under *trace*.

That is, the mitigated target implementation is secure against cache-side-channel attackers under *trace*. Our mitigation successfully removes the vulnerability described in Section 5.5.

Our hardened target implementation and a version of the original encoding implementation from [100] that is hardened based on our proposed mitigation would only differ in the scope of variables and the use of dynamically allocated pointer structures. Hence, the leakage bound of 0 bit transfers to the original encoding implementation if hardened as described in Section 5.5.2.

Decoding Our program analysis also returns the *trace* leakage bound

$$0\,\text{bit}$$

for the decoding implementation from Figure 5.2. In the decoding case, however, transferring the leakage bound to the original implementation from [100] is a bit more tricky.

Recall from Section 5.2 that the use of fixed-size arrays and a fixed parity matrix does not influence the leakage through cache traces. It remains to investigate the influence of performing only one iteration of `iterprp` and of our modifications to the implementation of overflow handling.

If the function `iterprp` is executed more than once, leakage might accumulate across the runs of the function. However, our leakage bound shows that there is zero leakage in a function run. Hence, no accumulation is possible. The leakage is not influenced by the reduced iterations.

The overflow handling in the original implementation from [100] checks whether the intermediate probability values `pr` and `H_re[e][j]` have the value N.a.N. (not a number) to detect overflows. In case an overflow occurs, the probabilities are reset to 50% (represented by the value

1). In case no overflow occurs, this step is skipped. Hence, the cache traces of executions of [100] reveal where overflows occurred. Our simplified overflow handling hides the information where overflows occurred from the cache traces by executing an overflow handling on dummy variables in case no overflow occurs. This is inspired by the program transformation unification [76].

The occurrence of an overflow does not reveal any secrets, because the probability that has overflowed is reset to 50%. Hence, potential leakage resulting from a dependence of cache traces on overflows would be a false positive. Removing such dependencies does not hide actual leakage.

Overall, our simplifications do not alter the leakage to *trace* compared to [100]. The leakage bound of 0 bit, hence, also holds for the original implementation.

Privacy Amplification Finally, our program analysis also yields the *trace* leakage bound

$$0\,\text{bit}$$

for our target implementation of privacy-amplification from Figure 5.3.

Recall from Section 5.2 that the simplifications we made compared to the original privacy-amplification implementation from [100] (fixed size and changed scope of variables, explicit marking of the Toeplitz matrix as uninitialized) do not influence the possible cache traces of the implementation. Hence, the leakage bound transfers to the original implementation.

Overall, we now have secure implementations of the QKD steps that compute on secret information: error correction (encoding and decoding) and privacy amplification. The security guarantees for these implementations are with respect to cache-side-channel attackers under *trace*. We broaden the scope to overall QKD solutions and to the other attacker models in the following.

5.6.2 Lifting of the Guarantees to the QKD Solution

In this section, we lift our security guarantees for the hardened software to an overall QKD solution that uses the hardened QKD software.

Recall from Section 2.3.3 that the conventional security guarantees for QKD solutions depend on the length l of the privacy-amplified-key blocks. The shorter l is compared to the attacker's remaining uncertainty $k - r - m$ (for r-bit leakage during the raw-key exchange and m parity bits used during error-correction) about the k-bit error-corrected key blocks, the more secure is the privacy-amplified key against conventional QKD attackers.

If an attacker can mount cache-side-channel attacks as modeled by *trace* in addition to the conventional attacks, his uncertainty might decrease further compared to $k - r - m$. The amount of bits that he learns, i.e., the additional decrease in his uncertainty is bounded by the leakage bound $\log_2(|Obs^{\overline{\mathcal{D}}_{64}}|)$ that our program analysis computes. In the worst case, the bits that the attacker learns via the cache side channel are distinct from the bits that he learns through conventional attacks, so that the leakage adds up. That is, in the worst case, the attacker's uncertainty about the secret key decreases to only $k - r - m - \log_2(|Obs^{\overline{\mathcal{D}}_{64}}|)$.

Privacy amplification, which is usually used to counter the leakage to conventional QKD attackers, can in principle also be used to counter additional cache-side-channel leakage. To this end, the target length l of the privacy-amplified key would have to be reduced further such that

$$l < k - r - m - \log_2(|Obs^{\overline{\mathcal{D}}_{64}}|).$$

Based on the above inequation, cache side channel leakage can be taken into account in the navigation of the security-performance trade-off with privacy amplification. The smaller l is chosen compared to the right-hand side of the inequality, the higher the security of the privacy amplified key (see Section 2.3.3). At the same time, the smaller l is chosen, the more particles have to be transmitted in the raw-key exchange to arrive at the same target key length. Possible bit-rates for the raw-key exchange range from 10^{-3} to 13 Mbit/s, depending on the setup [134].

The concrete security guarantees that we obtained for the hardened QKD software are $\log_2(|Obs^{\overline{\mathcal{D}}_{64}}|) = 0$ for each of the critical computations (encoding, decoding, privacy amplification). Hence, the target length $l < k - r - m - 0$ of the traditional privacy-amplification is sufficient to achieve a privacy-amplified key that is secure against an attacker who combines both, the capabilities of the conventional attacker model and the cache-side-channel attacker model *trace*. No additional strengthening of the privacy amplification is needed.

Note that, a complete leakage of the privacy-amplified key (e.g., through the vulnerability described in Section 5.5) cannot be compensated by privacy amplification alone. If $\log_2(|Obs^{\overline{\mathcal{D}}_{64}}|) = k$, the target length of the privacy amplification would have to be negative: $l < -r - m$, which is impossible. This is why a hardening on the implementation level as, e.g., in Section 5.5.2 is needed in addition to the traditional privacy amplification.

5.6.3 Other Attacker Models and Cache Configurations

The model of cache-side-channel attackers that we considered in the previous sections focuses on attackers who obtain cache traces as, e.g., in [1]. In this section, we extend our results to the other attacker models considered in this thesis: *acc*, *accd*, and *time*. Furthermore, we discuss the influence of the cache configuration on our security guarantees.

Vulnerability of the Encoding Function For the additional attacker models, we obtain the following leakage bounds on the implementation of encoding from Figure 5.1 with our analysis:

acc: 0 bit, *accd*: 0 bit, *time*: 2.4 bit.

These bounds suggest that there is also a vulnerability with respect to attackers under *time*.

Consider again the excerpt from the original implementation of the encoding step in Figure 5.4. As described in Section 5.5, the multiplication function `mod2dense_multiply` iterates over the secret sifted key and accesses the generator matrix only for the sifted-key bits that are set.

An attacker under the model *time* can observe the sum of durations for the cache accesses performed during the program execution. The attacker might use this information to compute the amount of cache accesses and, hence, deduce the amount of set bits in the sifted-key block. Thus, he might learn the Hamming weight of the sifted key.

The leakage of the sifted key's Hamming weight is reflected in the leakage bounds as follows: The Hamming weight of an k-bit block corresponds to $\log_2(k+1)$ bits of information. For the 4 bit sifted-key block in the analyzed implementation, these are $\log_2(5) \approx 2.4$ bit.

This leakage could in principle be compensated by more expensive privacy amplification. Given a leakage of $log_2(k+1)$ bits to cache-side-channel attackers under *time*, the privacy amplification would have to compress the error-corrected key by additional $log_2(k+1)$ bits.

Leakage Bounds for the Hardened Implementations When we analyze the hardened encoding function from Section 5.5.2 with respect to the additional attacker models, we obtain the following leakage bounds:

acc: 0 bit, *accd*: 0 bit, *time*: 0 bit.

That is, our mitigation from Section 5.5.2 is also effective against the leakage to *time*. With the hardened implementation, a strengthening of the privacy amplification and the resulting need for transmitting additional qbits can be avoided.

For the implementations of both, decoding and privacy amplification from Section 5.2, we obtain the following leakage bounds:

acc: 0 bit, *accd*: 0 bit, *time*: 0 bit.

Overall, we now have zero-leakage guarantees for our hardened software against the cache-side-channel attackers modeled by *trace*, *acc*, *accd*, and *time*.

Influence of the Cache Configuration As described in Section 5.4, we focused on one exemplary cache configuration to be able to provide concrete security guarantees. To assess the consequences of focusing on one configuration, we investigated the influence of the cache size, associativity, and replacement strategy on our security guarantees exemplarily.

For the cache size, we repeated our analysis of the hardened implementations for a smaller cache size of 8 KiB and a larger cache size of 64 KiB. Both lead to leakage bounds of 0 bit for all four attacker models: *trace*, *acc*, *accd*, and *time*. We also repeated our analyses with a smaller associativity of 4 and a larger associativity of 12. Again, we obtained 0 bit bounds across all attacker models. Finally, we repeated our analyses with alternative replacement strategies, namely Pseudo-LRU (PLRU) and FIFO. For these strategies, we also obtained 0 bit leakage bounds.

That the leakage bounds were stable under exemplary variation of the parameters of the cache configuration gives us confidence that our security guarantees are not limited to caches of the exact configuration that we considered, but are also relevant for a broader set of caches.

5.7 Summary

In this chapter, we investigated the security of Quantum Key Distribution against cache side channels at the example of the software part of the implementation of the BB84 protocol from [100]. Since QKD implementations need floating-point instructions and existing abstract domains were not suitable for the quantification of cache side channels in implementations with floating-point instructions, we developed the new abstract domain $\overline{\mathcal{D}}_{64}$, which tracks the interaction between the components related to the regular ALU, the FPU, and the cache at a suitable level of precision. Our analysis tool CacheAudit-FPU automates the program analysis based on $\overline{\mathcal{D}}_{64}$ and can quantify the cache-side-channel leakage of x86/87 binaries with floating-point instructions.

With our analysis we detected a vulnerability in the optimized matrix multiplication performed by the function `mod2dense_multiply` during the QKD encoding step in [100]. This vulnerability might reveal an entire sifted-key block and, hence, also the final, privacy-amplified secret key. The vulnerability not only affects the QKD implementation from [100], but also the encoding implementation from [97] that has been forked by many others. We hardened the vulnerable function and derived security guarantees for the hardened QKD software. We also lifted these guarantees to an overall hardened QKD solution. Our mitigations were successfully deployed by the physicists who maintain the QKD software from [100]. The new software version that is used for QKD setups at the Department of Physics at TU Darmstadt is, hence, already hardened against the detected vulnerability.

Chapter 6

Cache-Side-Channel Quantification on Platforms with an Instruction Pipeline

6.1 Introduction

In the previous chapters we focused on quantifying the leakage of implementations through cache side channels. In this chapter we broaden the scope to address side channels that arise from the combination of caching and pipelining. As explained in Section 2.1.3, even programs whose sequential in-order executions are secure against cache side channels can have severe cache-side-channel leakage when executed on a platform with instruction pipelining. Such leakage can be exploited in access-based attacks like, e.g., the prominent Spectre-PHT attack [72].

Instruction pipelining is very common in modern computer architectures and indispensable for performance reasons. While side channels that arise from the combination of caching and pipelining could, in theory, be mitigated by disabling speculative execution throughout a program execution (e.g., using so-called speculation barriers), this is undesirable in practice due to the resulting performance overhead. Luckily, there are possibilities for partial mitigations that have less impact on performance. For instance, the Intel C++ compiler [64, p.160] provides options for mitigating leakage on platforms with pipelining at different levels of completeness.

So far, there are no program analyses that can provide quantitative security guarantees, e.g., for partially mitigated programs, with respect to side channels based on the combination of caching and pipelining. Existing quantitative analyses, including [45, 44, 30, 86, 11] and our analyses from Chapter 3-5, do not take into account instruction pipelining. Existing analyses that do take pipelining into account, including [28, 13, 126, 57, 58, 27], are not quantitative.

In this chapter, we propose Spectrescope, the first quantitative program analysis that provides upper bounds on the leakage through side channels that arise from the combination of caching and pipelining. At the heart of our program analysis is a novel abstract domain that faithfully captures an instruction pipeline with branch prediction and out-of-order execution. The abstract domain is fine-grained enough to enable the computation of precise leakage bounds and, at the same time, coarse-grained enough for the computation to remain feasible.

The program analysis takes a program in a simple assembly-level language *p*ASM as input. It computes the reachable observations under the attacker model *acc* with respect to our abstract domain. It computes an upper leakage bound as the logarithm of the number of possible attacker observations. The abstract reachability analysis is based on an abstract semantics that faithfully captures pipelined out-of-order executions with implicit register renaming, a static

branch-prediction strategy, and a fully-associative data cache with FIFO replacement strategy.

With Spectrescope, we successfully quantify the leakage through a known vulnerability in a function from the Linux kernel (version 4.16.8). Furthermore, we verify the effectiveness of an existing mitigation, which is deployed in the Linux kernel starting from version 4.16.9. The mitigation that is deployed in the Linux kernel removes the leakage completely and Spectrescope verifies this successfully. However, Spectrescope is not limited to verifying complete mitigations.

Based on multiple small example programs, we demonstrate how Spectrescope can be used to support quantitative reasoning about partial mitigations and to compare multiple implementations with respect to their security. The leakage bounds in our evaluation range from 4 bit to 32 bit. For instance, we use our analysis successfully to verify that a partial mitigation based on a bit-mask reduces the leakage of an example program by 87.5% at relatively low performance cost.

In Section 6.2, we give an overview of the basic execution model underlying our analysis, including the corresponding assembly language *p*ASM, and describe our target implementations of the Linux-kernel excerpts in *p*ASM. In Section 6.3, we describe our pipeline abstract domain, the corresponding abstract semantics, and the overall program analysis Spectrescope. We define our analysis setup in Section 6.4 and discuss our analysis of the Linux-kernel excerpts in Section 6.5. In Section 6.6, we present examples for reasoning about partial mitigations and for comparing the security across different programs based on leakage bounds computed with Spectrescope.

6.2 Target Implementations

We analyze two excerpts from the Linux kernel with respect to their cache-side-channel leakage on an architecture with both, a cache and an instruction pipeline. Our analysis is based on a formal model of the target architecture, which is described in full detail in Appendix B. Our target implementations for the Linux-kernel excerpts are programs in the assembly language *p*ASM that is executed by our target architecture.

6.2.1 Overview of Target Architecture

For our target architecture, we specify both, an Instruction Set Architecture (ISA) and a corresponding Micro-Architecture (MA). The former specifies the instructions of the assembly language *p*ASM and the data types on which they operate. The latter specifies how the instructions are executed by the CPU.

Instruction Set Architecture

The data types specified in real-world ISAs differ, but usually include a set of general-purpose registers, immediate values, and memory. The latter can be organized Harvard-style (separate memories for data and instructions) or von-Neumann-style (joint memory for data and instructions). For instance, the Intel IA-32 ISA [63] features eight general-purpose registers and the RISC-V ISA [131] features 31 general-purpose registers. The Intel IA-32 ISA features von-Neumann-style memory and the Atmel AVR ISA [10] features Harvard-style memories.

In our formal model, the set of all immediate values is specified by $Vs = \{n \in \mathbb{N}_0 \mid n \leq 2^{32}-1\}$, i.e., it is the set of all 32-bit values including zero. The set of general-purpose registers is $Rs = \{e_n \mid n \in \mathbb{N} \wedge n \leq NUM\}$, i.e., the number of registers is parametric and can be specified by instantiating the constant NUM. The set of all data-memory addresses is $DAs = \{d_n \mid n \in \mathbb{N}\}$ and the set of all instruction-memory addresses is $IAs = \{ia_n \mid n \in \mathbb{N}\}$.

Overall, our model captures a 32-bit architecture with NUM general-purpose registers and Harvard-style memory organization. Both, the data and instruction memory are infinite in our model. We assume that out-of-memory conditions are treated in a different analysis step, outside of our side-channel analysis. To make our examples more readable, we use the symbols eax, ebx, ecx, and edx as synonyms for e_1, e_2, e_3, and e_4 throughout this chapter.

The range of instructions defined in real-world ISAs varies. Different instruction sets can be roughly categorized as either RISC-style or CISC-style. RISC-style instruction sets contain a reduced number of instructions, where the data memory is usually only accessible via separate load and store instructions. CISC-style instruction sets contain more complex instructions, which might also operate on the data memory directly. For both types of instruction sets, there are multiple possible modes in which memory entries can be addressed. Examples include direct addressing, where an immediate value is used to specify a memory address, and base-plus-offset addressing. In base-plus-offset addressing, the instruction operands include a register and an immediate value, which are added to obtain the target address for the memory access.

The instruction set in our model is the set $Insts$, such that

$$\begin{aligned} Insts \quad = \quad & \textbf{add-rc}\ e\ v \mid \textbf{sub-rc}\ e\ v \mid \textbf{shr-rc}\ e\ v \mid \textbf{sar-rc}\ e\ v \mid \\ & \textbf{add-rr}\ e\ e' \mid \textbf{sub-rr}\ e\ e' \mid \textbf{and-rr}\ e\ e' \mid \textbf{or-rr}\ e\ e' \mid \textbf{neg}\ e \mid \\ & \textbf{mov-rc}\ e\ v \mid \textbf{mov-rm}\ e\ d\ e' \mid \\ & \textbf{jge}\ ia\ e\ e' \mid \textbf{fence} \mid \textbf{nop} \end{aligned}$$

with $e, e' \in Rs$, $v \in Vs$, $ia \in IAs$, and $d \in DAs$. We write $opc(in)$ to extract the mnemonic from an instruction in, $dreg(in)$ to extract the destination register (the first register that occurs in the instruction) from an instruction that writes to a register, and $sregs(in)$ to extract the sequence of source registers (the sequence of registers that occur in the instruction, except for the first register in case of a move instruction) from an instruction.

The instructions **add-rc**, **sub-rc**, **shr-rc**, and **sar-rc** operate on the value of a register and on an immediate value. They write the result back to the register. The two shift instructions **shr-rc** and **sar-rc** shift the value of the register to the right by the offset specified by the immediate value. The former performs a logical shift (shifting in zeroes) and the latter performs an arithmetic shift (shifting in copies of the value's sign bit).

The instructions **add-rr**, **sub-rr**, **and-rr**, and **or-rr** perform binary arithmetic or bit-wise logical operations on the values of two registers, specified by the two operands of the instructions. The register specified as the first operand is also the destination to which the result is written. The instruction **neg** performs a bit-wise negation of the value of its operand register and writes the result back to the same operand register.

The instruction **mov-rc** moves an immediate value into a register. The instruction **mov-rm** retrieves a value from the data memory. The target address is computed by adding the immediate value provided as the second operand to the value of the register provided as the third operand. The value from the memory is written into the register specified by the first operand.

The instruction **jge** $ia\ e\ e'$ is a conditional jump instruction. If the value of the register e is greater than or equal to the value of the register e', **jge** triggers a jump to the instruction at the instruction-memory address ia. Otherwise, the execution proceeds regularly. The instruction **fence** is a speculation barrier. It cannot be executed out-of-order and prevents the speculative fetching of instructions that occur after the **fence** in program order (i.e., in the order in which instructions would be executed on a sequential in-order CPU). Finally, **nop** skips one clock cycle.

Overall, $Insts$ is a RISC-style instruction set that uses base-plus-offset addressing for the data memory and direct addressing for the instruction memory. The instruction set is a simplification of real-world instruction sets. Notably, there are no instructions in $Insts$ that write to memory. For our program analysis, this simplification is reasonable, because our target architecture is a single-processor system with a cache that implements the no-write-allocate policy. On such a system, new entries are only added to the cache when a read instruction encounters a cache miss. Write instructions might affect the values stored in the cache but not the addresses of the memory blocks in the cache. Since cache side channels arise from differences in the addresses of the cached blocks only, write instructions do not affect the cache-side-channel leakage.

In the assembly language of our execution model, which is called pASM, the set of all programs is $Ps = IAs \rightharpoonup Insts$, i.e., the set of all partial mappings from instruction addresses

to instructions. Throughout this chapter, we consider only a subset of Ps, namely the set $Ps_{wf} = \{pr \in Ps \mid wf(pr)\}$ of well-formed programs, where

$$\begin{aligned} wf(pr) = {} & ia_1 \in dom(pr) \wedge \\ & \forall n \in \mathbb{N}.\ ((ia_n \in dom(pr) \wedge n > 1) \Rightarrow ia_{n-1} \in dom(pr)) \wedge \\ & \forall ia \in IAs.\ \forall e, e' \in Rs.\ (\textbf{jge}\ ia\ e\ e' \in rng(pr) \Rightarrow ia \in dom(pr)) \wedge \\ & \forall n \in \mathbb{N}.\ ((ia_n \in dom(pr) \wedge ia_{n+1} \notin dom(pr)) \Rightarrow opc(ia_n) \neq \textbf{jge}) \end{aligned}$$

That is, we only consider programs that (1) begin at the instruction address ia_1, (2) are stored at consecutive instruction addresses, (3) contain only jump instructions whose targets are within the program, and (4) do not end with a loop (i.e., the last instruction is not a conditional jump instruction). If the last condition is not satisfied, the program can be transformed by adding a **nop** instruction in the end, such that the transformed program satisfies the condition.

Micro-Architecture

In addition to the architectural components specified above (registers, data memory, and instruction memory), our execution model captures multiple micro-architectural components. More concretely, it captures a regular ALU and components that implement caching and pipelining.

Caching In real-world MAs, caching is usually implemented by a hierarchy of multiple caches. The caches at each level can be Harvard-style (separate caches for data and instructions) or von-Neumann-style (a unified cache for data and instructions). The BOOM RISC-V MA, e.g., contains separate L1 caches and a unified L2 cache [29]. Caches can differ with respect to their overall size, line size, associativity, and replacement policy. The FIFO replacement policy is supported, e.g., by most Cortex-A CPUs that implement the ARMv7-A ISA and the LRU replacement policy is supported, e.g., by the Cortex-A15 CPU [8].

In our execution model, caching is implemented by one fully-associative L1 data cache with 32-bit line size and FIFO replacement. The number of cache lines is specified by the constant *SIZE*. That is, each memory block contains exactly one 32-bit memory entry and the model is parametric in the overall cache size.

Pipelining Pipelining with out-of-order execution is usually implemented by a set of multiple micro-architectural components. The core of an instruction pipeline are the actual pipeline stages. There are MAs with comparatively few pipeline stages like, e.g., the MIPS32 architecture with four pipeline stages [90, p.122], and MAs with longer pipelines like, e.g., the Intel Core architecture with a 14-stage pipeline [62, p.2-30]. Our execution model captures a pipeline with four stages.

Real-world pipeline implementations often make use of different prediction mechanisms to avoid pipeline stalls. For instance, branch prediction avoids pipeline stalls due to conditional branches, jump-target prediction avoids pipeline stalls due to indirect branches, and return-address prediction avoids pipeline stalls due to function returns. All three prediction mechanisms are, e.g., supported in the BOOM architecture. There are different strategies for implementing each prediction mechanism. Many strategies are based on historical information, e.g., about previous branching decisions. For branch prediction, there are also static strategies like, e.g., the Null Predictor that is supported by the BOOM architecture and that statically predicts for each branch that it will not be taken. Our execution model captures branch prediction with respect to a static always-not-taken prediction strategy, i.e., without a separate branch-prediction component.

In addition to prediction mechanisms, modern pipeline implementations often feature out-of-order execution. That is, they execute instructions in a different order than program order to avoid pipeline stalls when operands have to be fetched from the memory. The implementations of out-of-order execution differ across CPUs. For instance, name dependencies between registers

can be resolved by explicit register renaming or by implicit register renaming. The former is, e.g., supported by the BOOM architecture and is based on a set of hardware registers that is larger than the set of registers specified in the ISA. The latter is, e.g., supported by the Intel Core architecture and keeps track of in-flight instructions using so-called reservation stations. Each instruction is processed in one of the reservation stations, which stores either the operand values (if they are available) or pointers to the instructions that will produce these values (if they are not yet available) [125]. When all operand values for an instruction are available, it can be executed. The order in which instructions are scheduled for execution can differ across MAs. Our execution model captures implicit register renaming based on reservation stations and a scheduler that always schedules the first ready instruction in program order for execution.

Many MAs that support out-of-order execution preserve sequential consistency, i.e., the changes to the registers become visible at higher levels of abstraction in program order. To this end, register updates are buffered and committed in program order. This can happen either synchronously in a pipeline stage or asynchronously. Our execution model captures out-of-order execution with a reorder buffer for register updates and synchronous in-order commits.

6.2.2 Target Implementations of Linux-Kernel Excerpts

In 2018, multiple patches for the Linux kernel were developed to close possible vulnerabilities to side channels that arise from caching and pipelining. One of these vulnerabilities was located in Line 861-875 of the file /kernel/events/ring_buffer.c in the kernel version 4.16.8 [138]. The corresponding patch was deployed starting from version 4.16.9. For our analysis, we implement an excerpt from the kernel version 4.16.8 and an excerpt from the kernel version 4.16.9 in *p*ASM.

Excerpt from v. 4.16.8 The original code from kernel version 4.16.8 is shown in Figure 6.1. The function `perf_nmap_to_page` operates on an untrusted parameter `pgoff`, which is controlled from user space [138]. It uses the value `pgoff - rb->aux_pgoff` as an index to access the array `rb->aux_pages` in Line 11. The guards in Line 6 and 10 ensure that this index is within bounds of the array. On an architecture with branch prediction and out-of-order execution, there are possible executions in which the value of `pgoff` would lead to a memory access that is outside the bounds of `rb->aux_pages` and in which this out-of-bounds access is executed speculatively because the guard in Line 10 is predicted to evaluate to `true`. That is, there are possible executions that retrieve a value from the private kernel memory instead of the array `rb->aux_pages`. This value is processed by the function `virt_to_page` and then returned. If the returned value is used as an index for another array access, information is leaked into the cache. How much information is leaked depends on how the secret memory entry is processed by the function `virt_to_page`.

```
1  struct page *
2  perf_mmap_to_page(struct ring_buffer *rb, unsigned long pgoff)
3  {
4    if (rb->aux_nr_pages) {
5      /* above AUX space */
6      if (pgoff > rb->aux_pgoff + rb->aux_nr_pages)
7        return NULL;
8
9      /* AUX space */
10     if (pgoff >= rb->aux_pgoff)
11       return virt_to_page(rb->aux_pages[pgoff - rb->aux_pgoff]);
12   }
13   return __perf_mmap_to_page(rb, pgoff);
14 }
```

Figure 6.1: Excerpt from Linux Kernel v. 4.16.8

How the function `virt_to_page` processes the secret value depends on the target architecture. For instance, the implementation of `virt_to_page` might be of the form

```
(mem_map + ((__phys_addr((unsigned long) (x)) >> PAGE_SHIFT)−ARCH_PFN_OFFSET)),
```

where `__phys_addr` contains further additions and subtractions.

Figure 6.2 shows our target implementation for the kernel excerpt. We call this target implementation `p-kernel`. It captures the untrusted value `pgoff - rb->aux_pgoff` by the parameter register *eax* that we will consider attacker-controlled in our analysis. It captures the array `aux_pages` by an array `a1` that is stored in the data memory, starting from the data address d_2. The size of `a1` is stored at the data address d_1. The program `p-kernel` retrieves the size of `a1` from the data memory (Line 5) and checks whether the value of *eax* is within bounds of `a1` (Line 6). If the value of *eax* is within bounds, the program accesses `a1` at this index (Line 7). We call the value that is retrieved by this memory access `x`. To capture the pattern used to process the value `x`, `p-kernel` computes `const_1 + (x >>> const_2)- const_3` (Line 8-9), where `const_1` = 4 (Line 3), `const_2` = 12 (Line 8), `const_3` = 123460 (Line 2). To capture a potential leakage of the processed value, the program accesses an additional array `a2` at an index corresponding to the processed value (Line 10).

```
1  mov-rc  edx 0
2  add-rc  edx 123460
3  sub-rc  edx 4          // edx = 0 + 123460 - 4
4  mov-rc  ebx 0          // ebx = 0
5  mov-rm  ebx d1  ebx    // ebx = size(a1)
6  jge i11 eax ebx        // jump if eax >= ebx
7  mov-rm  eax d2  eax    // eax = a1[eax] (value x)
8  shr-rc  eax 12         // eax = eax >>> 12
9  sub-rr  eax edx        // eax = eax - edx
10 mov-rm  eax d6 eax     // eax = a2[eax]
11 nop
```

Figure 6.2: Program `p-kernel`

Excerpt from v 4.16.9 Figure 6.3 shows excerpts from the Linux kernel version 4.16.9. The index used to access the array `rb->aux_pages` in Line 32 is sanitized using the function `array_index_nospec`. We capture this mitigation in our target program `p-kernel`$^{\mathtt{f}}$ (Figure 6.4).

The mitigation is captured in Line 7-12. The rest of `p-kernel`$^{\mathtt{f}}$ is identical to `p-kernel`. In Line 7, `p-kernel`$^{\mathtt{f}}$ computes the maximum value that a valid index of an access to `a2` may have (i.e., it subtracts one from the size of `a1`). Subsequently, the program subtracts the value of *eax* (i.e., the attacker-controlled index) from this maximum value (Line 8) and computes the bit-wise OR between the resulting difference and the original index in *eax* (Line 9). If the index is out-of-bounds with respect to `a1`, then the result of the bit-wise OR will have the sign bit one. If the index is within bounds, the result will have the sign bit zero. The reason is that if the index is out-of-bounds, then either the index itself or its difference from the maximum index will be negative (i.e., have sign bit one, which is preserved by the bit-wise OR). The program negates the resulting value in Line 10, so that the sign bit is one exactly in the cases where the index is within bounds and zero exactly in the cases where the index is out-of-bounds. The sign bit is copied into the remaining 32 bit of the register by the arithmetic shift in Line 11. The resulting value is applied as a mask to the index in Line 12. That is, the index is masked out exactly in those cases where it is out-of-bounds. This is exactly the mitigation technique applied in the Linux kernel patch (see Line 7 of Figure 6.3).

In addition to masking out illegal indices, this mitigation introduces a dependence between the index of the access to `a1` and the computation of the mask. This ensures that the access cannot be executed out-of-order before the index has been sanitized.

```
1  //from /include/linux/nospec.h:
2
3  [...]
4  static inline unsigned long array_index_mask_nospec(unsigned long index,
       unsigned long size)
5  {
6    [...]
7    return ~(long)(index | (size - 1UL - index)) >> (BITS_PER_LONG - 1);
8  }
9  [...]
10 #define array_index_nospec(index, size)                                \
11 ({                                                                     \
12   typeof(index) _i = (index);                                         \
13   typeof(size) _s = (size);                                           \
14   unsigned long _mask = array_index_mask_nospec(_i, _s);              \
15   [...]                                                               \
16   (typeof(_i)) (_i & _mask);                                          \
17 })
18
19
20 //from /kernel/events/ring_buffer.c:
21
22 struct page *
23 perf_mmap_to_page(struct ring_buffer *rb, unsigned long pgoff)
24 {
25   if (rb->aux_nr_pages) {
26     /* above AUX space */
27     if (pgoff > rb->aux_pgoff + rb->aux_nr_pages)
28       return NULL;
29
30     /* AUX space */
31     if (pgoff >= rb->aux_pgoff) {
32       int aux_pgoff = array_index_nospec(pgoff - rb->aux_pgoff, rb->
              aux_nr_pages);
33       return virt_to_page(rb->aux_pages[aux_pgoff]);
34     }
35   }
36   return __perf_mmap_to_page(rb, pgoff);
37 }
```

Figure 6.3: Excerpt from Linux Kernel v. 4.16.9

```
1  mov-rc  edx 0
2  add-rc  edx 123460
3  sub-rc  edx 4           // edx = 0 + 123460 - 4
4  mov-rc  ebx 0           // ebx = 0
5  mov-rm  ebx d1  ebx     // ebx = size(a1)
6  jge i17 eax ebx         //jump if eax >= ebx
7  sub-rc  ebx 1                   // ebx = ebx -1 (maximum index in a1)
8  sub-rr  ebx eax                 // ebx = ebx - eax (maximum index - value x)
9  or-rr   ebx eax                 // ebx = ebx | eax
10 neg     ebx                     // ebx = ~ebx
11 sar-rc  ebx 31                  // ebx = ebx >> 31 (arithmetic shift)
12 and-rr  eax ebx                 // eax = eax & ebx (apply mask)
13 mov-rm  eax d2  eax     // eax = a1[eax]
14 shr-rc  eax 12          // eax = eax >>> 12
15 sub-rr  eax edx         // eax = eax - edx
16 mov-rm  eax d6 eax      // eax = a2[eax]
17 nop
```

Figure 6.4: Program `p-kernel`[f]

6.3 Program Analysis for Caching and Pipelining

Our program analysis is based on abstract interpretation with respect to our novel pipeline abstract domain and a corresponding abstract semantics. The concrete domain and concrete semantics from which the analysis abstracts are described in detail in Appendix B.

6.3.1 Abstract Domain

Our pipeline abstract domain captures the possible snapshots of a system with a cache and an instruction pipeline during a program execution at an abstract level. More concretely, it abstracts from the snapshots across all possible program executions in two dimensions. First, it abstracts from the possible values of the data that are processed during program executions. Second, it abstracts from the possible control flows that are taken during program executions.

Abstraction across Data Flows In the abstract domain, the possible snapshots of the instruction pipeline are represented by triples, which capture the possible snapshots of (1) the pipeline stages, (2) the reservation stations, and (3) the reorder buffer. More concretely, the set of all abstract pipeline configurations is the set $\overline{PCos} = StCos \times \overline{RSCos} \times \overline{BCos}$, where $StCos$ is the set of pipeline-stage configurations, $\overline{RSCos}$ is the set of abstract reservation-station configurations, and $\overline{BCos}$ is the set of abstract reorder-buffer configurations.

The set of abstract pipeline-stage configurations is the set $StCos = Sts \to StCs$. Each abstract pipeline-stage configuration maps the pipeline stages from the set $Sts = \{fet, dis, exe, com\}$ to elements of the set $StCs = UIAs \cup \{\bot, \top\}$. Here, the symbol fet captures the first pipeline stage, namely the fetch stage, dis captures the dispatch stage, exe captures the execute stage, and com captures the commit stage. The set $UIAs = IAs \times \mathbb{N}$ is the set of unique identifiers for instructions during a program execution. The pair $(ia, n) \in UIAs$ refers to the n-th execution of the instruction stored at instruction address ia. For loop-free programs, in which each instruction is executed at most once, we abbreviate $(ia, 1)$ by the short-hand ia. In addition to the unique instruction identifiers $UIAs$, the set $StCs$ contains the symbol $\bot$, which captures that a pipeline stage is stalled due to a pending barrier instruction, and the symbol $\top$, which captures that a pipeline stage is stalled because no instruction is ready to be processed (e.g., because each instruction has been processed already, because of unresolved register dependencies, or because of pending memory accesses). Overall, each element of $StCos$ captures a snapshot of the four pipeline stages by mapping each stage that processes an instruction to the identifier of this instruction and each stage that is stalled to a symbol that captures the reason for the stalling.

The set of abstract reservation-station configurations is the set $\overline{RSCos} = \overline{RSCs}^{\star}$. Each abstract reservation-station configuration is a list with elements from the set $\overline{RSCs} = UIAs \times \overline{MLs} \times RLs$. Each element $(ua, \overline{ml}, rl)$ in the list captures the information about one in-flight instruction ua that is stored in the reservation stations. We call $\overline{ml}$ the abstract memory-access list and rl the register dependence list of ua. Both lists are described in more detail below.

Each abstract memory-access list is an element of the set $\overline{MLs} = (\mathcal{P}(DAs) \times \mathbb{N}_0)^{\star}$, i.e., a list of pairs. Each pair (D, t) in the list models one memory access performed by the instruction ua. More concretely, D is the set of possible data-memory addresses that the memory access might target and t is the number of clock cycles that are still required to complete the memory access. The total number of clock cycles that is required to complete a memory access depends on whether the target address is available in the cache. The constant HIT captures the total number of clock cycles that are required to retrieve a value from a target address that is available in the cache. The constant $MISS$ captures the total number of clock cycles that are required to retrieve a value directly from the data memory in case of a cache miss.

Each register-dependence list is an element of the set $RLs = (Rs \times UIAs)^{\star}$, i.e., also a list of pairs. In this case, each pair (e, ua') captures the dependence of one register operand e of the instruction ua. More concretely, the instruction-address ua' is the address of the instruction

that produces the value of e that is read by ua. That is, the register-dependence list captures the pointers between reservation stations that are used to keep track of name dependencies in a pipelined architecture with implicit register renaming.

The set of abstract reorder-buffer configurations is the set $\overline{BCos} = \overline{BCs}^{\star}$. Like an abstract reservation-station configuration, each abstract reorder-buffer configuration is a list whose entries correspond to in-flight instructions. Each entry is an element of the set $\overline{BCs} = UIAs \times Rs \times \mathcal{P}(Vs)$, i.e., a triple. More concretely, each triple (ua, e, V) captures the register update that is triggered by the instruction ua. The register e is the destination register of the instruction ua and V is the set of all possible values that might be written to e by ua. That is, an abstract reorder-buffer configuration captures the register updates that are stored in a buffer before being written to the actual registers in a pipelined architecture with synchronous in-order commit.

All in all, each abstract pipeline configuration represents the possible in-flight instructions in the same clock cycle across executions with the same control flow. The instructions that are processed in each pipeline stage and reservation station are represented by concrete instruction identifiers, because they do not differ across executions with the same control flow. The data that is processed by the pipeline might differ across executions with the same control flow and is, hence, represented on an abstract level. The possible target addresses of each memory access and the possible values of each register update are represented by sets.

Note that, the remaining clock cycles for each memory access are represented by a concrete number in an abstract pipeline configuration. Cases in which the number of clock cycles differs across different target addresses are captured in the abstraction across control flows.

Example 6.3.1. Recall the Spectre-PHT code snippet from [72], which we described in Section 2.1.3. We consider a variant of this snippet, namely the program `c-simp`, shown in Figure 6.5.

```
1  if (x < array1_size)
2    y = array2[array1[x]];
```

Figure 6.5: Program `c-simp`

The program `c-simp` differs from the original Spectre-PHT code snippet in one aspect. It omits the multiplication by `4096` because the architecture captured by our execution model has 32 bit line size, so that the memory block that is loaded into the cache is already uniquely determined by the entry of `array2` that is accessed.

Figure 6.6 shows the program `p-simp`, which is a pASM implementation of `c-simp`. Let eax be a parameter register that contains the program input supplied by the attacker. Let `a1` be a public array with three entries that is stored at the data addresses d_2, d_3, and d_4. Let the size of `a1`, i.e., the value 3, be stored at the data address d_1. Let `a2` be another public array that is stored at the data addresses starting from the base address d_6. Finally, let a private memory entry be stored at the data address d_5 and let this be the only address that is cached initially.

```
1  mov-rc ebx 0
2  mov-rm ebx d1 ebx
3  jge i6 eax ebx
4  mov-rm ecx d2 eax
5  mov-rm ecx d6 ecx
6  nop
```

Figure 6.6: Program `p-simp`

The program `p-simp` retrieves the size of `a1` from the data memory in Line 2 and checks whether the value of eax is within bounds of `a1` in Line 3. If the index is within bounds (or

predicted to be within bounds), the program accesses `a1` at this index and stores the result in the register *ecx* (Line 4). It uses the value in *ecx* as the index for an access to `a2` in Line 5.

Consider an execution of `p-simp` in which *eax* has the value 3. When the instruction i_2 is executed, this triggers a memory access at the address d_1. Since the memory entry at address d_1 is not available in the cache, a cache miss occurs and the entry has to be retrieved from the data memory, which takes *MISS* clock cycles. The reservation-station entry for the instruction i_2 is $(i_2, \langle(\{d_1\}, \mathit{MISS})\rangle, \langle(ebx, i_1)\rangle)$. The abstract memory-access list of this entry captures the singleton set of possible target addresses $\{d_1\}$ and the number *MISS* of clock cycles. The register-dependence list captures the dependence of the register *ebx* on the instruction i_1.

While the memory access of instruction i_2 is ongoing, the instruction i_3 cannot be executed, because it depends on the value of *ebx* produced by i_2. Since we consider an architecture with out-of-order execution and static always-not-taken branch prediction, the instructions i_4 and i_5 are executed speculatively in parallel to the ongoing memory access.

When the instruction i_4 is executed, this triggers a memory access at the address $d_{2+3} = d_5$, because *eax* has the value 3. The data address d_5 is available in the cache, such that a cache hit occurs. The memory entry at the data address d_5 is secret and might have any value in the range $[0, 2^{32} - 1]$. This is reflected by the register update $(i_4, ecx, [0, 2^{32} - 1])$ in the reorder buffer.

When the instruction i_5 is executed, this triggers a memory access whose target address depends on the value of the register *ecx*. More concretely, the access might target any data address in the range $[d_6, d_{6+2^{32}-1}]$. Since none of the data addresses in this range is available in the cache, the reservation-station entry for i_5 is $(i_5, \langle([d_6, d_{6+2^{32}-1}], \mathit{MISS})\rangle, \langle(ecx, i_4)\rangle)$. ◊

Abstraction across Control Flows For each possible control flow, our abstract domain captures the possible snapshots of registers, data memory, and cache in one clock cycle.

The possible snapshots of the registers are captured by an abstract register configuration from the set $\overline{RCos} = Rs \to \mathcal{P}(Vs)$. That is, an abstract register configuration maps each register to the set of 32 bit values that might be stored in this register. Analogously, the set of abstract data-memory configurations is $\overline{MCos} = DAs \to \mathcal{P}(Vs)$ and each abstract data-memory configuration maps each data address to the set of 32 bit values that might be stored at this address.

The possible snapshots of the cache are captured by an abstract cache configuration from the set $\overline{CCos} = DAs \to \mathcal{P}([0, \mathit{SIZE}])$. Here, *SIZE* is a constant that captures the number of cache lines in the cache. Each abstract cache configuration maps each data-memory address to the set of cache lines in which it might be stored. The numbers from 0 to $\mathit{SIZE} - 1$ each represent one cache line. The number *SIZE* represents a situation in which the respective data-memory address is not available in the data cache.

Definition 6.1. *The set of abstract configurations (or the abstract domain) is the set*

$$\overline{Cos} = (Ps_{wf} \times \overline{PCos}) \rightharpoonup (\overline{RCos} \times \overline{MCos} \times \overline{CCos}).$$

Overall, our pipeline abstract domain is the set $\overline{Cos}$ of abstract configurations. Each abstract configuration maps pairs of a program and an abstract pipeline configuration to triples of abstract register, data-memory, and cache configurations. That is, an abstract configuration captures the relation between the pipeline configuration for each possible control flow and the corresponding configurations of the other components that can be reached under the respective control flow.

This abstract domain captures execution snapshots at a level of granularity that allows one to compute precise side-channel leakage bounds while, at the same time, the complexity of the leakage-bound computation remains feasible.

In theory, there is a wide spectrum of possible definitions for abstract domains. At the one end of the spectrum are very coarse-grained abstractions, e.g., abstractions that do not track memory accesses explicitly and assume that the instructions of a program might be executed in any order. Leakage bounds computed with respect to such an abstraction would quickly become very imprecise. At the other end of the spectrum are very fine-grained abstractions, e.g.,

abstractions that track sets of concrete configurations. The computation of leakage bounds with respect to such an abstraction would quickly become infeasible due to state-space explosion.

Within this spectrum, a candidate alternative to our abstract domain would be an abstraction that abstracts from the possible instructions processed in the pipeline stages and reservation stations by sets. However, our abstraction outperforms this alternative with respect to both, precision and complexity already for the following small example.

Example 6.3.2. Consider the following *p*ASM program.

```
1  mov–rm ebx d1 ebx
2  jge i9 eax ebx
3  mov–rm ecx d2 ecx
4  mov–rm ecx d3 ecx
5  mov–rm ecx d4 ecx
6  mov–rm ecx d5 ecx
7  mov–rm ecx d6 ecx
8  mov–rm ecx d7 ecx
9  mov–rm ecx d8 ecx
```

Suppose the program processes a secret value from the set $\{0, 1\}$, which is stored at the data address d_1. Suppose that all registers and all remaining memory entries initially contain the value 0 and that the cache is initially empty. For simplicity, assume that the number of clock cycles required to retrieve a memory entry is only $\mathit{MISS} = 2$.

When the instruction i_1 is executed, it triggers a memory access to d_1, which leads to a cache miss. While the memory entry is retrieved from the data memory, the instructions starting from i_3 are executed speculatively out-of-order because the instruction i_2 depends on the result of i_1. Given $\mathit{MISS} = 2$, the actual result of the branching decision in i_2 is available after the speculative execution of i_7 but before the speculative execution of i_8. Hence, $i_3 - i_7$ are executed speculatively and the data addresses $d_2 - d_6$ are loaded into the cache. Whether the instruction i_8 is executed and the data address d_7 is loaded into the cache depends on the value of the secret at d_1. If the value is 1, the execution proceeds with i_8 and i_9. If the value is 0, the speculative execution is rolled back and only i_9 is executed. Since the interactions with the cache are not rolled back, the cache contains the addresses d_2 - d_6 and d_8 after the program terminates in both cases. If the secret value is 1, the cache also contains d_7. If the secret is 0, the cache does not contain d_7. That is, the example program leaks 1 bit of information, namely the value of the secret stored at d_1.

With our abstract domain $\overline{Cos}$, the complexity of analyzing this program remains manageable throughout. For each clock cycle, the abstract configuration is defined for at most two pairs of program and abstract pipeline configuration (one pair for the case in which the conditional jump in i_2 is taken and one pair for the case in which the jump is not taken). Using the hypothetical alternative domain, the size of the abstract configuration grows quickly.

In the first six clock cycles, the instructions are fetched, dispatched, executed, and committed in program order and the respective abstract cache configurations represent one unique concrete cache configuration. In the seventh clock cycle, the execute stage processes instruction i_2, i.e., the conditional jump depending on the secret value from the data address d_1. While the domain $\overline{Cos}$ is fine-grained enough to track the abstract pipeline configurations that result from the two possible secret values separately, the hypothetical domain uses a set abstraction for the pipeline configuration. That is, in the eighth clock cycle, each pipeline stage is mapped to a set that contains two possibilities (two instructions in case of the fetch stage; one instruction and $\top$, i.e., stalling, in case of the other stages). The possible reservation-station configurations and reorder-buffer configurations are also captured using set abstractions. In the eighth clock cycle, there are two possible configurations in the sets for both, the reservation stations and the reorder buffer, respectively. In Cycles 8-10, there are two possibilities for the instruction dispatch and

the scheduling of an instruction for execution, which causes the number of possible reservation-station and reorder-buffer configurations to double. Moreover, the number of possibilities for the instruction to be committed increases in each clock cycle, starting from Cycle 9. The reason is that the possibility of stalling ($\top$) in the commit stage can never be ruled out based on the possible combinations of reservation-station and reorder-buffer configurations. This causes the number of possibilities for the reorder-buffer configurations to grow quickly, from 16 in Cycle 10 to 48 in Cycle 11 (because the commit stage in Cycle 10 might process any of i_1, i_3, i_4, i_7, or $\top$), to 61 in Cycle 12 (because the commit stage in Cycle 11 might process i_8, i_9, or any of the possibilities mentioned for Cycle 10), to 141 in Cycle 13, to 172 in Cycle 14-16 (because the commit stage in Cycles 12-16 might process i_5, i_6, or any of the possibilities mentioned for Cycles 10 and 11). Overall, the analysis with the hypothetical domain is significantly more complex than the analysis with $\overline{Cos}$ already for this small example program.

The leakage bound resulting from the analysis with $\overline{Cos}$ is $\log_2(2) = 1\,\text{bit}$, because the final abstract configuration captures two distinct final concrete cache configurations. Using the hypothetical alternative abstract domain, we obtain the leakage bound $\log_2(400) \approx 8.7\,\text{bit}$.

Since the pipeline stages are captured by sets of possible states during the analysis with the hypothetical domain, there are also multiple possibilities with respect to the dispatch of instructions that access the memory. This causes the abstract cache configuration to become increasingly imprecise starting from the ninth clock cycle, in which d_5 might or might not be loaded into the cache. In the tenth cycle, d_6 might or might not be loaded. In the eleventh cycle, d_7 and d_8 each might or might not be loaded. Finally, in the twelfth cycle, again d_8 might or might not be loaded. This leads to the following final abstract cache configuration:

$$\overline{c}(d) = \begin{cases} \{3,4,5,6,7\} & \text{if } d = d_1 \\ \{2,3,4,5,6\} & \text{if } d = d_2 \\ \{1,2,3,4,5\} & \text{if } d = d_3 \\ \{0,1,2,3,4\} & \text{if } d = d_4 \\ \{0,1,2,3,\mathit{SIZE}\} & \text{if } d = d_5 \\ \{0,1,2,\mathit{SIZE}\} & \text{if } d = d_6 \\ \{0,1,\mathit{SIZE}\} & \text{if } d = d_7 \\ \{0,\mathit{SIZE}\} & \text{if } d = d_8 \\ \{\mathit{SIZE}\} & \text{otherwise.} \end{cases}$$

This abstract cache configuration represents 400 possible concrete cache configurations that can occur in real executions (i.e., where each cache line contains at most one entry and where the lines are used consecutively). Examples include, e.g., the following cache configurations:

$$c_1(d) = \begin{cases} 0 & \text{if } d = d_5 \\ 1 & \text{if } d = d_3 \\ 2 & \text{if } d = d_2 \\ 3 & \text{if } d = d_1 \\ 4 & \text{if } d = d_4 \\ \mathit{SIZE} & \text{otherwise,} \end{cases} \quad c_1(d) = \begin{cases} 0 & \text{if } d = d_6 \\ 1 & \text{if } d = d_3 \\ 2 & \text{if } d = d_2 \\ 3 & \text{if } d = d_1 \\ 4 & \text{if } d = d_4 \\ \mathit{SIZE} & \text{otherwise,} \end{cases} \quad \text{and} \quad c_1(d) = \begin{cases} 0 & \text{if } d = d_4 \\ 1 & \text{if } d = d_5 \\ 2 & \text{if } d = d_2 \\ 3 & \text{if } d = d_1 \\ 4 & \text{if } d = d_3 \\ \mathit{SIZE} & \text{otherwise.} \end{cases}$$

Overall, the analysis with our pipeline abstract domain $\overline{Cos}$ captures the actual leakage of 1 bit much more precisely than the analysis with the hypothetical alternative domain. ◊

Notational Conventions We define the following short-hand notations to extract the components from an abstract pipeline-stage configuration $\overline{pi} \in \overline{PCos}$. We write $\mathit{stag}(\overline{pi})$ to extract

the abstract pipeline-stage configurations, $rst(\overline{pi})$ to extract the abstract reservation-station configuration, and $buf(\overline{pi})$ to extract the abstract reorder-buffer configuration.

To extract the content of each pipeline stage of $\overline{pi}$, we use the short-hand notations $fst(\overline{pi})$, $dst(\overline{pi})$, $est(\overline{pi})$, and $cst(\overline{pi})$. To extract the instruction identifier and the instruction address from a reservation-station or reorder-buffer entry, we use $uad(((ia, n), x, y)) = (ia, n)$ and $ad(((ia, n), x, y)) = ia$, respectively. Finally, we extract the set of all memory-access-list entries and the set of all register-dependence list entries from an abstract reservation-station configuration $\overline{rs}$ using $mcs(\overline{rs})$ and $rds(\overline{rs})$, respectively.

6.3.2 Abstract Semantics

Our abstract semantics captures the changes to the system state that occur during one clock cycle of the program execution with respect to the pipeline abstract domain. The overall abstract semantics $upd^\alpha : \overline{Cos} \to \overline{Cos}$ is based on component-wise abstract update functions.

Abstract Pipeline Update

The abstract update function for the first component, the instruction pipeline, is defined based on abstract update functions for the pipeline stages, the reservation stations, and the reorder buffer.

Pipeline Stages In each clock cycle, the fetch stage of the instruction pipeline fetches one instruction. Which instruction is fetched, i.e., enters the fetch stage, in each cycle is defined in our abstract semantics by the function $nxf^\alpha : \overline{PCos} \times Ps_{wf} \to StCs$, such that

$$nxf^\alpha(\overline{pi}, pr) = \begin{cases} \bot & \text{if } barr(\overline{pi}, pr) \\ \top & \text{if } term(\overline{pi}, pr) \\ bpr(fst(\overline{pi})) & \text{if } \neg barr(\overline{pi}, pr) \wedge \neg term(\overline{pi}, pr) \wedge fst(\overline{pi}) \neq \bot \\ bpr(est(\overline{pi})) & \text{otherwise.} \end{cases}$$

The first case in the definition of nxf^α captures the update to the fetch stage in case the stage is stalled due to a pending speculation barrier. Whether the there is a pending speculation barrier is captured by the predicate $barr : \overline{PCos} \times Ps_{wf} \to \mathbb{B}$, such that

$$\begin{aligned} barr(\overline{pi}, pr) = (&\textbf{fence} \in \{pr(fst(\overline{pi})), pr(dst(\overline{pi}))\} \vee \\ &\exists i \in \mathbb{N}_0.\ (i < |rst(\overline{pi})| \wedge pr(ad(rst(\overline{pi})[i])) = \textbf{fence})). \end{aligned}$$

This predicate holds if there is a **fence** instruction that is currently processed in the fetch or dispatch stage (first disjunct) or in a reservation station (second disjunct).

The second case in the definition of nxf^α captures the update to the fetch stage in case the stage is stalled because the program has already been fetched completely. Whether the program has been fetched completely is captured by the predicate $term : \overline{PCos} \times Ps_{wf} \to \mathbb{B}$, such that

$$\begin{aligned} term(\overline{pi}, pr) = (&fst(\overline{pi}) = \top \vee (fst(\overline{pi}) \in UIAs \wedge bpr(fst(\overline{pi})) = (ia, n) \wedge pr(ia)\uparrow)) \vee \\ &(fst(\overline{pi}) = \bot \wedge bpr(est(\overline{pi})) = (ia, n) \wedge pr(ia)\uparrow)) \wedge \neg barr(\overline{pi}, pr). \end{aligned}$$

This predicate holds if there is no pending speculation barrier and if either (1) the program had already been fetched completely before the current clock cycle (first disjunct), (2) the instruction that is currently processed in the fetch stage is the last instruction of the program (second disjunct), or (3) the fetch stage is stalled due to a **fence** that is the last instruction of the program and that is currently processed in the execute stage (third disjunct).

Here, the function $bpr : UIAs \to UIAs$ is used to determine whether an instruction is the last instruction during a program execution. It is defined by

$$bpr((ia_k, n)) = (ia_{k+1}, n).$$

Given a unique instruction identifier (ia_k, n), the function predicts the identifier of the next instruction to be fetched. If ia_k is a regular instruction that does not trigger conditional jumps, the next instruction is always ia_{k+1} . If ia_k is a conditional jump instruction, the function bpr predicts that the corresponding branch will not be taken. That is, it also predicts that the next instruction is ia_{k+1}. Hence, bpr captures a static always-not-taken branch-prediction strategy.

Situations in which the fetch stage is not stalled and the program has not been fetched completely are covered by the third and fourth case in the definition of nxf^α. The former captures the situation in which there is no pending **fence** instruction and in which the next instruction is predicted based on the current instruction in the fetch stage. The latter captures the situation in which the last pending **fence** is processed in the execute stage and in which the next instruction is predicted based on this **fence** instruction.

Since the instructions are dispatched to reservation stations in order, the dispatch stage is always updated based on the fetch stage. The instruction that leaves the fetch stage enters the dispatch stage. The instruction that leaves the dispatch stage enters the reservation stations. The out-of-order execution begins at the execute stage. The instruction that leaves the reservation stations and enters the execute stage is determined by the function $nxe^\alpha : \overline{RSCos} \times \overline{RSCos} \to StCs$, s.t.

$$nxe^\alpha(\overline{rs}, a) = \begin{cases} \top & \text{if } \overline{rs} = \langle\rangle \\ uad(\overline{rc}) & \text{if } \overline{rs} = \langle \overline{rc} \rangle \bullet \overline{rs''} \wedge re^\alpha(\overline{rc}, a) \\ ua & \text{if } \overline{rs} = \langle \overline{rc} \rangle \bullet \overline{rs''} \wedge \neg re^\alpha(\overline{rc}, a) \wedge ua = nxe^\alpha(\overline{rs''}, a \bullet \langle \overline{rc} \rangle). \end{cases}$$

Given an abstract reservation-station configuration $\overline{rs}$, the result of $nxe^\alpha(\overline{rs}, \langle\rangle)$ is either the first instruction that is processed in the reservation stations and that is ready to be executed or, if no instruction is ready to be executed, $\top$. Whether an instruction is ready to be executed is captured by the predicated $re^\alpha : \overline{RSCs} \times \overline{RSCos} \to \mathbb{B}$, such that

$$\begin{aligned} re^\alpha((ua, \overline{ml}, rl), \overline{rs}) = \forall i \in \mathbb{N}_0.\ (i < |\overline{ml}| \Rightarrow \exists D \in \mathcal{P}(DAs).\ \overline{ml}[i] = (D, 0)) \wedge \\ \forall j \in \mathbb{N}_0.\ (j < |rl| \Rightarrow \exists e \in Rs.\ \exists ua' \in UIAs. \\ (rl[j] = (e, ua') \wedge \forall k \in \mathbb{N}_0.\ (k < |\overline{rs}| \Rightarrow uad(\overline{rs}[k]) \neq ua'))). \end{aligned}$$

Given a reservation-station entry $(ua, \overline{ml}, rl)$ and the list $\overline{rs}$ of all reservation-station entries that belong to pending instructions that were dispatched before ua, the predicate holds if ua is ready to be executed. More concretely it holds if all memory accesses of ua have been completed (first conjunct) and all register-dependencies of ua have been resolved, i.e., the instructions that produce the values for the register operands are not pending anymore (second conjunct).

In our execution model register updates are committed synchronously in program order. This is captured by the function $nxc^\alpha : \overline{PCos} \to StCs$, such that

$$nxc^\alpha(\overline{pi}) = \begin{cases} uad(\overline{b}[0]) & \text{if } rc^\alpha(\overline{pi}, \overline{b}) \wedge \overline{b} = pb^\alpha(buf(\overline{pi}), cst(\overline{pi})) \\ \top & \text{if } \neg rc^\alpha(\overline{pi}, \overline{b}) \wedge \overline{b} = pb^\alpha(buf(\overline{pi}), cst(\overline{pi})). \end{cases}$$

The auxiliary function $pb^\alpha : \overline{BCos} \times StCs \to \overline{BCos}$ with

$$pb^\alpha(\overline{b}, sc) = \begin{cases} \overline{b} & \text{if } \overline{b} = \langle\rangle \\ pb^\alpha(\overline{b'}) & \text{if } \overline{b} = \langle \overline{bc} \rangle \bullet \overline{b'} \wedge sc = ad(\overline{bc}) \\ \overline{bc} \bullet pb^\alpha(\overline{b'}) & \text{if } \overline{b} = \langle \overline{bc} \rangle \bullet \overline{b'} \wedge sc \neq ad(\overline{bc}) \end{cases}$$

is used to remove the register update from the reorder buffer that is currently processed in the commit stage. This avoids that the same update is committed twice. The function nxc^α then checks the first register update from the remaining reorder buffer. If it is ready to be committed, the commit stage is updated to the instruction that triggered the update. If it s not ready to be committed, the commit stage is updated to $\top$. Whether an update is ready to be committed is captured by the predicate $rc^\alpha : \overline{PCos} \times \overline{BCos} \to \mathbb{B}$, such that

$$rc^\alpha(\overline{pi}, \overline{b}) = (|\overline{b}| > 0 \wedge \forall i \in \mathbb{N}_0.\ (i < |rst(\overline{pi})| \Rightarrow uad(\overline{b}[0]) \neq uad(rst(\overline{pi})[i]))).$$

The predicate holds if the instruction that triggered the first register update is not processed in a reservation station, i.e., if the instruction is currently executed or has already been executed.

Definition 6.2. *The abstract pipeline-stage update is* $upd^\alpha_{st} : \overline{PCos} \times Ps_{wf} \to StCos$*, such that*

$$upd^\alpha_{st}(\overline{pi}, pr)(s) = \begin{cases} nxf^\alpha(\overline{pi}, pr) & \text{if } s = fet \\ fst(\overline{pi}) & \text{if } s = dis \\ nxe^\alpha(rst(\overline{pi}), \langle\rangle) & \text{if } s = exe \\ nxc^\alpha(\overline{pi}) & \text{if } s = com. \end{cases}$$

The abstract pipeline-stage update uses the functions nxf^α, nxe^α, and nxc^α to capture the overall changes to the pipeline stages during one clock cycle if no pipeline flush occurs. The next predicted instruction (potentially speculatively) enters the fetch stage, the instruction that leaves the fetch stage enters the dispatch stage (in program order), the next ready instruction from the reservation stations (potentially out-of-order) enters the execute stage, and the instruction that triggered the next ready register update (in program order) enters the commit stage.

Definition 6.3. *The abstract pipeline flush is the function* $fl^\alpha_{st} : \overline{PCos} \times Ps_{wf} \to StCos$*, s.t.*

$$fl^\alpha_{st}(\overline{pi}, pr)(s) = \begin{cases} cor(\overline{pi}, pr) & \text{if } s = fet \\ cst(\overline{pi}) & \text{if } s = com \\ \top & \text{otherwise.} \end{cases}$$

The abstract pipeline flush captures the changes to the pipeline stages during one clock cycle if a pipeline flush occurs. In this case, the fetch stage is updated to the correct instruction address based on the branching decision that triggered the pipeline flush. The commit stage remains unchanged and the dispatch and execute stages are cleared. The correct instruction address for the fetch stage is captured by the auxiliary function $cor : \overline{PCos} \times Ps_{wf} \to StCs$, such that

$$cor(\overline{pi}, pr) = \begin{cases} (ia_k, n) & \text{if } pr(est(\overline{pi})) = \textbf{jge}\ ia_k\ e\ e' \wedge est(\overline{pi}) = (ia_{k'}, n) \wedge k > k' \\ (ia_k, n+1) & \text{if } pr(est(\overline{pi})) = \textbf{jge}\ ia_k\ e\ e' \wedge est(\overline{pi}) = (ia_{k'}, n) \wedge k \leq k' \\ \top & \text{otherwise.} \end{cases}$$

That is, if the instruction processed in the execute stage is a conditional jump that jumps forward, the instruction address is set to the target address of the jump. If the instruction in the execute stage is a conditional jump that jumps backward, the instruction address is set to the target address and the counter for instruction occurrence is increased by one.

Reservation Stations In each clock cycle, there are three changes that affect the reservation stations: The instruction that leaves the dispatch stage is dispatched to a new entry in the reservation stations, the ongoing memory accesses make progress, and the next ready instruction is scheduled for execution, i.e., leaves the reservation stations and enters the execute stage.

The first change, i.e., the dispatch of the instruction that leaves the dispatch stage, is captured by the function $disp^{\alpha} : StCs \times Ps_{wf} \times \overline{RCos} \times \overline{BCos} \times \overline{CCos} \to \mathcal{P}(\overline{RSCos})$, such that

$$disp^{\alpha}(sc, pr, \overline{r}, \overline{b}, \overline{c}) = \begin{cases} \{\langle\rangle\} & \text{if } sc \notin UIAs \\ \{\langle(sc, \langle\rangle, x)\rangle\} & \text{if } sc = (ia, n) \wedge opc(pr(ia)) \neq \textbf{mov-rm} \wedge x = rde(sc, pr) \\ X & \text{if } sc = (ia, n) \wedge opc(pr(ia)) = \textbf{mov-rm} \wedge \\ & \quad X = \{\langle(sc, mts, rde(sc, pr))\rangle \mid \exists e, e' \in Rs.\ \exists d \in DAs. \\ & \qquad (pr(ia) = \textbf{mov-rm}\ e\ d\ e' \wedge \\ & \qquad\ mts \in mtim(d, sc, e', \overline{r}, \overline{b}, \overline{c}))\}. \end{cases}$$

If there is no instruction to be dispatched, i.e., if the dispatch stage is stalled, $disp^{\alpha}(sc, pr, \overline{r}, \overline{b}, \overline{c})$ is a singleton set that contains only the empty list. If the dispatch stage processes an instruction that does not access the data memory, $disp^{\alpha}(sc, pr, \overline{r}, \overline{b}, \overline{c})$ is a singleton set that contains a reservation-station entry for the respective instruction. The memory-access list of this entry is the empty list. The register-dependence list contains the dependencies of the source registers. If the dispatch stage processes a **mov-rm** instruction, the set $disp^{\alpha}(sc, pr, \overline{r}, \overline{b}, \overline{c})$ might have multiple entries, namely one entry for each possible outcome (cache hit or miss) of the memory access.

The register dependencies of an instruction are defined by $rde : UIAs \times Ps_{wf} \to RLs$, s.t.

$$rde((ia, n), pr) = \begin{cases} \langle x\rangle & \text{if } sregs(pr(ia)) = \langle e\rangle \wedge x = (e, dep((ia, n), e, pr)) \\ \langle x, y\rangle & \text{if } sregs(pr(ia)) = \langle e, e'\rangle \wedge \\ & \quad x = (e, dep((ia, n), e, pr)) \wedge y = (e', dep((ia, n), e', pr)) \\ \langle\rangle & \text{if } sregs(pr(ia)) = \langle\rangle \wedge pr(ia) \neq \textbf{fence} \\ fd((ia, n)) & \text{if } pr(ia) = \textbf{fence}. \end{cases}$$

If an instruction ua has one or two source registers (first and second case in the definition, respectively), $rde(ua, pr)$ is a list that contains the dependence of each source register, which is determined using an auxiliary function $dep : UIAs \times Rs \times Ps_{wf} \to UIAs$. If ua has no source registers and is not a speculation barrier, then $rde(ua, pr)$ is the empty list. Finally, in case of a speculation barrier, the dependencies are determined by an auxiliary function $fd : UIAs \to RLs$, such that $fd(ua) = fdx(ua, \langle\rangle)$. Here, the function $fdx : UIAs \times RLs \to RLs$ is defined by

$$fdx(ua, rl) = \begin{cases} rl & \text{if } \neg(\exists ua' \in UIAs.\ (ua' <_{po} ua)) \\ fdx(ua, rl \bullet \langle(eax, x)\rangle) & \text{if } \exists ua' \in UIAs.\ (ua' <_{po} ua) \wedge x <_{po} ua \wedge \\ & \quad \neg(\exists ua'' \in UIAs.\ (x <_{po} ua'' \wedge ua'' <_{po} ua)), \end{cases}$$

where $<_{po}$: $UIAs \times UIAs \to \mathbb{B}$ is the program order, i.e., the order in which the instructions would be executed by a sequential in-order CPU. The function $fdx(ua, rl)$ uses rl as an accumulator to compute a list that contains a dependence of the register eax on each instruction that occurs in program order before ua. This definition of the dependencies for speculation barriers captures that instructions cannot be re-ordered across such barriers.

The possible memory-access lists for a freshly dispatched instruction ua are defined by $mtim : DAs \times StCs \times Rs \times \overline{RCos} \times \overline{BCos} \times \overline{CCos} \to \mathcal{P}(\overline{MLs})$, such that

$$mtim(d_k, sc, e, \overline{r}, \overline{b}, \overline{c}) = \{\langle(M, t)\rangle \mid M = \{d_{k+v} \mid v \in rv^{\alpha}(sc, e, \overline{r}, \overline{b})\} \wedge t \in \{cyc^{\alpha}(d, \overline{c}) \mid d \in M\}\}.$$

That is, $mtim(d_k, sc, e, \overline{r}, \overline{b}, \overline{c})$ contains one or two memory-access lists. It contains the list $\langle(M_1, \mathit{HIT})\rangle$ if there is at least one possible target address of the memory access that might result in a cache hit. The set M_1 contains all possible target addresses that might result in a cache hit. Analogously, $mtim(d_k, sc, e, \overline{r}, \overline{b}, \overline{c})$ contains a list $\langle(M_2, \mathit{MISS})\rangle$ if there is at least one

possible target address that might result in a cache miss. The set M_2 contains all possible target addresses that might result in a cache miss. The possible target addresses of a memory access are determined by computing all possible values of the offset in register e with the function rv^α and adding them to the base address d_k. The possible outcomes of the memory access for each possible target address are determined using the function cyc^α.

The function $rv^\alpha : UIAs \times Rs \times \overline{RCos} \times \overline{BCos} \to \mathcal{P}(Vs)$ is defined by

$$rv^\alpha(ua, e, \overline{r}, \overline{b}) = \begin{cases} \overline{r}(e) & \text{if } \overline{b} = \langle\rangle \\ V & \text{if } \overline{b} = \overline{b'} \bullet \langle(ua', e', V')\rangle \\ & \quad \wedge ((V = rv^\alpha(ua, e, \overline{r}, \overline{b'}) \ \wedge (e \neq e' \vee \neg(ua' <_{po} ua))) \\ & \quad\quad \vee (V = V' \wedge e = e' \wedge ua' <_{po} ua)). \end{cases}$$

The function $rv^\alpha(ua, e, \overline{r}, \overline{b})$ searches for the update in the reorder buffer $\overline{b}$ that belongs to the most recently dispatched instruction that (1) writes to the register e and (2) occurs before ua in program order. If such an update exists in the buffer, the function returns the set of values from this update. If no such update exists, the function returns the set of values of the register e according to the abstract register configuration $\overline{r}$. That is, the function captures the forwarding of uncommitted register values between the reservation stations on an architecture with implicit register renaming.

The function $cyc^\alpha : DAs \times \overline{CCos} \to \mathcal{P}(\mathbb{N}_0)$ is defined by

$$cyc^\alpha(d, \overline{c}) = \begin{cases} \{HIT\} & \text{if } \forall a \in \overline{c}(d).\ a < SIZE \\ \{MISS\} & \text{if } \forall a \in \overline{c}(d).\ a \geq SIZE \\ \{HIT, MISS\} & \text{otherwise.} \end{cases}$$

That is, if the data address d is cached in each concrete cache configuration represented by $\overline{c}$, then $cyc^\alpha(d, \overline{c})$ is the singleton set $\{HIT\}$. If d is definitely not cached, then $cyc^\alpha(d, \overline{c})$ is the singleton set $\{MISS\}$. If both cases are possible, then $cyc^\alpha(d, \overline{c})$ is the set $\{HIT, MISS\}$. The function cyc^α, thus, overapproximates the possible numbers of clock cycles that might be required to retrieve the value from the data address d.

Overall, the function $disp^\alpha$ overapproximates the possible outcomes of an instruction dispatch for the reservation stations based on the auxiliary functions defined above.

Definition 6.4. *The abstract reservation-station update is the function* $upd^\alpha_{rs} : \overline{RSCos} \times \overline{PCos} \times Ps_{wf} \times \overline{RCos} \times \overline{CCos} \to \mathcal{P}(\overline{RSCos})$*, such that*

$$upd^\alpha_{rs}(\overline{rs}, \overline{pi}, pr, \overline{r}, \overline{c}) = \{\overline{rs''} \mid \exists \overline{rs'} \in disp^\alpha(dst(\overline{pi}), pr, \overline{r}, buf(\overline{pi}), \overline{c}).\ \overline{rs''} = uts^\alpha(ps^\alpha(\overline{rs})) \bullet \overline{rs'}\}.$$

The abstract reservation station update combines the dispatch of a new instruction with the other possible changes to the reservation stations. The newly dispatched instruction is appended in the end of the abstract reservation-station configuration. The existing reservation-station entries are updated using the function ps^α to remove the entry of the next instruction that is scheduled for execution and the function uts^α to capture the progress of ongoing memory accesses.

The function $ps^\alpha : \overline{RSCos} \to \overline{RSCos}$ is defined by

$$ps^\alpha(\overline{rs}) = \begin{cases} \overline{rs} & \text{if } \overline{rs} = \langle\rangle \\ \overline{rs'} & \text{if } \overline{rs} = \overline{rc} \bullet \overline{rs'} \wedge ad(\overline{rc}) = nxe^\alpha(\overline{rs}, \langle\rangle) \\ \overline{rc} \bullet ps^\alpha(\overline{rs'}) & \text{if } \overline{rs} = \overline{rc} \bullet \overline{rs'} \wedge uad(\overline{rc}) \neq nxe^\alpha(\overline{rs}, \langle\rangle). \end{cases}$$

That is, it removes exactly the entry that belongs to the instruction $nxe^\alpha(\overline{rs}, \langle\rangle)$. Recall that this is also the instruction to which the execute stage is updated in upd^α_{st}. That is, the scheduling is captured faithfully and the instruction does not get lost from the reservation stations.

The function $uts^\alpha : \overline{RSCos} \to \overline{RSCos}$ is defined by

$$uts^\alpha(\overline{rs}) = \begin{cases} \overline{rs} & \text{if } \overline{rs} = \langle\rangle \\ \overline{rc} \bullet uts^\alpha(\overline{rs'}) & \text{if } \overline{rs} = \langle (ia, \overline{ml}, rl)\rangle \bullet \overline{rs'} \wedge \overline{rc} = (ia, u^\alpha(ml), rl) \end{cases}$$

with the auxiliary function

$$u^\alpha(\overline{ml}) = \begin{cases} \overline{ml} & \text{if } \overline{ml} = \langle\rangle \\ (D, n') \bullet u^\alpha(\overline{ml'}) & \text{if } \overline{ml} = \langle (D, n)\rangle \bullet \overline{ml'} \wedge n' = \max(n-1, 0). \end{cases}$$

It captures the progress made by the memory accesses of the instructions that are processed in the reservation station by decreasing the remaining number of clock cycles for each access by one (unless the number is already zero, i.e., the access has been completed already).

Overall, upd^α_{rs} models the changes to the reservation stations in case no pipeline flush occurs. If a pipeline flush occurs, the changes are captured by $fl^\alpha_{rs} : \overline{RSCos} \times StCs \to \overline{RSCos}$, such that

$$fl^\alpha_{rs}(\overline{rs}, sc) = \begin{cases} \langle\rangle & \text{if } \overline{rs} = \langle\rangle \\ fl^\alpha_{rs}(\overline{rs'}, sc) & \text{if } \overline{rs} = \langle \overline{rc}\rangle \bullet \overline{rs'} \wedge sc <_{po} uad(\overline{rc}) \\ \langle \overline{rc}\rangle \bullet x & \text{if } \overline{rs} = \langle \overline{rc}\rangle \bullet \overline{rs'} \wedge x = fl^\alpha_{rs}(\overline{rs'}, sc) \wedge \neg(sc <_{po} uad(\overline{rc})). \end{cases}$$

That is, all reservation-station entries that belong to speculatively dispatched instructions (i.e., instructions that occur after sc in program order) are removed from the reservation stations.

Reorder Buffer In each clock cycle, there are two changes that affect the reorder buffer: A new register update for the instruction that leaves the dispatch stage is added to the buffer and the register update for the instruction that leaves the commit stage is dropped from the buffer.

The first change, i.e., the addition of a new update to the reorder buffer, depends on the result of the instruction that leaves the dispatch stage. The new update to be added is captured by the function $res^\alpha : StCs \times Ps_{wf} \times \overline{BCos} \times \overline{RCos} \times \overline{MCos} \to \overline{BCos}$, such that

$$res^\alpha(sc, pr, \overline{b}, \overline{r}, \overline{m}) = \begin{cases} \langle\rangle & \text{if } sc \notin IAs \vee (sc = (ia, n) \wedge \\ & \quad opc(pr(ia)) \in \{\textbf{jge}, \textbf{fence}, \textbf{nop}\}) \\ \langle (sc, e, X)\rangle & \text{if } sc = (ia, n) \wedge e = dreg(pr(ia)) \wedge \\ & \quad X = resx^\alpha(pr(ia), sc, \overline{b}, \overline{r}, \overline{m}). \end{cases}$$

For **jge**, **fence**, and **nop** instructions, no new register update is added, because these instructions do not write to any registers. For each other instruction, a register update is added that consists of the instruction's identifier, its destination register, and the set of its possible results. The latter is defined by the function $resx^\alpha : Insts \times StCs \times \overline{BCos} \times \overline{RCos} \times \overline{MCos} \to \mathcal{P}(Vs)$, such that

$$resx^\alpha(in, sc, \overline{b}, \overline{r}, \overline{m}) = \begin{cases} \bigcup_{v \in V} \overline{m}(d_{k+v}) & \text{if } in = \textbf{mov-rm}\ e\ d_k\ e' \wedge \\ & \quad V = rv^\alpha(sc, e', \overline{r}, \overline{b}) \\ \bigcup_{v \in V} \{\sim v\} & \text{if } in = \textbf{neg}\ e \wedge V = rv^\alpha(sc, e, \overline{r}, \overline{b}) \\ \bigcup_{v \in V, v' \in V'} \{bop(in, v, v')\} & \text{if } sregs(in) = \langle e, e'\rangle \wedge V = rv^\alpha(sc, e, \overline{r}, \overline{b}) \wedge \\ & \quad V' = rv^\alpha(sc, e', \overline{r}, \overline{b}) \\ \bigcup_{v \in V} \{bop(in, v, v')\} & \text{if } in \in \{\textbf{add-rc}\ e\ v', \textbf{sub-rc}\ e\ v', \\ & \qquad \textbf{shr-rc}\ e\ v', \textbf{sar-rc}\ e\, v', \textbf{mov-rc}\, e\, v'\} \wedge \\ & \quad V = rv^\alpha(sc, e, \overline{r}, \overline{b}). \end{cases}$$

The set of possible results of a **mov-rm** instruction is the set of all values that might be stored at any possible target address of the memory access that is triggered by the instruction. For

a **neg** instruction, the possible results are the bit-wise negations of the possible values of the register. For instructions with two source registers, the possible results are computed by applying the function *bop*, which models the ALU, to the sets of possible values for both registers. For instructions with one source register and one immediate operand, the possible results are also computed by *bop*, but instead of the set of values for the second register a singleton set that contains the immediate value is used.

The function $bop : Insts \times Vs \times Vs \to Vs$ is defined by

$$bop(in, v, v') = \begin{cases} v + v' & \text{if } opc(in) \in \{\textbf{add-rr}, \textbf{add-rc}\} \\ v - v' & \text{if } opc(in) \in \{\textbf{sub-rr}, \textbf{sub-rc}\} \\ v\&v' & \text{if } opc(in) = \textbf{and-rr} \\ v\|v' & \text{if } opc(in) = \textbf{or-rr} \\ v >>> v' & \text{if } opc(in) = \textbf{shr-rc} \\ v >> v' & \text{if } opc(in) = \textbf{sar-rc} \\ v' & \text{if } opc(in) = \textbf{mov-rc}. \end{cases}$$

Here, $v + v'$ and $v - v'$ are the sum and difference of v and v', respectively, where v and v' are each interpreted as a value in two's complement representation. Furthermore, $v\&v'$ is the bit-wise AND, $v\|v'$ is the bit-wise OR, $\sim v$ is the bit-wise negation, and $v >>> v'$ and $v >> v'$ are the logic and arithmetic right-shift of v by v' positions, respectively. The function *bop* applies the arithmetic or logic operation that corresponds to the instruction's mnemonic to all possible combinations of source values and returns the corresponding set of result values. Thus, the function overapproximates all possible results for an instruction.

Definition 6.5. *The abstract reorder-buffer update is the function* $upd_b^\alpha : \overline{BCos} \times \overline{PCos} \times Ps_{wf} \times \overline{MCos} \times \overline{RCos} \to \overline{BCos}$*, such that*

$$upd_b^\alpha(\overline{b}, \overline{pi}, pr, \overline{m}, \overline{r}) = pb^\alpha(\overline{b}, cst(\overline{pi})) \bullet res^\alpha(dst(\overline{pi}), pr, \overline{b}, \overline{r}, \overline{m}).$$

The abstract reorder-buffer update appends the register update for the newly dispatched instruction in the end of the abstract reorder-buffer configuration. It also removes the update triggered by the instruction $cst(\overline{pi})$ in the commit stage using the function pb^α defined above.

The function upd_b^α captures the changes to the reorder buffer for clock cycles in which no pipeline flush occurs. The case of a flush is captured by $fl_b^\alpha : \overline{BCos} \times StCs \to \overline{BCos}$, such that

$$fl_b^\alpha(\overline{b}, sc) = \begin{cases} \langle\rangle & \text{if } \overline{b} = \langle\rangle \\ fl_b^\alpha(\overline{b'}, sc) & \text{if } \overline{b} = \langle \overline{bc} \rangle \bullet \overline{b'} \wedge ad(\overline{bc}) > sc \\ \langle \overline{bc} \rangle \bullet fl_b^\alpha(\overline{b'}, sc) & \text{if } \overline{b} = \langle \overline{bc} \rangle \bullet \overline{b'} \wedge ad(\overline{bc}) \leq sc. \end{cases}$$

The function removes all register updates from the buffer that were triggered by speculatively dispatched instructions, i.e., by instructions that occur after *sc* in program order.

Overall Pipeline The update function for an overall abstract pipeline configuration combines the update functions for the components that were described above.

Definition 6.6. *The abstract pipeline update is the function* $upd_{pi}^\alpha : \overline{PCos} \times Ps_{wf} \times \overline{RCos} \times \overline{MCos} \times \overline{CCos} \to \mathcal{P}(\overline{PCos})$*, such that*

$$\begin{aligned} upd_{pi}^\alpha(\overline{pi}, pr, \overline{r}, \overline{m}, \overline{c}) = &\{fl^\alpha(\overline{pi}, pr) \mid true \in isfl^\alpha(\overline{pi}, pr, \overline{r})\} \cup \\ &\{\overline{pi'} \mid \overline{pi'} \in up_{pi}^\alpha(\overline{pi}, pr, \overline{r}, \overline{m}, \overline{c}) \wedge false \in isfl^\alpha(\overline{pi}, pr, \overline{r})\}, \end{aligned}$$

where

$$fl^{\alpha}(\overline{pi}, pr) = (fl^{\alpha}_{st}(\overline{pi}, pr), fl^{\alpha}_{rs}(rst(\overline{pi}), est(\overline{pi})), fl^{\alpha}_{b}(buf(\overline{pi}), est(\overline{pi})))$$

and

$$up^{\alpha}_{pi}(\overline{pi}, pr, \overline{r}, \overline{m}, \overline{c}) = \{(stags, \overline{rs'}, \overline{b'}) \mid stags = upd^{\alpha}_{st}(\overline{pi}, pr) \wedge \overline{rs'} \in upd^{\alpha}_{rs}(rst(\overline{pi}), \overline{pi}, pr, \overline{r}, \overline{c}) \wedge \overline{b'} = upd^{\alpha}_{b}(buf(\overline{pi}), \overline{pi}, pr, \overline{m}, \overline{r})\}.$$

In case a pipeline flush is possible, the update function applies the component-wise update functions for the flush case. In case a regular update without flush is possible, the update function applies the component-wise update functions for the regular case. If both are possible, the set resulting from upd^{α}_{pi} contains the updated abstract pipeline configurations for both cases.

Whether a pipeline flush might take place during a clock cycle is captured by the function $isfl^{\alpha} : \overline{PCos} \times Ps_{wf} \times \overline{RCos} \to \mathcal{P}(\mathbb{B})$, such that

$$isfl^{\alpha}(\overline{pi}, pr, \overline{r}) = \begin{cases} \{\text{true}\} & \text{if } alljump(\overline{pi}, pr, \overline{r}) \\ \{\text{false}\} & \text{if } nojump(\overline{pi}, pr, \overline{r}) \\ \{\text{true}, \text{false}\} & \text{otherwise.} \end{cases}$$

The function returns the set {true} if a pipeline flush definitely occurs. This is captured by the predicate $alljump : \overline{PCos} \times Ps_{wf} \times \overline{RCos} \to \mathbb{B}$, such that

$$alljump(\overline{pi}, pr, \overline{r}) = \exists ia \in IAs.\ \exists e, e' \in Rs.\ pr(est(\overline{pi})) = \textbf{jge}\ ia\ e\ e' \wedge \forall v \in rv^{\alpha}(est(\overline{pi}), e, \overline{r}, buf(\overline{pi}))).\ \forall v' \in rv^{\alpha}(est(\overline{pi}), e', \overline{r}, buf(\overline{pi}))).\ v \geq v'.$$

This predicate holds if the execute stage processes a conditional jump instruction that definitely triggers a jump. That is, it holds if, for each possible combination of values for the source registers of the instruction, the value of the first register is greater than or equal to the value of the second register. A pipeline flush occurs in this case because the always-not-taken branch prediction strategy always predicts that no jump is triggered.

The function $isfl^{\alpha}$ returns the set {false} if definitely no pipeline flush occurs. This is captured by the predicate $nojump : \overline{PCos} \times Ps_{wf} \times \overline{RCos} \to \mathbb{B}$, such that

$$nojump(\overline{pi}, pr, \overline{r}) = (\exists ia \in IAs.\ \exists e, e' \in Rs.\ pr(est(\overline{pi})) = \textbf{jge}\ ia\ e\ e' \wedge \forall v \in rv^{\alpha}(est(\overline{pi}), e, \overline{r}, buf(\overline{pi}))).\ \forall v' \in rv^{\alpha}(est(\overline{pi}), e', \overline{r}, buf(\overline{pi}))).\ v < v') \vee \neg(\exists ia \in IAs.\ \exists e, e' \in Rs.\ pr(est(\overline{pi})) = \textbf{jge}\ ia\ e\ e').$$

This predicate holds if the execute stage processes an instruction that is no conditional-jump instruction or if the execute stage processes a conditional jump that definitely does not trigger a jump. The latter is the case if, for each possible combination of values for the source registers of the instruction, the value of the first register is less than the value of the second register.

If neither of the predicates holds, i.e., if a pipeline flush might or might not happen, the function $isfl^{\alpha}$ returns the set {true, false}. Hence, it overapproximates the occurrence of a pipeline flush with respect to the always-not-taken branch-prediction strategy.

Based on the definitions of all component-wise update functions and auxiliary functions, upd^{α}_{pi} overapproximates all possible changes to the abstract pipeline configuration in one clock cycle.

Abstract Update for Registers, Memory, and Cache

The update functions for abstract instruction-memory configurations and abstract data-memory configurations are identity functions.

Definition 6.7. *The abstract instruction-memory update is* $upd^{\alpha}_{pr} : Ps_{wf} \to Ps_{wf}$*, such that*

$$upd^{\alpha}_{pr}(pr) = pr.$$

Definition 6.8. *The abstract data-memory update is* $upd^{\alpha}_{mem} : \overline{MCos} \to \overline{MCos}$*, such that*

$$upd^{\alpha}_{mem}(\overline{m}) = \overline{m}.$$

For abstract register configurations, the update function is defined in terms of the instruction that leaves the commit stage.

Definition 6.9. *The abstract register update is* $upd^{\alpha}_{reg} : \overline{RCos} \times \overline{BCos} \times StCs \to \overline{RCos}$*, s.t.*

$$upd^{\alpha}_{reg}(\overline{r}, \overline{b}, sc)(e) = \begin{cases} V & \text{if } \exists i \in \mathbb{N}_0.\ (i < |b| \wedge \overline{b}[i] = (sc, e, V) \wedge \\ & \qquad \forall j \in \mathbb{N}_0.\ (j \neq i \Rightarrow uad(\overline{b}[j]) \neq sc)) \\ \overline{r}(e) & \text{otherwise.} \end{cases}$$

Given the instruction sc that leaves the commit stage, the corresponding register update is retrieved from the reorder buffer. Let e be the register from this update and V be the set of possible values from this update. The abstract register configuration is changed so that the register e is mapped to V. The values of all other registers remain unchanged.

The update function for abstract cache configurations captures the possible changes to the cache during a clock cycle. These changes are caused by the memory access that is triggered during the clock cycle. Since there might be more than one possible target address for the memory access, we first define an auxiliary function that captures the changes caused by the access to one target address and then define the overall abstract cache update based on this auxiliary function.

The auxiliary function $up^{\alpha}_{c} : \overline{CCos} \times DAs \to \overline{CCos}$ is defined by

$$up^{\alpha}_{c}(\overline{c}, acad)(d) = \begin{cases} lines_{acc}(\overline{c}, d) & \text{if } d = acad \\ lines_{other}(\overline{c}, acad, d) & \text{otherwise.} \end{cases}$$

Given the abstract cache configuration $\overline{c}$ and the target address $acad$ of the memory access, the updated abstract cache configuration $up^{\alpha}_{c}(\overline{c}, acad)$ maps each data address d to the set of cache lines in which it may be cached after the update.

If d is the target address of the memory access, the set of updated cache lines is defined by the function $lines_{acc} : \overline{CCos} \times DAs \to \mathcal{P}([0, SIZE])$, such that

$$lines_{acc}(\overline{c}, d) = \{n \mid cached(\overline{c}, d) \wedge n \in \overline{c}(d)\} \cup \{0 \mid uncached(\overline{c}, d)\},$$

where $cached(\overline{c}, d) = \exists n \in \mathbb{N}_0.\ n < SIZE \wedge n \in \overline{c}(d)$ and $uncached(\overline{c}, d) = SIZE \in \overline{c}(d)$. That is, if the address was already cached, it remains in the same cache line (first subset in $lines_{acc}$ above) and if the address was not cached yet, it is loaded into cache line number zero (second subset).

If d is the address of some other memory block, the set of updated cache lines is defined by the function $lines_{other} : \overline{CCos} \times DAs \times DAs \to \mathcal{P}([0, SIZE])$, such that

$$\begin{aligned} lines_{other}(\overline{c}, acad, d) = {} & \{SIZE \mid uncached(\overline{c}, d)\} \cup \{n \in \overline{c}(d) \mid cached(\overline{c}, acad)\} \cup \\ & \{n + 1 \mid cached(\overline{c}, d) \wedge uncached(\overline{c}, acad) \wedge n \in \overline{c}(d) \wedge n < SIZE\}. \end{aligned}$$

That is, if d was not cached so far, it remains uncached (first subset in the definition of $lines_{other}$). If the accessed address $acad$ was already cached, d remains in the same cache line in which it was cached so far (second subset). If d was already cached but the accessed address $acad$ was not cached yet, d is moved one cache line ahead (third subset). This captures the FIFO replacement strategy with respect to abstract cache configurations.

Definition 6.10. *The abstract cache update is* $upd_c^\alpha : \overline{CCos} \times \overline{RSCos} \to \overline{CCos}$*, such that*

$$upd_c^\alpha(\overline{c}, \overline{rs})(d') = \begin{cases} D'' & \text{if } D'' = \bigcup_{d \in D} up_c^\alpha(\overline{c}, d)(d') \wedge (D, MISS) \in mcs(\overline{rs}) \wedge \\ & \quad \neg(\exists D' \in \mathcal{P}(DAs).\ (D' \neq D \wedge (D', MISS) \in mcs(\overline{rs}))) \\ \overline{c}(d') & \text{otherwise.} \end{cases}$$

The abstract cache update combines the possible changes to the cache across different target addresses for the memory access. If there is no unique memory access that leads to a cache miss in the current clock cycle, the abstract cache configuration remains unchanged.[4] If there is a unique memory access that leads to a cache miss, the abstract cache configuration is updated based on the possible target addresses of this memory access. More concretely, the updated abstract cache configuration maps each data address to the union of all possible sets of cache lines in which the address might be cached after the memory access. That is, upd_c^α overapproximates all possible changes to the cache during one clock cycle.

Putting all Together

The overall abstract update captures the possible changes to the system components under each possible control flow. We first define an auxiliary function that captures the possible changes under one control flow and then lift the definition across all control flows subsequently.

The auxiliary function $up^\alpha : \overline{PCos} \times Ps_{wf} \times \overline{RCos} \times \overline{MCos} \times \overline{CCos} \to \overline{Cos}$ is defined by

$$up^\alpha(\overline{pi}, pr, \overline{r}, \overline{m}, \overline{c})((pr', \overline{pi'})) = \begin{cases} up_{\text{sto}}^\alpha(\overline{r}, \overline{m}, \overline{c}) & \text{if } pr' = upd_{\text{pr}}^\alpha(pr) \wedge \overline{pi'} \in upd_{\text{pi}}^\alpha(\overline{pi}, pr, \overline{r}, \overline{m}, \overline{c}) \\ \uparrow & \text{otherwise} \end{cases}$$

based on the function $up_{\text{sto}}^\alpha(\overline{r}, \overline{m}, \overline{c}) = (upd_{\text{reg}}^\alpha(\overline{r}, buf(\overline{pi}), cst(\overline{pi})), upd_{\text{mem}}^\alpha(\overline{m}), upd_{\text{c}}^\alpha(\overline{c}, rst(\overline{pi})))$.

The updated abstract configuration is a function that maps each possible updated pipeline configuration to the possible configurations of the other components. It is undefined on each pipeline configuration that does not occur in the set of possible updated pipeline configurations.

Definition 6.11. *The abstract update (or abstract semantics) is* $upd^\alpha : \overline{Cos} \to \overline{Cos}$*, such that*

$$upd^\alpha(\overline{co}) = \bigsqcup \left\{ up^\alpha(\overline{pi}, pr, \overline{co}(pr, \overline{pi})) \mid (pr, \overline{pi}) \in Ps_{wf} \times \overline{PCos} \wedge \overline{co}(pr, \overline{pi})\downarrow \right\}.$$

The overall abstract semantics combines the abstract updates for the individual control flows. To this end, it applies the join function $\bigsqcup : \mathcal{P}(\overline{Cos}) \to \overline{Cos}$ of our pipeline abstract domain, where

$$\bigsqcup(\overline{S})(pr, \overline{pi}) = \begin{cases} \bigsqcup_{\text{aux}}(\{\overline{co}(pr, \overline{pi}) \mid \overline{co} \in \overline{S} \wedge \overline{co}(pr, \overline{pi})\downarrow\}) & \text{if } \exists \overline{co} \in \overline{S}.\ \overline{co}(pr, \overline{pi})\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

with the auxiliary function $\bigsqcup_{\text{aux}} : \mathcal{P}(\overline{RCos} \times \overline{MCos} \times \overline{CCos}) \to \overline{RCos} \times \overline{MCos} \times \overline{CCos}$, s.t.

$$\begin{aligned} \bigsqcup_{\text{aux}}(\overline{X}) = \Big(& \bigsqcup_{\text{r}} \{\overline{r} \mid \exists \overline{m} \in \overline{MCos}.\ \exists \overline{c} \in \overline{CCos}.\ (\overline{r}, \overline{m}, \overline{c}) \in \overline{X}\}, \\ & \bigsqcup_{\text{m}} \{\overline{m} \mid \exists \overline{r} \in \overline{RCos}.\ \exists \overline{c} \in \overline{CCos}.\ (\overline{r}, \overline{m}, \overline{c}) \in \overline{X}\}, \\ & \bigsqcup_{\text{c}} \{\overline{c} \mid \exists \overline{r} \in \overline{RCos}.\ \exists \overline{m} \in \overline{MCos}.\ (\overline{r}, \overline{m}, \overline{c}) \in \overline{X}\} \Big). \end{aligned}$$

A set of abstract configurations is combined into one abstract configuration as follows. Each pair of program and pipeline configuration for which at least one of the abstract configurations

[4] If there is a cache miss during the abstract interpretation, it should always be unique because at most one instruction is dispatched in each clock cycle and, hence, at most one memory access is triggered in each clock cycle.

is defined is mapped to the triple resulting from the component-wise join across the component configurations that can occur for the respective pipeline configuration.

The join functions for the components are defined by $\bigsqcup_{\mathrm{r}}(\overline{R}) = (\lambda e.\ \{v \mid \exists \overline{r} \in \overline{R}.\ v \in \overline{r}(e)\})$, $\bigsqcup_{\mathrm{m}}(\overline{M}) = (\lambda d.\ \{v \mid \exists \overline{m} \in \overline{M}.\ v \in \overline{m}(d)\})$, and $\bigsqcup_{\mathrm{c}}(\overline{C}) = (\lambda d.\ \{v \mid \exists \overline{c} \in \overline{C}.\ v \in \overline{c}(d)\})$.

In a nutshell, the join $\bigsqcup$ combines the triples of component configurations to which the same program and pipeline are mapped in different abstract configurations by propagating the set-union operator along the structure of the triples.

The overall abstract semantics upd^{α}, thus, overapproximates the possible changes that might occur in one clock cycle during a program execution across all possible control flows at the level of our abstract pipeline domain $\overline{Cos}$. The semantics of a complete program execution is overapproximated by the fixed point $fix(upd^{\alpha}, \overline{init})$ reached by the repeated application of upd^{α} based on an initial abstract configuration $\overline{init}$.

6.3.3 Analysis Procedure

Our program analysis, which is based on the pipeline abstract domain $\overline{Cos}$ and the abstract semantics upd^{α}, is called Spectrescope. The name Spectrescope indicates that the analysis investigates quantitative properties of programs across abstraction layers like a spectroscope, which breaks down abstract light beams to measure their properties based on spectral lines.

The analysis allows one to use the abstract domain and semantics to obtain a security guarantee in the form of a leakage bound for a pASM program in an end-to-end fashion. The guarantee depends on the set of possible initial states with respect to which the pASM program might be executed. This set of initial states is specified as a set of elements from the concrete domain

$$Cos = Ps_{wf} \times RCos \times MCos \times CCos \times PCos.$$

Here, $RCos = Rs \to Vs$ are the concrete register configurations, $MCos = DAs \to Vs$ are the concrete data-memory configurations, $CCos = DAs \to [0, SIZE]$ are the concrete cache configurations, and $PCos = StCos \times RSCos \times BCos$ are the concrete pipeline configurations. Furthermore, $RSCos = RSCs^{\star}$ and $BCos = BCs^{\star}$ are the concrete reservation-station and reorder-buffer configurations, respectively, with $RSCs = UIAs \times MLs \times RLs$, $MLs = (DAs \times \mathbb{N}_0)^{\star}$, and $BCs = UIAs \times Rs \times Vs$ (see also the detailed explanation in Appendix B).

The set $Init \subset Cos$ for the analysis of a program pr is a set of states from the concrete domain, such that the instruction memory in each state is equal to pr, the fetch stage of the pipeline is mapped to i_1, all other pipeline stages are mapped to $\top$, and the reservation-station and reorder-buffer configurations are empty lists. The register and data-memory configurations differ across the states in $Init$ and capture the possible scenarios in which the program might be run.

For instance, if the memory contains a secret that might have any 32 bit value, the set $Init$ contains one state for each possible secret value. Furthermore, if the attacker controls the value of registers or memory entries, the values assigned to these registers or memory entries in the states from the set $Init$ reflect the attacker's choice. If the attacker might choose between multiple values for a register or memory entry, the set $Init$ contains multiple states that differ in the respective value. Thereby, the active capability of the attacker to supply certain program inputs, e.g., in a Spectre-PHT attack, is captured implicitly through the choice of initial states.

Spectrescope Analysis (Input: $Init$)

1. compute the initial abstract configuration $\alpha(Init)$
2. apply upd^{α} to $\alpha(Init)$ until a fixed point $fix(upd^{\alpha}, \alpha(Init))$ is reached
3. compute the concrete configurations $\gamma(fix(upd^{\alpha}, \alpha(Init)))$ that are represented by the final abstract configuration
4. extract the cache configurations from the resulting set of concrete configurations
5. count the number of distinct cache configurations
6. apply $\log_2$ and return the result as the leakage bound

Based on the set $Init$ of initial states for the program pr, the Spectrescope analysis consists of six steps. The first step is to compute the representation of the set $Init$ in the abstract domain. It is based on the abstraction function α, which formally defines the connection between the concrete and abstract domain. It is defined by $\alpha : \mathcal{P}(Cos) \to \overline{Cos}$, such that

$$\alpha(S)(pr, \overline{pi}) = \begin{cases} (\overline{r}, \overline{m}, \overline{c}) & \text{if } \exists (pr', r', m', c', pi') \in S.\ (pr = pr' \wedge \overline{pi} = \alpha_{\mathrm{pi}}(pi')) \wedge \\ & \quad \overline{r} = \bigsqcup_{\mathrm{r}}(\{\alpha_{\mathrm{r}}(r) \mid \exists (pr', r', m', c', pi') \in S. \\ & \qquad (pr = pr' \wedge \overline{pi} = \alpha_{\mathrm{pi}}(pi') \wedge r' = r)\}) \wedge \\ & \quad \overline{m} = \bigsqcup_{\mathrm{m}}(\{\alpha_{\mathrm{m}}(m) \mid \exists (pr', r', m', c', pi') \in S. \\ & \qquad (pr = pr' \wedge \overline{pi} = \alpha_{\mathrm{pi}}(pi') \wedge m' = m)\}) \wedge \\ & \quad \overline{c} = \bigsqcup_{\mathrm{c}}(\{\alpha_{\mathrm{c}}(c) \mid \exists (pr', r', m', c', pi') \in S. \\ & \qquad (pr = pr' \wedge \overline{pi} = \alpha_{\mathrm{pi}}(pi') \wedge c' = c)\}) \\ \uparrow & \text{otherwise.} \end{cases}$$

The abstract state $\alpha(S)$ is defined for each pair of program pr and abstract pipeline configuration $\overline{pi}$ that represents a pair of program and concrete pipeline configuration that can occur in the set S of concrete states. It maps each such pair $(pr, \overline{pi})$ to the abstract register, memory, and cache configurations that represent the concrete register, memory, and cache configurations of each state in S whose pipeline configuration is represented by $\overline{pi}$. The abstract configurations of the pipeline, registers, memory, and cache are defined by auxiliary abstraction functions. The auxiliary functions for the three latter components are $\alpha_{\mathrm{r}}(r) = (\lambda e.\ \{r(e)\})$, $\alpha_{\mathrm{m}}(m) = (\lambda d.\ \{m(d)\})$, and $\alpha_{\mathrm{c}}(c) = (\lambda d.\ \{c(d)\})$. That is, they simply abstract from each value by a singleton set. The abstract representations of register, memory, and cache configurations that can occur for pipeline configurations represented by the same $\overline{pi}$ are combined using the the respective join operations.

The abstract pipeline configuration is defined by

$$\alpha_{\mathrm{pi}}((stags, rs, b)) = (stags, \alpha_{\mathrm{rs}}(rs), \alpha_{\mathrm{buf}}(b)),$$

which is defined based on the reorder-buffer abstraction function

$$\alpha_{\mathrm{buf}}(b) = \begin{cases} \langle\rangle & \text{if } b = \langle\rangle \\ \langle (e, \{v\}) \rangle \bullet \alpha_{\mathrm{buf}}(b') & \text{if } b = \langle (e, v) \rangle \bullet b' \end{cases}$$

and the reservation-station abstraction function

$$\alpha_{\mathrm{rs}}(rs) = \begin{cases} \langle\rangle & \text{if } rs = \langle\rangle \\ \langle (ia, \alpha_{\mathrm{ml}}(ml), rl) \rangle \bullet \alpha_{\mathrm{rs}}(rs') & \text{if } rs = \langle (ia, ml, rl) \rangle \bullet rs' \end{cases}$$

with

$$\alpha_{\mathrm{ml}}(ml) = \begin{cases} \langle\rangle & \text{if } ml = \langle\rangle \\ \langle (\{d\}, n) \rangle \bullet \alpha_{\mathrm{ml}}(ml') & \text{if } ml = \langle (d, n) \rangle \bullet ml'. \end{cases}$$

In a nutshell, the auxiliary function α_{pi} abstracts from the data addresses in the memory lists of the reservation stations and from the values in the reorder buffer by singleton sets.

The second step of the analysis procedure is to repeatedly apply the abstract semantics upd^{α} to the abstract representation of the initial states. Once a fixed point $fix(upd^{\alpha}, \alpha(Init))$ is reached, the second step terminates. The fixed point is an abstract configuration that overapproximates all concrete configurations that can occur after the execution of the program pr.

In the third step, Spectrescope computes the concrete configurations that are represented by the abstract configuration $fix(upd^{\alpha}, \alpha(Init))$. The computation is based on the concretization function $\gamma : \overline{Cos} \to \mathcal{P}(Cos)$, such that

$$\gamma(\overline{co}) = \{(pr, r, m, c, pi) \mid \exists \overline{pi} \in \overline{PCos}.\ \exists \overline{r} \in \overline{RCos}.\ \exists \overline{m} \in \overline{MCos}.\ \exists \overline{c} \in \overline{CCos}.\ \overline{co}(pr, \overline{pi}) \downarrow \wedge\ \overline{co}(pr, \overline{pi}) = (\overline{r}, \overline{m}, \overline{c}) \wedge pi \in \gamma_{\mathrm{pi}}(\overline{pi}) \wedge r \in \gamma_{\mathrm{r}}(\overline{r}) \wedge m \in \gamma_{\mathrm{m}}(\overline{m}) \wedge c \in \gamma_{\mathrm{c}}(\overline{c})\}.$$

The set of concrete states $\gamma(\overline{co})$ contains each concrete configuration that consists of elements of the component-wise concretizations for the pipeline, registers, memory, and cache, such that the pipeline is represented by an abstract pipeline configuration that is mapped to abstract configurations that represent the remaining concrete components. The component-wise concretization is defined by auxiliary functions. The functions $\gamma_{\mathrm{r}}(\overline{r}) = \{r \mid \forall e \in Rs.\ r(e) \in \overline{r}(e)\}$ and $\gamma_{\mathrm{m}}(\overline{m}) = \{m \mid \forall d \in DAs.\ m(d) \in \overline{m}(d)\}$ concretize the registers and memory, respectively, and

$$\begin{aligned}\gamma_{\mathrm{c}}(\overline{c}) =&\{c \mid (\forall d \in DAs.\ c(d) \in \overline{c}(d)) \wedge \\ &\forall n \in \mathbb{N}_0.\ (n < 512 \Rightarrow \forall d, d' \in DAs.\ (d \neq d' \wedge c(d) = n \Rightarrow c(d') \neq n)) \wedge \\ &\forall n \in \mathbb{N}_0.\ (n < 512 \Rightarrow (\exists d \in DAs.\ c(d) = n \wedge n > 0 \Rightarrow \exists d' \in DAs.\ c(d') = n - 1))\}\end{aligned}$$

concretizes the cache lines. The function γ_{c} captures that in any reachable concrete configuration (1) two different memory blocks cannot be cached in the same cache line simultaneously (second conjunct) and (2) the occupied cache lines must be contiguous (third conjunct).

For the instruction pipeline, the auxiliary concretization function

$$\gamma_{\mathrm{pipe}}((stags, \overline{rs}, \overline{b})) = \{(stags, rs, b) \mid rs \in \gamma_{\mathrm{rs}}(\overline{rs}) \wedge b \in \gamma_{\mathrm{buf}}(\overline{b})\}$$

is defined with respect to the reorder-buffer concretization function

$$\begin{aligned}\gamma_{\mathrm{buf}}(\overline{b}) = \{b \mid |b| = |\overline{b}| \wedge \forall i \in \mathbb{N}_0.\ \forall e \in Rs.\ \forall v \in Vs.\\ ((i < |b| \wedge b[i] = (e, v)) \Rightarrow \exists V \in \mathcal{P}(Vs).\ (\overline{b}[i] = (e, V) \wedge v \in V))\}\end{aligned}$$

and with respect to the reservation-station concretization function

$$\begin{aligned}\gamma_{\mathrm{rs}}(\overline{rs}) = \{rs \mid\ & |rs| = |\overline{rs}| \wedge \forall i \in \mathbb{N}_0.\ \forall ia \in IAs.\ \forall ml \in MLs.\ \forall rl \in RLs.\\ & ((i < |rs| \wedge rs[i] = (ia, ml, rl)) \Rightarrow \exists \overline{ml} \in \overline{MLs}.\ (\overline{rs}[i] = (ia, \overline{ml}, rl) \wedge ml \in \gamma_{\mathrm{ml}}(\overline{ml})))\}\end{aligned}$$

with the auxiliary memory-list concretization function

$$\begin{aligned}\gamma_{\mathrm{ml}}(\overline{ml}) = \{ml \mid\ & |ml| = |\overline{ml}| \wedge \forall i \in \mathbb{N}_0.\ \forall d \in DAs.\ \forall n \in \mathbb{N}_0.\\ & ((i < |ml| \wedge ml[i] = (d, n)) \Rightarrow \exists D \in \mathcal{P}(DAs).\ (\overline{ml}[i] = (D, n) \wedge d \in D)))\}.\end{aligned}$$

All in all, the pipeline concretization function returns the set of all concrete pipeline configurations that only contain values in the reorder buffer and data addresses in the memory lists that occur in the sets of possible values and addresses in the abstract configuration, respectively.

In the fourth step, the analysis computes the set of all cache configurations that occur in the possible final configurations resulting from the concretization. That is, the analysis computes the set $\{c \mid\ \exists r \in RCos.\ \exists m \in MCos.\ \exists pi \in PCos.\ (pr, r, m, c, pi) \in \gamma(fix(upd^{\alpha}, \alpha(Init)))\}$.

To compute the final leakage bound in the fifth and sixth step, the analysis counts the number of elements in the set and applies the logarithm. That is, the final result of the analysis is

$$\begin{aligned}\log_2 |\{c \mid\ & \exists r \in RCos.\ \exists m \in MCos.\ \exists pi \in PCos.\\ & (pr, r, m, c, pi) \in \gamma(fix(upd^{\alpha}, \alpha(Init)))\}|.\end{aligned}$$

This overapproximates the logarithm of the number of possible side-channel observations $\log_2 |\{c \mid \exists init \in Init.\ \exists r \in RCos.\ \exists m \in MCos.\ \exists pi \in PCos.\ fix(upd, init) = (pr, r, m, c, pi)\}|$ under *acc*, which is an upper bound on the min-entropy leakage of *pr* to *acc* (see Section 2.2.1).

Overall, Spectrescope leverages our pipeline abstract domain to compute quantitative security guarantees in the form of upper bounds on the *acc* leakage of programs during pipelined executions.

6.4 Analysis Setup

We apply Spectrescope manually to our target implementations in the *p*ASM programs `p-kernel` and `p-kernel`$^{\texttt{f}}$. The set of initial states for our analysis is defined such that

- the size of `a1`, which is stored at data address d_1, is 3,
- a secret that might have any 32 bit value is stored at data address d_5,
- the cache initially only contains the memory entry at data address d_5,
- the attacker sets *eax* to 3 in order to target the secret at d_5, and
- all other registers and memory entries have the value 0.

We instantiate the parameters of Spectrescope by $HIT = 1$, $MISS = 5$, $SIZE = 512$, and $NUM = 4$.

6.5 Analysis of Vulnerability and Mitigation

The leakage bounds computed with Spectrescope for the programs `p-kernel` and `p-kernel`$^{\texttt{f}}$ are shown in Table 6.1. For the unmitigated program `p-kernel`, the leakage bound is 19.9 bit.

	`p-kernel`	`p-kernel`$^{\texttt{f}}$
leakage bound	19.9 bit	0 bit

Table 6.1: Leakage Bounds for the Programs `p-kernel` and `p-kernel`$^{\texttt{f}}$

The non-zero leakage bound shows that our analysis detects the known vulnerability corresponding to the leakage of a private value from the kernel memory. Given the size 32 bit of memory entries in our execution model, the leakage bound corresponds to 62% of one secret memory entry. That is, our analysis also captures the effect that the processing of the private value (in Line 8 and 9 of `p-kernel`) after the retrieval from the memory (in Line 7 of `p-kernel`) has on the leakage. The program `p-kernel` does not leak all 32 bit of the secret memory entry but only the partial information about these bits that remains after the processing.

The leakage bound for the hardened program `p-kernel`$^{\texttt{f}}$ is 0 bit. That is, we obtain a zero-leakage guarantee for the hardened program with respect to our attacker model. Indeed, the mitigation in `p-kernel`$^{\texttt{f}}$ eliminates the leakage completely, because it masks out exactly those indices that are out-of-bounds with respect to the array `a1` and because it introduces a dependence that prevents an out-of-order execution of the access to `a1` before the sanitization. Our analysis successfully verifies the effectiveness of this complete mitigation.

Overall, Spectrescope successfully detects a known vulnerability and verifies the effectiveness of the existing mitigation technique that is used in the Linux kernel. In principle, the detection of vulnerabilities and the verification of the absence of leakage is also possible with qualitative program analyses. The distinguishing feature of our analysis is that it provides quantitative security guarantees in the presence of leakage. In this case, it guarantees that at most 19.9 bit of a secret memory entry are leaked to a cache-side-channel attacker under *acc*. We illustrate the benefits of such quantitative security guarantees for cache-side-channels with respect to pipelined executions in more detail in the next section.

6.6 Analysis across Program Variants

Quantitative leakage bounds computed with Spectrescope can provide a basis for evaluating the effectiveness of partial mitigations and for comparing different implementations with respect to their security. We demonstrate both at the example of small *p*ASM programs.

6.6.1 Reasoning about Partial Mitigations

Recall the program `p-simp` from Example 6.3.1. The program uses the attacker-controlled input value in the register *eax* as the index for an access to the array `a1` and then uses the resulting value as the index for an access to the array `a2`. We analyze this program with Spectrescope, using the same analysis setup as described in Section 6.4. That is, the attacker supplies the input value 3 in order to trigger a leak of the secret memory entry stored at data address d_5.

The analysis of `p-simp` with Spectrescope yields the leakage bound 32 bit. This bound is precise, because the program might, indeed, leak the entire secret 32 bit value stored at d_5 in an execution on an architecture with a cache and an instruction pipeline. The unmodified value retrieved from d_5 is used as the index for the access to `a2` and the entry of `a2` uniquely determines the memory block that is accesses and, hence, loaded into the cache.

Consider a scenario in which the array `a2` has ten entries, i.e., spans the data addresses d_6 - d_{15}. In this scenario, the leakage of `p-simp` can be mitigated partially using a bit-mask. Any valid access to the array `a2` will use an index that is less than ten, i.e., an index in whose binary representation only the four least-significant bits might be set to one. The remaining 28 bit of the index can be masked out without affecting the functionality of the program.

The program `p-mask`, shown in Figure 6.7, implements this mitigation. The program only differs from the original `p-simp` by two instructions, namely the **mov-rc** instruction in Line 2 and the **and-rr** instruction in Line 6. That is, the mitigation introduces a constant, relatively low overhead of two clock cycles with respect to our execution model. Depending on the number of clock cycles required for a memory access (in our analysis setup, we consider a duration of five clock cycles), this overhead is likely lower than the overhead induced by a complete mitigation, e.g., by inserting a **fence** instruction as a speculation barrier between Line 3 and Line 4 of `p-simp`.

```
1  mov-rc ebx 0
2  mov-rc ecx 15
3  mov-rm ebx d1 ebx
4  jge i8 eax ebx
5  mov-rm eax d2 eax
6  and-rr eax ecx
7  mov-rm eax d6 eax
8  nop
```

Figure 6.7: Program `p-mask`

With Spectrescope, we obtain the leakage bound 4 bit for the program `p-mask`. That is an improvement of 87.5% compared to the unmitigated program `p-simp`.

Like the bound for `p-simp`, the leakage bound for `p-mask` is tight. When the program speculatively accesses the secret 32 bit memory entry, the 28 most-significant bits of the secret are masked out and the four least-significant bits are leaked into the cache through the access to the array `a2` in Line 7. Hence, the improvement of 87.5% in the security guarantee corresponds to an actual improvement of 87.5% in the security of the program with respect to side channels that arise from the combination of caching and pipelining.

The leakage bounds for the programs `p-simp` and `p-mask` support the reasoning about the security-performance trade-off in the mitigation of the side-channel leakage. For instance, the bounds allow one to consider meaningful questions like, e.g., (1) Is the remaining potential leakage of 4 bit tolerable in the context where the program is used? (2) Is the 87.5% improvement in security worth the overhead of two clock cycles? (3) Is the remaining leakage of 4 bit worth the clock cycles that are saved by the partial mitigation compared to a complete mitigation?

Overall, the leakage bounds allow one to consider the security dimension based on concrete quantitative reference numbers in the same way as the performance dimension of the trade-off.

6.6.2 Comparison across Implementations

Figure 6.8 shows the small example program `p-shift`. The program differs from `p-simp` by a **shr-rc** instruction in Line 5. If the program is executed with the input value $eax = 3$ and the access to `a1[eax]` is executed speculatively, the program `p-shift` retrieves a secret memory entry in the same way as `p-simp`. However, in `p-shift`, the 15 least-significant bits of the secret are shifted out before the secret is leaked into the cache in Line 6. That is, `p-shift` leaks only the 17 most-significant bits of a secret 32 bit memory entry.

```
1  mov-rc ebx 0
2  mov-rm ebx d1 ebx
3  jge i7 eax ebx
4  mov-rm eax d2 eax
5  shr-rc eax 15
6  mov-rm eax d6 eax
7  nop
```

Figure 6.8: Program `p-shift`

Applying Spectrescope to the program `p-shift` yields the leakage bound 17 bit. This corresponds exactly to the actual leakage of the program. The analysis captures the leakage precisely.

Based on the leakage bounds for `p-simp`, `p-mask`, and `p-shift`, we show how Spectrescope can be used to support the comparison of implementations and the prioritization of mitigation efforts. All three leakage bounds are shown in comparison in Table 6.2. The bounds give rise to a security ranking in which the security guarantee for `p-mask` is the best, followed by the security guarantee for `p-shift`, and finally by the guarantee for `p-simp`. The ranking based on these security guarantees reflects the actual relation between the programs in terms of leakage. The program `p-mask`, which leaks only 4 bit is the most secure, followed by `p-shift`, which leaks 17 bit, and `p-simp`, which leaks an entire 32 bit memory entry.

	`p-simp`	`p-shift`	`p-mask`
leakage bound	32 bit	17 bit	4 bit

Table 6.2: Leakage Bounds for the Programs `p-simp`, `p-shift`, and `p-mask`

Consider a bigger program in which all three smaller programs, `p-mask`, `p-shift`, and `p-simp`, occur as code snippets. Suppose these snippets occur at equally security-critical locations in the program and are executed equally frequently. The three code snippets could be identified as vulnerabilities in the bigger program using a traditional, qualitative analysis with respect to side channels that arise from the combination of caching and pipelining. However, such a qualitative analysis would reject the bigger program as insecure until all leakage is eliminated from the code snippets. With Spectrescope, the code snippets in which vulnerabilities were detected can be analyzed quantitatively. The resulting leakage bounds can provide a basis for prioritizing the mitigation of vulnerabilities meaningfully. The leakage in the code snippet corresponding to `p-simp` should be mitigated with the highest priority. Depending on the security and performance requirements for the bigger program, the mitigation efforts should proceed in decreasing order of leakage bounds with the snippets corresponding to `p-shift` and `p-mask`.

6.7 Summary

In this chapter, we presented our program analysis Spectrescope, which targets side channels that arise from the combination of caching and pipelining. The analysis can verify the absence of leakage through such side channels and, if leakage is present, provide quantitative security guarantees in the form of upper leakage bounds. This distinguishes Spectrescope from all prior program analyses for side channels based on the combination of caching and pipelining, because these analyses can only verify the absence of leakage, but cannot quantify leakage that is present.

Spectrescope is based on the novel abstract domain $\overline{Cos}$ that captures the possible states of an instruction pipeline with branch prediction and out-or-order execution, as well as their relation to the states of registers, memory, and cache. The domain is fine-grained enough to allow for precise leakage bounds. At the same time, it is coarse-grained enough to make the program analysis feasible. We successfully applied the analysis to the *p*ASM programs `p-kernel` and `p-kernel`$^{\mathtt{f}}$, which capture two Linux-kernel excerpts. We quantified the leakage through a known vulnerability that occurs in one of these excerpts and verified the effectiveness of the mitigation that is implemented in the second excerpt.

Based on small example programs, we demonstrated the additional benefits of the quantitative analysis. For instance, we showed that the partial mitigation in the program `p-mask`, which induces only very limited performance overhead, reduces side-channel leakage by 87.5% compared to the program `p-simp`. Based on the quantitative analysis with Spectrescope, we were also able to compare the leakage across the programs `p-simp`, `p-shift`, and `p-mask` and to obtain a meaningful security ranking that could support the prioritization of mitigation efforts.

Chapter 7

Related Work

In this chapter, we compare this thesis to related work. Recall that this thesis is motivated by the security-performance trade-off with respect to cache-side-channel leakage in crypto implementations. On the one hand, it has been demonstrated in numerous attacks that entire secret keys can be recovered through cache side channels. Hence, cache-side-channel leakage is a serious security risk for cryptographic implementations. On the other hand, caches are an integral part of modern computer architectures and the use of caching is indispensable from a performance perspective. Being able to reliably quantify the cache-side-channel security of a crypto implementation is crucial for enabling an informed navigation of this security-performance trade-off. Therefore, this thesis focuses on program analyses that can provide reliable quantitative bounds on the cache-side-channel leakage of crypto implementations in Chapter 3-5. We expanded the scope to side channels that arise from the combination of caching and pipelining in Chapter 6.

In Section 7.1 and Section 7.2, we discuss qualitative program analyses. The goal of these analyses is very different from the analyses in this thesis, because qualitative analyses can only provide security guarantees in the absence of leakage. In case the absence of leakage cannot be verified, qualitative analyses can point out potential vulnerabilities, but they cannot provide quantitative guarantees and, hence, do not support the navigation of the security-performance trade-off. Instead, the qualitative analyses that we discuss are related to the program analyses from this thesis because they target the same types of side channels.

In Section 7.3, we discuss the program analyses that are most closely related to this thesis. More concretely, we discuss quantitative program analyses with respect to cache side channels. Like our analyses, these analyses can provide quantitative information about the cache-side-channel security of implementations, even in the presence of leakage. We first discuss the analyses that are based on the same information-theoretic notion of leakage as our analyses (but use different abstractions) and then we discuss analyses that are based on the complementary notion of dynamic leakage. Throughout Section 7.3, we focus only on analyses with respect to cache side channels, i.e., analyses that are related to our program analyses from Chapter 3-5. We are not aware of any existing quantitative program analysis, other than our analysis from Chapter 6, that takes into account side channels that arise from the combination of caching and pipelining.

In Section 7.4, we discuss how the attacker models considered in Chapter 3-6 relate to existing attacks. Finally, in Section 7.5, we discuss how the mitigations considered in Chapter 3-5 relate to existing side-channel countermeasures.

7.1 Qualitative Cache-Side-Channel Analyses

There are multiple qualitative tools that analyze programs with respect to cache-side-channel leakage. These tools focus either on verifying the absence of leakage or on the detection of leakage.

Verifying the Absence of Leakage The tools CaSym [26] and CacheS [128] focus on verifying the absence of leakage and, in case the verification fails, on pointing out potential leakage in the analyzed program. Both tools are based on program analyses that consider only cache side channels without instruction pipelining. The cache-side-channel attacker model of CaSym is a more powerful variant of the attacker model *acc*. Under CaSym's attacker model, the attacker can observe not only the final cache state after a program execution, but also the cache states at intermediate points of the execution which can be chosen during the analysis setup. The attacker model underlying CacheS is even more powerful. It allows the attacker to observe the outcome of each branching decision, as well as a trace of all memory addresses that the analyzed program accesses. The granularity of the trace can be configured. That is, a parameter L can be specified in order to hide the L least-significant bits of each address in the trace from the attacker.

Both, CaSym and CacheS verify that the attacker observations made for each program run are independent of secret inputs. To this end, both tools use abstractions in their analyses. CaSym uses symbolic execution in combination with SMT solving and CacheS uses abstract interpretation in combination with SMT solving. The abstract domain used in CacheS is called Secret-Augmented Symbolic Domain (SAS). Unlike our abstract domains, SAS does not track the possible states of the cache. This is not necessary due to the very powerful attacker model of CacheS. In the SAS domain, all public values are abstracted from by the same symbol and secret values are tracked at a more fine-grained level, based on different symbols for different parts of the secret. The symbolic execution in CaSym tracks the cache symbolically and is parametric in the cache model. It is supported by taint tracking in order to increase the precision of treating array contents and in order to modularize the analysis.

The tool CaSym works on programs in LLVM intermediate representation. The tool CacheS translates x86 binaries into the REIL intermediate language for its analysis. Both tools have been used to analyze different cryptographic implementations qualitatively, but, in contrast to our analyses, the tools do not provide quantitative leakage bounds.

Detecting Leakage The tools DATA [133] and CacheD [129] focus on detecting cache-side-channel leakage in programs based on concrete execution traces. DATA uses dynamic binary instrumentation to record traces of the data and instruction addresses that are accessed during program executions across multiple secret inputs. These traces are analyzed in multiple stages. First, the traces are compared at Byte-address granularity to detect potential leaks. The resulting list of potential leaks is filtered (using additional traces for the same secret input) to reduce false positives caused by random variations in the traces. The remaining potential leaks are classified with respect to different leakage models (e.g., the Hamming weight model). This classification could also be seen as an implicit quantification, because the leakage models restrict the amount of information that might leaked. However, the classification is very different from our cache-side-channel quantification, because it is based on dynamically recorded execution traces and can therefore not provide upper leakage bounds like our static analysis.

The tool CacheD uses a dynamically recorded execution trace to optimize cache-side-channel detection by symbolic execution. The tool assumes an attacker model that is similar to the attacker model of CacheS. The attacker can observe the trace of memory-access addresses modulo the L least-significant bits. CacheD uses symbolic execution and SMT solving to check for secret-dependent memory-access addresses. It optimizes this analysis by replacing secret-independent memory accesses with concrete values from the execution trace. The tool has been applied to analyze different implementations of AES, RSA, and ElGamal. Like CaSym and CacheS, CacheD is purely qualitative and does not provide leakage bounds for the analyzed crypto implementations.

7.2 Qualitative Analyses for Caching and Pipelining

While CaSym, CacheS, and CacheD focus only on cache side channels, there are also some qualitative program analyses that take into account pipelining in addition to caching.

The tool Spectector [57, 58], e.g., uses symbolic execution to verify a relative non-interference property. More concretely, the tool verifies that a program is non-interferent under speculative execution if it is non-interferent under sequential execution. The analysis is based on a formal semantics that is parametric in both, the execution mode and the attacker model. The execution mode is captured based on a prediction oracle and instantiated to either sequential execution or speculative execution with "always-mispredict" branch-prediction strategy. The latter overapproximates the instructions that might be executed speculatively and is therefore more general than our semantics, which captures one concrete branch-prediction strategy. However, the abstraction to the always-mispredict strategy might hide leakage that is caused by the presence or the depth of speculative execution. The attacker model in Spectector is captured based on traces that the attacker can observe during a program execution. These traces either contain only the program counter and the addresses of memory accesses, or they contain the values of memory loads in addition. That is, the attacker model is much more powerful than the model underlying our Spectrescope analysis from Chapter 6. However, unlike Spectrescope, the analysis in Spectector does not provide quantitative security guarantees in case leakage is present.

The tool SpecSafe [27] also captures speculative execution using a prediction oracle. It verifies a variant of Spectector's speculative non-interference property that is absolute instead of relative and that quantifies over all possible prediction oracles. To this end, the tool transforms the code of the analyzed program to make speculation explicit. The transformed program is then analyzed using the tool CaSym (described in Section 7.1). Like CaSym, SpecSafe is not quantitative.

Pitchfork [28] also verifies the absence of cache-side-channel leakage under speculative execution. It is based on an operational semantics that models a three-stage pipeline, consisting of a fetch stage, an execute stage, and a retire stage. The semantics captures a broad range of micro-architectural features, including branch prediction, out-of-order execution, jump-target prediction, return-address prediction, and memory-disambiguation prediction. Both, the scheduling for the out-of-order execution and the predictions are not modeled explicitly but assumed to be under attacker control. The attacker observations in the semantics underlying Pitchfork are traces that include control-flow information (on branch targets, rollbacks, and the use of store forwarding) and the addresses of memory reads and writes. That is, the Pitchfork semantics captures more powerful attackers and a broader range of prediction strategies than our semantics. To deal with the resulting complexity in the program analysis, Pitchfork considers a sound subset of all possible schedules and performs symbolic execution for a bounded speculation depth. However, the tool can only verify the absence of side-channel leakage and not provide quantitative guarantees.

A recent extension to the Jasmin verification framework [13] also supports the verification of a speculative non-interference property. It is based on an operational semantics that captures a pipeline with branch prediction and out-of-order execution. As in the Pitchfork semantics, the scheduling and the branch prediction are assumed to be under attacker control. The attacker observations are also traces, but in case of the Jasmin extension the traces not only include control-flow information and the addresses of memory accesses, but they also include snapshots of the entire memory contents at any point where an out-of-bounds access occurs.

The tool Blade [126] is based on an operational semantics that captures a three-stage pipeline as well as just-in-time compilation. The compilation translates high-level commands into low-level commands and inserts bound checks before each array access. Blade is based on an attacker model under which the attacker can observe the addresses of memory reads and writes, control-flow information about branches and rollbacks, as well as information about exceptions raised due to out-of-bounds memory accesses. The analysis in Blade is based on a type system that is sound with respect to a relative non-interference property. If the property is not satisfied, the type system does not provide leakage bounds, but it provides a set of commands at which mitigations need to be inserted to ensure that the hardened program satisfies the property.

Finally, there are also models of speculative execution that are not used in program analyses, but which are used for reasoning about different types of vulnerabilities and mitigations [42, 56] or for the machine-checked verification of micro-architectures with respect to security properties [56].

7.3 Quantitative Cache-Side-Channel Analyses

We are not aware of any quantitative program analyses that compute leakage bounds with respect to side channels that arise from the combination of caching and pipelining. However, there are some quantitative analyses that consider cache-side-channel leakage without pipelining and are, thus, related to our analyses from Chapter 3, Chapter 4, and Chapter 5. The analyses differ in their underlying information-theoretic leakage model.

Bounds across Side-Channel Output Values Most closely related to our work are other analyses that compute upper leakage bounds across all side-channel output values. This approach is used by multiple other analyses in the CacheAudit framework: CacheAudit 0.1 [43], CacheAudit 0.2 [45], CacheAudit-MemoryTrace [44], and CacheAudit 0.3 [86]. CacheAudit 0.1 was developed by Doychev, Feld, Köpf, Mauborgne, and Reineke, CacheAudit 0.2 was developed by Doychev, Köpf, Mauborgne, and Reineke, and CacheAudit-MemoryTrace was developed by Doychev and Köpf. CacheAudit 0.3 was developed by us and our collaborators as part of the toolchain RiCaSi for cache-side-channel mitigation via selective circuit compilation [86].

All of these analyses follow the same approach as the analyses in this thesis. That is, they use abstract interpretation to overapproximate the reachable attacker observations and then compute the logarithm of the number of reachable observations as a leakage bound. This logarithm bounds the leakage measured in terms of channel capacity with respect to min-entropy [77] and, hence, also with respect to any other leakage measure in the g-framework [5, "Miracle" Theorem].

The abstractions used differ across the analyses. The abstractions in CacheAudit 0.1 and CacheAudit 0.2 are closely related to our abstraction from Chapter 3, but do not track the state of the CPU flags sufficiently precisely for analyzing a broad range of off-the-shelf AES implementations. They track only the Carry and Zero Flags at a fine-grained level and abstract from the Sign and Overflow flags completely. CacheAudit 0.1 has been used to quantify the cache-side-channel leakage of the AES implementation from mbedTLS (in a version where the library was still called PolarSSL) and an implementation of Salsa20. CacheAudit 0.2 has been used in addition to quantify the leakage of the stream ciphers HC-128, Rabbit, and Sosemanuk.

In their analysis of PolarSSL AES, Doychev, Köpf, Mauborgne, and Reineke observed, e.g., that the leakage bounds with respect to *acc*, *trace*, and *time* decrease with increasing cache size, and that the leakage bounds with respect to *accd* increase [45]. They also observed a convergence of the *acc* and *accd* leakage bounds for large caches and a positive effect of preloading. These observations match our results for mbedTLS AES. In our study, we consider a broader range of AES implementations and derive insights beyond mbedTLS, e.g., about the effects of design choices in AES implementations on the leakage bounds and about the effect of bitslicing.

CacheAudit 0.3 is based on CacheAudit-FPU (see Chapter 5), but its analysis features an abstract semantics that supports instructions that occur in circuit-based binaries and its implementation is optimized to scale to large circuit binaries. It is used as part of the toolchain RiCaSi to guide and to verify the application of circuit compilation to mitigate cache side channels.

CacheAudit-MemoryTrace features an additional cache-side-channel attacker model. The attacker model is more powerful than the attacker models considered in this thesis, because the attacker can observe the trace of memory-access addresses. Like for CacheS, the granularity at which the attacker observes these addresses can be configured. Based on this attacker model and the novel Masked-Symbol Abstract Domain, CacheAudit-MemoryTrace can quantify the cache-side-channel leakage across different implementations of ElGamal decryption. In addition, CacheAudit-MemoryTrace features a symbolic treatment of public input values.

We are not aware of program analyses outside the CacheAudit framework that compute upper bounds on cache-side-channel leakage across all side-channel output values. However, the quantitative program analysis from [112] computes such leakage bounds with respect to side channels that are based on execution time, memory usage, or network traffic in Java programs. The analysis is based on a symbolic execution that tracks the reachable attacker observations

based on cost models. To compute leakage bounds across the possible values of public inputs, the symbolic execution is combined with Max-SMT solving. Newer variants of the analysis support the synthesis of side-channel attacks (in terms of program inputs that maximize the leakage to an adaptive multi-run attacker) [110] and the analysis of probabilistic programs [85].

Dynamic Leakage of Individual Output Values The tools CHALICE [30] and Abacus [11] take an alternative approach at cache-side-channel quantification. They quantify cache-side-channel leakage based on the notion of dynamic leakage [19]. That is, they quantify the leakage based on the amount of information that is revealed by individual side-channel output values.

Both tools select the cache-side-channel output values for which to quantify leakage based on dynamically measured execution traces. Abacus is based on the same attacker model as CacheS (described in Section 7.1), i.e., an attacker who observes each branching decision and the address of each memory access modulo the L least-significant bits. CHALICE is parametric in the attacker model and supports both, trace-based and access-based models.

Both tools quantify the leakage of an output value in terms of the number of secret inputs that are ruled out by the output value. CHALICE uses the absolute number, while Abacus works on a logarithmic scale. To compute the number of ruled-out secret inputs, both tools use symbolic execution. CHALICE symbolically executes the target program and combines the resulting path conditions and symbolic representations of memory accesses with a cache model to determine which inputs might cause the cache-side-channel output value of interest. Abacus performs symbolic execution only along one path of the target program that leads to the output value of interest. Based on the branching decisions and memory accesses that occur along this path, Abacus determines which inputs might cause the output value of interest along this path.

7.4 Cache-Side-Channel Attacks

Cache-side-channel attacks were first considered by Page in 2002 [103]. Page considers an attacker who can observe cache hits and cache misses during the execution of the victim program. Similar attacks that exploit cache traces have been mounted on multiple crypto implementations, including OpenSSL AES [1] and reference implementations of the block ciphers CAMELLIA [111] and CLEFIA [115]. This type of attacks is captured by the attacker model *trace* in this thesis.

In 2005, Bernstein proposed an attack that exploits the trace of cache hits and cache misses indirectly through the overall execution time of the victim implementation [16]. The attacker model *time* in this thesis captures such attacks on an abstract level. More concretely, the duration of cache hits, cache misses, and steps without memory accesses is assumed to be constant. We abstract from the influence of further micro-architectural details on the duration.

In the above-mentioned trace- and time-based attacks, the attacker obtains information about the victim's interaction with the cache indirectly, e.g., by measuring time or power consumption. He does not interact with the cache directly. An attacker who controls a spy process on the same system as the attacked program might also mount access-based attacks, in which the attacker's spy process interacts with the cache directly. To enable such attacks, the attacker could, e.g., force the co-location of his spy process and the target program on the same machine in the cloud [117].

Two early access-based attack techniques are Prime+Probe and Evict+Time, proposed by Osvik, Shamir, and Tromer in 2006 [102]. The key idea is that the attacker enforces a controlled initial state of the cache before the victim executes and then monitors the changes to the cache state during the victim execution. Multiple, increasingly sophisticated techniques following this pattern have been developed since then. Gullasch, Bangerter, and Krenn describe an asynchronous attack technique [59]. Here, the victim and attacker process are interleaved. The attacker can manipulate the cache state at a more fine-grained level and make observations about changes to the cache state also during the victim execution. Yarom and Falkner build on this technique and use a flush instruction to efficiently remove individual entries from the cache for a fine-grained

monitoring [135]. They call their technique Flush+Reload. The Flush+Reload technique has been used as a building block in multiple other attacks, including the first side-channel attack on a lattice-based signature scheme [54] and the initial Meltdown [81] and Spectre attacks [72].

While techniques like, e.g., Prime+Probe and Flush+Reload use memory accesses by the attacker either to control or to monitor the cache state, Flush+Flush [55] is the first technique that works without memory accesses by the attacker. It uses a flush instruction to control the cache state and at the same time exploits the secret-dependent execution time of the flush instruction to monitor changes to the cache state. Reload+Refresh [25] is a stealthy technique for access-based cache-side-channel attacks, which exploits knowledge about the replacement policy of the shared cache to avoid detection by mechanisms that check for unusually high cache-miss rates. The very recent Prime+Prune+Probe technique [113] enables optimized access-based attacks that can even target caches that are protected by randomization techniques.

Our access-based attacker model *acc* is independent of the technique used to monitor the cache state. It captures synchronous attacks that apply any technique to extract the contents of the final cache. The model does, however, not capture attacks that apply the techniques asynchronously, i.e., interleaved with the execution of the attacked program. The model *accd* captures the same attacks as *acc*, but in the case where the attacker's spy process and the process of the attacked program do not share common memory blocks.

Recently, cache-side-channel attacks have been exploited in combination with instruction pipelining. Prominent examples include Spectre-PHT [72], Spectre-BTB [72], Spectre-RSB [84], and Spectre-STL [60]. All of these attacks exploit cache-side-channel leakage that happens during speculative execution based on a prediction mechanism in the instruction pipeline. Spectre-PHT exploits branch prediction, Spectre-BTB exploits jump-target prediction, Spectre-RSB exploits return-address prediction, and Spectre-STL exploits memory-disambiguation prediction.

Our attacker model in Chapter 6 captures cache-side-channel leakage to access-based attackers in the presence of an instruction pipeline with out-of-order execution and branch prediction. This includes, e.g., the Spectre-PHT attack, but also covers any other synchronous access-based attack that is mounted on a system captured by our pipeline model.

7.5 Cache-Side-Channel Countermeasures

Countermeasures against cache-side-channel leakage can be taken at multiple levels of abstraction, ranging from the hardware level (e.g., [105, 130]), to the hypervisor level (e.g., [70]), to the software level. In the following, we focus on software-level countermeasures only, because they are most closely related to the cache-side-channel mitigations considered in this thesis.

Program rewriting can be used to eliminate secret-dependent control flow or memory accesses, e.g., with formal program transformations. Prominent examples include, e.g., the cross-copying transformation [2] and its improved variant called unification [76]. These techniques establish a uniform timing behavior across two branches by inserting copies of instructions from each of the branches into the respective other branch. To preserve functionality, the copied instructions operate on dummy variables. Conditional assignment is a program transformation that replaces conditional branches by secret-dependent masking [95]. Traditionally, the effectiveness of program transformations is evaluated with respect to formal models of execution. More recently, this approach has been complemented with experimental evaluation of both, the effectiveness of program transformations and the overhead that they induce [88].

While the above-mentioned program transformations were originally designed to counter timing side channels, the cache-side-channel mitigations in Chapter 4 and Chapter 5 are inspired by these transformations. The mitigations in the functions `test_rejection` and `test_w` of the ring-TESLA implementation use a technique similar to conditional assignment. The mitigations avoid branches that depend on the secret coefficients of a polynomial, because these branches would lead to the secret-dependent presence of memory accesses. To this end, the mitigations

encode the branching conditions into bit masks based on each coefficient. They use these masks to accumulate the result of the rejection sampling across all coefficients of the polynomial. Similarly, the mitigation in the function `mod2dense_multiply` of the QKD implementation uses a bit-mask to avoid the secret-dependent presence of memory accesses in a matrix multiplication. The program transformation unification inspired our technique for avoiding false positives in the analysis of the function `iterprp` of the QKD implementation. To avoid secret-dependent cache traces during the handling of overflows, we inserted instructions for overflow handling on dummy values to unify the behavior of the implementation across cases with and without overflows.

The above-mentioned program transformations locally address instructions within an implementation that might lead to side-channel leakage. There are also mitigation techniques that affect the implementation style as a whole. Bitslicing [20] is a technique that consists of (1) implementing an algorithm by simulating a circuit of logical gates in software and (2) processing multiple inputs in parallel by slicing them into individual bits and combining the i-th bit of each input together as one input to the circuit. Bitslicing has been used, e.g., to implement AES [68]. This original bitslicing technique is one of the countermeasures that we investigated in Chapter 3. Since creating a bitsliced implementation requires significant time and expertise, there are also initial efforts to leverage circuit compilers from the area of secure computation to automatically transform insecure crypto implementations into circuit-based variants without parallelization. The toolchain RiCaSi leverages CacheAudit 0.3 (described in Section 7.3) for this purpose [86].

There are also mitigation techniques that are specialized to certain algorithms, e.g., masking, blinding, and preloading. The masking technique randomizes secret values, e.g., by Boolean masking, which adds the masks to the values using XOR [118], or by masking based on secret sharing, which splits secret values into shares that are processed separately and that require an attacker to learn all shares in order to recover the secret [53]. Blinding is a technique that applies a transformation to the secret value before applying the algorithm and a corresponding inverse transformation to the result [31]. For instance, exponent blinding and ciphertext blinding can be used in RSA implementations [71]. Preloading as, e.g., in [24] is a countermeasure for lookup-table-based implementations, e.g., implementations of AES, that might leak due to secret-dependent table lookups. The key idea is to access all table elements before starting the actual computation, so that they will be cached independently of the secret. Out of these algorithm-specific countermeasures, we considered preloading (in combination with cache locking [91]) in our comparative study across AES implementations in Chapter 3.

Chapter 8

Conclusion and Outlook

8.1 Conclusion

Cache side channels pose a serious threat to cryptographic implementations. This thesis was motivated by the tension between the conflicting goals of securing implementations against cache-side-channel leakage and optimizing implementations for maximum performance.

Reliable quantitative information about both, security and performance, would allow one to navigate this security-performance trade-off in an informed fashion. Quantitative information about the performance of crypto implementations can be obtained, even for complex implementations, by measuring average cycle counts. Quantifying the security of complex implementations reliably is, however, very challenging. In principle, reliable upper bounds on the leakage across all possible cache-side-channel output values can be computed by combining information theory and abstract interpretation. However, the standard abstractions for the computation of such bounds, as well as the state-of-the-art abstraction from [45], were very limited in their applicability to real-world crypto implementations. The goal of this thesis was to enable the reliable quantification of cache-side-channel leakage across real-world cryptographic implementations.

The limiting factor of existing analyses was the underlying abstract formal execution model. As usual for formal models, the existing abstractions captured only selected parts of the instruction-set architecture and micro-architecture of execution platforms. The execution of increasingly complex crypto implementations, however, involves an increasing range of instructions and micro-architectural components. For instance, the execution of different AES implementations makes use of different CPU status flags. The signature scheme ring-TESLA [3] uses large parameters (the maximum key size is 192 times bigger than for AES), so that its implementation makes use of instructions that handle large operands. The software parts of quantum key distribution are based on the computation of probabilities, so that the corresponding implementations like, e.g., the implementation from [100] make use of floating-point instructions.

In this thesis, we proposed multiple novel abstract domains and abstract semantics to incrementally lift the restrictions posed by the existing abstractions. Our abstract semantics $\texttt{aux-upd}_{\overline{\mathcal{D}}_{32}}$, proposed in Chapter 3, captures the sign and overflow status flags of x86 CPUs. Our abstract semantics $\texttt{upd}'_{\overline{\mathcal{D}}_{32}}$, proposed in Chapter 4, captures the behavior of multiple x86 instructions that handle large operands. Our abstract domain $\mathcal{D}_{64}$, proposed in Chapter 5, captures the possible states of an x86 CPU with an FPU execution unit alongside the regular ALU.

We automated the program analyses based on all three of our abstractions in the form of analysis tools in the CacheAudit framework. We applied the analyses across multiple case studies that allowed us to obtain quantitative security guarantees for real-world cryptographic implementations, ranging from different AES implementations, to the ring-TESLA implementation, to the QKD implementation from [100]. That is, our case studies cover a variety of cryptographic

tasks, ranging from encryption, to digital signatures, to key exchange. All crypto primitives that we considered are relevant also in a post-quantum scenario. The attacker models that we considered capture a broad range of attacks, including, e.g., synchronous access-based attacks independently of the measurement technique (e.g., PRIME+PROBE or EVICT+TIME) they apply.

In our case studies, we computed multiple quantitative security guarantees in the form of leakage bounds. Table 8.1 gives an overview of the leakage bounds presented in this thesis. For the AES implementations, we computed leakage bounds with respect to multiple cache sizes, but for readability we list the bounds with respect to one cache size (128 KiB) only.

	attacker model			
crypto implementation	*acc*	*accd*	*trace*	*time*
AES encryption				
OpenSSL	64.0 bit	64.0 bit	196.0 bit	7.7 bit
OpenSSL (with preloading)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
mbedTLS	69.0 bit	69.0 bit	199.0 bit	7.7 bit
mbedTLS (with preloading)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
NaCl (bitsliced)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
Nettle	69.0 bit	69.0 bit	199.0 bit	7.7 bit
Nettle (with preloading)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
LibTomCrypt	129.0 bit	129.0 bit	198.0 bit	7.7 bit
LibTomCrypt (with preloading)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
ring-TESLA				
signature generation	12.9 bit	2.6 bit	51.6 bit	9.5 bit
`test_w` (unmitigated)	31.0 bit	31.0 bit	49 152.0 bit	19.3 bit
`test_w` (mitigatied)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
`test_rejection` (unmitigated)	31.0 bit	31.0 bit	10.1 bit	10.1 bit
`test_rejection` (mitigatied)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
QKD implementation				
encoding (unmitigated)	0.0 bit	0.0 bit	4.0 bit	2.4 bit
encoding (mitigated)	0.0 bit	0.0 bit	0.0 bit	0.0 bit
decoding	0.0 bit	0.0 bit	0.0 bit	0.0 bit
privacy amplification	0.0 bit	0.0 bit	0.0 bit	0.0 bit

Table 8.1: Overview of Cache-Side-Channel Leakage Bounds across Implementations

As visible in the overview of the leakage bounds, our analyses were able to verify the absence of leakage in multiple cases, e.g., for the AES implementations with bitslicing and preloading. For the implementations in which leakage is present, our analyses were able to compute quantitative security guarantees. These guarantees can serve as a basis for reasoning about the leakage across different crypto implementations that are only partially secure. For instance, we used the leakage bounds for the AES implementations without preloading and bitslicing as a basis to build up an improved understanding about the security impact of design choices in the implementations.

The insights that we gained in our study across AES implementations, e.g., clarify the impact of the lookup tables that are used to implement the last round of AES encryption. Out of the implementation techniques considered in our study, the best security guarantees with respect to access-based attackers were achieved by reusing the same tables as in the main rounds and masking out the effect of the MixColumns transformation. Across all considered implementation techniques, the security guarantees with respect to access-based attackers stabilize as soon as the mapping from the memory blocks to cache sets becomes injective. This insight, e.g., simplifies the navigation of the security-performance trade-off for lookup-table-based AES implementations

across platforms or hardware generations with different cache sizes.

In the cases of the ring-TESLA and QKD implementations, our leakage bounds provided the basis for the detection of multiple unknown vulnerabilities. The vulnerabilities detected in the ring-TESLA implementation might allow an attacker to break the signature scheme based on cache-side-channel observations. The vulnerability that we detected in the QKD implementation might leak the entire secret key that is established during the key exchange to a cache-side-channel attacker. In both cases, the detected vulnerabilities were considered very serious by the cryptographers and the physicists, respectively, who maintain the implementations. The mitigations that we discussed in this thesis and whose effectiveness we verified with our analyses were integrated into both implementations. The implementation of the ring-TESLA successor qTESLA [22], which also includes the new cache-side-channel mitigations, advanced to Round 2 of the NIST PQC standardization process [4]. The hardened version of the QKD implementation is used for research on QKD setups at the Department of Physics at TU Darmstadt [33].

In Chapter 6, we broadened the scope of this thesis to quantifying the leakage with respect to side channels like, e.g., Spectre-PHT [72] that arise from the combination of caching and pipelining. With Spectrescope, we proposed the first analysis that computes quantitative leakage bounds with respect to this type of side channels. The analysis is based on our abstract domain $\overline{Cos}$ and abstract semantics upd^{α} that capture the relation between the instruction pipeline and multiple other architectural and micro-architectural components, including the cache.

The definition of $\overline{Cos}$ is a sweet spot in terms of both, precision and performance of the leakage-bound computation. It allowed us to compute a bound on the leakage through a known vulnerability that occurred in a prior version of the Linux kernel and to verify the effectiveness of the mitigation that is deployed in the Linux kernel. Moreover, the quantitative nature of our analysis allowed us to verify that a partial mitigation in an example program reduced the side-channel leakage by 87.5% at comparatively low performance cost.

Overall, the analysis suite presented in this thesis enables the quantification of cache-side-channel leakage across a range of real-world crypto implementations and, for the first time, the quantification of side-channel leakage that arises from the combination of caching and pipelining.

8.2 Outlook

Based on the program analyses and case studies presented in this thesis, there are multiple interesting directions for future work that we discuss briefly in the following.

Quantifying Side Channels across Additional Types of Crypto Throughout this thesis we experienced that different types of crypto implementations pose different challenges to quantitative cache-side-channel analysis. We covered multiple key types of cryptography in this thesis. In the future, it would be interesting to target additional crypto implementations and to explore which additional requirements different types of crypto pose for the analysis. Moreover, it would be interesting to enable the quantitative evaluation of architecture-specific side-channel mitigation techniques. For instance, the different levels of mitigation against Spectre attacks supported by the Intel C++ compiler [64, p.160] would be an interesting target. Quantitative leakage bounds across the mitigation levels would support the informed selection between these levels. A candidate approach to supporting different architecture-specific mitigation techniques would be to develop an abstract domain and semantics that are based on a parametric representation of mitigation techniques and that can be refined with models of the relevant instructions, e.g., Intel's CMOVcc instruction.

Augmenting Spectrescope across Prediction Techniques To explore the quantification of side channels that arise from the combination of caching and pipelining, we developed the program analysis Spectrescope. In the underlying abstract domain, we focused on a four-stage

pipeline with static branch prediction. Since the results that we obtained with Spectrescope are very promising, it will be an interesting direction to generalize the analysis beyond the pipeline architecture that we considered in this thesis. For instance, making the execution model parametric in the amount of pipeline stages or in the branch-prediction strategy would allow one to obtain analysis results for a broader range of architectures. To include a parametric branch-prediction strategy, one possible approach might be to adapt the prediction-oracle technique used in the qualitative analyses of Spectector [57] and SpecSafe [27]. In addition to covering more branch prediction strategies, it would be interesting to augment the abstract domain and semantics to capture additional prediction techniques like, e.g., jump-target prediction and return-address prediction.

Improving the Information-Theoretic Leakage Bounds In this thesis, we incrementally augmented the abstract interpretation that overapproximates the reachable attacker observations for the cache-side-channel quantification. Throughout the process, we relied on the same information-theoretic concepts to compute leakage bounds based on these observations. In the future, it would be interesting to explore how different information-theoretic notions can help to improve the leakage bounds. To improve precision, incorporating a more fine-grained notion of information worth, e.g., based on the concepts from [6] would be an interesting direction. For the analysis of quantum cryptography, it might also be interesting to explore connections to QQIF [7], which applies concepts from the context of quantum systems to information theory.

Bibliography

[1] O. Aciiçmez and Ç. K. Koç. Trace-Driven Cache Attacks on AES (Short Paper). In *Proceedings of the 8th International Conference on Information and Communications Security (ICICS)*, LNCS 4307, pages 112–121. Springer, 2006.

[2] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 40–53. Association for Computing Machinery, 2000.

[3] S. Akleylek, N. Bindel, J. Buchmann, J. Krämer, and G. A. Marson. An Efficient Lattice-Based Signature Scheme with Provably Secure Instantiation. In *Proceedings of the 9th International Conference on Cryptology in Africa (AFRICACRYPT)*, LNCS 9646, pages 44–60. Springer, 2016.

[4] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone. NIST Internal Report 8240: Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process, 2019. URL: `https://doi.org/10.6028/NIST.IR.8240`.

[5] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring Information Leakage using Generalized Gain Functions. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 265–279. IEEE Computer Society, 2012.

[6] M. S. Alvim, A. Scedrov, and F. B. Schneider. When Not All Bits Are Equal: Worth-Based Information Flow. In *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST)*, LNCS 8414, pages 120–139. Springer, 2014.

[7] A. Américo and P. Malacaria. QQIF: Quantum Quantitative Information Flow (invited paper). In *Proceedings of the 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 261–270. IEEE Computer Society, 2020.

[8] ARM Ltd. ARM® Cortex™-A Series - Programmer's Guide. Version 4.0, 2013.

[9] ARM Ltd. mbed TLS (Version 2.2.1-gpl). `https://tls.mbed.org/download/mbedtls-2.2.1-gpl.tgz`, 2016. [Online; accessed Feb-18-2022].

[10] Atmel Corporation. Rad Tolerant Devices. `http://www.atmel.com/products/rad-hard/rad-tolerant-devices/`, 2016. [Online; accessed Mar-21-2017].

[11] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu. Abacus: Precise Side-Channel Analysis. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 797–809. IEEE Computer Society, 2021.

[12] E. Barker. NIST Special Publication 800-57 Part 1, Revision 4: Recommendation for Key Management - Part 1: General, 2016. URL: `https://doi.org/10.6028/NIST.SP.800-57pt1r4`.

[13] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. High-Assurance Cryptography in the Spectre Era. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, pages 1884–1901. IEEE Computer Society, 2021.

[14] C. H. Bennett and G. Brassard. Quantum cryptography: Public key distribution and coin tossing. In *Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing (CSSP)*, pages 175–179. IEEE Computer Society, 1984.

[15] C. H. Bennett, G. Brassard, and J.-M. Robert. Privacy Amplification by Public Discussion. *SIAM Journal on Computing (SICOMP)*, 17(2):210–229, 1988.

[16] D. J. Bernstein. Cache-timing attacks on AES. Technical report, University of Illinois at Chicago, 2005.

[17] D. J. Bernstein, J. Breitner, D. Genkin, L. Groot Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Proceedings of the 19th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 10529, pages 555–576. Springer, 2017.

[18] D. J. Bernstein, T. Lange, and P. Schwabe. The Security Impact of a New Cryptographic Library. In *Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, LNCS 7533, pages 159–176. Springer, 2012.

[19] N. Bielova. Short Paper: Dynamic leakage: A Need for a New Quantitative Information Flow Measure. In *Proceedings of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 83–88. Association for Computing Machinery, 2016.

[20] E. Biham. A Fast New DES Implementation in Software. In *Proceedings of the 4th International Workshop on Fast Software Encryption (FSE)*, LNCS 1267, pages 260–272. Springer, 1997.

[21] N. Bindel. *On the Security of Lattice-Based Signature Schemes in a Post-Quantum World*. Phd thesis, TU Darmstadt, 2018.

[22] N. Bindel, S. Akleylek, E. Alkim, P. S. L. M. Barreto, J. Buchmann, E. Eaton, G. Gutoski, J. Krämer, P. Longa, H. Polat, J. E. Ricardini, and G. Zanon. Submission to NIST's post-quantum project (2nd round): lattice-based digital signature scheme qTESLA. `https://qtesla.org/wp-content/uploads/2019/11/qTESLA_round2_11.09.2019.pdf`, 2019. [Online; accessed Feb-18-2022].

[23] N. Bindel, J. Buchmann, J. Krämer, H. Mantel, J. Schickel, and A. Weber. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In *Postproceedings of the 10th International Symposium on Foundations & Practice of Security (FPS)*, LNCS 10723, pages 225–241. Springer, 2017.

[24] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006(52):1–17, 2006.

[25] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, pages 1967–1984. USENIX Association, 2020.

[26] R. L. Brotzman, S. L. Liu, D. Zhang, G. Tan, and M. T. Kandemir. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, pages 505–521. IEEE Computer Society, 2018.

[27] R. L. Brotzman, D. Zhang, M. T. Kandemir, and G. Tan. SpecSafe: Detecting Cache Side Channels in a Speculative World. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 129:1–129:28. Association for Computing Machinery, 2021.

[28] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 913–926. Association for Computing Machinery, 2020.

[29] C. Celio, J. Zhao, A. Gonzalez, and B. Korpan. RISCV-BOOM Documentation. Technical report, University of California, Berkeley, 2021.

[30] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the Information Leakage in Cache Attacks via Symbolic Execution. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(1):1–27, 2019.

[31] D. Chaum. Blind Signatures for Untraceable Payments. In *Proceedings of the 2nd International Cryptology Conference (CRYPTO)*, pages 199–203. Springer, 1982.

[32] J. Y. Cho, T. Szyrkowiec, and H. Griesser. Quantum key distribution as a service. In *Proceedings of the 7th International Conference on Quantum Cryptography (QCrypt)*, pages 1–3, 2017.

[33] Collaborative Research Center CROSSING. Project Area Primitives - Project P4 - Quantum Key Hubs. `https://www.crossing.tu-darmstadt.de/research_crossing/project_areas/primitives/p4/index.en.jsp`, 2020. [Online; accessed Feb-25-2022].

[34] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, pages 238–252. Association for Computing Machinery, 1977.

[35] P. Cousot, R. Cousot, and L. Mauborgne. Theories, solvers and static analysis by abstract interpretation. *Journal of the ACM (JACM)*, 59(6):1–56, 2013.

[36] T. M. Cover and J. A. Thomas. *Elements of Information Theory, Second Edition*. Wiley-Interscience, New York, NY, USA, 2006.

[37] J. Daemen and V. Rijmen. AES submission document on Rijndael, Version 2, 1999.

[38] U. Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. Phd thesis, Universität des Saarlandes, 2012.

[39] Deutscher Bundestag. Hochsicheres Quantennetzwerk QuNET. Drucksache 19/18355, 2020.

[40] F. Dewald, H. Mantel, and A. Weber. AVR Processors as a Platform for Language-Based Security. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*, LNCS 10492, pages 427–445. Springer, 2017.

[41] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[42] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely. The Code that Never Ran: Modeling Attacks on Speculative Evaluation. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, pages 1238–1255. IEEE Computer Society, 2019.

[43] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, pages 431–446. USENIX Association, 2013.

[44] G. Doychev and B. Köpf. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementations (PLDI)*, pages 406–421. Association for Computing Machinery, 2017.

[45] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Transactions on Information and System Security (TISSEC)*, 18(1):4:1–4:32, 2015.

[46] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice Signatures and Bimodal Gaussians. In *Proceedings of the 33rd International Cryptology Conference (CRYPTO)*, LNCS 8042, pages 40–56. Springer, 2013.

[47] L. Ducas and P. Q. Nguyen. Learning a Zonotope and More: Cryptanalysis of NTRUSign Countermeasures. In *Proceedings of the 18th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, LNCS 7658, pages 433–450. Springer, 2012.

[48] S. Euler, M. Beier, M. Sinther, and T. Walther. Spontaneous Parametric Down-Conversion in Waveguide Chips for Quantum Information. *AIP Conference Proceedings*, 1363(1):323–326, 2011.

[49] F. Flam. Quantum Cryptography's Only Certainty: Secrecy. *Science*, 253(5022):858–858, 1991.

[50] C.-H. F. Fung, X. Ma, and H. F. Chau. Practical issues in quantum-key-distribution postprocessing. *Physical Review A*, 81(1):012318–1 – 012318–15, 2010.

[51] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, 1962.

[52] GitHub, Inc. Forks of radfordneal/LDPC-codes. `https://github.com/radfordneal/LDPC-codes/network/members`, 2019. [Online; accessed Feb-18-2022].

[53] L. Goubin and A. Martinelli. Protecting AES with Shamir's Secret Sharing Scheme. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 6917, pages 79–94. Springer, 2011.

[54] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. Flush, Gauss, and Reload – A Cache Attack on the BLISS Lattice-Based Signature Scheme. In *Proceedings of the 18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 9813, pages 323–345. Springer, 2016.

[55] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, LNCS 9721, pages 279–299. Springer, 2016.

[56] R. Guanciale, M. Balliu, and M. Dam. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, pages 1853–1869. Association for Computing Machinery, 2020.

[57] M. Guarnieri, B. Köpf, J.-F. Morales, J. Reineke, and A. Sanchez. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, pages 1–19. IEEE Computer Society, 2020.

[58] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. Hardware-Software Contracts for Secure Speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, pages 1868–1883. IEEE Computer Society, 2021.

[59] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, pages 490–505. IEEE Computer Society, 2011.

[60] J. Horn. Issue 1528: speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018. [Online; Accessed Feb-02-2022].

[61] Institute for Quantum Optics and Quantum Information, Austrian Academy of Sciences. QUAPITAL: Building the first reliable Quantum Internet on top of Europe's glass fiber network. `https://quapital.eu/`, 2020. [Online; Accessed Feb-18-2022].

[62] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-032, 2016.

[63] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. Order Number: 325462-069US, 2019.

[64] Intel Corporation. Intel® C++ Compiler Classic Developer Guide and Reference. Compiler version 2021.5, 2021.

[65] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, pages 591–604. IEEE Computer Society, 2015.

[66] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a Minute! A fast, Cross-VM Attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, LNCS 8688, pages 299–319. Springer, 2014.

[67] P. Jouguet and S. Kunz-Jacques. High performance error correction for quantum key distribution using polar codes. *Quantum Information & Computation (QIC)*, 14(3-4):329–338, 2014.

[68] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 5747, pages 1–17. Springer, 2009.

[69] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, Boca Raton, FL, USA, 2015.

[70] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security)*, pages 189–204. USENIX Association, 2012.

[71] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference (CRYPTO)*, LNCS 1109, pages 104–113. Springer, 1996.

[72] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, pages 1–19. IEEE Computer Society, 2019.

[73] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO)*, LNCS 1666, pages 388–397. Springer, 1999.

[74] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Deconstructing New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures (CSAW)*, pages 25–34. Association for Computing Machinery, 2008.

[75] J. Kong, O. Aciiçmez, J.-P. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th International Conference on High-Performance Computer Architecture (HPCA)*, pages 393–404. IEEE Computer Society, 2009.

[76] B. Köpf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *International Journal of Information Security (IJIS)*, 6(2–3):107–131, 2007.

[77] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic Quantification of Cache Side-Channels. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, LNCS 7358, pages 564–580. Springer, 2012.

[78] B. Köpf and G. Smith. Vulnerability Bounds and Leakage Resilience of Blinded Cryptography under Timing Attacks. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, pages 44–56. IEEE Computer Society, 2010.

[79] LibTom Projects. LibTomCrypt (Version 1.17). `https://github.com/libtom/libtomcrypt/archive/1.17.tar.gz`, 2010. [Online; accessed Feb-18-2022].

[80] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 549–564. USENIX Association, 2016.

[81] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 973–990. USENIX Association, 2018.

[82] V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In *Proceedings of the 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, LNCS 6110, pages 1–23. Springer, 2010.

[83] D. J. C. MacKay and R. M. Neal. Near Shannon limit performance of low density parity check codes. *Electronics Letters*, 32(18):1645–1646, 1996.

[84] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2109–2122. Association for Computing Machinery, 2018.

[85] P. Malacaria, M. Khouzani, C. S. Păsăreanu, Q.-S. Phan, and K. S. Luckow. Symbolic Side-Channel Analysis for Probabilistic Programs. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*, pages 313–327. IEEE Computer Society, 2018.

[86] H. Mantel, L. Scheidel, T. Schneider, A. Weber, C. Weinert, and T. Weißmantel. RiCaSi: Rigorous Cache Side Channel Mitigation via Selective Circuit Compilation. In *Proceedings of the 19th International Conference on Cryptology and Network Security (CANS)*, LNCS 12579, pages 505–525. Springer, 2020.

[87] H. Mantel, J. Schickel, A. Weber, and F. Weber. How Secure is Green IT? The Case of Software-Based Energy Side Channels. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, LNCS 11098, pages 218–239. Springer, 2018.

[88] H. Mantel and A. Starostin. Transforming Out Timing Leaks, More or Less. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*, LNCS 9326, pages 447–467. Springer, 2015.

[89] H. Mantel, A. Weber, and B. Köpf. A Systematic Study of Cache Side Channels across AES Implementations. In *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS)*, LNCS 10379, pages 213–230. Springer, 2017.

[90] MIPS Technologies Inc. MIPS® Architecture For Programmers - Volume I-A: Introduction to the MIPS32® Architecture. Document Number: MD00082. Revision 6.01, 2014.

[91] S. Mittal. A Survey of Techniques for Cache Locking. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 21(3):49:1–49:24, 2016.

[92] S. F. Mjølsnes. *A Multidisciplinary Introduction to Information Security.* CRC Press, Boca Raton, FL, USA, 2017.

[93] O. K. J. Mohammad and S. Abbas. Detailed quantum cryptographic service and data security in cloud computing. In *Proceedings of the 1st International Conference on Computing (ICC)*, pages 43–56. Springer, 2019.

[94] N. Möller. Nettle (Version 3.2). `https://ftp.gnu.org/gnu/nettle/nettle-3.2.tar.gz`, 2016. [Online; accessed Feb-18-2022].

[95] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Proceedings of the 9th International Conference on Information Security and Cryptology (ICISC)*, LNCS 3935, pages 156–168. Springer, 2006.

[96] National Institute of Standards and Technology. FIPS PUB 197: Advanced Encryption Standard (AES), 2001.

[97] R. M. Neal. Software for Low Density Parity Check Codes, Version 2012-02-11. `http://www.cs.utoronto.ca/~radford/ftp/LDPC-2012-02-11/`, 2012. [Online, accessed Feb-18-2022].

[98] P. Q. Nguyen and O. Regev. Learning a Parallelepiped: Cryptanalysis of GGH and NTRU Signatures. In *Proceedings of the 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, LNCS 4004, pages 271–288. Springer, 2006.

[99] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 2011.

[100] P. Notz, O. Nikiforov, and T. Walther. Software bundle for data post-processing in a quantum key distribution experiment. Technical report, TU Darmstadt, 2020. URL: `https://doi.org/10.25534/tuprints-00014042`.

[101] OpenSSL Software Foundation. OpenSSL (Version 1.0.1t). `https://www.openssl.org/source/openssl-1.0.1t.tar.gz`, 2016. [Online; accessed Feb-18-2022].

[102] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the 6th Cryptographer's Track at the RSA Conference (CT-RSA)*, LNCS 3860, pages 1–20. Springer, 2006.

[103] D. Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive*, 2002(169):1–23, 2002.

[104] D. Page. Defending Against Cache-Based Side-Channel Attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

[105] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive*, 2005(280):1–14, 2005.

[106] D. Pearson. High-speed QKD Reconciliation using Forward Error Correction. *AIP Conference Proceedings*, 734(1):299–302, 2004.

[107] C. Percival. Cache missing for fun and profit. In *Proceedings of the 2nd BSD Conference (BSDCan)*, pages 1–13, 2005.

[108] R. A. Perlner and D. A. Cooper. Quantum Resistant Public Key Cryptography: A Survey. In *Proceedings of the 8th Symposium on Identity and Trust on the Internet (IDtrust)*, pages 85–93. Association for Computing Machinery, 2009.

[109] P. Pessl, L. Groot Bruinderink, and Y. Yarom. To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1843–1855. Association for Computing Machinery, 2017.

[110] Q.-S. Phan, L. Bang, C. S. Păsăreanu, P. Malacaria, and T. Bultan. Synthesis of Adaptive Side-Channel Attacks. In *Proceedings of the 30th Computer Security Foundations Symposium (CSF)*, pages 328–342. IEEE Computer Society, 2017.

[111] R. Poddar, A. Datta, and C. Rebeiro. A cache trace attack on CAMELLIA. In *Proceedings of the 1st International Conference on Security Aspects in Information Technology (InfoSecHiComNet)*, LNCS 7011, pages 144–156. Springer, 2011.

[112] C. S. Păsăreanu, Q.-S. Phan, and P. Malacaria. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*, pages 387–400. IEEE Computer Society, 2016.

[113] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, pages 987–1002. IEEE Computer Society, 2021.

[114] A. Purnal, F. Turan, and I. Verbauwhede. Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, pages 2906–2920. Association for Computing Machinery, 2021.

[115] C. Rebeiro and D. Mukhopadhyay. Differential cache trace attack against CLEFIA. *IACR Cryptology ePrint Archive*, 2010(012):1–11, 2010.

[116] R. Renner. *Security of Quantum Key Distribution.* Phd thesis, ETH Zürich, 2005.

[117] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212. Association for Computing Machinery, 2009.

[118] M. Rivain and E. Prouff. Provably Secure Higher-Order Masking of AES. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 6225, pages 413–427. Springer, 2010.

[119] R. L. Rivest, A. Shamir, and L. M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, 1978.

[120] M. Sasaki, M. Fujiwara, H. Ishizuka, W. Klaus, K. Wakui, M. Takeoka, S. Miki, T. Yamashita, Z. Wang, A. Tanaka, K. Yoshino, Y. Nambu, S. Takahashi, A. Tajima, A. Tomita, T. Domeki, T. Hasegawa, Y. Sakai, H. Kobayashi, T. Asai, K. Shimizu, T. Tokura, T. Tsurumaru, M. Matsui, T. Honjo, K. Tamaki, H. Takesue, Y. Tokura, J. F. Dynes, A. R. Dixon, A. W. Sharpe, Z. L. Yuan, A. J. Shields, S. Uchikoga, M. Legré, S. Robyr, P. Trinkler, L. Monat, J.-B. Page, G. Ribordy, A. Poppe, A. Allacher, O. Maurhart, T. Länger, M. Peev, and A. Zeilinger. Field test of quantum key distribution in the Tokyo QKD network. *Optics Express*, 19(11):10387–10409, 2011.

[121] T. Schmitt-Manderbach, H. Weier, M. Fürst, R. Ursin, F. Tiefenbacher, T. Scheidl, J. Perdigues, Z. Sodnik, C. Kurtsiefer, J. G. Rarity, A. Zeilinger, and H. Weinfurter. Experimental demonstration of free-space decoy-state quantum key distribution over 144 km. *Physical Review Letters*, 98(1):010504–1 – 010504–4, 2007.

[122] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing (SICOMP)*, 26(5):1484–1509, 1997.

[123] G. Smith. On the Foundations of Quantitative Information Flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, LNCS 5504, pages 288–302. Springer, 2009.

[124] K. Tiri, O. Aciiçmez, M. Neve, and F. Andersen. An Analytical Model for Time-Driven Cache Attacks. In *Proceedings of the 14th International Workshop on Fast Software Encryption (FSE)*, LNCS 4593, pages 399–413. Springer, 2007.

[125] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[126] M. Vassena, C. Disselkoen, K. von Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. In *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 49:1–49:30. Association for Computing Machinery, 2021.

[127] G. Vest, M. Rau, L. Fuchs, G. Corrielli, H. Weier, S. Nauerth, A. Crespi, R. Osellame, and H. Weinfurter. Design and evaluation of a handheld quantum key distribution sender module. *IEEE Journal of Selected Topics in Quantum Electronics (JSTQE)*, 21(3):131–137, 2015.

[128] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pages 657–674. USENIX Association, 2019.

[129] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, pages 235–252. USENIX Association, 2017.

[130] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 83–93. IEEE Computer Society, 2008.

[131] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA. RISC-V Foundation. Document Version 20191213, 2019.

[132] A. Weber, O. Nikiforov, A. Sauer, J. Schickel, G. Alber, H. Mantel, and T. Walther. Cache-Side-Channel Quantification and Mitigation for Quantum Cryptography. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS)*, LNCS 12973, pages 235–256. Springer, 2021.

[133] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl. DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, pages 603–620. USENIX Association, 2018.

[134] F. Xu, X. Ma, Q. Zhang, H.-K. Lo, and J.-W. Pan. Secure quantum key distribution with realistic devices. *Reviews of Modern Physics*, 92(2):025002–1–025002–60, 2020.

[135] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pages 719–732. USENIX Association, 2014.

[136] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: A Timing Attack on OpenSSL Constant-Time RSA. *Journal of Cryptographic Engineering (JCEN)*, 7(2):99–112, 2017.

[137] Q. Zhang, F. Xu, L. Li, N.-L. Liu, and J.-W. Pan. Quantum information research in China. *Quantum Science and Technology*, 4(040503):1–7, 2019.

[138] P. Zijlstra. perf/core: Fix possible Spectre-v1 indexing for ->aux_pages[]. `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/kernel?h=v5.15.5&id=4411ec1d1993e8dbff2898390e3fed280d88e446`, 2018. [Online; accessed Feb-02-2022].

Appendix A

Raw Leakage Bounds

In this appendix, we present the leakage bounds visualized in the figures and tables of Chapter 3.

Raw Leakage Bounds for Figure 3.3 (mbedTLS AES without Preloading)

	cache size [KiB]						
attacker model	2	4	8	16	32	64	128
acc (- ▲ -)	92.6 bit	114.5 bit	91.8 bit	71.2 bit	69.6 bit	69.6 bit	69.0 bit
accd (- ● -)	17.7 bit	36.2 bit	52.8 bit	67.0 bit	69.0 bit	69.0 bit	69.0 bit
trace (- ■ -)	256.0 bit	256.0 bit	199.0 bit	199.0 bit	199.0 bit	199.0 bit	199.0 bit
time (- ◆ -)	8.7 bit	8.7 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit

Raw Leakage Bounds for Figure 3.4 (AES across Libraries without Preloading)

	cache size [KiB]						
	2	4	8	16	32	64	128
attacker model *acc*							
LibTomCrypt (- ◬ -)	n.a.[5]	177.8 bit	203.3 bit	165.5 bit	132.6 bit	129.0 bit	129.0 bit
mbedTLS (- ◈ -)	92.6 bit	114.5 bit	91.8 bit	71.2 bit	69.6 bit	69.6 bit	69.0 bit
Nettle (- ⊡ -)	90.7 bit	112.9 bit	91.3 bit	72.4 bit	69.6 bit	69.0 bit	69.0 bit
OpenSSL (- ⊖ -)	87.6 bit	105.4 bit	82.3 bit	65.8 bit	64.6 bit	64.0 bit	64.0 bit
attacker model *accd*							
LibTomCrypt (- ◬ -)	n.a.[5]	35.9 bit	69.5 bit	99.4 bit	122.4 bit	129.0 bit	129.0 bit
mbedTLS (- ◈ -)	17.7 bit	36.2 bit	52.8 bit	67.0 bit	69.0 bit	69.0 bit	69.0 bit
Nettle (- ⊡ -)	17.3 bit	35.9 bit	52.8 bit	67.0 bit	69.0 bit	69.0 bit	69.0 bit
OpenSSL (- ⊖ -)	17.3 bit	35.9 bit	50.8 bit	64.0 bit	64.0 bit	64.0 bit	64.0 bit
attacker model *trace*							
LibTomCrypt (- ◬ -)	n.a.[5]	256.0 bit	214.0 bit	198.0 bit	198.0 bit	198.0 bit	198.0 bit
mbedTLS (- ◈ -)	256.0 bit	256.0 bit	199.0 bit	199.0 bit	199.0 bit	199.0 bit	199.0 bit
Nettle (- ⊡ -)	256.0 bit	256.0 bit	199.0 bit	199.0 bit	199.0 bit	199.0 bit	199.0 bit
OpenSSL (- ⊖ -)	256.0 bit	256.0 bit	196.0 bit	196.0 bit	196.0 bit	196.0 bit	196.0 bit
attacker model *time*							
LibTomCrypt (- ◬ -)	n.a.[5]	8.9 bit	7.8 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit
mbedTLS (- ◈ -)	8.7 bit	8.7 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit
Nettle (- ⊡ -)	9.1 bit	9.1 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit
OpenSSL (- ⊖ -)	8.9 bit	8.8 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit	7.7 bit

[5]The analysis ran out of memory.

Raw Leakage Bounds for Table 3.6 (AES across Libraries with Preloading)

	cache size [KiB]						
	2	4	8	16	32	64	128
attacker model *acc*							
LibTomCrypt	n.a.[6]	176.1 bit	137.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
mbedTLS	91.3 bit	57.9 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
Nettle	90.6 bit	64.6 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
OpenSSL	88.6 bit	47.9 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
attacker model *accd*							
LibTomCrypt	n.a.[6]	35.9 bit	41.2 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
mbedTLS	17.7 bit	15.7 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
Nettle	17.3 bit	17.7 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
OpenSSL	17.3 bit	13.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
attacker model *trace*							
LibTomCrypt	n.a.[6]	256.0 bit	217.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
mbedTLS	256.0 bit	256.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
Nettle	256.0 bit	256.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
OpenSSL	256.0 bit	256.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
attacker model *time*							
LibTomCrypt	n.a.[6]	8.9 bit	7.8 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
mbedTLS	8.9 bit	8.9 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
Nettle	9.3 bit	9.3 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit
OpenSSL	8.9 bit	8.8 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit	0.0 bit

[6] The analysis ran out of memory.

Appendix B

Execution Model for Systems with Caching and Pipelining

In this appendix, we present the full definitions for the concrete domain and concrete semantics of the execution model underlying the program analysis in Chapter 6.

B.1 Concrete Domain

Our concrete domain captures the set of possible execution snapshots with respect to the execution of *p*ASM programs on an architecture with *NUM* general purpose registers, Harvard-style instruction and data memory, a fully-associative Level 1 data cache, and an instruction pipeline. The instruction pipeline has four stages and features branch prediction with a static always-not-taken prediction strategy, out-of-order execution with implicit register renaming based on reservation stations, and synchronous in-order commit based on a reorder buffer. Figure B.1 gives an overview of the architectural and micro-architectural components captured by our execution model. A snapshot of the system state during a program execution is captured by a concrete configuration in the model. A concrete configuration consists of concrete component configurations that capture snapshots of the states of the individual components.

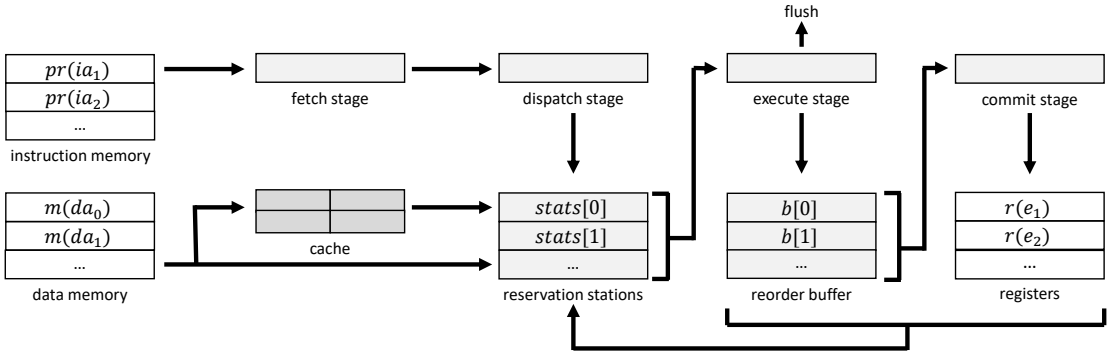

Figure B.1: Overview of our Execution Model for an Architecture with Caching and Pipelining

Pipeline

A concrete pipeline configuration consists of concrete configurations for the micro-architectural components that belong to the instruction pipeline, namely the pipeline stages, the reservation stations, and the reorder buffer.

Pipeline Stages The set of concrete pipeline-stage configurations is the set $StCos = Sts \rightarrow StCs$. Each pipeline-stage configuration is a function from pipeline stages to the set $StCs = UIAs \cup \{\bot, \top\}$. The four pipeline stages fetch, dispatch, execute, and commit are captured by the symbols in the set $Sts = \{fet, dis, exe, com\}$. Each pipeline stage is mapped either to the instruction that is currently processed in the stage or to a symbol that captures a reason why the pipeline stage is stalled. Instructions are represented by elements of the set $UIAs = IAs \times \mathbb{N}$ of pairs, where each pair uniquely identifies an instruction based on the instruction's address in the program and a counter by which multiple occurrences of the same instruction during a program execution can be distinguished. The symbol $\bot$ captures that a pipeline stage is stalled due to a pending speculation barrier. The symbol $\top$ captures that a pipeline stage is stalled because no instruction is ready to be processed in the respective stage.

Reservation Stations The set of concrete reservation-station configurations is the set $RSCos = RSCs^\star$ with $RSCs = UIAs \times MLs \times RLs$. Each concrete reservation-station configuration is a list of triples, where each triple captures one in-flight instruction that is processed in the reservation stations. Each triple consists of the instruction's identifier, the instruction's memory-access list, and the instruction's register-dependence list. The memory-access list is an element of the set $MLs = (DAs \times \mathbb{N}_0)^\star$, i.e., a list of pairs. Each pair corresponds to one memory access that is triggered by the instruction and consists of the target address of the access and the number of clock cycles that are still required until the value has been retrieved from the target address. The symbol *HIT* captures the number of clock cycles that are required to serve a memory access from the cache and the symbol *MISS* captures the number of clock cycles that are required to serve a memory access from the main memory. The register-dependence list of an instruction is an element of the set $RLs = (Rs \times UIAs)^\star$, i.e., also a list of pairs. Each pair in the register-dependence list of an instruction ua consists of one source register of ua and of the identifier of the instruction that will produce the value that ua reads from this register.

Overall, a concrete reservation-station configuration captures the information about each in-flight instruction that is processed in the reservation stations. The memory-access lists capture the state of all pending memory accesses that are processed in parallel. The register-dependence lists capture the pointers between the reservation stations that are used to keep track of name dependencies, i.e., that implement the implicit register renaming.

Reorder Buffer The set of concrete reorder-buffer configurations is the set $BCos = BCs^\star$ with $BCs = UIAs \times Rs \times Vs$. The state of the reorder buffer is captured by a list of triples, where each triple models one register update that is buffered for the synchronous in-order commit to the actual register. Each triple consists of the identifier of the instruction that triggered the update, the register that is affected by the update, and the value to which the register shall be updated.

Overall Pipeline The set of concrete pipeline configurations is the set $PCos = StCos \times RSCos \times BCos$, i.e., each concrete pipeline configuration is a triple that consists of a pipeline-stage configuration, a reservation-station configuration, and a reorder-buffer configuration.

Notational Conventions To extract individual components from concrete pipeline configurations, we use the same notation as for abstract pipeline configurations. More concretely, we write $stag(pi)$, $rst(pi)$, and $buf(pi)$ to extract the pipeline-stage configuration, reservation-station

configuration, and reorder-buffer configuration from a concrete pipeline configuration pi. We write $fst(pi)$, $dst(pi)$, $est(pi)$, and $cst(pi)$ to extract the content of each pipeline stage. To extract the instruction identifier and the instruction address from a reservation-station or reorder-buffer entry, we use $uad(((ia, n), x, y)) = (ia, n)$ and $ad(((ia, n), x, y)) = ia$, respectively. Finally, we extract the set of all memory-access-list entries and the set of all register-dependence list entries from a concrete reservation-station configuration rs using $mcs(rs)$ and $rds(rs)$, respectively.

Cache

The set of concrete cache configurations is the set $CCos = DAs \to [0, SIZE]$. Each concrete cache configuration is a function that maps each data address to the cache line in which it is cached. The numbers 0 to $SIZE - 1$ capture the individual cache lines within the cache. The number $SIZE$ captures that the data address is not cached. Our model can be instantiated for different values of the number $SIZE$ to capture caches with different sizes, i.e., different numbers of cache lines.

Memory and Registers

The set of concrete instruction-memory configurations is the set Ps_{wf} of well-formed programs. The set of concrete data-memory configurations is the set $MCos = DAs \to Vs$, where each concrete data-memory configuration maps each data address to the 32 bit value that is stored at this address. Analogously, the set of concrete register configurations is the set $RCos = Rs \to Vs$. Each concrete register configuration maps each register to the 32 bit value stored in this register.

Putting all Together

Snapshots of the overall system state are captured by concrete configurations.

Definition B.1. *The set of concrete configurations (or the concrete domain) is the set*

$$Cos = Ps_{wf} \times RCos \times MCos \times CCos \times PCos.$$

That is, each concrete configuration is a five-tuple consisting of a concrete configuration for each architectural and micro-architectural component that is captured by our model. That is, it captures a snapshot of the system state during the execution of a pASM program on a 32 bit architecture with a fully-associative Level 1 data cache and a four-stage instruction pipeline with static branch prediction and out-of-order execution.

B.2 Concrete Semantics

We capture the changes to the system state that occur in one clock cycle during the execution of a pASM program with respect to our concrete domain by a function $upd : Cos \to Cos$, which is based on auxiliary concrete update functions for each component in a concrete configuration.

Pipeline

The concrete pipeline update function is defined with respect to concrete update functions for concrete pipeline-stage configurations, concrete reservation-station configurations, and concrete reorder-buffer configurations.

Pipeline Stages In each clock cycle, the fetch stage of the pipeline is updated. The instruction to which it is updated depends on the branch-prediction strategy. In our model, the branch prediction is captured by the function $bpr : UIAs \to UIAs$, such that

$$bpr((ia_k, n)) = (ia_{k+1}, n).$$

The function simply increments the instruction address of the current instruction identifier by one. That is, each branch is predicted to not be taken. Our concrete execution model captures a static always-not-taken branch-prediction strategy.

For the dispatch stage, we do not define a separate auxiliary function, because it always processes the most recently fetched instruction and does not reorder.

In each clock cycle, the execute stage is updated to one instruction from the reservation stations, namely to the instruction that is scheduled for execution in the next clock cycle. Which instruction is scheduled for execution is defined by $nxe : RSCos \times RSCos \to StCs$, such that

$$nxe(rs, rs') = \begin{cases} \top & \text{if } rs = \langle\rangle \\ uad(rc) & \text{if } rs = \langle rc\rangle \bullet rs'' \ \wedge re(rc, rs') \\ nxe(rs'', rs' \bullet \langle rc\rangle) & \text{if } rs = \langle rc\rangle \bullet rs'' \ \wedge \neg re(rc, rs'). \end{cases}$$

The function nxe searches the reservation stations for the first instruction that is ready to be executed. If no instruction is ready, it returns $\top$. Whether an instruction is ready to be executed is captured in the concrete semantics by the predicate $re : RSCs \times RSCos \to \mathbb{B}$, such that

$$\begin{aligned} re((ua, ml, rl), rs) = {} & \forall i \in \mathbb{N}_0.\ (i < |ml| \Rightarrow \exists d \in DAs.\ ml[i] = (d, 0)) \wedge \\ & \forall j \in \mathbb{N}_0.\ (j < |rl| \Rightarrow \exists e \in Rs.\ \exists ua' \in UIAs. \\ & \qquad (rl[j] = (e, ua') \wedge \forall k \in \mathbb{N}_0.\ (k < |rs| \Rightarrow uad(rs[k]) \neq ua'))). \end{aligned}$$

This predicate holds if all memory operands of the instruction are available, i.e., zero clock cycles are required to complete each memory access (first conjunct), and all register operands of the instruction are available, i.e., all instructions that produce operand values are executed (second conjunct). The latter is captured with respect to a reservation-station configuration rs that is supplied as the second parameter of re and should contain the reservation-station entries of all instructions that were dispatched before ua and are still pending.

For the commit stage, we capture the next instruction to be processed by the function $nxc : PCos \to StCs$, such that

$$nxc(pi) = \begin{cases} uad(b[0]) & \text{if } b = pb(buf(pi), cst(pi)) \wedge rc(pi, b) \\ \top & \text{if } \neg rc(pi, b). \end{cases}$$

The function checks whether the first register update in the reorder-buffer is ready to be committed. To avoid that the same update is committed twice, the update that belongs to the instruction that is currently processed in the commit stage is removed beforehand using the auxiliary function $pb : BCos \times StCs \to BCos$, such that

$$pb(b, sc) = \begin{cases} b & \text{if } b = \langle\rangle \\ pb(b') & \text{if } b = \langle bc\rangle \bullet b' \wedge sc = ad(bc) \\ bc \bullet pb(b') & \text{if } b = \langle bc\rangle \bullet b' \wedge sc \neq ad(bc). \end{cases}$$

Whether the first update from the buffer is ready to be committed is captured by the predicate $rc : PCos \times BCos \to \mathbb{B}$, such that

$$rc(pi, b) = |b| > 0 \wedge \forall i \in \mathbb{N}_0.\ (i < |rst(pi)| \Rightarrow uad(b[0]) \neq uad(rst(pi)[i])).$$

This predicate holds if the instruction that triggered the update is already executed, i.e., if the instruction is not processed in any reservation station anymore.

The fact that only the first register update from the reorder buffer is considered for the next commit models a synchronous in-order commit, which ensures that the register updates become visible in the actual registers in program order.

Definition B.2. *The concrete pipeline-stage update is* $upd_{st} : PCos \times Ps_{wf} \to StCos$*, such that*

$$upd_{st}(pi, pr)(s) = \begin{cases} \bot & \text{if } s = fet \wedge barr(pi, pr) \\ \top & \text{if } s = fet \wedge term(pi, pr) \\ bpr(fst(pi)) & \text{if } s = fet \wedge \neg barr(pi, pr) \wedge \neg term(pi, pr) \wedge fst(pi) \in UIAs \\ bpr(est(pi)) & \text{if } s = fet \wedge \neg barr(pi, pr) \wedge \neg term(pi, pr) \wedge fst(pi) \notin UIAs \\ fst(pi) & \text{if } s = dis \\ nxe(rst(pi), \langle\rangle) & \text{if } s = exe \\ nxc(pi) & \text{if } s = com. \end{cases}$$

The first four cases in the definition of the pipeline-stage update capture the update of the fetch stage. If the fetch stage is stalled intentionally (due to a pending speculation barrier), it is updated to $\bot$. If the fetch stage is stalled because the program has already been fetched completely, it is updated to $\top$. The predicates $barr : PCos \times Ps_{wf} \to \mathbb{B}$ and $term : PCos \times Ps_{wf} \to \mathbb{B}$ are:

$$\begin{aligned} barr(pi, pr) = {} & \textbf{fence} \in \{pr(fst(pi)), pr(dst(pi))\} \vee \\ & \exists i \in \mathbb{N}_0.\ (i < |rst(pi)| \wedge pr(ad(rst(pi)[i])) = \textbf{fence}) \\ term(pi, pr) = {} & \neg barr(pi, pr) \wedge (fst(pi) = \top \vee \\ & (fst(pi) \in UIAs \wedge bpr(fst(pi)) = (ia, n) \wedge pr(ia)\uparrow) \vee \\ & (fst(pi) = \bot \wedge bpr(est(pi)) = (ia, n) \wedge pr(ia)\uparrow)). \end{aligned}$$

The former captures whether there is a pending speculation barrier in the fetch stage, dispatch stage, or reservation stations. The latter captures whether the program has already been fetched completely, i.e., there is no more instruction to fetch based on *bpr*.

If the fetch stage is not stalled, it fetches the next instruction according to the prediction modeled by *bpr*. The prediction is based on the current instruction in the fetch stage or (in case the fetch stage was stalled so far) based on the current instruction in the execute stage.

The fifth, sixth, and seventh case in the definition of upd_{st} capture the updates of the dispatch stage, execute stage, and commit stage, respectively. The dispatch stage is updated to the current instruction in the fetch stage. The updates of the execute and commit stage are based on the auxiliary functions defined above. Overall, the function upd_{st} captures the changes to the pipeline stages that occur if the pipeline is not flushed.

Definition B.3. *The concrete pipeline flush function is* $fl_{st} : PCos \times Ps_{wf} \to StCos$*, such that*

$$fl_{st}(pi, pr)(s) = \begin{cases} (ia, n) & \text{if } s = fet \wedge est(pi) = (ia_k, n) \wedge pr(est(pi)) = \textbf{jge}\, ia_{k'}\, e\, e' \wedge k' > k \\ (ia, n+1) & \text{if } s = fet \wedge est(pi) = (ia_k, n) \wedge pr(est(pi)) = \textbf{jge}\, ia_{k'}\, e\, e' \wedge k' \leq k \\ cst(pi) & \text{if } s = com \\ \top & \text{otherwise.} \end{cases}$$

The function fl_{st} captures the changes to the concrete pipeline-stage configuration in case of a pipeline flush. In this case, the fetch stage is updated to the correct target of the conditional jump instruction that caused the pipeline flush (the first case captures forward jumps, the second case captures backward jumps). The commit stage remains unchanged, because it processes instructions in program order and is not affected by speculatively executed instructions. The dispatch and execute stages are cleared, i.e., mapped to $\top$.

Reservation Stations The changes to the reservation stations in one clock cycle consist of (1) the dispatch of the instruction that leaves the dispatch stage, (2) the progress made by the ongoing memory accesses, and (3) the removal of the reservation-station entry of the instruction that is scheduled for execution, i.e., that enters the execute stage.

The dispatch is captured by $disp : StCs \times Ps_{wf} \times CCos \times RCos \times BCos \to RSCos$, s.t.

$$disp(sc, pr, c, r, b) = \begin{cases} \langle\rangle & \text{if } sc \notin UIAs \\ \langle(sc, \langle\rangle, \langle\rangle)\rangle & \text{if } sc = (ia, n) \wedge pr(ia) \in \{\textbf{mov-rc}\, e\, v, \textbf{nop}\} \\ \langle(sc, \langle\rangle, fd(sc))\rangle & \text{if } sc = (ia, n) \wedge pr(ia) = \textbf{fence} \\ \langle(sc, \langle\rangle, \langle(e, ua)\rangle)\rangle & \text{if } sc = (ia, n) \wedge pr(ia) \in \{\textbf{neg}\, e, \textbf{add-rc}\, e\, v, \\ & \quad \textbf{sub-rc}\, e\, v, \textbf{shr-rc}\, e\, v, \textbf{sar-rc}\, e\, v\} \wedge \\ & \quad ua = dep(sc, e, pr) \\ \langle(sc, \langle\rangle, \langle(e, ua), (e', ua')\rangle)\rangle & \text{if } sc = (ia, n) \wedge pr(ia) \in \{\textbf{jge}\, ia'\, e\, e', \\ & \quad \textbf{add-rr}\, e\, e', \textbf{sub-rr}\, e\, e', \textbf{and-rr}\, e\, e', \\ & \quad \textbf{or-rr}\, e\, e'\} \wedge ua = dep(sc, e, pr) \wedge \\ & \quad ua' = dep(sc, e', pr) \\ \langle(sc, \langle(d', t')\rangle, \langle(e', ua')\rangle)\rangle & \text{if } sc = (ia, n) \wedge pr(ia) = \textbf{mov-rm}\, e\, d_k\, e' \wedge \\ & \quad ua' = dep(sc, e', pr) \wedge v = rv(sc, e', r, b) \wedge \\ & \quad d' = d_{k+v} \wedge t' = cyc(d', c). \end{cases}$$

If the dispatch stage is idle, then $disp(sc, pr, c, r, b)$ is the empty list. If the dispatch stage processes an instruction sc, then $disp(sc, pr, c, r, b)$ is a list with one element. This element is a reservation-station entry that consists of the instruction identifier sc, the memory-access list of sc, and the register-dependence list of sc.

The memory-access list is empty for all instructions except **mov-rm** instructions. If sc is a **mov-rm** instruction, its memory-access list contains one pair, which consists of a target address and a number of clock cycles. The number of clock cycles required for the memory access depends on whether the target address is available in the cache. This is captured by the function $cyc : DAs \times CCos \to \mathbb{N}_0$, such that

$$cyc(d, c) = \begin{cases} MISS & c(d) \geq SIZE \\ HIT & c(d) < SIZE. \end{cases}$$

The target address itself is the result of adding the base address d_k to the offset from the operand register e'. The value that the instruction sc reads from the register e' depends on the pending register updates in the reorder buffer. It is defined by $rv : UIAs \times Rs \times RCos \times BCos \to Vs$, s.t.

$$rv(ua, e, r, b) = \begin{cases} r(e) & \text{if } b = \langle\rangle \\ rv(ua, e, r, b') & \text{if } b = b' \bullet \langle(ua', e', v)\rangle \wedge (e' \neq e \vee \neg(ua' <_{po} ua)) \\ v & \text{if } b = b' \bullet \langle(ua', e, v)\rangle \wedge ua' <_{po} ua, \end{cases}$$

where $<_{po}$: $UIAs \times UIAs \to \mathbb{B}$ is the program order, i.e., the order in which the instructions would be executed by a sequential, in-order CPU. To determine the value that an instruction ua reads from a register e, the function rv checks whether there are pending updates for e in the reorder-buffer configuration b that belong to instructions that occur before ua in program order. If such updates exist, the update that belongs to the instruction that is closest to ua in program order determines the value that ua reads from e. If no such updates exist, the value for e is read directly from the concrete register configuration r.

The register-dependence list is empty for **mov-rc** and **nop** instructions. For **fence** instructions, it is determined by an auxiliary function $fd : UIAs \to RLs$, which returns a list that contains a

pair (eax, ua) for each instruction ua that occurs before the **fence** instruction in program order. This list reflects that no reordering of instructions across the **fence** instruction is possible. For all other instructions, the register dependence list contains one pair for each source register of the instruction. The dependence of each source register is determined using an auxiliary function $dep : UIAs \times Rs \times Ps_{wf} \to UIAs$, such that $dep(ua, e, pr)$ returns the identifier of the instruction that produces the value of e that is processed by ua.

The second change to the reservation-station configuration is caused by the ongoing memory accesses of the instructions that are processed in the stations. This change is captured by the function $uts : RSCos \to RSCos$, such that

$$uts(rs) = \begin{cases} rs & \text{if } rs = \langle\rangle \\ rc \bullet uts(stats') & \text{if } rs = \langle (ua, ml, rl)\rangle \bullet rs' \wedge rc = (ua, u(ml), rl), \end{cases}$$

with

$$u(ml) = \begin{cases} ml & \text{if } ml = \langle\rangle \\ (d, t-1) \bullet u(ml') & \text{if } ml = \langle (d,t)\rangle \bullet ml' \wedge t > 0 \\ (d,t) \bullet u(ml') & \text{if } ml = \langle (d,t)\rangle \bullet ml' \wedge t \leq 0. \end{cases}$$

This function simply decreases the remaining number of clock cycles for each access by one.

Finally, the change caused by the removal of the instruction that is scheduled for execution is captured by the function $ps : RSCos \to RSCos$, such that

$$ps(rs) = \begin{cases} rs & \text{if } rs = \langle\rangle \\ rs' & \text{if } rs = rc \bullet rs' \wedge uad(rc) = nxe(rs, \langle\rangle) \\ rc \bullet ps(rs') & \text{if } rs = rc \bullet rs' \wedge uad(rc) \neq nxe(rs, \langle\rangle). \end{cases}$$

The function ps removes the entry from the reorder-buffer configuration that belongs to the instruction $nxe(rs, \langle\rangle)$, i.e., the instruction to which the execute stage of the pipeline is updated.

Definition B.4. *The concrete reservation-station update is the function* $upd_{rs} : RSCos \times PCos \times Ps_{wf} \times CCos \times RCos \to RSCos$, *such that*

$$upd_{rs}(rs, pi, pr, c, r) = uts(ps(rs)) \bullet disp(dst(pi), pr, c, r, buf(pi)).$$

The concrete reservation-station update covers the addition of a new reservation-station entry, the progress of the ongoing memory accesses, and the removal of the entry for the instruction that is scheduled for execution. It does not enforce an explicit upper limit on the number of instructions that can be processed in the reservation stations at the same time. Such a limit is captured implicitly in the execution model, because the instruction that is scheduled for execution is always the first ready instruction in the order of the instruction dispatch. The total number of clock cycles *MISS* that is required to resolve a cache miss, hence, bounds for how many clock cycles each instruction can remain in the reservation stations.

While upd_{rs} defines the changes to the reservation-station configuration during clock cycles without pipeline flushes, the changes during clock cycles with flushes are captured by the function $fl_{rs} : RSCos \times StCs \to RSCos$, such that

$$fl_{rs}(rs, sc) = \begin{cases} \langle\rangle & \text{if } rs = \langle\rangle \\ fl_{rs}(rs', sc) & \text{if } rs = \langle rc\rangle \bullet rs' \wedge sc \in UIAs \wedge sc <_{po} uad(rc) \\ \langle rc\rangle \bullet fl_{rs}(rs', sc) & \text{if } rs = \langle rc\rangle \bullet rs' \wedge sc \in UIAs \wedge \neg(sc <_{po} uad(rc)). \end{cases}$$

This function drops the reservation-station entries for each instruction that was dispatched speculatively, i.e., for each instruction that occurs after sc in program order.

Reorder Buffer The changes to the reorder buffer during a clock cycle are (1) the addition of a register update for the instruction that leaves the dispatch stage and (2) the removal of the register update for the instruction that leaves the commit stage.

The addition of the new register update to the buffer is captured based on the function $res : StCs \times Ps_{wf} \times BCos \times MCos \times RCos \to BCos$, such that

$$res(sc, pr, b, m, r) = \begin{cases} \langle\rangle & \text{if } sc \notin UIAs \vee (sc = (ia, n) \wedge \\ & \quad pr(ia) \in \{\textbf{jge}\, ia'\, e\, e', \textbf{fence}, \textbf{nop}\}) \\ \langle(sc, e, v)\rangle & \text{if } sc = (ia, n) \wedge pr(ia) = \textbf{mov-rm}\, e\, d_k\, e' \wedge \\ & \quad v = m(d_{k+rv(sc,e',r,b)}) \\ \langle(sc, e, \sim v)\rangle & \text{if } sc = (ia, n) \wedge pr(ia) = \textbf{neg}\, e \ \wedge v = rv(sc, e, r, b) \\ \langle(sc, e, x)\rangle & \text{if } sc = (ia, n) \wedge \\ & \quad (pr(ia) \in \{\textbf{add-rc}\, e\, v', \textbf{sub-rc}\, e\, v', \textbf{shr-rc}\, e\, v', \textbf{sar-rc}\, e\, v'\} \vee \\ & \quad (pr(ia) \in \{\textbf{add-rr}\, e\, e', \textbf{sub-rr}\, e\, e', \textbf{and-rr}\, e\, e', \textbf{or-rr}\, e\, e'\} \wedge \\ & \qquad v' = rv(sc, e', r, b)) \wedge x = bop(pr(ia), rv(sc, e, r, b), v'), \end{cases}$$

where $bop : Insts \times Vs \times Vs \to Vs$ is defined by

$$bop(in, v, v') = \begin{cases} v + v' & \text{if } opc(in) \in \{\textbf{add-rr}, \textbf{add-rc}\} \\ v - v' & \text{if } opc(in) \in \{\textbf{sub-rr}, \textbf{sub-rc}\} \\ v \& v' & \text{if } opc(in) = \textbf{and-rr} \\ v \| v' & \text{if } opc(in) = \textbf{or-rr} \\ v >>> v' & \text{if } opc(in) = \textbf{shr-rc} \\ v >> v' & \text{if } opc(in) = \textbf{sar-rc} \\ v' & \text{if } opc(in) = \textbf{mov-rc}. \end{cases}$$

That is, $res(sc, pr, b, m, r)$ is the empty list if the dispatch stage is stalled or processes an instruction sc that does not write to a register. If sc is an instruction that writes to a register, $res(sc, pr, b, m, r)$ is a list with one element. This one element is a register update that consists of the instruction identifier sc, the destination register of the instruction, and the result that shall be written to the destination register. For a **mov-rm** instruction, the result is a value from the data memory. The value is retrieved from a target address that consists of the base address d_k and the offset from the operand register e'. For a **neg** instruction, the result is the bit-wise negation of the value from the operand register. For instructions that compute arithmetic or logical binary operations, the result is captured by the function *bop*.

Definition B.5. *The concrete reorder-buffer update is* $upd_b : BCos \times PCos \times Ps_{wf} \times MCos \times RCos \to BCos$, *such that*

$$upd_b(b, pi, pr, m, r) = pb(b, est(pi)) \bullet res(dst(pi), pr, b, m, r).$$

The reorder-buffer update combines the addition of the new register update, which is appended to the buffer, with the removal of the committed update. The latter is based on the function *pb*, which is defined in the paragraph on reservation stations above. The definition does not enforce a limit on the number of updates that can be stored in the buffer at the same time. This limit is enforced implicitly by the number of clock cycles *MISS*, which limits how long each instruction may be kept in a reservation station before it is executed.

The function upd_b models the changes to the reorder buffer in case no pipeline flush occurs. If

a flush occurs, the changes are modeled by the function $fl_b : BCos \times StCs \to BCos$, such that

$$fl_b(b, sc) = \begin{cases} \langle\rangle & \text{if } b = \langle\rangle \\ fl_b(b', sc) & \text{if } b = \langle bc\rangle \bullet b' \wedge sc \in UIAs \wedge sc <_{po} uad(bc) \\ \langle bc\rangle \bullet fl_b(b', sc) & \text{if } b = \langle bc\rangle \bullet b' \wedge sc \in UIAs \wedge \neg(sc <_{po} uad(bc)). \end{cases}$$

The function drops all entries from the reorder-buffer configuration that were triggered by speculatively dispatched instructions, i.e., instructions that occur after sc in program order.

Overall Pipeline The update functions for all three pipeline components (the pipeline stages, reservation stations, and reorder buffer) are combined in the overall concrete pipeline update.

Definition B.6. *The concrete pipeline update is the function* $upd_{pi} : PCos \times Ps_{wf} \times RCos \times MCos \times CCos \to PCos$*, such that*

$$upd_{pi}(pi, pr, r, m, c) = \begin{cases} fl(pi, pr) & \textit{if } isfl(pi, pr, r) \\ up_{pi}(pi, pr, r, m, c) & \textit{otherwise,} \end{cases}$$

where

$$fl(pi, pr) = (fl_{st}(pi, pr), fl_{rs}(rst(pi), est(pi)), fl_b(buf(pi), est(pi)))$$
$$up_{pi}(pi, pr, r, m, c) = (upd_{st}(pi, pr), upd_{rs}(rst(pi), pi, pr, c, r), upd_b(buf(pi), pi, pr, m, r)).$$

The concrete pipeline update either applies the regular update functions or, if a pipeline flush occurs, the update functions that capture pipeline flushes. Whether a pipeline flush occurs is defined by the predicate $isfl : PCos \times Ps_{wf} \times RCos \to \mathbb{B}$, such that

$$isfl(pi, pr, r) = \exists ia \in IAs.\ \exists e, e' \in Rs.\ (pr(est(pi)) = \textbf{jge}\ ia\ e\ e' \wedge rv(est(pi), e, r, buf(pi)) \geq rv(est(pi), e', r, buf(pi))).$$

This predicate holds if the execute stage is processing a conditional jump instruction and the value of the first operand register is greater than or equal to the value of the second operand register. Thus, a pipeline flush occurs exactly in those cases in which the conditional jump instruction triggers a jump, i.e., the cases in which the corresponding branch is taken. This reflects the always-not-taken branch-prediction strategy.

Note that, the predicate $isfl$ is used within the concrete pipeline update. That is, it affects only the concrete pipeline configuration and not the configurations of the registers, memory, and cache. This reflects that the state of the cache is not rolled back during a pipeline flush. The register rollback is captured by the flush function for the reorder buffer. Updates that are already committed are not rolled back.

Cache

The cache configuration is updated if a cache miss occurs during the current clock cycle.

Definition B.7. *The concrete cache update is* $upd_c : CCos \times RSCos \to CCos$*, such that*

$$upd_c(c, rs)(d) = \begin{cases} 0 & \textit{if } (d, MISS) \in mcs(rs) \wedge c(d) = SIZE \\ c(d) + 1 & \textit{if } (d, MISS) \notin mcs(rs) \wedge c(d) < SIZE \wedge \\ & \quad \exists d' \in DAs.\ (d' \neq d \wedge (d', MISS) \in mcs(rs) \wedge c(d') = SIZE) \\ c(d) & \textit{otherwise.} \end{cases}$$

Given the concrete reservation-station configuration rs, the updated concrete cache configuration $upd_c(c, rs)$ maps each data address d to its updated position if a unique cache miss occurs according to the memory-access lists of rs. The updated position is the cache line zero if d is the address on which the cache miss occurred and d was not cached so far. If the cache miss was triggered by an access to another data address d' that was not cached so far but d was already cached, then d moves one cache line ahead. If d was not cached so far or if there is no cache miss on any uncached data address, the position of d remains unchanged. Overall, the concrete cache update function captures the FIFO replacement policy.

Memory and Registers

The instruction memory and data memory remain unchanged throughout a program execution. Hence, the corresponding update functions are identity functions.

Definition B.8. *The concrete instruction-memory update is* $upd_{pr} : Ps_{wf} \to Ps_{wf}$*, such that*

$$upd_{pr}(pr) = pr$$

Definition B.9. *The concrete data-memory update is* $upd_{mem} : MCos \to MCos$*, such that*

$$upd_{mem}(m) = m$$

The content of the registers changes based on the instruction processed in the commit stage.

Definition B.10. *The concrete register update is* $upd_{reg} : RCos \times BCos \times StCs \to RCos$*, s.t.*

$$upd_{reg}(r, b, sc)(e) = \begin{cases} v & \text{if } \exists i \in \mathbb{N}_0.\ (i < |b| \wedge b[i] = (sc, e, v)) \wedge \\ & \qquad \forall j \in \mathbb{N}_0.\ ((j < |b| \wedge j \neq i) \Rightarrow uad(b[j]) \neq sc)) \\ r(e) & \text{otherwise.} \end{cases}$$

That is, the register r is updated if the instruction processed in the commit stage triggered a unique register update that affects this register. In this case, r is updated to the new value defined by the corresponding update in the reorder buffer. If the instruction that is processed in the commit stage did not trigger such an update, the register remains unchanged.

Putting all Together

The concrete semantics is the combination of the concrete component update functions.

Definition B.11. *The concrete update (or concrete semantics) is* $upd : Cos \to Cos$*, such that*

$$\begin{aligned} upd(pr, r, m, c, pi) = (&upd_{pr}(pr), upd_{reg}(r, buf(pi), cst(pi)), upd_{mem}(m), \\ &upd_c(c, rst(pi)), upd_{pi}(pi, pr, r, m, c)). \end{aligned}$$

The concrete semantics captures the changes to the architectural and micro-architectural state of a system with respect to our concrete domain. More concretely, it captures the changes that occur in one clock cycle during the execution of a pASM program. The semantics of an entire program execution based on the initial configuration $co \in Cos$ is captured by the fixed point $fix(upd, co)$ reached by the repeated application of upd.