Unique Program Execution Checking: A Novel Approach for Formal Security Analysis of Hardware

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Mohammad Rahmani Fadiheh

geboren in Mashhad, Iran

D 386

Datum der mündlichen Prüfung	:	05.09.2022
Dekan des Fachbereichs	:	Prof. Dr. rer. nat Marco Rahm
Vorsitzender der Prüfungskommission	:	Prof. Dipl.-Ing. Dr. Gerhard Fohler
Gutachter	:	Prof. Dr.-Ing. Wolfgang Kunz
		Prof. Dr. phil. nat. Rolf Drechsler

Acknowledgments

This thesis documents several years of research carried out at the Electronic Design Automation Group at Technische Universität Kaiserslautern. I would like to thank my supervisors, my colleagues, and my collaborators for their immense help and support during this period.

Firstly, I would like to express my sincere gratitude to Prof. Wolfgang Kunz for giving me a great opportunity to learn and grow under his supervision. I am truly indebted to him for his unflagging support and guidance during my research. I would also like to especially thank Prof. Dominik Stoffel for his invaluable ideas during my research, as well as for all his constructive comments for this thesis. During this research, I had the privilege to work in collaboration with Prof. Subhasish Mitra and his Group at Stanford University. I am truly grateful for his valuable feedback and helpful insights for the past five years.

Many thanks to Prof. Rolf Drechsler for his interest in reviewing my thesis as well as for the important and helpful feedback. I also thank Prof. Gerhard Fohler for chairing my thesis committee.

This research has greatly benefited from various industrial collaborations. I would like to sincerely extend my gratitude to Dr. Jason Fung and Dr. Sayak Ray from Intel Co. for their invaluable insights. Furthermore, I am truly grateful to Dr. Jörg Bormann and Dr. Keerthikumara Devarajegowda from Siemens EDA, for all the interesting discussions.

Next, I would like to thank my colleagues Mr. Johannes Müller, Ms. Anna Duque Anton, Mr. Lucas Deutschmann, Mr. Alex Wezel, Mr. Tobias Jauch, and Mr. Dino Mehmedagic for all the interesting discussions and fruitful collaborations. I would like to especially thank Mr. Tobias Jauch for his help in writing the German summary of this thesis. I would like to thank Carmen Vicente-Fess, Matthias Legrom, and Andreas Christmann for their kind support.

Last but not least, I would like to thank my family, who are my greatest source of support and motivation: my parents, my beloved wife, and my dear sisters. I am always grateful for their love and support.

Kaiserslautern, 05.09.2022
Mohammad Rahmani Fadiheh

Table of Contents

Chapter 1

Introduction

Ever since the invention of the first microprocessor in 1971, our reliance on electronic computing systems has been increasing at an accelerating pace. Computing systems, ranging from small embedded systems to high-end server computers, serve as the critical infrastructure for almost any modern technology that our world depends on. Our trust in the computing systems means that our security, whether it be the proper functioning of a power grid or keeping our credit card numbers out of the reach of hackers, depends mainly on provided safety and security features of the underlying computing infrastructure. The System-on-Chips (SoC) used in many embedded systems with their abundance of connectivity features created a new attack surface for cyber-attacks. Malicious cyber-activities can nowadays range from stealing a car [11] to hacking a commercial jet [12].

Although the majority of the advanced security features, e.g., end-to-end encryption, is implemented at the software level, they all rely on basic hardware primitives to deliver the intended functionalities. For example, encryption can be rendered useless if the hardware system does not provide secure memory isolation which keeps the encryption keys confidential. These hardware primitives and features form the root-of-trust for the computing system, i.e., a set of trusted functionalities that are used to ensure the security of the system. Any security flaw in the hardware root of trust can affect many different applications running on top of the system, which makes the design verification of this part of the hardware critical. Hardware systems are very difficult or, in some cases, even impossible to patch, which further exacerbates the challenge of dealing with hardware security flaws. Countless reports in recent years on vulnerabilities at the hardware level(e.g., [13]) attest to the fact that hardware security flaws can pose a genuine threat to the overall system security. The Common Weakness Enumeration database(CWE) has acknowledged this problem recently by including hardware vulnerabilities as a separate category of security weaknesses [14].

The role of hardware in system security is not limited to providing security-related functionalities. Hardware designs can indeed introduce new vulnerabilities to the computing system: the hardware circuit executing a security-critical software application may leak confidential information through side channels,

such as timing of the software execution [15], power consumption [16], etc. Microarchitectural timing side channels are especially important since they do not require physical access to the system or advanced measurement tools.

1.1 Microarchitectural Side Channel Attacks

In design flows commonly used in industry, the digital designer describes the logic behavior of the processor in a clock cycle-accurate way and defines for each instruction the elementary steps of its execution based on the processor's registers. Such design descriptions at the "Register-Transfer Level (RTL)" are often referred to as the *microarchitecture* of a processor. Numerous degrees of freedom exist for the designer in choosing an appropriate microarchitecture for a specification of the processor given at the level of its instruction set architecture (ISA).

However, the same degrees of freedom that the designer uses for optimizing a processor design may also lead to *microarchitectural side effects* that can be exploited in security attacks. In fact, it is possible that, depending on the data that is processed, one and the same program may behave slightly differently in terms of what data is stored in which registers and at which time points. These differences only affect detailed timing at the microarchitectural level and have no impact on the correct functioning of the program at the ISA level, as seen by the programmer. However, if these subtle alterations of program execution at the microarchitectural level can be caused by secret data, this may open a "side channel". An attacker may trigger and observe these alterations to infer secret information, which is called a microarchitectural side channel attack.

In microarchitectural side channel attacks, the possible leakage of secret information is based on a microarchitectural resource which creates a timing information channel between different software processes that share this resource. For example, the cache can be such a shared resource, and an attacker can observe timing variations when accessing the cache, based on the victim's cache access pattern. Various cache-based attacking schemes have been reported that can deduce critical information from the footprint of an encryption software on the cache [15, 17–19]. Also, other shared resources can be (mis-)used as the channel in a side-channel attack, as has been shown for DRAMs [20] and other shared functional units [21].

In these attacks, the attacker process by itself is not capable of controlling both ends of a side channel. In order to steal secret information, it must interact with another process initiated by the system, the "victim process", which manipulates the secret. In addition, the attacker must possess detailed knowledge about the victim software, in order to make a meaningful correlation between the observed side channel and the victim's secret assets. Because of these prerequisites, the scope of a side channel attack may be limited to specific software components, which allows for remedies at the software level. These remedies are typically applied to security-critical software components like encryption algorithms. Common measures include constant-time encryption [22] and cache access pattern

obfuscation [23]. They prohibit the information flow at one end of the channel, namely the one owned by the victim process.

Although securing encryption software against these attacks is challenging as it requires consideration of microarchitectural details, the threat of microarchitectural side channels was generally perceived to be limited to a small set of software applications. This general intuition was significantly changed by the discovery of transient execution side channel (TES) attacks.

1.2 Transient Execution Side Channel Attacks

Despite using similar channels for exfiltrating information, TES attacks are fundamentally different compared to classical side channels. TES attacks exploit side effects of *transient instruction execution*, a phenomenon not visible in the sequential execution semantics of the ISA. Processors may transiently execute instructions ahead of time, without ensuring whether or not the flow of the program actually reaches those instructions. If it turns out that the transiently executed instructions are indeed not part of the correct flow of the program, e.g., due to a mispredicted branch, the processor discards their results.

In a TES attack, an attacker exploits certain microarchitectural features, such as speculative execution, to transiently execute a sequence of instructions that leaks the secret through timing side channels, without affecting the ISA-level results of the program. This makes the TES attacks more threatening since the attacker controls *both* ends of the channel, the part that triggers the side effect and sends out the information, and also the part that observes it. In this scenario, a single user-level attacker program can establish a microarchitectural side channel, leaking parts of the memory which is not accessed by any other program. Such hardware *covert* channels not only can destroy the usefulness of encryption and secure authentication schemes, but can steal data essentially anywhere in the system. As a result, unlike classical side channel attacks, TES attacks threaten the overall security of the system and its root of trust.

TES attacks first emerged after the discovery of the Spectre [24] and Meltdown [25] attacks in 2018. The variety of attacks discovered since then (e.g., MDS attacks [26–28], speculative store bypass [29], speculative interference [30], etc.), with many of them targeting a previously patched system (e.g., Fallout attack [26])), has proven that the threat is not limited to Spectre and Meltdown and calls for more attention towards hardware security.

The first response to the threat of TES attacks were software-level patches, such as [31]. Such countermeasures are crucial for fixing legacy systems, but they are not a panacea. Lack of concrete knowledge about the root cause of the problem can lead to software being patched based on the known attacks, potentially rendering systems vulnerable to future attacks (see, e.g., the case of the speculative store bypass attack [29]). Alternatively, the software may be fixed conservatively, preventing also future attacks, however, at the cost of possibly prohibitive performance overhead (up to 200% [32]). Moreover, finding the vulnerable code segments in software requires new software verification

techniques in which the ISA semantics is enhanced based on the speculative execution features of the underlying microarchitecture [33–35]. Analyzing the kernel software with these techniques is a computationally expensive verification task.

Designing secure microarchitectures is the more favorable solution against TESs in the long run. Such microarchitectural designs try to hide or erase the observable footprint produced by transient execution, without completely turning off the speculation to keep some of the performance gains of the corresponding microarchitectural features (e.g., [36]). Besides improving the performance overhead [36] compared to software-only patches, fixing the problem at the microarchitectural level relieves the software developer from having to consider hardware details in security analysis. Consequently, the ISA can, again, be used as a golden model, leading to more scalable software security verification such as by symbolic execution.

The emergence of TES attacks and the need for hardware-level countermeasures add a new objective to the hardware security verification task: the microarchitecture must be analyzed w.r.t. TES attacks. This problem is even more challenging than the conventional hardware security verification, since current verification flows are mostly designed to verify the functional requirements of the system, while TES attacks, by definition, only relate to timing behaviors. Furthermore, there is a clear lack of formal specification for security against TES attacks.

1.3 Challenges in Hardware Security Verification

Hardware verification is an integral part of any hardware design flow. Just like the functional requirements of the design, also the security requirements must be verified by the design team before sign-off. The semiconductor industry currently employs a variety of techniques to verify security aspects of the designs, including simulation, emulation, formal verification, and code audit [37]. Although most of the industry standard verification tools were originally geared towards functional verification, many of them have incorporated security verification apps. Examples of such tools include Incisive [38], Solidify [39], Questa Simulation and Questa Formal [40], OneSpin 360 [41], and JasperGold [42].

In the academic community, despite the rich body of work in software security verification, there has been limited work done regarding security-focused hardware analysis and verification (see Chap. 3 for a detailed review) and there are still many open research questions and challenges that need to be addressed.

Decades of hardware design and verification have led to mature functional verification techniques that can deliver hardware designs of high quality w.r.t. a given functional specification (e.g., see Complete-Interval Property Checking [43–45]). However, the same cannot be said about the security aspects of the hardware design. Hardware security verification faces certain unique challenges that have not yet been properly addressed by either academic or industry-standard approaches. These challenges include:

- **Timing side channels:** Integrity requirements of hardware designs can be expressed by the functional specification and, thus, functional verification techniques can cover these requirements to a large extent. For the case of confidentiality, however, in addition to the functional or explicit information leakages that can violate the requirement, also the *implicit information flows* through timing side channels must be considered. This complicates the verification process significantly, because the functional specification, which is *untimed*, cannot cover such requirement. Therefore, there is a lack of proper specification for security against timing side channels. Most of the commercial formal verification tools were designed to verify functional properties and are not suitable for checking such non-functional requirements.
- **Cross-modular vulnerabilities:** Hardware designs usually consist of several interconnected modules. Many security issues are introduced into the system through the integration of these components and a vulnerability in one component may only be exploited through its communication with other components. Detecting such vulnerabilities requires analyzing information flows across multiple components which is usually a computationally expensive task for formal verification techniques.
- **Hardware/Firmware interactions:** Many hardware security issues are only exposed if triggered by a specific interaction between hardware and the low-level firmware. For example, a bug in the access control mechanism of the bus may only be triggered if the firmware configures the access control unit in a specific way. These bugs may escape many hardware verification techniques if the hardware/firmware interaction is not properly modeled. On the other hand, a complete hardware/firmware co-verification is a very challenging or even infeasible task for most designs.
- **Diversity of the threat models:** Considering the vast variety of applications in which SoCs are deployed, there is an ever-expanding attack surface. Hardware verification engineers need to consider different threat models to cover different use case scenarios and each threat model may impose some unique challenges on the verification task.
- **Drastic effect of escaped bugs:** Despite the fact that software security flaws can have dramatic impact on the computing system, the possibility of deploying software patches in a fast and efficient way has led to the adoption of a *patch-and-pray* approach for software security by many development teams. For the case of hardware security flaws, patching is either very expensive (in terms of sacrificing performance or limiting functionality of the design) or simply impossible for technical reasons. This makes it crucial for a hardware security verification technique to be exhaustive in nature and to deliver well-defined security guarantees. Achieving this goal is only possible by developing scalable formal verification techniques, which are, at the same time, amenable to adoption by current industrial hardware design flows.

1.4 Motivation and Envisioned Approach

The discovery of Spectre and Meltdown marked a turning point in the field of hardware security, for two particular reasons: First, these attacks showed the possibility of circumventing all existing security mechanisms by exploiting a microarchitectural feature that existed for decades in almost any desktop and server computer in the world (see the universal read gadget in Spectre variant 1 [24]). Second, these attacks showed that a security flaw in hardware, i.e., in the root of trust of the system, can render security features across the abstraction layers ineffective.

Ignited by the drastic implications of the discovery of TES attacks, there is a significant interest towards a hardware security verification methodology that can contribute to a well-defined security sign-off process in hardware design flows. Such a verification approach must fulfill certain requirements in order to prevent security flaws in the deployed system and re-instantiate the trust in hardware. Derived from the described challenges in hardware security, a good verification technique should fulfill the following criteria:

- **Exhaustiveness:** Hardware security must be able to *exhaustively* capture all security flaws and provide well defined guarantees about the security at the hardware level. It is crucial to prove the absence of vulnerabilities rather than to merely hunt for the known ones. As a result, a good hardware security verification technique must not be based on expert knowledge about previous attacks and vulnerabilities.
- **Applicability to Cycle-Accurate Models:** Detecting any vulnerability based on a timing side channel is only possible if the verification is applied to cycle-accurate models of the hardware. More abstract models fail to provide enough microarchitectural details to capture timing side channels exhaustively.
- **Adaptability to Current Hardware Design Flows:** To make security verification an integral part of the design sign-off, such a technique must be adaptable to the current hardware design practice. This includes being applicable to Register Transfer Level (RTL) designs written in industry standard HDLs.
- **Scalability:** Hardware security verification techniques must be scalable to a wide range of microarchitectural designs with different levels of complexity. Especially when targeting TESs, it is important to analyze advanced microarchitectural features such as speculative execution, Out-of-Order (OOO) execution, non-blocking data caches, etc.
- **Assisting Designers in Patching the Design:** The security verification process should not end with capturing the vulnerabilities. A good verification approach should provide enough information to the designer to pinpoint the root cause of the vulnerability and assist him/her to develop a design patch that is secure but not overly conservative. This is especially

important for TES attacks, since an efficient microarchitectural countermeasure against these vulnerabilities requires detailed knowledge about the possible side channels. Providing a complete list of TES side channels through formal analysis enables designers to develop secure microarchitectures without resorting to conservative solutions.

- **Reasoning about Hardware/Firmware Interactions:** A scalable hardware security verification technique should also model security-critical hardware/firmware interactions. This is crucial for detecting security bugs that are triggered through specific hardware/firmware interactions. Efficiently modeling the hardware/firmware interaction can also enable the hardware designer to incorporate the firmware-level patches into the hardware verification process and verify the security of the design at the presence of such countermeasures.

1.4.1 Unique Program Execution Checking

Originally motivated by the emerging threat of TES attacks, in this thesis we work towards a unified hardware security verification methodology targeting confidentiality requirements of SoCs. Considering the existing gap in the hardware verification flows regarding security, the main goal of this research is to develop a formal security verification approach that can contribute to hardware design sign-off with well-defined security guarantees.

In this thesis, we propose Unique Program Execution Checking (UPEC), a formal security verification approach for confidentiality requirements of hardware designs. UPEC fully covers the class of TES attacks, as well as confidentiality violations caused by functional design bugs. UPEC is applicable to RTL designs in a wide range of complexity, from simple processors with in-order pipelines to deep out-of-order speculative execution processors. Given a confidentiality requirement, UPEC either certifies the security of the design w.r.t. this requirement, or it automatically generates a counterexample pointing to possible vulnerabilities. The provided security guarantee can significantly boost the trust in the hardware and relieves the software developer from considering complex microarchitectural details for the software security verification.

The key contributions of this thesis are as follows:

(I) We propose a formal definition for microarchitectural vulnerabilities violating confidentiality through explicit or implicit information flows. The definition, unlike previous works, considers a concrete microarchitecture at the Register Transfer Level and does not restrict the leakage to known side channels such as cache footprint. As a result, it also covers so far unknown TES attacks. The formal definition forms the foundation of UPEC and enables us to deliver a well-defined security guarantee to the software level.

(II) The proposed formal definition of confidentiality is transformed into a set of interval properties of finite length. This enables a feasible proof

methodology based on Interval Property Checking (IPC) which can deliver unbounded proofs based on bounded circuit models. Adapting the property to a standard proof methodology contributes to a more feasible, scalable verification methodology which can build upon the existing proof technologies and verification expertise.

(III) UPEC defines a structured and systematic formal methodology for hardware security verification targeting all microarchitectural vulnerabilities including TES attacks in RTL processor designs. UPEC verification methodology is exhaustive, provides security guarantees without requiring a radically different design approach (such as security driven HDLs), and does not rely on expert knowledge about different classes of microarchitectural attacks. UPEC therefore can be easily integrated into existing design flows and can serve as a basis for hardware security verification sign-off at the RT level. The provided guarantee, which stems from our formal definition of security, is a key factor towards re-establishing the hardware as a root-of-trust.

(IV) We introduce an overapproximation of functional correctness in out-of-order speculative execution pipelines which is called microequivalence. Microequivalence allows to address the problem of false counterexamples in UPEC proofs in a systematic way and helps the scalability of the proof by reducing the search space. This is crucial in making UPEC scalable to high-end processors, which are the main targets of TES attacks due to out-of-order and speculative execution features.

(V) So far TES attacks were found through offensive security research techniques, i.e., by mimicking the clever thinking of a human attacker and trying to break into the system. UPEC, for the first time, is able to find vulnerabilities to TES attacks in a highly automated and systematic way. UPEC automatically provides snippets of the attack vector and an execution trace of the information leakage without relying on expert knowledge about the attacks. The provided execution trace also hints towards the root cause of the vulnerability.

(VI) The UPEC verification methodology can accommodate for many of the features found in today's advanced processors, including virtual memory translation, dynamic register mapping, nested speculation and load-store dependency speculation.

(VII) The experiments reported throughout this thesis provide new insights into TES attacks. Through experimental evidence, we show that: (1) TESs are not exclusive to high-end processors with advanced features, such as out-of-order and speculative execution. They can also exist in simple in-order processors. (2) TES vulnerabilities can be introduced by low-level design decisions. Such a vulnerability is extremely hard to detect since it is nearly impossible for the designer to anticipate the security implications of every design decision. (3) Patching TES vulnerabilities, even for known TES

attacks, can be challenging and is prone to design mistakes. Design patches may block certain side channels but at the same time introduce new ones. This motivates the need for a formal security verification technique that can assist designers in the patch process.

(VIII) The effectiveness of the UPEC approach is shown by verifying various RTL designs with different levels of complexity. In our experiments, by merit of the exhaustiveness of our approach, previously unknown vulnerabilities have been found in RocketChip [46], Ariane [47] and BOOM [48]. It should be noted that, to the best of our knowledge, BOOM is the most complex open source RTL design within the RISC-V [49] community, which can deliver performance comparable to ARM Cortex A9 to A15 depending on the configurations. The reported runtime of UPEC confirms the scalability of the approach to processors of realistic size.

(IX) By being applicable to standard RTL designs, UPEC is a perfect candidate for analyzing legacy hardware designs to collect all the necessary information about the TES vulnerabilities in deployed hardware designs. This information may assist software developers to provide efficient countermeasures against TES attacks at the software level and avoid overconservative solutions.

(X) The UPEC computational model allows for modeling certain software behaviors using input constraints. This allows to exclude irrelevant software behaviors and also to incorporate software constraints that are provided by the software developers. In case the designer decides to patch a vulnerability at the software level, UPEC can verify the hardware under appropriate input constraints which model the designed software level countermeasure. This can be the starting point to develop more advanced hardware/software contracts with formal guarantees at the hardware level.

(XI) The UPEC verification methodology and computational model bears promise for hardware verification targeting different security goals (e.g., security against classical side channels, data oblivious computing). UPEC also showed the scalability of 2-safety models, which can be useful in functional verification. In this thesis, we describe a UPEC-based verification approach for capturing all security-critical design bugs as well as a technique for verifying data-obliviousness of hardware modules. A processor verification technique is also described, which benefits from the 2-safety model, similar to UPEC computational model.

1.5 Thesis Overview

The rest of the thesis is organized as follows: Chapter 2 provides the background information necessary to understand the topic of this thesis, including relevant computer architecture topics, examples of side channel and TES attacks and a

brief overview of formal hardware verification. In Chapter 3, a detailed overview of the literature regarding hardware-related security issues is provided.

The main contributions of this thesis are presented starting with Chapter 4.

Chapter 4 describes the theoretical foundation for UPEC. In this chapter, a formal definition for hardware vulnerabilities violating the confidentiality requirement is proposed. Chapter 5 and 6 propose a feasible and scalable proof methodology and verification flow based on the UPEC formal definition of confidentiality. These chapters describe how the UPEC property can be refined into interval properties of finite length and how it can be verified using IPC and bounded circuit models.

In Chapter 7, our verification methodology is further extended to accommodate features of high-end processors, including out-of-order and speculative execution, dynamic register mapping, virtual memory translation and multi-core architectures. To deal with the complexity of out-of-order processors, a novel invariant called *microequivalence* is proposed, which overapproximates the functional correctness of the design and significantly contributes to the scalability of the UPEC approach in complex processors.

Case studies with UPEC produced new insights into hardware security. They are described in detail in Chapter 8. These insights can show the unique challenge of ensuring security at the hardware level and motivate using UPEC as an exhaustive verification methodology. A detailed account of our UPEC case studies is presented in chapter 9. These case studies include verifying three different RISC-V cores, RocketChip [46], Ariane [47] and BOOM [48], using the UPEC verification methodology.

Chapter 10 explores UPEC-based verification techniques pursuing different security goals. These include a verification technique exhaustively targeting security-critical design bugs, and a formal approach for verifying data-oblivious execution of hardware modules. In this chapter, the benefit of a 2-safety model, similar to the computational model of UPEC, for functional verification is evaluated and a novel processor verification technique is proposed.

A summary of the work as well as a discussion on the results achieved conclude the main contributions of the thesis in Chapter 11.

Publications

We already published several aspects of this disseration, as shown by the following bibliography.

[1] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz. "Symbolic Quick Error Detection using Symbolic Initial State for Pre-silicon Verification". In: *Design, Automation & Test in Europe Conference (DATE)*. IEEE. 2018, pp. 55–60.

[2] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *Design, Automation & Test in Europe Conf. (DATE)*. 2019, pp. 994–999.

[3] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz. "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors". In: *IEEE/ACM Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[4] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz. "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors". In: *IEEE Transactions on Computers* (2022), pp. 1–1. DOI: 10.1109/TC.2022.3152666.

[5] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz. "Gap-free Processor Verification with S²QED and Property Generation". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France*. Mar. 2020.

[6] J. Müller, M. R. Fadiheh, A. L. Duque Anton, T. Eisenbarth, D. Stoffel, and W. Kunz. "A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 991–996.

[7] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz. "Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing". In: *2022 59th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2022.

[8] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *arXiv preprint arXiv:1812.04975* (2018).

[9] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. Barrett, W. Ecker, et al. "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1000–1005.

[10] K. Ganesan, F. Lonsing, S. S. Nuthakki, E. Singh, M. R. Fadiheh, W. Kunz, D. Stoffel, C. Barrett, and S. Mitra. "Effective Pre-Silicon Verification of Processor Cores by Breaking the Bounds of Symbolic Quick Error Detection". In: *arXiv preprint arXiv:2106.10392* (2021).

Chapter 2

Background

In this chapter, a brief overview of the computer architecture, security, and verification concepts relevant to this thesis is provided. Out-of-order and speculative execution are described in more detail, which are the basic features that enable most of the known TES attacks. An example of a cache side channel attack as well as the Spectre and Meltdown attacks are elaborated to give an intuition about the nature of side channel attacks. Finally, the concept of Interval Property Checking (IPC) is explained, which serves as the backend for the UPEC verification methodology.

2.1 Out-of-Order and Speculative Execution

Out-of-order execution is a microarchitectural feature for optimizing processor utilization by avoiding unnecessary stalls in the pipeline due to data or structural hazards and by exploiting parallelism inherent in the instruction stream [50].

Processors usually employ several functional units, each dedicated to a certain type of instructions/operations. In an in-order pipeline, it is hard to fully utilize all the existing functional units at all times, due to structural or data hazards. Typically, in an in-order pipeline, instructions are issued to functional units in *program order*. This means a structural or data hazard for one instruction can make all of the subsequent instructions wait and thus, a large part of the available functional units in the pipeline will not be utilized until the hazard for the instruction at the bottleneck is resolved.

Tomasulo developed an algorithm for dynamic scheduling of instructions to enable efficient out-of-order execution [51]. The basic idea is to issue instructions as soon as the operands and the corresponding functional unit are available, regardless of the program order. The results of instructions, which may be produced in an arbitrary order, are temporarily stored in a buffer called *Reorder Buffer* (ROB). In the instruction commit stage, the ROB writes back the results to the architectural registers in program order. As a result, although the processor executes instructions in an arbitrary order, in the view of the software level the

results are generated in program order.

The Tomasulo scheme can address many different hazard scenarios without stalling the pipeline, by introducing various bookkeeping mechanisms such as reservation stations and register renaming.

Conditional branch and jump instructions pose extra challenge in out-of-order execution processors since the next instruction after a jump or conditional branch is only known after the branch condition and target address have been computed. This can significantly limit processor utilization as each control flow-related instruction can induce several stalls in the pipeline. The main solution to address the issue is to predict the outcome of a conditional branch/jump ahead of its execution and then *speculatively* execute the next instructions based on the prediction [50].

For the case of a conditional branch instruction, the first prediction must be made on the outcome of the branch (i.e., taken vs. not taken). In the next step, for both branch and jump instructions, also the target address must be predicted, i.e., the address from which the next instruction will be fetched. There is a variety of techniques to make rather accurate predictions for each of these cases. These prediction techniques employ dynamic approaches in which the predictor learns through the course of the program to make a more accurate prediction as the program runs.

Once the conditional branch or jump instruction is actually executed, the processor checks whether the prediction was correct. In case of a misprediction, the processor must discard all the instructions that have been speculatively executed and repeat the program execution with the correct flow.

Out-of-Order and speculative execution allow processors to execute instructions in a *transient* manner. This means an instruction may be executed before the processor is certain whether the instruction is part of the program. As a result, different events in a pipeline, such as an exception, may render the younger in-flight instructions invalid and the processor has to throw away their results although they have already finished execution. This is the key factor that has enabled TES attacks in modern processors.

More information about out-of-order and speculative execution as well as other relevant microarchitectural features can be found in [50].

2.2 Cache Side Channel Attacks

Memory access is usually the slowest operation within a processor core and therefore it can easily create a bottleneck limiting the performance. To speed up memory transactions, processors typically employ a hierarchy of caches, which buffer frequently accessed instructions and data. Cache mitigates the latency of slow memory transactions by working on buffered copies of the data/instructions rather than accessing the slow main memory.

This speedup comes with the side effect that the pattern of memory accesses leaves a footprint in the cache: reading the data that has been accessed by the previous process can be faster because of a cache hit. This timing difference

is exploited by the cache side channel attacks that mainly target encryption algorithms. These attacks try to deduce information about the encryption keys based on the footprint of the encryption software on the cache hierarchy.

A variety of cache side channel attacks have been developed by security researchers (e.g., [15, 17, 19]). These attacks typically consist of the following phases:

1. The attacker primes the cache into a known clean state, for example by flushing all cache lines or filling the cache with certain known addresses.
2. The victim software begins its computation and changes the state of the cache in the course of its execution.
3. The attacker takes back control and tries to decipher the cache state by measuring the time for accessing certain addresses. A fast access indicates a cache hit and a slow one a cache miss. Based on this information, the attacker is able to find out which cache lines have been evicted or which cache lines are refilled by the victim.
4. The attacker deduces information about certain confidential assets of the victim by correlating the refilled/evicted cache lines with the program run by the victim.

In the next section we present a more detailed example of such an attack in order to provide the reader with an intuition into the key elements.

2.2.1 FLUSH+RELOAD Attack on RSA

RSA is a public cryptosystem for encryption and signing. Here we describe how certain implementations of the RSA decryption function can leak information about private keys through the cache side channel at the example of the FLUSH+RELOAD attack [15].

A common optimization for RSA decryption function is to use a square-and-multiply algorithm for exponentiation [52]. This algorithm performs fast exponentiation by iterating through a binary representation of the exponent. For each bit, if the bit is set, an additional multiply-and-modulo reduce operation is conducted. Figure 2.1 shows a pseudo-code implementation of the square-and-multiply algorithm.

The sequence of operations executed in the square-and-multiply algorithm directly corresponds with the bits of the exponent. Considering the fact that within RSA decryption, the exponent is part of the private key, this correspondence can create a cache side channel through the instruction cache.

In a FLUSH+RELOAD attack, the attacker and the victim run on two separate cores with a shared last-level cache. Furthermore, the attacker is able to flush cache lines holding the instruction memory of the RSA victim software. This is usually possible due to the page sharing mechanism in most operating systems [15].

A round of attack runs in three phases:

```
1: procedure EXPONENT(b, e, m)                    ▷ computes b^e mod m
2:     x ← 1
3:     for i ← ⌊log2(e)⌋ + 1 down to 1 do
4:         x ← x^2
5:         x ← x mod m
6:         if e_i = 1 then                        ▷ i'th bit of binary representation of e
7:             x ← x * b
8:             x ← x mod m
9:     return x
```

Figure 2.1: Pseudo-code of square-and-multiply algorithm

1. The attacker *flushes* the cache lines holding the instructions for lines 7 and 8 of the square-and-multiply algorithm (Fig. 2.1).

2. The attacker waits for the victim to run one iteration of the square-and-multiply algorithm, which may or may not execute lines 7 and 8 of the algorithm.

3. The attacker *reloads* the memory addresses holding lines 7 and 8 of the square-and-multiply algorithm and measures the time needed to do so. In case of a cache hit, it means that the memory lines have been brought to the cache by the victim, meaning the corresponding bit of the exponent is one. Otherwise, the corresponding bit is zero.

These three phases are repeated to capture every bit of the exponent in RSA decryption, which enables the attacker to reconstruct the entire private key of the victim.

This example shows how security-critical applications can leave an observable footprint that can be exploited to deduce information about confidential assets of the system. Other side channel attacks may exploit different microarchitectural features, such as a floating point unit [21] or DRAM ports [20]. It is important to note that, in a side channel attack, the attacker must have detailed knowledge about the victim, and the attacker can only steal confidential data that is being actively processed by the victim.

2.3 Transient Execution Side Channel Attacks

The research in the field of TES attacks was initiated by the discovery of Spectre [25] and Meltdown [24] and led to finding many different variants of these attacks. In this section, we describe Spectre and Meltdown in more details to give a better intuition about TES attacks. These attacks not only provide good examples to understand the basic mechanism of a TES attack, but also they define the two main categories of TES attacks. A formal definition of these categories is provided in Chapter 4

```
1   // protected_ptr: a pointer to a protected memory content
2   x = *(protected_ptr); // raises exception
3   // the line below is never reached
4   temp = probe_array[x]);
```

Figure 2.2: Snippet of the Meltdown attack in C-code

2.3.1 Meltdown

The Meltdown attack exploits the way certain out-of-order execution processors handle exceptions to leak the content of the kernel memory through microarchitectural side channels. The attack affects many of the available commodity processors and has drastic security implications since it leaks the kernel memory without depending on any software vulnerability. Meltdown can technically destroy the entire memory isolation of the system. In this section, we describe a simplified version of the attack.

CPUs provide virtual address translation to isolate the memory dedicated to each process. Virtual addresses are translated to physical addresses by a memory management unit based on a translation table. These translation tables are called *page tables* and are stored in the data memory. Each process has its own page table, which makes it possible for the operating system to easily isolate the memory space of each process. Upon every context switch, also the page table must be switched to the new process. In most operating systems, to avoid unnecessary overhead in the frequent context switches between a user-level process and the kernel, the mapping for the kernel address space also exists within the user-level page tables, so a switch between user level and kernel level processes does not involve switching the page table. The content of the kernel memory is protected from user-level accesses by throwing an exception whenever the user-level process tries to read the content of the kernel memory. The way this exception is raised and handled by the processor is the key factor enabling the Meltdown attack.

As explained earlier, out-of-order execution allows for instructions to be executed out of the program order as long as the program-visible events, e.g., an exception, occur in order. An instruction may be executed out-of-order, but in case it encounters an exception, that exception will only be raised when the faulting instruction reaches the head of the reorder buffer. For the case of a load instruction that causes an exception due to accessing kernel memory, this delay in raising the exception creates a time window in which the data accessed by the load instruction can be used by other transient instructions to leave a microarchitectural footprint. This is the basic mechanism enabling a Meltdown attack.

Fig. 2.2 shows a snippet of the attack. The code snippet reads a secret from a protected part of the memory, and uses the secret as an index to access an element within the probe array. Although the exception prevents the second

```
1   // x is an attacker controller input.
2   if (x < array1_size) {
3     y = array2[array1[x] * 4096];
4   }
```

Figure 2.3: Spectre gadget in C-code

memory access, it may still alter the state of the cache by refilling or evicting a cache line. Similar to the FLUSH+RELOAD attack, the attacker flushes any cache line holding elements of the probe array prior to the attack. Once the code snippet of Fig. 2.2 is executed and the attacker takes back control through the exception handler, the attacker reloads the entire probe array and measures the timing of each access. For only of one the elements, there will be a fast access due to a cache hit, which is the element referenced by the secret. The attacker can repeat this process to read the entire memory as long as an address mapping for the targeted address exists within its page table.

2.3.2 Spectre

Spectre attacks exploit speculation features in modern microarchitectures to trick the processor into transiently executing pre-selected gadgets (=exploitable instruction sequences existing within the kernel software, triggered by the attacker and running in privileged mode). These gadgets access the secret and leak it through microarchitectural side channels. Exploiting gadgets within the victim software is a known practice that has also been used in *return-oriented programming* attacks [53]. Unlike return-oriented programming, Spectre attacks do not require the presence of software vulnerabilities such as stack overflows to exploit the gadgets, but they rather exploit the gadgets by tricking the processor into executing them transiently.

Fig. 2.3 shows an example of a Spectre-exploitable gadget. It shows a coding practice that is commonly used across different libraries: before accessing an internal array through an index variable which comes from inputs, the index is checked against the size of the array to avoid invalid out-of-range accesses.

In a Spectre attack, the attacker starts with (mis-)training the branch predictor so that the *if condition* will be predicted to be true before array_size1 is known. Then, the attacker calls the victim software which contains the selected gadget and selects a value for x such that $x >$ array1_size and array1[x] points to a secret within the victim's memory. From an ISA-level point of view, the code inside the if block will never execute since x is greater than the size of the array1. However, due to branch predictor mis-training, the processor will speculatively execute the code snippet while waiting for array1_size to be loaded from the memory. This transient execution leaves a footprint in the data cache that can be observed using various techniques, such as FLUSH+RELOAD.

The same effect can be achieved by mis-training the prediction unit predicting

the target address in a function call. In this way, the attacker can trick the processor into transiently executing gadgets within the attacker program, which can create a *universal read gadget*. This can have catastrophic consequences since it means an attacker can read the entire memory without relying on any software vulnerability.

2.4 Formal Hardware Verification Techniques

Hardware verification is the part of the design flow in which the design is checked against its specification. In this realm, formal verification techniques aim to check the design correctness by mathematical reasoning rather than simulation or testing. In this process, the specification is first transformed into a set of temporal and logic formulas called *properties* and then a mathematical proof method checks whether the design fulfills these formulas under every legal input.

While the rest of the thesis is primarily focused on how to formally specify certain security requirements as properties, in this section we outline the proof methodology that is used for verifying UPEC properties. We first describe the Bounded Model Checking (BMC) approach and then proceed to elaborate Interval Property Checking (IPC), which is the proof method used within the UPEC verification approach.

2.4.1 Bounded Model Checking

Bounded Model Checking (BMC) [54] is a model checking approach which leverages the power of modern satisfiability solving (SAT solvers) to verify temporal properties of sequential circuits.

A SAT solver determines, for a given Boolean function, whether or not the function is satisfiable, i.e., whether or not there exists an input combination under which the function evaluates to "true". In case the function is satisfiable, the SAT solver returns a set of values for the input variables under which the function evaluates to "true". The computational complexity of this problem is, in the worst case, exponential in the number of variables in the function. Modern SAT solvers, however, employ various heuristic methods and can often deliver a solution in a reasonable time for rather large Boolean functions [55–58]. The advancement of these techniques motivated the use of SAT solvers for hardware verification by bounded model checking.

In bounded model checking, as the name suggests, the model checking problem is reduced to a finite *bounded* time interval. To this end, the temporal property is formulated for a finite time window of length k starting from a known state of the system, often the reset state. To transform the proof problem into a SAT problem, we need to create a Boolean function such that any valuation to its variables that satisfies the function is a counterexample to the specified property. In this way, if the function is proven to be unsatisfiable, it is equivalent to proving the absence of counterexamples within the bounded time window. This problem can be formulated by creating an unrolled circuit model.

A sequential circuit can be modeled as shown in Fig. 2.4, with I as the set of inputs and O as the set of outputs. The transition function δ computes the next state and the output function λ computes the output signal values as a function of input and current state. The circuit behavior over a finite time interval of length k can be expressed by unrolling the combinational part of the circuit and connecting the next-state output of λ to the current-state input of the next δ and λ functions (Fig. 2.5). Using this model, the temporal property can be expressed as a simple Boolean function, with the inputs of the circuit being the free variables of the function. Based on the unrolled circuit model, the SAT solver checks whether there exists any valuation to the inputs which can violate the property within k clock cycles from the state s_0.

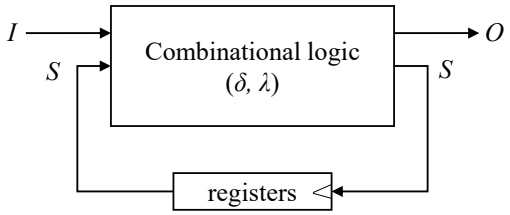


Figure 2.4: Sequential design model

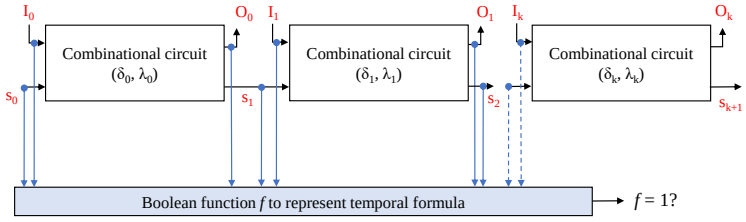


Figure 2.5: BMC iterative design model for a bound depth of k

BMC is a powerful bug detection technique as it exhaustively searches through all possible valuations of inputs within the considered time window. However, when the property holds, it is only proven for a time window of k clock cycles, i.e., the property is only proven to be valid for states reachable from s_0 within k clock cycles. To prove a property for all reachable states using BMC, k must be chosen such that it is at least as large as the sequential depth of the circuit. The sequential depth can be defined as the minimum number of clock cycles needed to reach all reachable states. It is usually infeasible for BMC to prove properties for the entire sequential depth of the circuit, due to the complexity of the unrolled model. As a result, BMC usually falls short of making guarantees about the correctness of the design and can only provide bounded proofs. A clear statement on the absence of bugs requires an *unbounded* proof, i.e., a proof

that is globally valid and is not restricted to a bounded time interval.

2.4.2 Interval Property Checking

Interval Property Checking (IPC) is a SAT-based model checking approach that, similar to BMC, works with an unrolled circuit model. However, unlike BMC, it is capable of providing unbounded proofs for a certain class of properties called *interval properties* [44]. An interval property is a temporal formula that describes an implication such that both assumption (antecedent) and commitment (consequent) are described over a finite time interval.

The key characteristic that enables IPC to deliver an unbounded proof based on a bounded circuit model is using a *symbolic* initial state in the unrolled circuit model.

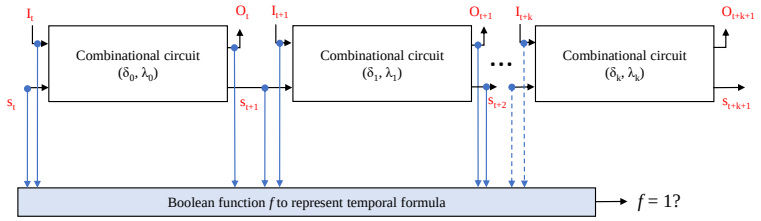


Figure 2.6: IPC iterative design model for a bound depth of k

Fig. 2.6 shows the computational model used for IPC. Similar to BMC, the circuit is unrolled for the time window of the property. The main difference here is that the starting state of the unrolled circuit model is a symbolic variable s_t rather than the reset state of the system. s_t is a free variable for the SAT solver similar to input variables, which can only be constrained by the property assumptions. With this computational model, the SAT solver also explores all possible valuations for s_t, which as a subset includes all reachable states of the system. Therefore, the proof result is unbounded and valid for all reachable states of the system.

The unbounded proof of IPC comes at the cost of possible false counterexamples in the proof. By considering any possible valuation for s_t, the proof also includes unreachable states as the starting state of the unrolled model. If the property fails for a valuation of s_t which is unreachable, the provided counterexample is a false one since it cannot happen within the normal operation of the system. This problem must be addressed by adding reachability information to the starting state s_t in the form of invariants [59].

It worth noting that in IPC, *false holds*, i.e., false proof of absence of property violation, cannot occur since the proof result is valid for any reachable state.

This characteristic makes IPC a perfect proof methodology for our security verification technique that can enable delivering formal security guarantees.

Chapter 3

State of the Art in Hardware Security

In this chapter, we summarize major works addressing hardware-related security issues, with a focus on TES attacks.

3.1 Addressing TES Attacks at the Software Level

Most of the attempts for formally defining TES attacks have been made at the software level to enable effective software verification against Spectre attacks. The main goal of such verification techniques is to find the exploitable gadgets in order to apply proper hardening, e.g., by inserting synchronization barriers such as fence instructions. Although software verification is a fundamentally different problem compared to hardware verification which is the main focus of this work, it is important to look into relevant software verification approaches to create a thorough overview of formal models describing TES attacks.

The work of Mcilroy et al. [60] was among the first attempts for developing a mathematical model for microarchitectural side channels and TES attacks from a software perspective. The model provides a good basis for understanding the security implications of microarchitectural optimizations. However, it may not be suitable to form a basis for a verification methodology. In this model, a microarchitectural leak consists of writing a secret into the final microarchitectural state of a program which is observable by another program, e.g., the state of the cache. This model does not cover all possible TES attacks and misses attack scenarios in which the secret is leaked through the timing of the victim program rather than its final state (see, e.g., the Spectre-STC attack in Sec. 8.2).

Transient instructions are not observable by analyzing the software based on the semantics provided by the ISA. As a result, new techniques are proposed to verify software against speculative attacks. Most of these techniques extend the existing software verification approaches by augmenting the ISA semantics with some abstract annotations of underlying hardware features (e.g., [33, 61–63]).

Cauligi et al. [33] defined a new operational semantics based on speculative execution and extended the constant-time programming paradigm to capture exploitable gadgets for speculative attacks. KLEESpectre [61], OO7 [62] and SpecuSym [63] are other examples of such methods which extend symbolic execution semantics with the modeling of speculative execution.

Cheang et al. [34] formally defined *Secure Speculation* as an observational determinism property over four traces. A related approach was proposed by [35], who formalized *Speculative Non-interference* as the requirement for security against speculative execution attacks. The basic idea is to verify at the software level whether there is any security violation (w.r.t. a defined security policy) that can only occur if the program is executed using the speculative semantics. Therefore, these formulations, which can be seen as a mathematical definition for Spectre attacks, are effective in verifying software. However, secure speculation, speculative non-interference, and other efforts for formalizing TES attacks [64, 65] do not provide a generic method for detecting TES attacks in hardware since they do not take the microarchitecture into account. Adopting software security concepts to hardware verification is not trivial, as shown for the example of constant time execution by [66].

Patrignani et al. [67] formally specified the security requirement for compiler-level countermeasures against speculative attacks and the proposed property is used to verify certain countermeasures such as speculative load hardening.

New hardware/software contracts have been developed in [68] as a framework to reason more precisely about what information hardware leaks and what consequences this has for software security requirements. This provides a useful framework to evaluate security at the software level, without leaving a gap due to certain microarchitectural features. However, it has not yet been examined how the hardware side of the high-level contracts can be mapped to the RTL implementation by RTL properties and whether these properties scale for state-of-the-art commercial property checking.

Non-formal approaches have also been applied to verify software security against speculative attacks. SpecFuz [69] is a fuzzing-based technique which aims to target speculative vulnerabilities in software by forcing the system to execute wrong branches. A Machine learning based technique is proposed in [70] to verify software using a neural network classifier that can detect Spectre gadgets.

3.2 Addressing TES Attacks at the Hardware Level

Software-level patches, such as in [31], are crucial for protecting legacy systems against TES attacks, but they are not a panacea. Lack of concrete knowledge about the root cause of the problem can lead to software being patched based on the known attacks, potentially rendering systems vulnerable to future attacks (see, e.g., the case of speculative store bypass attack [29]). Alternatively, the software may be fixed conservatively, preventing future attacks, however, at the cost of possibly prohibitive performance overhead (up to 200% [32]).

Compared to software-only patches, enhancing security at the hardware level

is the more favorable protection against TES attacks in the long run. Besides improving the performance overhead [36], microarchitectural countermeasures can relieve the software developer from having to consider hardware details in the security analysis.

A variety of techniques have been proposed to design Spectre-resistant cache subsystems, by hiding or preventing speculative memory transactions in the cache. InvisiSpec [71] proposed a cache architecture which makes the speculation invisible in the cache, by simply buffering the effect of speculative loads in some shadow registers until the speculation is resolved. Gonzalez et al. [72] implemented a similar approach on a RISC-V SoC design. MounTrap [73] also pursued a related idea by introducing a new L0 cache into the cache hierarchy between the processor and L1 Cache, to transiently buffer the speculative cache state, which led to improved performance. This kind of approach was also pursued in [74]. CleanupSpec [75] took a different approach by removing the footprint on the cache after each misprediction rather than preventing the footprint in the first place. In this way, CleanUpSpec only incurs overhead in case of a misprediction which is less often. ReversiSpec [76] extended this idea to multi-core cache systems, by introducing a novel reversible cache coherency protocol. Delay-on-miss was another novel cache design, which only blocks speculative loads in case of a cache miss [77]. As a result, the state of the cache cannot be altered through speculative cache refills. This idea was further enhanced by value recomputation techniques to reduce the number of load instructions in a program [78]. Further evaluation of these cache designs and comparison with other, similar, cache architectures can be found in [79, 80]. Zhao et al. [81] developed a Spectre-resistant cache controller based on the Delay-on-miss approach and evaluated the performance and area overhead at the RTL. These cache architectures incur relatively low performance overhead compared to the baseline design. All these techniques, however, block only the TES attacks that are based on a cache side channel.

A more comprehensive approach for blocking speculative attacks is by re-designing the speculative execution feature of the microarchitecture. The goal of these approaches is to protect not only against the cache-based Spectre attacks, but also against any other possible TES attack.

NDA [82] and SpecShield [83] prevent TES attacks by blocking forwarding between speculative instructions. SpecGuard [84] and ConTExT [85] further improve this idea by incorporating an additional bit in the page table to mark sensitive data, and only block speculative propagation of the data loaded from sensitive memory pages. Although these techniques block the information leakage at its source and protect against all variants of attacks, they are rather conservative and incur significant overhead, more than 300% in certain security-critical applications.

Speculative Taint Tracking (STT) [36] further improves the previous techniques by selectively forwarding the results of speculative load instructions (or any instruction that accesses the secret), as long as the forwarded data does not reach a timing side channel. This is ensured by identifying possible side channels in a microarchitecture and the instructions that can trigger them. The work is further extended by speculative data-oblivious execution [86]: The idea is to allow

speculative execution of unsafe instructions, i.e., instructions that can leak their operands, but only based on secret-independent prediction of their operands rather than their actual value. This prediction will be resolved as soon as the instruction is no longer speculative. Another information flow tracking approach is pursued in DOLMA [87], which employs various performance optimizations to improve the performance overhead of secure speculation. Secure speculation architectures such as STT and DOLMA can be augmented by compile-time optimizations to minimize the number of stalls in the pipeline and improve performance [88, 89].

Evaluating the performance of these design ideas at an abstract level (e.g., using the Gem5 simulator [90]) indicated that they can deliver competitive performance compared to vulnerable baseline designs. However, the claimed security guarantees can only be delivered if an RTL verification sign-off is carried out on the implemented microarchitecture. The variety of vulnerabilities in older approaches that have been found and patched by each new approach attest to this. (For example, refer to DOLMA [87] compared to STT [36].) Furthermore, in approaches such as DOLMA [87] and STT [36], achieving the security guarantee without sacrificing performance requires identifying all micro-operations that can form a side channel. Although some of these micro-operations may be known a priori to the designer, our experiments (especially in the case of Spectre-STC attack) have shown that detailed RTL design decisions can create new side channels for certain micro-operations. It is nearly impossible for the designer to anticipate all these scenarios. Thus, achieving the desired level of security and performance on the RTL design requires a scalable formal security analysis. For these reasons, we believe our proposed methodology can provide synergy with hardware-based Spectre countermeasures. Each counterexample in our proof methodology points to a TES which helps the designer to implement the security countermeasures only for the necessary scenarios so that conservative decisions are avoided. Furthermore, our formal verification approach can relieve the designer from a detailed analysis of all instructions for identifying the possible side channels, since the UPEC formal proof automatically finds all relevant cases.

HyperFlow [91] is an example of an SoC completely designed using a special, security-driven HDL. The non-interference property implemented by HyperFlow is a strong security measure that can block many attack scenarios in a computing system, including both classical side-channel attacks and TES attacks. However, the security guarantee comes at a high price in performance and memory overhead, and imposes drastic changes in the overall design flow, from HW design to OS development.

3.3 Hardware Verification Techniques

Cabodi et al. [92] present one of the first hardware security verification approaches targeting speculative attacks, extending over their previous work [93] and the work of [94], and pioneering the adoption of *taint analysis* in the hardware domain. Wei et al. [95] also proposed a model for precise taint propagation and

taint property verification. It is, however, in the nature of taint analysis that it requires assumptions about the paths to be analyzed. As a result, such approaches can be effective in checking a design for known variants of Spectre, but face their limitations with respect to unknown variants. Furthermore, in contrast to the analysis of [92], which is conducted on an abstract model of the processor, the approach proposed in this thesis is applied to its RTL implementation.

Iodine [66] proposes an approach for verifying constant-time properties for hardware at the RTL. The tool receives a hardware design, a set of source and sink nodes and a set of assumptions, and proves that under the given assumptions, the timing for information propagation between the given source and sink is constant. The constant-time execution for hardware defines a more conservative security requirement for the system, compared to our approach which targets specifically the absence of TES attacks. Constant-time execution in hardware aims at providing the underlying infrastructure for data-oblivious implementation of certain applications such as encryption. As a result, although Iodine has been proven effective in verifying the absence of timing leakage for certain security-critical operations or security-critical components, the targeted security requirement may be too conservative and thus not applicable to general application processors in an SoC. Furthermore, similar to taint property verification techniques, the approach faces its limitations in finding previously unknown timing vulnerabilities because the user needs to specify the source and sink of a leakage path.

Zhang et al [96] proposed a novel approach in verifying security properties, by translating the hardware design and its security assertions into an LLVM [97]model and then verifying the assertions using backward symbolic execution. The proposed approach is orthogonal to UPEC, since it improves the efficiency of the backend solver rather than specifying the required security properties.

Another line of research is language-based security. It advocates the use of more expressive security-driven hardware description languages. These languages usually use a type system that forbids explicit (direct value assignment) and/or implicit (conditional assignment) information flows between certain security types (security labels), according to a security policy. Caison [98] and Sapper [99] are examples of security HDLs which enable the user to design hardware with the desired information flow properties. In Caisson and Sapper, the designer must annotate each register in the design with security types (labels), and the code can be checked with the designed type system for security violations. Caisson uses a static type system, which forces the designer to duplicate the logic for certain paths violating the type system. Sapper improved Caisson by using dynamic types. However, using a completely new HDL language is usually prohibitive for design teams. Bidmeshki and Markis [100] developed the VeriCoq-IFT framework which automatically converts the design to the Coq formal language [101] and generates a security property theorem based on an information flow policy. Although this removes the need for using a new HDL, the designer still needs to annotate the Coq code with security types. Furthermore, all of the above languages employ a conservative information tracking scheme in

their type system, which creates an overestimate about the possible information flows and therefore leads to conservative and possibly inefficient designs.

SecVerilog [102] extends the Verilog language with a security type system. The designer needs to label storage elements with security types, which enables enforcing information flow properties. SecVerilog implements precise information flow tracking by using predicate analysis and constraint solving. This solves the overestimation problem of other languages. Similar to SecVerilog, ChiselFlow is proposed in [91] which extends the Chisel Language [103] with a security type system. ChiselFlow partially automates the labeling process to mitigate the incurred manual effort. Although using Verilog and Chisel as the base of the language eases the adoption of the method, the labeling process is complicated and security violations are hard to debug in these approaches. Furthermore, the designer may need to adapt the labels in the design in order to verify different security properties. In addition to the effort associated with the system-wide labeling of the RTL design, changing design methodologies in established design teams usually is not desirable.

Other approaches have targeted verifying hardware security against TES attacks at abstract levels. InSpectre [104] provided a framework for formally reasoning about different security countermeasures, using a formal microarchitectural model with speculative and out-of-order execution semantics. UCLID5 [105] can also be used to verify the security of different microarchitectural design schemes. CheckMate [106] is a program-synthesis based technique to synthesize attacks based on certain execution patterns and abstract microarchitecture models. The synthesized attacks can be used to test the security of the system. All these techniques provide important insight into the design flow by early capturing of security problems and consequently delivering RTL designs with higher quality leading to lower RTL verification efforts. However, they cannot serve as a sign-off verification tool as they do not operate on the RTL design. Moreover, they fail to detect security vulnerabilities that are introduced due to detailed RTL design decisions, e.g., the attacks described in Chapter 8.

Furthermore, there are promising techniques based on hardware fuzzing, such as IntroSpectre [107], Osiris [108], and HyperFuzzing [109]. Their non-exhaustive nature, however, does not allow them to deliver formal guarantees when signing off a hardware design.

Chapter 4

Unique Program Execution Checking

The first step towards a security verification approach with mathematically proven security guarantees is to formally define the security requirement and relate it to the hardware model. The formal definition forms the basis for the hardware properties that shall be proven for security verification. This first step is especially challenging when considering the confidentiality requirement since it must provide a general definition without relying on a priori knowledge about previous attacks.

In this section, we propose a mathematical definition for *Unique Program Execution* as a requirement for confidentiality at the hardware level and refine the formulation such that it exclusively targets the class of TES attacks. Since UPEC should be applicable to design implementations at the RTL, all elements of the following definitions directly relate to components of a CPU's microarchitecture.

4.1 Auxiliary Definitions

The contents of the program-visible registers and data memory (including cache) are assumed to be partitioned into two sets: a set of public information, X_p, and a set of confidential information, X_c. A program P is the content of instruction memory together with a start address and an end address. It receives X_c and X_p as its inputs. In the microarchitectural implementation of our system, we consider a microarchitectural state S, which is defined for all state bits other than those holding the inputs, X_p and X_c, to the program. S_0 is the initial (starting) state for executing a program P. It should be noted that S_0 is different from the reset state of the CPU since a program can be started at any point during the run time of the system.

Based on the aforementioned concepts, we distinguish the following notions:

Definition 1 (Architectural Observation)**.** *The architectural observation $O(P, X_p, X_c)$ of a program P is the (time-abstract) sequence of valuations to the program-visible (architectural) registers, as they are produced by committing instructions according to the ISA specification.* □

The architectural observation is independent of any hardware implementation of the ISA and does not relate to any microarchitectural features such as cache or pipelining. It can be seen as the result of simulating the program by an ISA simulator.

Definition 2 (Microarchitectural Execution)**.** *The microarchitectural execution $\xi(S_0, P, X_p, X_c)$ of a program P is the (clock-cycle-accurate) sequence of valuations to the program-visible (architectural) registers, as they are produced during execution of P on a specific hardware implementation/microarchitecture.* □

The microarchitectural execution can be seen as the clock-cycle-accurate result of simulating the program on the hardware design using an RTL simulator.

Definition 3 (Microarchitectural Observation)**.** *The microarchitectural observation $\mu(S_0, P, X_p, X_c)$ of a program P is the (time-abstract) sequence of valuations to the program-visible (architectural) registers, as they are produced by committing instructions on a specific hardware implementation/microarchitecture.* □

In this definition, committing instructions are those instructions that finish their execution and leave the pipeline, i.e., they have written to the program-visible registers.

The microarchitectural observation can be obtained from the microarchitectural execution by keeping in the sequence all valuations to the architectural registers at the time points they are written (due to instruction commitment) and discarding all other "intermediate" valuations. For a functionally correct microarchitecture, for any tuple (P, X_p, X_c), the microarchitectural and architectural observation are the same.

4.2 Unique Program Execution as a Confidentiality Requirement

Using these notions we can now formulate *Unique Program Execution* which is the security requirement for an SoC to provide the root of trust for confidentiality.

Definition 4 (Unique Program Execution)**.** *A program executes uniquely w.r.t. a set of confidential information X_c if and only if the sequence of valuations to the set of architectural state variables is independent of X_c, in every clock cycle of program execution.* □

We can use this notion to formulate the requirement for confidentiality as a trace property:

$$\begin{aligned}&\forall P, S_0, X_p, X_c, X_c' : \\ &\qquad O(P, X_p, X_c) = O(P, X_p, X_c') \\ &\Rightarrow \xi(S_0, P, X_p, X_c) = \xi(S_0, P, X_p, X_c')\end{aligned} \tag{4.1}$$

The property is a "non-interference property" which considers an arbitrary program running for different sets of secret data, X_c and X_c'. It states that if the execution of a program as defined by the ISA-level specification does not depend on the secret, then its cycle-accurate execution as defined by the microarchitecture must not depend on the secret either. This way it is ensured that the hardware provides the same level of confidentiality as its specification, i.e., the ISA.

It should be noted that, in this formulation, there is no restriction on what can be considered confidential information. Confidential information can be selected according to the security requirement of the system, for example: a memory page marked unreadable by the page table, a security-critical memory-mapped IO device or a security-critical system register.

A counterexample to this property documents a confidentiality violation: it shows a scenario in which reading or observing a confidential part of memory or architectural registers is possible at the hardware level, while the ISA specification does not allow that. The confidentiality violation can be caused by either *TESs* or *functional leakages*. In case of a TES, only the timing of the program execution at the hardware level depends on the confidential information and the computation results are still independent of the secret. Therefore, the secret is leaked through a timing side channel. In the case of functional leakage, the secret directly leaks to the architectural registers and the confidentiality violation is caused by design bugs that also violate the functional ISA specification of the system. For example, reading a security-critical system register by a user-level program must lead to an exception according to the ISA, but a bug in the hardware implementation may prevent properly raising the exception and, as a result, the user-level program is able to directly read the content of the system register.

UPEC covers both types of confidentiality violations. A functional leakage is a counterexample not only to the property of Eq. 4.1 but also to the following, weaker property:

$$\begin{aligned}&\forall P, S_0, X_p, X_c, X_c' : \\ &\qquad O(P, X_p, X_c) = O(P, X_p, X_c') \\ &\Rightarrow \mu(S_0, P, X_p, X_c) = \mu(S_0, P, X_p, X_c')\end{aligned} \tag{4.2}$$

This means, when targeting functional leakages, we do not consider the detailed timing behavior of the hardware implementation. We only need to check if the actual content of the program-visible registers can be influenced by secret data. Functional leakages, in principle, can be identified by any functional verification technique, such as conventional property checking or functional simulation. This

is due to the fact that most functional verification techniques check the hardware implementation in terms of the produced results or time-abstract I/O behavior. It is worth noting that it is still a challenge to provide guarantees regarding the absence of functional leakages and conventional verification techniques are not always effective for this purpose. This motivates further examining the potential of UPEC in detecting functional confidentiality violations, which is briefly reviewed in Chapter 10.

For the case of TESs, the security verification is exceptionally challenging since none of the conventional functional verification techniques can address the issue. Most of these techniques are geared towards checking the time-abstract results against the specification and thus, they cannot directly cover timing-related issues.

4.3 Security against TES Attacks

The property of Eq. 4.1 provides a formal definition of confidentiality at the hardware level. To foster a formal understanding of TES attacks, we need to further refine this property. A *TES attack* is a counterexample to the following trace property. It results from Eq. (4.1) by excluding functional leakages from consideration:

$$\begin{aligned}
&\forall P, S_0, X_p, X_c, X_c' : \\
&\quad [\ O(P, X_p, X_c) = O(P, X_p, X_c') \wedge \\
&\quad\ \ O(P, X_p, X_c) = \mu(S_0, P, X_p, X_c) \wedge \\
&\quad\ \ O(P, X_p, X_c') = \mu(S_0, P, X_p, X_c')\ ] \\
&\Rightarrow \xi(S_0, P, X_p, X_c) = \xi(S_0, P, X_p, X_c')
\end{aligned} \tag{4.3}$$

It should be noted that this property verifies concrete hardware while considering any possible software for both the attacker process and the underlying operating system. This is different from software verification approaches (e.g., [34, 35, 65, 110]) which always consider a concrete kernel software that is being verified to be secure against TES attacks. The property also differs from formulations such as in [104] that formalize TES security based on abstract execution semantics rather than a concrete microarchitecture.

For the remainder of this thesis, it is helpful to observe that any valid counterexample to the property in Eq. 4.3 fulfills the following condition:

$$\mu(S_0, P, X_p, X_c) = \mu(S_0, P, X_p, X_c') \tag{4.4}$$

This condition is referred to as *microequivalence* in the following chapters.

Spectre vs. Meltdown The property of Eq. 4.3 can also be used to formally categorize TES attacks into two main classes.

Consider a TES attack, given as a counterexample (S_0, P, X_p) to the trace property of Eq. 4.3, executing *a concrete instruction sequence* A on the processor.

The attack is called a Spectre attack if there exists a subsequence of A called A_{victim} such that A_{victim} is executed under an execution context with the proper privilege to access some confidential program input X_c, without an exception being raised. Otherwise, if no such A_{victim} exists, the attack is called a Meltdown attack.

This categorization helps to develop proofs that exclusively target one class, which may help scalability of the proofs as well as assisting the designer for only patching the system against one class while leaving the other one to software-level countermeasures.

4.4 Further Evaluation of the UPEC Property

4.4.1 TES Attack as UPEC Counterexample

The full attack vector in a TES attack usually consists of hundreds or even thousands of instructions. This casts doubt over the feasibility of finding TES attacks through formal analysis of the RTL hardware model as considering traces of hundreds of instructions is mostly infeasible in formal proof methods. However, the UPEC property formulation and the employed proof methodology, allow for modeling most of the trace implicitly in the microarchitectural starting state S_0 which drastically simplifies the proof complexity.

TES attacks are usually carried out in multiple phases (for example see Sec. 2.3.1 and 2.3.2). For example, the attacker usually primes the system into a specific state before attempting to leak the secret and then later measures the timing of certain operations to deduce the secret value from the side channel. This entire process constructs a successful TES attack. The counterexample to the described trace properties is not necessarily the complete attack vector, but rather can be the critical part of the attack exfiltrating the secret and creating the side channel.

In a UPEC counterexample, the initial part of the attack prior to the secret leakage is implicitly represented by the microarchitectural starting state S_0, which makes the counterexample trace significantly shorter compared to the concrete attack vector. For example, the part of the Spectre attack which poisons the branch prediction unit (by bad training) can be excluded from P and can be implicitly represented by S_0.

The final phase of the attack in which the attacker tries to deduce information about the secret through the side channel (e.g., measuring the time of cache accesses) is not represented in the counterexample. Although this phase is critical for the attack since it may define the extent and the bandwidth of possible information leakage, it may not be relevant to the hardware design process. Regardless of the strength of the attack, if the design fails the property of Eq. 4.1, it is vulnerable and the vulnerability must be properly and thoroughly analyzed. Therefore, it is a reasonable simplification to only focus on the part of the attack which exfiltrates the secret.

4.4.2 TES Attacks vs. Classical Side Channel Attacks

The trace properties of Eq. 4.1 and Eq. 4.3 only consider programs with secret-independent architectural observation. This requirement means that the considered program must not read the secret and must not perform any computation using the secret as an input. This, clearly, excludes all programs that perform secure computations using secret information, e.g., encryption software.

Constraining the property with this requirement as an assumption allows us to exclusively target TES attacks and hardware vulnerabilities and automatically excludes all security weaknesses that may arise due to vulnerable programs that interact with confidential assets. For a verification methodology that is developed for assisting hardware design flows, it is important to make a distinction between hardware security issues and vulnerabilities that should be addressed at the software level.

It should be noted that this assumption does not restrict the generality of the property and the security guarantee that it provides, since the assumption is not assumed as a global constraint for the system and only applies to the time window of the considered trace. Within the time points prior to or after the trace, there is no constraint on what program can be executed. In other words, the property does not forbid secure computation based on secret values. However, vulnerabilities due to specific security-critical software components are out of the scope of the property. Remember that it belongs to the specific characteristics of TES attacks that they are independent of any specific security-critical software component, as explained in Sec. 2.3. Therefore, assuming the equivalence of architectural observations in our property does not exclude any TES channels.

Using the same S_0 in both traces excludes the footprint left by any previous secret-dependent program runs. As a result, we exclude classical timing side channels from consideration, i.e., side channels which are not based on transient executions. Those rely on exploiting a specific victim software, such as encryption software, whose footprint would need to be represented by a difference between the starting states of the two traces. This is a valid distinction because classical side channels are fundamentally different from TES attacks since they are observable at the ISA level, e.g. through secret-dependent memory access patterns, while TES are not.

4.4.3 UPEC Security Guarantee at the Software Level

The ISA is the standard hardware/software contract for software development. However, the ISA does not specify the timing of instruction execution and, thus, leaves a gap regarding microarchitectural timing side channels. As a result, it is important to establish a contract with software developers on what will be leaked through timing side channels for a given microarchitectural implementation [68]. Such contracts contribute to software security by defining which elements of a software process (e.g., program counter, accessed memory locations, etc.) may be *exposed*, i.e., may be leaked through microarchitectural timing side channels. This knowledge enables effective software verification, without leaving a gap due

to microarchitectural side channels.

Previous work [68] defines a framework for hardware/software contracts w.r.t. timing side channels. A contract in this framework is defined based on *execution semantics* and an *adversarial observation*:

- **Execution semantics** defines whether the software execution is based on *sequential* semantics (SEQ) or *speculative* semantics (SPEC).
- **Adversarial observation** defines what an attacker can observe through timing side channels from a victim software process running, based on the defined execution semantics. The adversarial observation is either *constant-time* (CT), i.e., the attacker can only observe *addresses* of memory accesses, or *architectural* (ARCH), i.e., the attacker can observe both *address* and *data* for every memory access.

Using these paradigms, one can define the following contracts:

1. **SEQ-CT**: The microarchitecture exposes the address of every memory access in the sequential (non-speculative) execution of a program.
2. **SEQ-ARCH**: In addition to what is exposed based on the SEQ-CT contract, in this contract the microarchitecture exposes the data that is read from the memory in the sequential (non-speculative) execution of a program.
3. **SPEC-CT**: The microarchitecture exposes the address of every memory access in the speculative execution of a program, i.e., the address of any memory access that is executed due to a mispredicted branch is also exposed.
4. **SPEC-ARCH**: In addition to what is exposed based on the SPEC-CT contract, in this contract the microarchitecture exposes the data that is read from the memory in the speculative execution of a program, i.e., the data that is read from the memory due to a mispredicted branch is also exposed.

It should be noted that SEQ-CT and SEQ-ARCH contracts allows the microarchitecture to support speculative execution, as long as the speculatively executed instructions do not expose any information.

Although the SEQ-ARCH contract does not seem to provide a strong security guarantee since it effectively exposes all the information in a sequential program run, it does guarantee the confidentiality of the data that is accessed *transiently*. This relieves the software verification from considering any non-sequential semantics, which contributes to more scalable and effective software security verification.

The hardware/software contracts framework in [68] provides extensive reasoning on the implication of each contract for software security verification. However, verifying the hardware side of the guarantee is not addressed in [68]. It

is, therefore, worthwhile to evaluate which contract in this framework relates to the UPEC hardware security property (Eq. 4.3). This provides the opportunity for UPEC to utilize this framework to make a well-defined connection between the hardware-level guarantee and the software-level security requirements.

According to the property of Eq. 4.3, any instruction that does not affect the architectural or microarchitectural observation of the program must not have any effect on the microarchitectural execution (i.e., the timing of the program). This includes all instructions that are executed transiently but discarded due to misprediction or exception. Therefore, the property of Eq. 4.3 requires that speculative or out-of-order execution of instructions must not leak any information. In other words, the program execution based on SPEC execution semantics must not leak any information. However, this property does not impose any restriction on instructions that are executed on a non-speculative path (or a correctly predicted path). As a result, a microarchitecture that is verified by UPEC does not leak any information by the instructions that are only executed speculatively and later discarded. However, the microarchitecture is free to expose both address and data of memory accesses that are not transient since the property provides no restriction regarding that. Therefore, UPEC guarantees that a given microarchitecture conforms to the SEQ-ARCH contract. This contract relieves the software verification from complex modeling of speculative or out-of-order execution at the ISA level.

It should be noted that a microarchitecture fulfilling the property of Eq. 4.3 can also conform to the stronger SEQ-CT contract, if additional properties regarding the timing of non-transient instruction execution are verified.

Chapter 5

UPEC on a Bounded Model

Proving the property of Eq. 4.1 by generic unbounded model checking is infeasible for SoCs of realistic size. Therefore, in this chapter, we develop a SAT-based approach which is tailored to the properties considered in UPEC. Our approach is based on "*any-state proofs*" in a bounded circuit model. It is related to Interval Property Checking (IPC) [44, 111] (cf. Sec. 2.4.2) which is applied to the UPEC problem in a similar fashion as in [1] for functional processor verification. This proof methodology is used at the heart of the UPEC verification methodology, described in Chapter 6.

The property of Eq. 4.1 relates to two separate execution traces for a given program that start from the same microarchitectural state. Consequently, in our IPC-based approach, we use a tailor-made computational model, shown in Fig. 5.1. The model consists of two identical instances of the SoC under verification, SoC_1 and SoC_2, that each produce an execution trace. In the terminology of software verification, UPEC pursues an approach based on *2-safety hyperproperties* [112]. A 2-safety formulation provides similar benefits for UPEC as for information flow and taint property verification [93, 113]. In hardware equivalence checking, the underlying self-compositional model is widely referred to as "miter" [114, 115]. Owing to the resemblance of UPEC with hardware equivalence checking we refer to the UPEC computational model as "miter for side channel detection".

In the computational model of Fig. 5.1 as well as in the rest of this thesis we assume, without loss of generality, that X_c resides in the data memory, unless otherwise noted. Verifying the SoC for confidentiality of information in the register file is analogous. It should be noted that protecting against TES attacks that are targeting secret in the memory is more critical than the case of data in general purpose registers. The former case can lead to the creation of *universal read gadgets*, while in the latter the attacker can only leak the data that the victim has recently accessed.

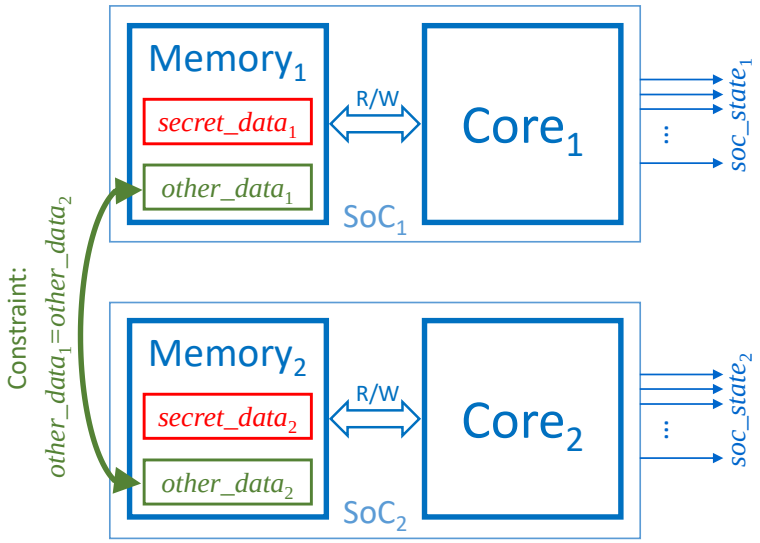


Figure 5.1: Computational model for UPEC: The content of the memory is the same, except for the secret data (X_c).

5.1 UPEC Interval Property

Fig. 5.2 shows the UPEC property formulated as an interval property [44]. It is proven on the unrolled miter model depicted in Fig. 5.3. In this property, *micro_soc_state* is the vector of all state variables inside the SoC, excluding those parts of memory that hold the public information X_P and the confidential information X_c. The state variables of memory and cache holding X_P are denoted with *public_mem* in the UPEC property.

Similar to Eq. 4.1, the first two assumptions of the property guarantee that both SoC instances execute the same, arbitrary, program and that the only difference between the two execution traces at the start state is the set of secret data. Since microarchitectural behavior at the RTL is deterministic, including that of cache eviction and branch prediction, any discrepancy between the state of SoC_1 and SoC_2 that may occur in later clock cycles is due to the propagation of the secret. Furthermore, by having arbitrary values in microarchitectural state variables while fulfilliing the equality of *micro_soc_state* between SoC_1 and SoC_2, we can exhaustively model any possible history (program context) in the system.

For a productive verification methodology, we need to avoid formulating the property with reference to the architectural observation of a program, since this would require formalizing the ISA. Instead, we formulate two constraints *secret_data_protected* and *secret_load_speculative*. They are expressed in terms of the considered execution traces, and serve the purpose of taking into account only traces with the same architectural observation, without actually formalizing

assume:
  at t: $micro\_soc\_state_1 = micro\_soc\_state_2$;
  at t: $public\_mem_1 = public\_mem_2$;
  at t: $secret\_data\_protected()$;
  during $t..t+k$: $secret\_load\_speculative()$;
prove:
  at $t+k$: $soc\_state_1 = soc\_state_2$;

Figure 5.2: UPEC property formulated as interval property: it can be proven based on a bounded model of length k using Satisfiability (SAT) solving and related techniques.

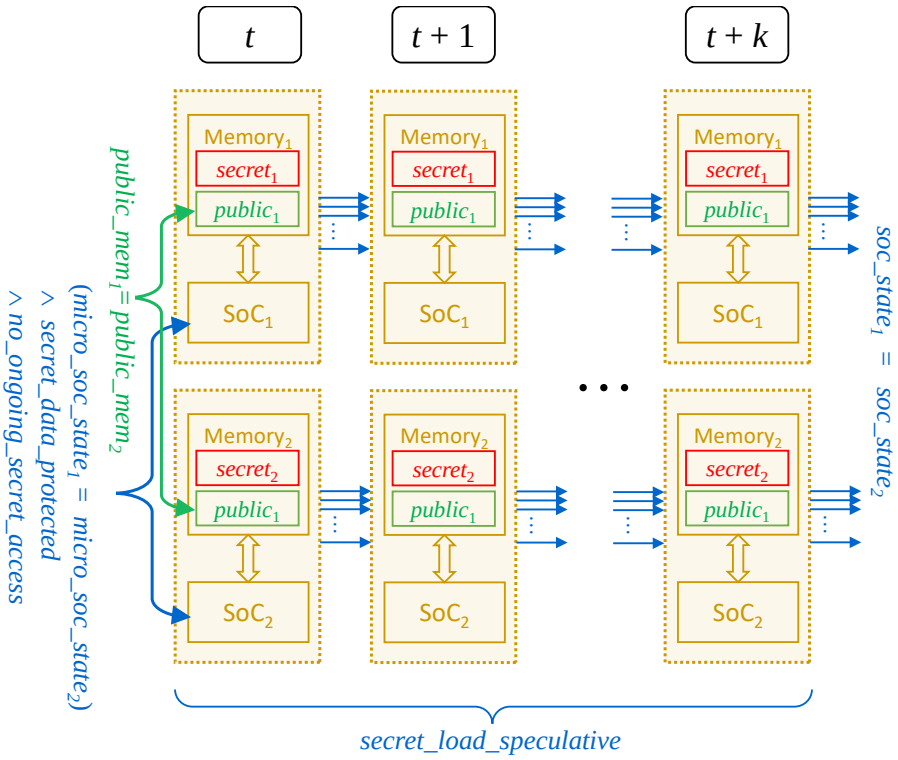


Figure 5.3: Unrolled miter for side channel detection: two almost identical instances of the computing system are considered. They contain the same data, except for the secret data that may differ between the two instances. The memory block in this figure stands for any resource that may carry secret data, including peripheral devices.

the ISA.

***secret_data_protected*:** The macro *secret_data_protected* states that a hardware protection mechanism is enabled to protect the memory locations holding X_c. This means that, provided that the protection mechanism is implemented correctly, any access to the secret within a user-level process is blocked by an exception. This allows for considering Meltdown-style attacks (according to Sec. 4.3).

This macro can be refined for a given RTL model, by constraining the registers holding the configuration of the memory protection unit, so that a specific address, which holds X_c, is marked unreadable. The memory protection mechanism can be an ISA-specific access control, e.g., *physical memory protection* of RISC-V ISA [49], or the virtual memory translation. Examples of how this can be done can be found in [116].

***secret_load_speculative*:** The macro *secret_load_speculative* assumes that any access to X_c within an OS-level process happens transiently and is discarded before it commits. The instruction can be discarded due to an earlier exception or branch misprediction. This allows for considering Spectre-style attacks (according to Sec. 4.3).

This macro can be refined for a given RTL model, based on the pipeline signals controlling the speculation level. For X_c residing in memory, any load instruction targeting X_c must be a speculative instruction. In in-order pipelines, spotting the signals determining the speculation level of each instruction is usually straightforward. However, for out-of-order cores it can be more challenging. We describe how to specify this macro as well as other aspects of applying UPEC to out-of-order cores in section 7.1 and a template is provided in [116].

With the help of *secret_data_protected* and *secret_load_speculative* macros, only execution traces are considered in which any access to X_c is either blocked by an exception or discarded due to an earlier exception or misprediction. Since all other sources of instruction operands are the same between the two instances, the two execution traces considered by the property always yield the same architectural observation.

Although the macros guarantee a secret-independent architectural observation, it is important to understand that the described macros do not over-constrain the interval property compared to the property of Eq. 4.1. In any execution trace that violates *secret_data_protected* or *secret_load_speculative*, the system executes a load instruction targeting the secret that becomes visible as part of the architectural observation, since the instruction is neither misspeculated nor blocked by an exception. Consequently, the architectural observation is secret-dependent which is a violation of the assumption in Eq. 4.1. It should also be noted that *secret_data_protected* and *secret_load_speculative* macros do not provide a global constraint on the system, but rather constrain program execution within the property time window between t and $t + k$. The system is still free to execute any program, even the ones violating the macros, in the

clock cycles before t. This is taken into account by considering any microarchitectural state as the starting state of the program execution for the trace property of Eq. 4.1 and the interval property of Fig. 5.2. Furthermore, we note that *secret_data_protected* and *secret_load_speculative* macros do not prevent detecting any functional bug in the design which causes confidentiality violation by creating secret-dependent microarchitectural observation. These macros only constrain the architectural observation, however, the microarchitectural observation still depends on the hardware implementation.

In the prove part of the property, *soc_state* is a vector of state variables which includes all architectural (program-visible) state variables, as required by Eq. 4.1, but additionally it can also include some or all other microarchitectural state variables. Observing also non-architectural registers is a key concept in UPEC which serves to obtaining an unbounded and scalable proof method. This will be further developed in the following subsections.

5.2 Spurious Counterexamples in UPEC Proof

Generally, in the IPC approach, the symbolic starting state at the beginning of the interval also includes unreachable states of the system. This may lead to false counterexamples which cannot happen in the normal operation of the system after reset (*spurious counterexamples*). This is addressed by strengthening the property using invariants, which excludes certain problematic unreachable states from the consideration.

For the case of UPEC, program execution is unique also in most unreachable states, and this leads to a low number of spurious counterexamples, and consequently, low number of required invariants. Moreover, since security verification is usually done in addition to a functional verification flow, the invariants developed for functional verification can be reused in the UPEC flow too.

Although developing invariants is usually a design-dependent and ad-hoc process, the UPEC computational model and property formulation allow us to address the problem of spurious counterexamples in a more structured way. We distinguish between two types of invariants within the UPEC methodology:

- **Design Invariants** relate to *the actual design* under verification and hold for the design regardless of the running software (they can be shared with functional verification).
- **Miter Invariants** relate to *the miter* rather than the design and are based on security properties of the design which hold assuming the running software is secure w.r.t. protecting the secret. In other words, miter invariants excludes certain miter states that are unreachable, if a realistic operating system is running on each SoC.

Developing design invariants require careful analysis of each spurious counterexample and they are design-dependent. However, the miter invariants, which

```
assume:
    at t:          reset;
    at t:          secret_data_protected();
prove:
    at t+1:        no_ongoing_protected_access();
```

Figure 5.4: Induction base property to prove *no_ongoing_protected_access*() as a miter invariant

can be developed in a more structured way, prevent many false counterexamples and thus save the required effort for developing design invariants.

The UPEC property requires two main miter invariants:

Miter Invariant 1, "no ongoing protected access". Protected memory regions are inaccessible to certain user processes but the operating system or kernel process can freely access these regions. This can create a scenario for confidentiality violation in which the kernel process issues a load instruction to read secret data from a protected region and then instantly branches into a malicious user level process. The user level process will start while there is a pending memory transaction accessing the secret, which is issued by the operating system and will not be discarded by the memory protection unit. Such an explicit revelation of secret data can be considered as a spurious counterexample and excluded from consideration since it cannot happen in a SoC running an operating system within its normal operation. To exclude these trivial leakage scenarios, the proof must be constrained to exclude such initial states that implicitly represent ongoing memory transactions in which protected memory regions are accessed. This invariant can be formally expressed by assuming that, at the starting time point, the buffers holding the addresses for ongoing valid/authorized memory transactions do not contain protected addresses. An authorized transaction is any memory transaction that has successfully passed the access permission checks done by the memory protection unit. These buffers can be easily identified by inspecting the fan-in of the memory interface address port. This invariant must be proven inductively on the miter, by constraining the proof by the *secret_data_protected* macro, as shown by properties of Fig. 5.4 and 5.5. This inductive proof ensures that no bug in the memory management unit can lead to violation of this invariant. The *secret_data_protected* macro in functionally correct RTL implementation must prevent any secret access from being authorized by the memory protection unit and thus, none of the buffers holding ongoing authorized memory transactions can possibly hold a secret access transaction.

Miter Invariant 2, "secure system software". Memory protection mechanisms are meant to protect the content of the memory from user level software, however, kernel level software can freely configure these mechanisms. Since it

```
assume:
    at t:          no_ongoing_protected_access();
    at t:          secret_data_protected();
prove:
    at t + 1:      no_ongoing_protected_access();
```

Figure 5.5: Induction step property to prove *no_ongoing_protected_access*() as a miter invariant

is not possible to have the complete OS/kernel in our bounded model (due to complexity), and the kernel software is modeled as symbolic input to the system, this may lead to trivial false counterexamples in which the kernel software turns the memory protection off (e.g., by disabling address translation) and makes the secret accessible to any user process. In order to exclude such trivial cases, a miter invariant is needed for restricting the proof to systems with secure system software. It should be noted this miter invariant must be proven at the software level based on the employed operating system.

Miter invariants do not restrict the generality of our proof. They are, actually, invariants of the global system. Their validity follows from the functional correctness of the OS and the SoC.

5.3 Security Alerts in the UPEC Proof

For a full proof of the UPEC property on the bounded computational model of Fig. 5.3, in principle, we need to consider a time window as large as the sequential depth, d_{soc}, of the examined SoC. This is infeasible in most practical cases. Fortunately, employing a symbolic initial state allows the solver to often capture hard-to-detect vulnerabilities within much smaller time windows.

Any information leakage starts with the propagation of confidential information into some microarchitectural register. As a result, if we include every microarchitectural state variable in *soc_state*, the any-state proof on our bounded model guarantees detection of any violation of the UPEC property in only a single clock cycle. This is due to the fact that the symbolic initial state of the proof includes, as a subset, all possible reachable states.

However, unique program execution only requires that the secret data must not be observable through the architectural state variables. Therefore, proving that the secret data cannot propagate to any state variable in a superset of the architectural state variables represents a sufficient but not a necessary condition for certifying confidentiality.

We therefore distinguish two kinds of "alerts" to the UPEC property. In the following discussion, an alert a_k is a counterexample of length k to the UPEC property where $soc\_state_1 \neq soc\_state_2$ in the ending state, i.e., at time k.

Definition 5 (L-alert)**.**
An alert, a_k, is called a leakage alert (L-alert) iff the differing state variables of

soc_state_1 and soc_state_2 include architectural state variables. □

L-alerts demonstrate how secret data can affect the sequence of architectural states. Two cases can be distinguished. In the first case, secret data propagates directly into an architectural register, i.e., the microarchitectural observation is changed. If this happens the design contains a functional bug. In the second case, the problem is more subtle. The secret changes the timing and/or the values of the sequence without violating the functional design correctness and without leaking the secret directly. In other words, it changes the microarchitectural execution without affecting the microarchitectural observation. UPEC detects the hardware vulnerability in both cases.

Definition 6 (P-alert)**.**
An alert, a_k, is called a propagation alert (P-alert) iff the differing state variables of soc_state_1 and soc_state_2 are not architectural state variables. □

P-alerts show possible propagation paths of secret data from the cache or memory to program-invisible, internal state variables of the system, such as pipeline buffers. Hence, no secret data is *leaked*. However, a P-alert can be a precursor to an L-alert, because the secret often traverses internal, program-invisible buffers before it is propagated to an architectural state variable like a register in the register file. In fact, any L-alert that is not (yet) visible within the time window of the unrolled model has one or more shorter precursor P-alerts that are. Based on this observation, we realize that proving the absence of P-alerts and L-alerts within one clock cycle in an any-state proof is sufficient for verifying confidentiality. In the following chapter, we present the UPEC methodology for verifying systems that do not fulfill this sufficient condition for confidentiality.

Chapter 6

UPEC Verification Methodology

In this chapter, we describe the UPEC security verification methodology. For a given RTL design, the verification methodology either certifies the absence of microarchitectural vulnerabilities causing confidentiality violation, or produces an execution trace pointing to a detected vulnerability. It employs the UPEC interval property, describe in Sec. 5.1, at the heart of its proof procedure.

We first elaborate on optimizations for the UPEC property that contribute to the scalability of the technique. Afterward, the iterative proof procedure based on inductive reasoning is described.

6.1 Modeling the Propagation of Secrets

In the UPEC verification methodology, P-alerts provide valuable information on the "locations" to which secret information can propagate.

Definition 7 (P-location)**.** *For a P-alert a_k the P-location, Ploc(a_k), is the set of all microarchitectural state variables of the SoC whose values in the ending state of a_k (at time point k) depend on the secret, i.e., whose valuations differ between the SoC instances 1 and 2.* □

In the following, we use P-locations to distinguish different scenarios of secret information flows. In our proof method, the P-locations are represented by Boolean state predicates.

Definition 8 (P-location Predicate)**.** *A P-location predicate, Λ_P, for a set of microarchitectural state variables, P, is a state predicate over all microarchitectural variables of the miter, given as a conjunction of equalities and inequalities of corresponding variables in the SoC instances 1 and 2:*

$$\Lambda_P = \prod_i (v_{i,1} \neq v_{i,2}) \cdot \prod_j (w_{j,1} = w_{j,2})$$

specifies

- *inequality for every variable* $v_i \in P$, *and*
- *equality for every other variable* $w_j \notin P$. □

A P-alert a_k satisfies its P-location predicate Λ_P with $P = Ploc\,(a_k)$ at time point k, i.e., in its ending state.

Using these notions, we can collect and enumerate all possible locations in the SoC to which secret information can propagate in k clock cycles. Similar to an all-SAT enumeration, after finding a P-alert a_k, a blocking clause is added to the property preventing the solver from generating another counterexample fulfilling the same Λ_P, i.e., reaching the same P-location.

In the following, we will use the complete set of P-locations to decompose the UPEC proof problem into scalable sub-problems. Before this idea is elaborated, we introduce a cone-of-influence reduction for UPEC, as an additional optimization for the proof procedures to be presented.

6.2 Cone-of-Influence Reduction

Cone-of-influence reduction is a well-known technique for reducing the complexity of proof tasks in formal verification by structurally removing parts from the computational model that are irrelevant for the proof at hand [117]. This process usually starts from the signals mentioned in the prove part of the property and removes parts of the model that cannot functionally affect the values in these signals within the time window of the property.

In the original UPEC interval property of Fig. 5.2, however, the prove part includes every state variable in the design; as a result, the unrolled model cannot be simplified by the solver through cone-of-influence reduction.

Fortunately, this changes when we decompose the problem, as presented in the following subsection. In the decomposed UPEC proof on a bounded model we are interested in finding P-alerts and L-alerts of a specific length k while the precursor P-alerts and their P-locations $Ploc\,(a_{k-1})$ are known. Therefore, only the subset of next-state variables that lie in the fanout of the given locations need to be considered as the possible candidates for a new alert.

Definition 9 (Alert Candidate)**.** ***Alert_candidate***$_P$ *is the set of state variables of the SoC which are in the immediate fanout cone of* $P = Ploc\,(a_{k-1})$.

We can use the set Alert_candidate$_P$ to simplify the proof obligation for the property of Fig. 5.2. Assuming a_{k-1} is a P-alert of length $k-1$, for the UPEC proof of length k, we can replace *soc_state* in the prove part of the property with Alert_candidate$_P$. This helps the solver in many cases to conduct an effective cone-of-influence reduction.

UPEC_Base (P, k, blocking_clause):
  assume:
    at t: $micro\_soc\_state_1 = micro\_soc\_state_2$;
    at t: $public\_mem_1 = public\_mem_2$;
    at t: $secret\_data\_protected()$;
    during $t..t+k$: $secret\_load\_speculative()$;
    at $t+k-1$: Λ_P = *true*;
    at $t+k$: blocking_clause = *true*;
  prove:
    at $t+k$: $Alert\_candidate_{P1} = Alert\_candidate_{P2}$;

Figure 6.1: UPEC property for induction base procedure

6.3 Iterative UPEC Proof Procedure

The any-state proof of UPEC guarantees to find all P-alerts of a given temporal length, if the model is unrolled for the corresponding number of clock cycles. As already observed in Sec. 5.3, any P-alert a_k with $k > 1$ has a pre-cursor P-alert a'_{k-1} which explains how the secret propagates to SoC variables in the immediate fanin variables of *Ploc* (a_k). This information can be used to guide the solver towards the next counterexamples and mitigate the computational complexity of the proof by decomposing the UPEC proof into simpler steps.

In our general procedure, we enumerate P-alerts by starting with the ones for $k = 1$, and then incrementing k until no new alerts are found. We decompose each UPEC proof for a particular k along the different precursors for P-alerts, by enumerating all P-locations reached within $k-1$ clock cycles. The proposed decomposition does not violate the completeness of the proof, since the any-state proof on the $k-1$ clock cycle time window guarantees finding all possible P-alerts of that length and, hence, every counterexample to the UPEC property is covered by a P-location computed by the proof procedure.

Fig. 6.1 shows the UPEC property used in our iterative procedure. The property receives a P-location P, as reached by a precursor P-alert, and verifies whether the secret can propagate further and produce a new alert. The property also receives a blocking clause which prevents the solver from generating the same counterexamples again when calling the proof procedure repeatedly.

The UPEC verification methodology is based on proof by induction, which consists of an induction base and an induction step. Alg. 1 shows the base proof. It iteratively verifies the design against the property of Fig. 6.1, collecting all possible P-locations, and searches for any L-alert. The "IPC_Solver" is an interval property checker which returns a counterexample if the given property fails or, otherwise, returns $\emptyset$.

The first step (lines 2 to 11) is to compute the initial set of P-locations, A_1, which can be reached in one clock cycle. "UPEC_Base($\emptyset$, 1, bc)" is a special case of the UPEC property where $k = 1$ and there is no predecessor P-alert. Therefore, Alert_candidate needs to be determined based on the initial location

Algorithm 1 UPEC_Induction_Base

```
1:  procedure UPEC_INDUCTION_BASE
2:      k ← 1
3:      A_k ← ∅
4:      bc ← true
5:      a_k ← IPC_Solver(UPEC_Base(∅, 1, bc))
6:      while a_k ≠ ∅ do
7:          if a_k is L-alert then return a_k
8:          P = Ploc(a_k)
9:          A_k ← A_k ∪ {P}
10:         bc ← bc ∧¬ Λ_P
11:         a_k ← IPC_Solver(UPEC_Base(∅, 1, bc))
12:     while A_k ≠ ∅ do
13:         k ← k + 1
14:         A_k ← ∅
15:         bc ← true
16:         for P' in A_{k−1} do
17:             a_k ← IPC_Solver(UPEC_Base(P', k, bc)
18:             while a_k ≠ ∅ do
19:                 if a_k is L-alert then return a_k
20:                 P = Ploc(a_k)
21:                 if P ∉ ⋃_{i=1}^{k} A_i then
22:                     A_k ← A_k ∪ {P}
23:                 bc ← bc ∧¬ Λ_P
24:                 a_k ← IPC_Solver(UPEC_Base(P', k, bc))
25:     return ⋃_{i=1}^{k} A_i
```

of the secret (e.g., data memory/data cache). The design is repeatedly verified using the property. Every time, after finding a counterexample, the blocking clause is updated to search for new P-alerts. The loop is terminated as soon as the solver returns $\emptyset$, which means there is no new P-location that is reachable within one clock cycle.

Once A_1 is computed, the procedure continues to find more alerts with longer time windows, until it reaches a time window in which it determines no new P-alert or L-alert (lines 12 to 24). The *for* loop (line 16) iterates through all P-locations determined in the previous iteration for $k-1$, and, for each one of them, computes all successor alerts of length k. In line 21, for the newly found alert a_k, we check whether the same P-location has been detected by a shorter P-alert. This avoids generating the same P-location twice and makes sure that circular propagation of the secret between a set of state variables will not cause the algorithm to become stuck in an infinite loop.

Every time "IPC_Solver" returns a counterexample, it is also checked if it is an L-alert. If this happens, a true security violation has been detected. The L-alert is returned immediately to the designer for debugging and repair. If no L-alert is detected, the algorithm returns all the P-locations found in the iterations.

Once the design successfully passes the UPEC induction base proof, it is guaranteed that, starting from *any possible initial state*, there is no information leakage possible within k clock cycles. The any-state proof within *UPEC_Induction_Base* implicitly considers any possible pipeline context and system state that is required for initiating secret propagation. As a result, for any information leakage possible within a time window larger than k clock cycles, *UPEC_Induction_Base* is guaranteed to find at least one precursor P-alert. This ensures exhaustive coverage of yet undiscovered P-alerts and L-alerts by the following algorithm for the induction step.

The goal of the induction step is to prove that, if the secret propagates to some non-observable microarchitectural state variables, it will not leak to architectural state variables at any time in the future. In other words, we need to prove that a P-alert cannot be extended further to reach a new P-location or to an L-alert. Therefore, in contrast to the induction base, the induction step property does not assume equality of all microarchitectural state variables at time t but instead assumes that the secret has already propagated to some of these variables, represented by a P-location P'.

Fig. 6.2 shows the property which is the heart of *UPEC_Induction_Step* (Alg. 2). In this property, the starting state (at time point t) is the ending state of some P-alert found within the base proof. The property proves that any P-alert (of any length) that reaches P and that does not reach a new P-location in the subsequent clock cycle, will also (inductively) not reach a new P-location thereafter. Note that a counterexample to the step property is always of length 2.

The procedure over-approximates the history prior to the ending state of a considered P-alert and, thus, includes all reachable behaviors of the design, including possible L-alerts. The over-approximation may lead to unreachable

UPEC_Step (P, blocking_clause):

assume:	
at t:	$public\_mem_1 = public\_mem_2$;
at t:	$secret\_data\_protected()$;
during $t..t+2$:	$secret\_load\_speculative()$;
at t:	Λ_P = *true*
at $t+1$:	$Alert\_candidate_{P1} = Alert\_candidate_{P2}$;
at $t+2$:	blocking_clause = *true*;
prove:	
at $t+2$:	$Alert\_candidate_{P1} = Alert\_candidate_{P2}$;

Figure 6.2: UPEC property for induction step procedure

Algorithm 2 UPEC_Induction_Step

```
1:  procedure UPEC_INDUCTION_STEP(A)
2:      for P' in A do
3:          bc← true
4:          a2 ← IPC_Solver(UPEC_Step(P', bc))
5:          while a2 ≠ ∅ do
6:              if a2 is L-alert then return a2
7:              P = Ploc(a2)
8:              if P ∉ A then
9:                  A ← A ∪ {P}
10:             bc ← bc ∧¬ Λ_P
11:             a2 ← IPC_Solver(UPEC_Step(P', bc))
12:     return ∅
```

counterexamples. This is a standard problem and can be addressed by extending the proof to a k-step induction [118], or by strengthening the initial state of the proof by invariants. In our experiments, only for one of the designs (RocketChip), due to several uninitialized state variables, a few simple invariants had to be added manually.

Alg. 2 describes the procedure for the UPEC induction step. It receives the set of P-locations determined by the induction base computation of Alg. 1. For each P-location in this set, it computes all subsequent P-alerts using the "UPEC_Step" property. If the property holds, it is guaranteed that there is no subsequent P-alert. If the property fails, it is checked whether the counterexample is an L-alert. If it is not an L-alert and also yields a new P-location, the new P-location is added to the set (lines 8 and 9) and used in another "UPEC_Step" proof.

The latter situation reflects that the induction base computation of Alg. 1 terminates as soon as incrementing k does not yield any new P-location. The bounded model of *UPEC_Induction_Base* misses a P-location in case it is reached only at a later time point, after having been propagated through already known P-locations. However, such a P-location is detected afterwards, in *UPEC_Induction_Step*, by iteratively computing all successor P-locations of already known ones. Alg. 2, therefore, always completes the set of P-locations. If Alg. 2 terminates without an L-alert, it means there cannot be any information leakage, and the procedure returns $\emptyset$, certifying the security of the design.

In most practical cases, already Alg. 1 determines all P-locations that exist. This implies that *UPEC_Induction_Step* mostly serves as a check for proof completeness and usually does not incur additional debugging effort. The possible P-locations typically comprise only a small fraction of the microarchitectural state variables, entailing only a small number of iterations in Alg. 1 and Alg. 2. Although the above procedures are fully automated, there can be a benefit in manually inspecting the counterexamples for every generated P-alert that points to a new P-location. This helps the user to understand the security implications of microarchitectural optimizations, detect undesired information flows and, even more importantly, capture security violations early if the security compromise is already obvious from the given P-alert.

6.4 Blackboxing in the UPEC Flow

Blackboxing, i.e., abstracting away the internal logic of a sub-component while its inputs and outputs are modeled as primary inputs and outputs, is a common technique for mitigating proof complexity in BMC and IPC. It usually leads to a smaller number of state variables and less complex logic circuit in the unrolled model, which leads to a smaller SAT problem and shorter proof time. It comes, however, with the problem that the solver generally over-approximates the sub-component's I/O behavior, as its outputs are modeled as free variables, rather than as a function of the inputs. This over-approximation can lead to generating invalid/spurious counterexamples. Verification engineers must manually develop

constraints to restrict the output behavior to valid behaviors, to avoid the over-approximation issue. This process is usually ad-hoc, time-consuming, and prone to errors. The verification engineer may under-approximate the sub-component's behavior and exclude certain legal outputs.

In UPEC, the computational model facilitates addressing this issue and provides a good opportunity for automatic blackboxing, without the risks associated of under- or over-approximating the design behavior.

In UPEC, a component can be blackboxed soundly using only a simple constraint. It states that the component outputs are equal between SoC instance 1 and 2 in every clock cycle of the proof as long as the component's inputs are equal between SoC instance 1 and 2 in every clock cycle of the proof. If a P-alert occurs during the verification procedure of Sec 6.3 which violates this condition, i.e., if the secret propagates to the inputs of the blackboxed component, then this component must be unblackboxed. Otherwise, it is proven that the blackbox is sound and no leakage can be missed.

This blackboxing technique can, for example, be applied effectively to system components containing large memory arrays, like the data array in a cache. This is particularly beneficial for complex out-of-order processors since their microarchitectures contain many data structures with large numbers of memory elements for all kinds of book-keeping purposes. It should be noted that in the design examples we have considered so far, such components have a state-dependent timing behavior. Consequently, secret propagation to their internal state can alter the timing of instruction execution. Therefore, a P-alert which shows propagation of the secret to the inputs of such a component can be seen as an early security alarm.

Blackboxing does not always improve proof complexity: components with a large number of outputs in relation to the number of internal state bits are not suitable for blackboxing. The reason is mostly the fact that by blackboxing such a component, the number of free variables in the SAT problem increases (outputs become free variables), which can, in fact, rather increase proof complexity than reduce it.

Chapter 7

UPEC for Advanced Microarchitectural Features

The UPEC verification methodology (Chapter 5 and Chapter 6) provides a general framework for verifying confidentiality for SoCs. In this chapter, we further extend the UPEC methodology to accommodate for advanced microarchitectural features and the associated threat models. In Sec. 7.1, a new invariant and its refinement is described to deal with false counterexamples when applying UPEC to complex out-of-order processors. We further elaborate on how to refine the definition of L-alerts in processors with dynamic register mapping in Sec. 7.2. A UPEC extension for additional threat models relevant for processors with virtual memory and multicore systems is discussed in Sec. 7.3 and Sec. 7.4, respectively.

7.1 UPEC for Out-of-Order Processors

Our experiments (cf. 9.1) have shown that for processors of medium complexity with *in-order* pipelining, the miter construction of Fig. 5.3 entails that only little effort is required for manually creating invariants to eliminate unreachable counterexamples [2]. However, this changes when out-of-order processors are considered and a systematic solution is required to address the problem of spurious counterexamples in a feasible way.

The symbolic initial state in the UPEC proof can include unreachable starting states. Such a starting state may, for example, represent a pipeline state with inconsistent instruction tags or IDs in different bookkeeping structures, so that invalid execution orders are considered by the proof engine. As a result, secret-dependent microarchitectural observations can be generated which are spurious counterexamples to the UPEC property.

For a simple spurious counterexample, consider the code snippet in Fig. 7.1. A functionally correct out-of-order processor implementation resolves the *write-after-read* hazard between the ADD and LOAD instructions automatically by

```
1   ADD X1, X2, X3 // X1 = X2 + X3
2   BLT X4, X5, branch_taken // branch if X4 < X5
3   LOAD X2, X4 // X2 = mem[X4], X4 is address of the secret
4   ...
5   branch_taken:
6   ...
```

Figure 7.1: Code snippet representing a typical scenario for a UPEC false counterexample

renaming the source and destination registers. In the UPEC proof, however, the solver may consider an unreachable pipeline state in which the physical registers for the destination register of the LOAD and the operand register of the ADD instruction are the same and the LOAD is scheduled for execution before the ADD. With such an inconsistent execution order, the ADD instruction commits a secret-dependent (wrong) result and thus the UPEC property fails because of a *secret-dependent microarchitectural observation*. Such a scenario is always an unreachable (spurious) counterexample unless there exist functional bugs (ISA incompliances) in the implementation.

The key idea to tackle this problem is to constrain the UPEC proof by *microequivalence* (Eq. 4.4). This means that, instead of verifying the trace property of Eq. 4.1, we propose to target Eq. 4.3 when dealing with out-of-order processors. Therefore, the solver only considers microarchitectural observations that are independent of the secret. Note that this does not restrict the generality of the UPEC proof w.r.t. TES attacks, and only excludes functional bugs (i.e., ISA incompliances) from consideration. This simplifies the proof procedure significantly, as it reduces the search space for the SAT solver. Excluding functional bugs from consideration is legitimate since functional verification, i.e., checking compliance of the microarchitectural implementation with the functional ISA specification, is a separate and mandatory step in standard design flows, regardless of the security requirements.

In the following, we provide an exemplary description of how to specify microequivalence.

The symbolic initial state models the reservation stations of the pipeline with arbitrary instructions having arbitrary operands, IDs and other bookkeeping information. The information about how this is linked by the renaming mechanism to the original instructions, as they were processed by the previous decode stage, does not exist in the symbolic initial state. This is how the solver can create the above-mentioned false counterexample: an older, non-speculative ADD instruction in a reservation station may receive an operand from the destination register of a younger speculative LOAD instruction, although the decode stage forbids such a false data dependency. Obviously, this can lead to false alarms.

Invariants for microequivalence can be created that remove such spurious behavior by relating the entries of the reservation stations to the relevant operand

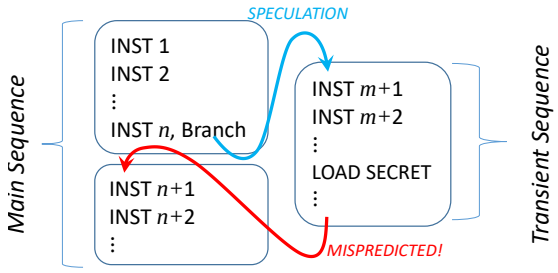


Figure 7.2: General model for TES attack: microarchitectural flow

and bookkeeping information of the reorder buffer (ROB) such that only feasible orders and operand dependencies are taken into consideration. These invariants must be formulated, however, for each pair of pending instructions in the pipeline. This, unfortunately, can be quite laborious.

A better understanding of transient instruction execution in out-of-order processors helps us to reduce this manual effort drastically. Fig. 7.2 shows a general model of how instructions execute in a TES attack. The *main sequence* of instructions reaches a branch instruction, initiating the *transient sequence* that is executed speculatively and discarded eventually due to misprediction. Since the program must have a secret-independent architectural observation, there must be no instruction in the main sequence that depends on the secret. However, instructions in the transient sequence (in a Spectre-style attack these typically belong to a privileged process called by the attacker) can access the secret but cannot commit.

In the following, we assume that the transient instructions are discarded due to a misprediction event. For the case that the transient instructions are discarded due to an exception (Meltdown-style attacks) or for the case that other kinds of speculations (e.g., load-to-store dependency prediction) are the cause of misprediction, the approach is analogous.

For an information leakage to happen, the transient sequence must affect the behavior of the main sequence, i.e., it must affect the microarchitectural execution of the program. For example, in a Spectre-STC attack (Sec. 8.2), the instructions in the transient sequence affect the timing of when instructions commit in the first block of the main sequence. As another example, for the original Spectre attack, the instructions in the transient sequence create a secret-dependent cache footprint which induces timing variations in the second block of the main sequence.

We observe that a spurious behavior, such as the one described above, can only lead to a false counterexample to the UPEC proof if it creates a false interrelation between instructions of the transient and the main sequence (e.g., a secret value being forwarded from the transient sequence to the main sequence). Due to the UPEC miter structure and the fact that the only difference between the two instances is the value of the secret, any spurious behavior within each block is irrelevant for the proof and cannot produce a false L-alert. The reason is

that within the main sequence, no execution order can create a secret-dependent result and within the transient sequence, none of the results can be committed.

This means that the bookkeeping mechanisms must be constrained to ensure the program order and corresponding operand dependencies only between the three code blocks in Fig. 7.2, but not necessarily within each block. This can be achieved by enforcing that the instructions in the main sequence do not use a result of transient instructions as an operand. Note that this is a valid constraint because the main sequence block before the branch never reads an operand from instructions after the branch instruction, and the main sequence block after the branch never reads a result from a transient (and as such always discarded) instruction.

This observation is key and allows us to approximate microequivalence effectively by using the branch instruction to split the ROB into the slots of the main sequence and the slots of the transient sequence.

Using this partitioning, false operand dependencies between the blocks can be avoided without formulating complex invariants for source and destination registers of each in-flight instruction. For example, to avoid the false operand dependence for the ADD instruction of the above example, the proof must be constrained such that if an ADD instruction from the main sequence (based on its ROB slot) is being executed by the ALU in both SoC instances, its operands must be independent of the secret. This element of the microequivalence specification is expressed in Eq. 7.1 using a pseudo-code notation. The approach is similar for other functional units.

$$\begin{aligned} &(ALU_1.instr\_ID = ALU_2.instr\_ID) \wedge \\ &ALU_1.instr\_ID \textit{ is assigned to main sequence ROB slot} \\ &\rightarrow (ALU_1.instr\_operands = ALU_2.instr\_operands) \end{aligned} \tag{7.1}$$

A more detailed description on how to specify microequivalence for any processor with a reorder buffer, as well as how to deal with other features of out-of-order processors such as nested speculation, is presented in the following sections. We start with an abstract model of an out-of-order processor with a focus on the bookkeeping mechanisms within the pipeline, and then we further elaborate on how to concretely specify microequivalence in an efficient and systematic way.

7.1.1 General Out-of-Order Processor Model

The abstract out-of-order processor model consists of 1) different functional units (FU) and sets of buffers for in-flight instructions (e.g., reservation stations), 2) a reorder buffer (ROB) which ensures that the instructions commit in program order and 3) a prediction unit (PU) which controls speculation.

Every instruction in the pipeline (in the ROB or pipeline bookkeeping buffers) has two identifiers:

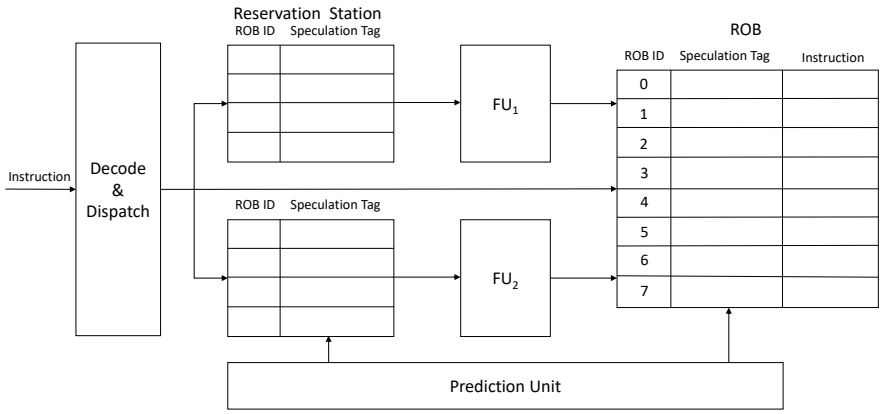


Figure 7.3: Abstract model of an out-of-order processor.

- **ID of the assigned ROB Slot (ROB ID)**: A unique identifier that reflects the program order.
- **Speculation tag (T)**: An identifier that reflects the level of speculation, i.e., the number of unresolved branches before the instruction.

Fig. 7.3 shows the block diagram of the abstract out-of-order model.

Reorder Buffer (ROB):

The ROB is a circular buffer and every instruction in the pipeline is assigned to a slot in the ROB with a unique ROB ID. The ROB IDs are assigned in program order and instructions commit their results in the order of their ROB IDs. The ROB has two pointers to maintain the order: ROB tail, which points to the next available slot for the newly dispatched instructions, and ROB head which points to the oldest uncommitted instruction. The instruction at ROB head commits when its result is available.

Prediction Unit:

A subset of instructions, such as branch, are considered *speculation initiating instructions* (SPI), i.e., they can initiate speculation, marking the next instructions as *speculative*. The speculation tag (T) of each in-flight instruction distinguishes speculative and non-speculative execution.

In order to enable nested speculation, speculation tags must be defined in a hierarchical way which allows multiple valuations for instruction tags with hierarchical relations between them. For simplicity of the following discussion, we model T as an integer.

A speculation tag T_i is *above* tag T_j in the nesting hierarchy, denoted as $T_i < T_j$, if any misprediction which will discard instructions with speculation tag T_i will also discard all instructions with tag T_j. Since there can be different tags of in-flight instructions, each SPI, besides having its own tag T, generates an additional "spawn" tag T' which becomes the speculation tag of subsequent instructions. The prediction unit (PU) determines the correctness of each prediction. Its outputs include a single-bit flag ($PU.mispred$), which is set in case of a misprediction, the spawn tag ($PU.T'_{spi}$) and the speculation tag ($PU.T_{spi}$) of the corresponding SPI instruction. For each in-flight instruction with tag T, the speculation resolution works as follows:

$$
\begin{array}{ll}
if(PU.mispred) & \\
\quad if \quad (T \geq PU.T'_{spi}) & \qquad discard\, instruction \\
else & \\
\quad if \quad (T = PU.T'_{spi}) & \qquad T \leftarrow PU.T_{spi}
\end{array}
$$

Functional Units:

The data path of the out-of-order processor consists of several functional units (FU), which receive for each instruction they process its ROB ID, $FU.inst\_ID$, and a set of operands, $FU.inst\_operands$, and return the result of the instruction, $FU.inst\_result$.

This model will be used to develop a systematic technique for pruning the state space explored in UPEC. The model provides the means for understanding the spurious counterexample problem in our verification methodology for out-of-order processors, and enables efficient development of the microequivalence invariant, without need for detailed knowledge of every aspect of an out-of-order processor.

It should be noted that in the described model, all other aspects of an out-of-order processor, such as operation scheduling, register renaming, fetch width, frontend, etc., are abstracted away. The model and, consequently, the microequivalence invariant includes only a small subset of the microarchitectural features that are typically implemented in an out-of-order processor.

7.1.2 Template for Microequivalence

In this section, we evaluate how the idea of partitioning the ROB can help us to develop a template for microequivalence based on our abstract out-of-order model. The template can be refined for any concrete out-of-order processor implementation.

In the following, the ROB ID of the highest (= oldest in terms of program order) mispredicted SPI in the hierarchy, *Inst* n in Fig. 7.2, is denoted by $root\_ID$. The ROB is partitioned into two sets:

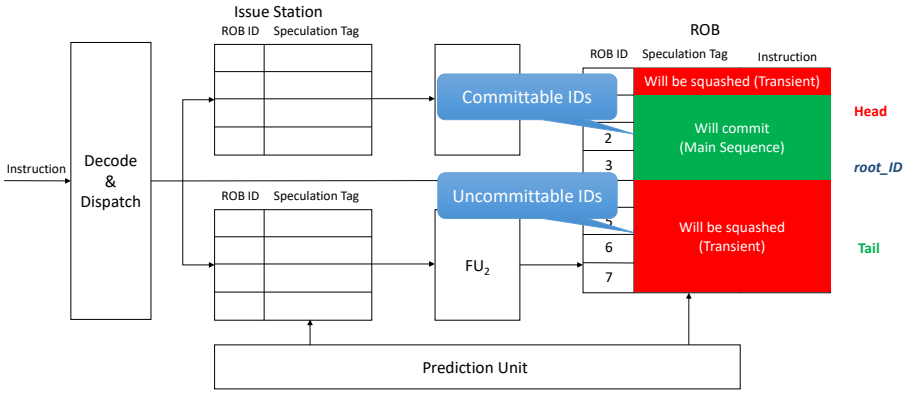


Figure 7.4: Partitioning of the ROB

- *Committable set*: ROB slots with an ID that is before the *root_ID*, i.e., ROB IDs which are between ROB head and *root_ID*. Instructions in these slots can commit their results.
- *Uncommittable set*: ROB slots with an ID that is after the *root_ID*, i.e., ROB IDs which are not between ROB head and *root_ID*. Instructions in these slots cannot commit their results. They are squashed when the misprediction signal is asserted.

Fig. 7.4 visualizes the idea of partiotining the ROB using the abstract out-of-order model.

Based on the partitioning of the ROB, we describe three sets of assumptions which together create an over-approximation of the microequivalence requirement without restricting the generality of the proof. There are a total of six assumptions denoted, in the following, by ME-1, ME-2,. . . , ME-6.

ROB Consistency

The bookkeeping mechanisms must be constrained to prevent spurious behaviors in speculative execution scenarios, in which instructions commit before speculation resolution. The following set of assumptions prevents the occurrence of spurious counterexamples in which instructions in the uncommittable ROB slots commit their results.

ME-1 (Root Instruction Pending):

- The ROB slot with *root_ID* contains an SPI instruction.
- The SPI is mispredicted.

- It remains valid (pending) until the misprediction is signaled by the prediction unit.

Note that this condition restricts the search of the solver to instruction sequences containing transient executions, which are the root cause of the attacks targeted by UPEC. This constraint on the search space contributes significantly to the tractability of UPEC. A condition similar to ME-1 can be formulated for the case that transient executions are terminated by an exception in an analogous way.

ME-2 (Uncommittable Slots Invalidated): In the clock cycle following the misprediction of the SPI with *root_ID*, every valid ROB slot in the uncommittable set is invalidated.

The ROB tail is critical for maintaining relevant information on program order. A newly fetched instruction is assigned to the ROB tail slot and its speculation tag is determined by the latest branch instruction. In a functionally correct design, the latest instruction belongs to the uncommittable set, and the ROB tail must also point to a slot in the uncommittable set. If this condition is violated in the symbolic initial state, this can create a spurious scenario in which a mispredicted branch in the transient sequence can discard an instruction in the main sequence.

ME-3 (ROB tail Consistency): Until misprediction, ROB tail points to an uncommittable ROB slot.

Note that, for our purposes, it does not matter which of the uncommittable ROB slots the ROB tail points to, exactly. This strongly simplifies the specification of ME-3.

Functional Unit Consistency

Due to our assumption that the program must have a secret-independent architectural observation (cf. Sec. 4.3), the two SoC instances in the UPEC miter are not allowed to commit different values. (The only difference can be the time point of commit.) Consequently, the instructions in the committable ROB slots must always produce the same results. We must prevent spurious counterexamples that violate this condition. Instead of specifying the details of a correct data hazard handling and operand forwarding as an invariant, it is sufficient to formulate the following assumption for every functional unit.

ME-4 (FU Consistency): At any time point prior to the misprediction, every functional unit must fulfill the following assumption:

$$
\begin{aligned}
&(FU_1.inst\_ID = FU_2.inst\_ID \wedge \\
&(FU_1.inst\_operands \neq FU_2.inst\_operands \vee \\
&FU_1.inst\_results \neq FU_2.inst\_results) \\
&\rightarrow FU_1.inst\_ID \textit{ is assigned to uncommittable ROB slot}
\end{aligned}
$$

This is based on the observation that the same instruction on the same FU can only have different operands and/or results if the secret has propagated

to this FU. This can only occur in transient executions and the instruction must be assigned to the uncommittable set. The assumption only constrains the instructions that are being executed at the same time point on both SoC instances, while allowing committable instructions that are executed at different timepoints between SoC 1 and 2 to receive secret-dependent operands. This is a valid overapproximation, since the scheduling of the instructions can only be different between SoC 1 and 2 if the secret has propagated to the control logic of the pipeline, and if such secret propagation can create secret-dependent scheduling of committable instructions, this is already a valid security alert (L-alert) regardless of the value of the committed results.

Speculation Consistency

Nested speculation, i.e., allowing speculation while older branches are still not resolved, is a common feature in out-of-order processors. To support this feature in our verification flow, the microequivalence invariant must address the scenarios with secret-dependent branches. Secret-dependent SPIs are SPIs that receive the secret as their operand. They can occur within the transient sequence. Their misprediction must not lead to discarding instructions of the main sequence. If the initial state violates this condition due to tag inconsistency, this produces a false counterexample in which one CPU instance commits an instruction while the other one discards the same instruction.

We address this issue by identifying the tag T_{main} which is the tag of the youngest (or the last) instruction within the first block of the main sequence. Based on our definition of integer speculation tags (Sec. 7.1.1), T_{main} is the highest tag between the tags of instructions in the committable set. For T_{main} to be useful in the following, we need to make sure that spurious initial states are excluded from consideration where instructions of the uncommittable set have tags smaller than or equal to T_{main}.

This requires relating to the various storage elements within the pipeline which buffer pending instructions at different stages. These buffers, among other information, store the ROB ID and speculation tag of the pending instruction.

ME-5 (Consistent Speculation Tag): Every in-flight instruction with an ROB ID in the uncommittable set must have a speculation tag greater than T_{main}.

Based on a T_{main} fulfilling ME-5, we can ensure that counterexamples are generated such that SPIs in the transient sequence only spawn tags which are greater than T_{main}.

ME-6 (Consistent Spawn Tag):

$$((PU_1.T_{spi} > T_{main}) \rightarrow PU_1.T'_{spi} > T_{main}) \wedge$$
$$(PU_2.T_{spi} > T_{main})) \rightarrow PU_2.T'_{spi} > T_{main})))$$

Implementing these conditions in a property language only involves identifying the buffers containing tags and IDs of in-flight instructions. It is worth noting that the template for microequivalence does not relate to how the branch predictor

UPEC_Base (P, k, blocking_clause):
  assume:
    at t: $micro\_soc\_state_1 = micro\_soc\_state_2$;
    at t: $public\_mem_1 = public\_mem_2$;
    at t: $secret\_data\_protected()$;
    during $t..t+k$: $secret\_load\_speculative()$;
    during $t..t+k$: $microequivalence()$;
    at $t+k-1$: Λ_P = *true*;
    at $t+k$: blocking_clause = *true*;
  prove:
    at $t+k$: $Alert\_candidate_{P1} = Alert\_candidate_{P2}$;

Figure 7.5: UPEC base property for out-of-order processors

works (e.g., a 1-bit predictor vs. 2-bit predictor) and completely abstracts away branch target prediction mechanism as well as instruction fetch logic. This significantly reduces the amount of white-box knowledge required for developing the microequivalence and saves the manual effort in the verification process.

A concrete example of refinement of the microequivalence invariant for the BOOM processor [48] is available at [116].

7.1.3 UPEC Proof for Out-of-Order Processors

ME-1 to ME-6 together form the assumption called *microequivalence*() in the UPEC base and step property for out-of-order processors, as shown in Fig. 7.5 and Fig. 7.6. The rest of the UPEC verification methodology for out-of-order processors is the same as the methodology described in Chapter 6. The conditions formulated in *microequivalence*() constrain the search to programs with transient execution. The only general assumption we make for our approach is that transient executions are either terminated by misspeculation of any kind, or by exception. We are not aware of any out-of-order architectures violating this assumption. In all other respects, ME-1 to ME-6 are obviously fulfilled by any correct out-of-order architecture. Their sufficiency for removing spurious counterexamples from the UPEC proof procedures has been made plausible in the previous section and is confirmed experimentally through our case studies.

The developed formulation of microequivalence is a template that can be refined to any concrete implementation of an out-of-order processor, as long as its architecture ensures in-order commit using a reorder buffer and complies with the general model of Sec. 7.1.1. This can be expected for the large majority of out-of-order processors.

It should be noted that in these properties, the starting microarchitectural state of the proof, which is constrained by microequivalence, is equal between the two SoC instances.

UPEC_Step (P, blocking_clause):
  assume:
    at t: $public\_mem_1 = public\_mem_2$;
    at t: $secret\_data\_protected()$;
    during $t..t+2$: $secret\_load\_speculative()$;
    during $t..t+2$: $microequivalence()$;
    at t: $\Lambda_P = \mathit{true}$
    at $t+1$: $Alert\_candidate_{P1} = Alert\_candidate_{P2}$;
    at $t+2$: blocking_clause = *true*;
  prove:
    at $t+2$: $Alert\_candidate_{P1} = Alert\_candidate_{P2}$;

Figure 7.6: UPEC step property for out-of-order processors

7.2 Dynamic Register Mapping

ISAs usually define a set of logical registers (or general-purpose registers) which are used as source and destination of different instructions. In in-order pipelines, these logical registers are statically mapped to a set of physical registers. They are the architectural (program-visible) registers of the design. Out-of-order pipelines, on the other hand, rely on register renaming, and some microarchitecturs (e.g., BOOM [48]) realize this using *dynamic register mapping*. The microarchitecture implements a physical register file, which usually has more registers than specified in the ISA, as well as a map table which holds, at each time point of program execution, the mapping between logical ISA registers and physical registers. This allows for storing both speculative and committed values in the same register file.

This creates an additional challenge for the UPEC approach: The proof methodology relies on making the distinction between L-alerts and P-alerts. This means that UPEC must know the architectural state of the system which, for a processor with dynamic register mapping, depends on the state of the ROB and the map tables.

Fortunately, there is an efficient solution to this problem. In our proof methodology for out-of-order processors, microequivalence constrains the search to only TES attacks, i.e., UPEC analyzes execution traces in which the microarchitectural observation is independent of the secret. This means that the same sequence of values is committed to the architectural registers in both SoC instances and only the timing of instruction commit can be secret-dependent. This allows us to modify the definition of an L-alert to be used in Alg. 1 (lines 7, 19) and Alg. 2 (line 6). Instead of comparing values of architectural registers, as in Def. 5, we check, as given by Eq. 7.2, whether or not the instructions in the two SoC instances reach the head of the ROBs simultaneously.

$$\texttt{soc1.core.rob.head} == \texttt{soc2.core.rob.head} \tag{7.2}$$

7.3 Confidentiality of Page Tables

Protecting physical addresses that are stored in the page table is important to avoid attack scenarios such as RowHammer [119]. The Spoiler [120] attack shows an example of how TES attacks can target the page table and form the precursor for a powerful RowHammer attack.

We can use UPEC to verify the absence of any vulnerability that reveals the physical address to a user-level process. So effectively, in such an experiment, the confidential information in our miter model is a physical page number stored in the Translation Lookaside Buffer (TLB).

It is well known that, theoretically, a brute-force approach exists in which different processes operating on shared pages can enumerate the virtual address space and can obtain information about virtual-to-physical mappings by observing cache hits and misses. (Two virtual addresses that are mapped to the same physical address, also point to the same cache line.) This, however, yields limited information about the physical address (i.e., just the fact that two virtual addresses are mapped to the same physical address is revealed).

Also, due to its complexity, this brute-force approach is commonly considered infeasible in practice which is why we wish to exclude this scenario from the UPEC proof.

The goal of the proof is to verify unique program execution between the two SoC instances, under the assumption that a single pair of physical addresses (a_1, a_2) is mapped to different virtual addresses in the two SoC instances, i.e., only a single entry in the page table is different between SoC 1 and 2. This allows us to exclude the above-mentioned brute-force scenario, by constraining the memory subsystem to always produce the same response to a_1 and a_2, in terms of the returned data for loads and cache hits and misses. This constraint blocks the ability of the attacker to observe physical addresses through the state of the memory subsystem by a simple brute-force attack. It should be noted, however, that it still allows capturing scenarios in which information about physical addresses is leaked through timing side channels. The reason is that, for all the addresses other than a_1 and a_2, the memory subsystems of SoC 1 and 2 are allowed to return different responses, if different addresses are being accessed. In our proof procedure, a_1 and a_2 are modeled through symbolic variables to reach a complete proof.

A case study based on this approach at the example of Ariane [47] and BOOM [48] is reported in Chapter 9.

7.4 UPEC for Verifying Multicore Systems

A big hurdle in the verification of multicore SoCs is the sheer complexity of such designs. A multicore system typically consists of multiple processors connected through a shared bus to shared memory or L2 cache while each core has its own separate L1 cache.

The coherency between the caches is maintained by sending and receiving a

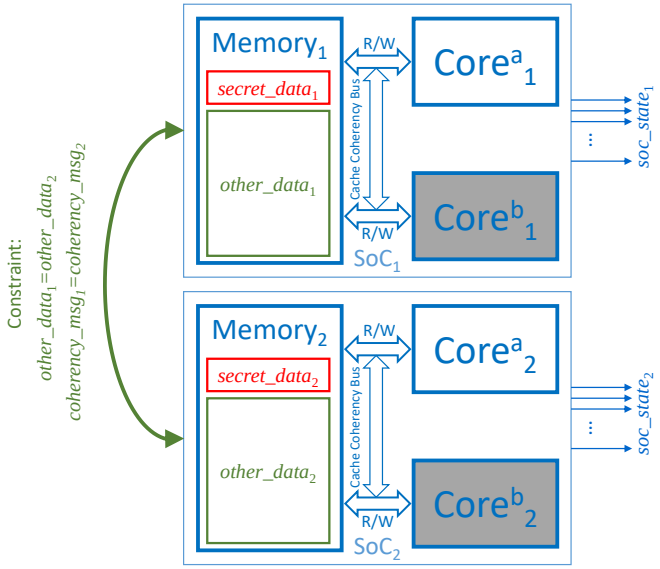


Figure 7.7: UPEC miter model for a multicore system

set of cache coherency messages that are defined by the cache coherency protocol. In such systems, the only possibility for a core to affect program execution on the other core is through the cache coherency messages that are produced as a result of load and store instructions.

When verifying a homogenous multicore SoC using the UPEC verification methodology, the UPEC miter model allows for effective blackboxing which significantly reduces the complexity of the model. The blackboxing approach is based on the observation that, for any confidentiality violation, it is sufficient for the secret to affect program execution in one core. Therefore, we only need to consider one core together with the cache coherency infrastructure.

Fig. 7.7 shows the UPEC miter for the case of a 2-core SoC, with the second core in each SoC blackboxed.

The considered threat model is that the attacker controls both cores while trying to read the secret residing in the memory. With this threat model, the cache coherency messages do not contain the secret (they are controlled by the attacker) and thus they are constrained to be equal between SoC 1 and SoC 2. With this threat model, UPEC can detect any security violation that may arise due to interaction between the memory transactions of *core A* and cache coherency messages coming from *core B*. This model can be easily extended to any number of cores for a similar threat model.

The same blackboxing approach can also be applied for other threat models that are relevant for multicore systems. An example of such a threat model can be the case in which the victim is controlling the second core and the attacker

is trying to snoop the cache coherency bus and leak the data/address that is being read/accessed by the victim. In this threat model, the program execution of the victim does not need to be considered and the second core can still be blackboxed. The threat model can be expressed by appropriate constraints on the cache coherency messages that are coming from the blackboxed core, by allowing them to have different attributes between SoC 1 and SoC 2. Developing these constraints is the subject of future work.

Chapter 8

Novel Insights into Hardware Security

We have conducted multiple UPEC case studies on various open-source designs. The exhaustive nature of UPEC enabled us to evaluate corner case scenarios and look into previously unknown security issues. UPEC does not rely on any a priori knowledge or intuition about the possible security issues from the user, and as a result, our experiments have found multiple new vulnerabilities which do not automatically fall into a category of known security issues. These findings, besides showing the power and necessity of formal verification in the hardware security domain, provide new insights about hardware security which can be useful to hardware designers and security analysts.

In this chapter, we describe some of the security insights gained from our UPEC experiments. Sec. 8.1 describes the threat of transient execution side channels existing even in in-order pipeline processors. In Sec. 8.2, we describe a new variant of the Spectre attack, which expands the threat of port contention side channels to single-threaded processors. We finally discuss the challenges associated with patching a vulnerability at the RTL in Sec. 8.3.

8.1 TES Attacks in In-Order Pipelines

The emergence of Spectre [24] and Meltdown [25] showed the threat of TES attacks in modern processors. These attacks and their variants [26–30] all exploit certain microarchitectural features typically found in high-end processors. All of these attacks rely on either out-of-order or speculative execution features which created the intuition that the threat of TES attacks were limited to high-end out-of-order/speculative execution pipelines.

In this section, we prove this intuition wrong and provide experimental proof that TES attacks are possible also in average-complexity processors using the classical in-order pipelining scheme. Our experiments show that hardware vulnerabilities by TES are not only a consequence of early architectural decisions

on the features of a processor, such as out-of-order execution or speculative execution. We discovered that small (and functionally correct) design changes, as they occur regularly when optimizing a design at the RTL, for example inserting or removing a buffer, may open or close a side channel. Such a vulnerability can have severe implications for a wide range of applications in Embedded Systems and Internet-of-Things (IoT) where simple in-order processors are commonly used.

Fig. 8.1 presents an intuitive example of how a hardware vulnerability may be created by certain design decisions even in a simple in-order pipeline. Consider two instructions as they are executed by a 5-stage pipeline, and let us assume that register R1 contains the address of some secret data that should not be accessible for a user-level program. Let us consider the routine situation where the cache holds a copy of the secret data from an earlier execution of operating system-level code. The first instruction, as it enters the memory (M) stage of the pipeline, makes a READ request to the protected address, attempting to read the secret data. The second instruction at the same time enters the execute (EX) stage of the pipeline and uses the secret data in an address calculation. For performance reasons, a cached copy of the requested secret data may be forwarded directly to the EX stage. This is a common and correct microarchitectural forwarding [121] feature and, by itself, doesn't cause a problem if the memory protection scheme in place makes sure that neither instruction #1 nor any of the subsequent instructions in the pipeline can complete. We assume that such a memory protection scheme is correctly implemented in our scenario.

In the next clock cycle, as shown in the two bottom parts of Fig. 8.1, instruction #1 enters the write-back (WB) stage and instruction #2 enters the M stage of the pipeline. Concurrently, the memory protection unit has identified in this clock cycle that instruction #1 attempts access to a protected location. This raises an exception for illegal memory access and causes the pipeline to be flushed. From the perspective of the programmer at the ISA level the shown instructions are ignored by the system. In place of these instructions a jump to the exception handler of the operating system is executed.

However, depending on certain design decisions, the instruction sequence may have a *side effect*. Instruction #2 triggers a READ request on the cache. Let us assume that this request is a *cache miss*, causing the cache controller to fill the corresponding cache line from main memory. Even though the READ request by the pipeline is never fulfilled because it is canceled when the exception flushes the pipeline, it may have measurable consequences in the states of the microarchitecture. In the scenario on the left ("vulnerable design") the state of the cache is changed because the cache line is updated.

Such a cache foot print dependent on secret data is the basis for the Meltdown attack. It can be exploited as detailed in [25]. However, in Meltdown this critical step to open a TES is only accomplished exploiting the out-of-order execution of instructions. In our example, however, the behavior of the cache-to-memory interface may cause the security weakness. Not only the core-to-cache transaction but also the cache-to-memory transaction must be canceled in case of an exception to avoid any observable side effect ("secure design" variant on the right). This

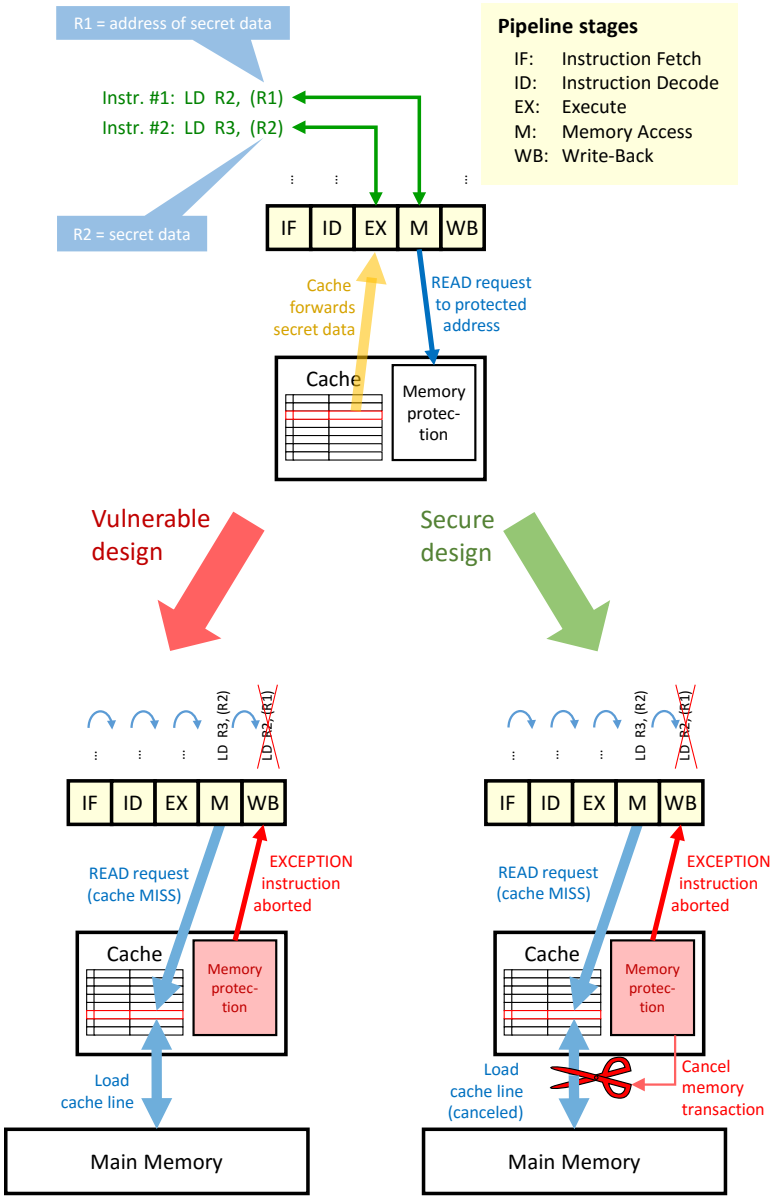


Figure 8.1: In-order pipeline vulnerability: the design decision whether or not a cancellation mechanism is implemented for the transaction between memory and cache does neither have an impact on functional design correctness nor on the correct implementation of the memory protection. However, it may insert a microarchitectural side effect that can form the basis for a transient execution attack.

is not at all obvious to a designer. There can be various reasons why designers may choose not to implement cancellation of cache updates. First of all, both design variants are functionally correct. Moreover, the vulnerable design prohibits access to protected memory regions. Implementing the cancellation feature incurs additional cost in terms of design and verification efforts, hardware overhead, power consumption, etc. There is no good reason for implementing the feature unless the designer is aware of the vulnerability.

This example shows that a Meltdown-style attack can be based on even subtler side effects than those resulting from out-of-order execution. This not only opens new possibilities for known Meltdown-style attacks in processors with in-order pipelines. These subtle side effects can also form the basis for new types of TES attacks which have been unknown so far, as demonstrated in the following section.

8.1.1 Orc: A New TES Attack

The term *Read-After-Write (RAW) Hazard* denotes a well-known design complication resulting directly from pipelining certain operations in digital hardware [121]. The hazard occurs when some resource is to be read after it is written, causing a possible stall in the pipeline. RAW hazards exist not only in processor execution pipelines but also elsewhere, e.g., in the core-to-cache interface.

For reasons of performance, many cache designs employ a pipelined structure which allows the cache to receive new requests while still processing older ones. This is particularly beneficial for the case of store instructions, since the core does not need to wait until the cache write transaction has completed. However, this can create a RAW hazard in the cache pipeline, if a load instruction tries to access an address for which there is a pending write.

A RAW hazard needs to be properly handled in order to ensure that the correct values are read. A straightforward implementation uses a dedicated hardware unit called *hazard detection* that checks for every read request whether or not there is a pending write request to the same cache line. If so, all read requests are removed until the pending write has completed. The processor pipeline is stalled, repeating to send read requests until the cache interface accepts them.

In the following, we show an example how such a cache structure can create a security vulnerability allowing an attacker to open a transient execution side channel. Let's assume we have a computing system with a cache with write-back/write-allocate policy and the RAW hazard resolution just described. In the system, some confidential data (*secret data*) is stored in a certain protected location (*protected address*).

For better understanding of the example, let us make some more simplifying assumptions that, however, do not compromise the generality of the described attack mechanism. We assume that the cache holds a valid copy of the secret data (from an earlier execution of privileged code). We also simplify by assuming that each cache line holds a single byte, and that a cache line is selected based

```
1: li x1, #protected_addr    // x1 ← #protected_addr
2: li x2, #accessible_addr   // x2 ← #accessible_addr
3: addi x2, x2, #test_value  // x2 ← x2 + #test_value
4: sw x3, 0(x2)              // mem[x2+0] ← x3
5: lw x4, 0(x1)              // x4 ← mem[x1+0]
6: lw x5, 0(x4)              // x5 ← mem[x4+0]
```

Figure 8.2: Example of an Orc attack: in this RISC-V code snippet, *accessible_addr* is an address within the accessible range of the attacker process. Its lower 8 bits are zero. The address within its 256 bytes offset is also accessible. *test_value* is a value in the range of 0...255.

on the lower 8 bits of the address of the cached location. Hence, in our example, there are a total of $2^8 = 256$ cache lines.

The basic mechanism for the Orc attack is the following. Every address in the computing system's address space is mapped to some cache line. If we use the secret data as an address, then the secret data also points to some cache line. The attacker program "guesses" which cache line the secret data points to. It sets the conditions for a RAW hazard in the pipelined cache interface by writing to the guessed cache line. If the guess was correct then the RAW hazard occurs, leading to slightly longer execution time of the instruction sequence than if the guess was not correct. Instead of guessing, of course, the attacker program iteratively tries all 256 possible cache locations until successful.

Fig. 8.2 shows the instruction sequence for one such iteration. The shown `#test_value` represents the current guess of the attacker and sets the RAW hazard conditions for the guessed cache line. The sequence attempts an illegal memory access in instruction #5 by trying to load the secret data from the protected address into register `x4`. The processor correctly intercepts this attempt and raises an exception. Neither is the secret data loaded into `x4` nor is instruction #6 executed because the exception transfers control to the operating system with the architectural state of instruction #5. However, before control is actually transferred, instruction #6 has already entered the pipeline and has initiated a cache transaction. The cache transaction has no effect on the architectural state of the processor. But the execution time of the instruction sequence depends on the state of the cache. When probing all values of `#test_value`, the case will occur where the read request affects the same cache line as the pending write request, thus creating a RAW hazard and a stall in the pipeline. It is this difference in timing that can be exploited as a side channel.

Let us look at the sequence in more detail. The first three instructions are all legal for the user process of the attacker. Instruction #1 makes register `x1` a pointer to the secret data. Instruction #2 makes register `x2` a pointer to some array in the user space. The address of this array is aligned such that the 8 least significant bits are 0. Instruction #3 adds `#test_value` as an offset to `x2`. This value is in the range 0...255.

Instruction #4 is a *store* of some value `x3` into the user array at `x2`. This is a legal instruction that results in a write request to the cache. Note that the

destination cache line is determined by `#test_value` since the lower 8 bits of the write address are used for cache line addressing. The cache accepts the write request and immediately becomes ready for a new request. The cache controller marks this write transaction as pending until it is complete. (This takes a few clock cycles in case of a cache hit and even significantly longer in case of a miss.)

In the next clock cycle, instruction #5 attempts an illegal load from the secret address, producing a read request to the cache. It will take a small number of clock cycles before the instruction has progressed to the write-back (WB) stage of the processor pipeline. In this stage the exception will be raised and control will be transferred to the OS kernel. Until the instruction reaches the WB stage, however, all components including the cache keep working. Since the cache has a valid copy of the secret data, it instantly answers the read request and returns the secret data to the core where it is stored in some internal buffer (inaccessible to software).

Even though the illegal memory access exception is about to be raised, instruction #6 is already in the pipeline and creates a third read request to the cache. This request is created by a forwarding unit using the value stored in the internal buffer. Note that the request is for an address value equal to the secret data. (It does not matter whether an attempt to load from this address is legal or not.)

Let us assume that this address value happens to be mapped to the same cache line as the write request to the user array from instruction #4. This will be a read-after-write (RAW) hazard situation, in case the write request from instruction #4 is still pending. The read transaction must wait for the write transaction to finish. The cache controller stalls the core until the pending write transaction has completed.

In case that the read request affects a different cache line than the pending write request there is no RAW hazard and the processor core is not stalled.

In both cases, the processor will eventually raise the exception and secret data will not appear in any of the program-visible registers. However, the execution time of the instruction sequence differs in the two cases because of the different number of stall cycles. The execution time depends on whether or not a RAW hazard occurs, i.e., whether or not `#test_value` is equal to the 8 lower bits of the secret data.

Assuming the attacker knows how many clock cycles it takes for the kernel to handle the exception and to yield the control back to the parent process, the attacker can measure the difference in execution time and determine whether the lower 8 bits of the secret are equal to `#test_value` or not. By repeating the sequence for up to 256 times (in the worst case), the attacker can determine the lower 8 bits of the secret. If the location of the secret data is byte-accessible, the attacker can reveal the complete secret by repeating the attack for each byte of the secret. Hardware performance counters can further ease the attack since they make it possible to explicitly count the number of stalls.

This new transient execution side channel can be illustrated at the example of the RISC-V RocketChip [46]. The original RocketChip design is not vulnerable to the Orc attack. However, with only a slight modification and without corrupting

the functionality, it was possible to insert the vulnerability. The modifications actually optimized the performance of the design by bypassing a buffer in the cache, by which an additional stall between consecutive load instructions with data dependency was removed. The modified design passes the standard ISA tests provided by the RISC-V framework and successfully runs a Linux operating system, which attests to the functional correctness of the design.

The attack does not require the processor to start from a specific state: any program can precede the code snippet of Fig. 8.2. The only requirement is that `protected_addr` and `accessible_addr` reside in the cache before executing the code in Fig. 8.2.

The described vulnerability is a very subtle one, compared to Meltdown and Spectre, and it is feasible in an in-order non-speculative pipeline. Previous TES attacks in in-order pipelines exploited speculative execution [122] or compile-time branch optimizations that mimic the speculative execution in VLIW processors [123].

The Orc attack is caused by a RAW hazard not in the processor pipeline itself but in its interface to the cache. It is very hard for a designer to anticipate an attack scenario based on this hazard. The timing differences between the scenarios where the RAW hazard is effective and those where it isn't are small. Nevertheless, they are measurable and can be used to open a covert communication channel.

This new type of transient execution side channel, discovered by UPEC, gives some important messages:

- Subtle design changes in standard RTL processor designs, such as adding or removing a buffer, can open or close a side channel. Although specific to a particular design, the newly discovered vulnerabilities may inflict serious damage, once such a vulnerability becomes known in a specific product.
- The *Orc* attack is based on the interface between the core (a simple in-order core in this case) and the cache. This provides the insight that the *orchestration* of component communication in an SoC, such as RAW hazard handling in the core-to-cache interface, may also open or close side channels. Considering the complex details of interface protocols and their implementation in modern SoCs, this can further complicate verifying security of the design against TES attacks.
- The new insight that the existence of transient execution side channels does not rely on certain types of processors but on decisions in the RTL design phase underlines the challenge in capturing such vulnerabilities and calls for methods dealing with the high complexity of RTL models.
- The presented attack is based on a so far unsuspicious microarchitectural feature. This makes it resistant to most existing techniques of security verification, as discussed in Sec. 3. The verification method, therefore, should be exhaustive and must not rely on *a priori* knowledge about the possible attacks.

These challenges motivate the proposed UPEC approach as an exhaustive verification technique. It is meant to be used by the designer during the RTL design phase to detect all possible cases of vulnerabilities, even the ones that exploit previously unknown side channels.

8.2 Spectre-STC: A New Variant of Spectre

Architectural resources and ports shared between threads running simultaneously on a core are known to form communication side channels in processors with simultaneous multithreading (SMT). Resource or port contention side channels in SMT processors have been exploited by classical side channel attacks to spy on security-critical sofware components [124], as well as by TES attacks to create powerful Spectre gadgets [125]. (A *gadget* is an exploitable instruction sequence existing in the software stack that can be triggered by the attacker and executed in privileged mode.) These attacks created the general intuition that port contention were a threat only relevant to SMT processors and that by turning this feature off the system could be protected against port contention attacks.

The Spectre-STC attack scenario demonstrates that also single-threaded processors can be vulnerable to contention-based Spectre attacks. We describe the attack at the example of the BOOM processor. A hypothetical gadget is presented to show the feasibility of the attack. (Analyzing other, commercial processors for this vulnerability or searching existing kernel software for relevant gadgets is out of the scope of this thesis.)

The attack's transient instruction sequence consists of two main components:

1. **Secret-Dependent Branch:** Assuming a processor allows nested branch speculation, there can be branch instructions in a speculative sequence which depend on the secret value brought to the pipeline through a speculative load. This causes the CPU to perform *different operations* depending on the value of the secret. It should be noted that this is different from the secret-dependent control flow vulnerability in cryptographic software, since a speculative secret-dependent branch may not depend on any sensitive value in the ISA-level flow of the program and dependency on the secret can only be created through speculation.

2. **Port Contention:** The ports to access microarchitectural resources are usually shared between many operations, and port contention can alter the timing of operations. This has security implications in SMT processors, in which two threads run concurrently and share all the resources [125]. In addition, Spectre-STC shows that even single-threaded processors may be vulnerable to leaking information through contention.

The Spectre-STC vulnerability was first found by verifying the BOOM design using the UPEC verification methodology. Here, we also use the BOOM design to illustrate how this attack works.

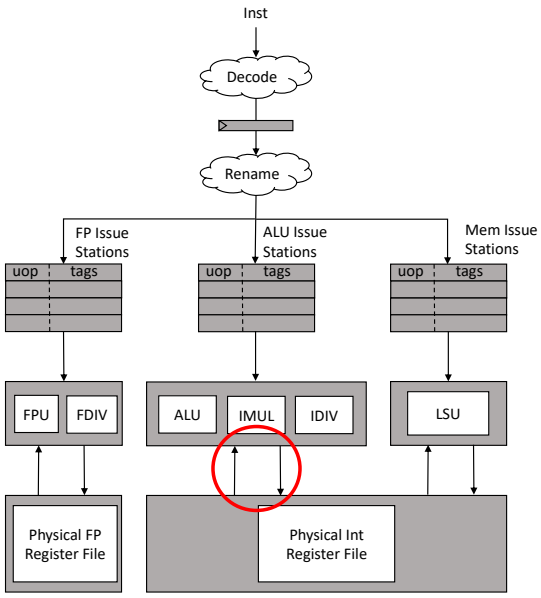


Figure 8.3: Resource sharing between different components in BOOM microarchitecture

Fig. 8.3 shows the block diagram of decode, register renaming, execute and writeback stage of the BOOM microarchitecture. In this figure (as marked by the red circle), the ALU, the integer multiplier and the integer division unit use the same write port to access the register file. In case of contention, there is a fixed-priority arbitration, in which the ALU has the highest and the division unit the lowest priority, regardless of the program order. This means a division instruction can be delayed by a later issued multiply or add instruction due to port contention.

Fig. 8.4 shows a possible gadget for the Spectre-STC attack. The basic idea is to start a division before the transient execution, and then, in the transient time window, using a secret-dependent branch, perform multiplications and additions (and create port contention) if the secret is equal to a probing value. The attacker can probe the secret by measuring how long it takes to execute the gadget code. It should be noted that this is not the only possible gadget, but rather the most simple one, used to demonstrate the attack.

SmotherSpectre [125] is a Spectre variant which exploits port contention in SMT processors. However, unlike SmotherSpectre, Spectre-STC is feasible on a single-threaded processor, which further expands the threat of port contention-based attacks. Allocating only mutually trusting threads to a physical core, which was believed to be a solution for the port contention side channels, is shown to be ineffective in a scenario like ours.

```
1: DIV x1,x2,x3              // x1 ← x2 ÷ x3
2: BR x1 ≥ 42, Line_m        // if x1 ≥ 42 jump to line m
3: LB x4, addr_of_secret     // x4 ← mem[addr_of_secret]
4: BR x4 ≠ probe_val, Line_n // if x4 ≠ probe_val jump to line n
5: MUL                       // a mult. instruction with arbitrary operands
6: ADD                       // an add. instruction with arbitrary operands
   ...                       // more such mult.'s and add.'s
n: ...                       // an arbitrary sequence of instructions...
   ...                       // ...other than MUL and ADD
m: ...
```

Figure 8.4: Pseudo-code of a Spectre-STC gadget: *addr_of_secret* and *probe_val* are values (typically stored in registers) that can be controlled by attacker-provided inputs. The division instruction takes multiple clock cycles, causing lines 3 to m to execute speculatively (based on wrong prediction). Lines 5 to n will only execute if the secret value is equal to the probing value *probe_val*. Assuming that the execution of sequences between lines 5 to n and n to m, will take longer than the DIV in line 1, register file write port contention between DIV and MUL/ADD instructions (and an additional delay) may or may not happen based on the value of the secret (line 4).

The Spectre-STC vulnerability can be introduced in case a speculative instruction is prioritized over an earlier non-speculative instruction when resolving resource contention. This can occur not only due to detailed microarchitectural design decisions, such as the fixed priority arbitration for write port access in BOOM, but also due to high-level architectural features. In fact, any out-of-order processor containing a non-pipelined functional unit with multi-clock-cycle operations may be vulnerable to Spectre-STC due to resource contention. This may happen, for example, if the execution order between a non-speculative division and a speculative division located in a secret-dependent branch is reversed. The security implication of non-pipelined functional units has also been observed by two independent researches, which have shown the possibilitiy of Speculative interference [30] and SpectreRewind [126] attacks. The Speculative intereference attack also exploited this vulnerability to create a persistent state side channel through the cache. It should be noted that, unlike these two attacks, Spectre-STC has been identified based on a formal analysis using UPEC on the BOOM RTL design, which enabled us to find different root causes of the vulnerability.

Spectre-STC is another experimental proof to the fact that RTL design decisions can have security implication w.r.t. TES attacks. It further emphasizes the need for an exhaustive verification approach such as UPEC to assist the designers in evaluating the security implications of RTL design optimizations.

The described scenario may be exploited by a malicious software/hardware provider to create an invisible "backdoor" in the system, as a new kind of Trojan. Such a Trojan can be extremely hard to find, since it does not conform to the conventional Trojan models (trigger and payload), and more importantly, it neither corrupts the functionality of the design, nor does it add any redundant logic. With the increasing role of shared hardware and software infrastructures involving components from numerous providers as well as open-source domains,

such a risk must be given appropriate attention.

8.3 Challenges in Patching Hardware

We have elaborated on the threat of TES attacks and the challenges associated with detecting these vulnerabilities in hardware. Our case studies have also resulted in another interesting security insight: The challenge of addressing the threat of TES attacks does not end with detecting the vulnerability. In fact, patching a security issue in the RTL design can be complicated and tricky, even if the root cause of the vulnerability is known to the designer. The main reason is that it is hard for designers to anticipate the implications of design decisions in terms of side channels. Furthermore, design flows, typically, face a myriad of non-functional requirements which can cause numerous design updates, each possibly interfering with the security countermeasures that have been implemented previously and rendering them useless.

In this section, we elaborate on the challenge of patching vulnerabilities to TES attacks, based on two case studies on BOOM v3.0.0. It should be noted that these case studies are different from the ones reported in Chapter 9, which is based on an older version of BOOM.

8.3.1 Patching against Spectre-STC

In Sec. 8.2, a vulnerability to Spectre-STC in the BOOM design is described, which enables an attacker to exploit port contention between ADD and DIV instructions in a speculative attack. The root cause of the described vulnerability is the fixed priority of arbitration between ADD and DIV instructions in case of contention for writing to the register file. In this section, we report on a case study on BOOM v3.0.0 in which a dynamic arbitration based on program order for access to register file is implemented, to address the root cause of Spectre-STC.

For a straightforward solution to mitigating this vulnerability, we implemented an arbitration scheme at the write port of the register file shared between the ALU and DIV units. The arbiter receives the ROB IDs of the DIV and ALU instructions, and in case of simultaneous accesses by both ALU and DIV units, it grants the port to the older instruction in terms of the program order.

This design modification, at first glance, should patch the vulnerability against the Spectre-STC. The older non-speculative DIV is no longer delayed by a speculative ADD instruction. An unaware designer, who is not assisted by formal verification techniques, may now sign off the design as secure. The patched design, however, creates a new contention-based side channel at a different point.

In the original BOOM design, ALU operations always have the priority to write back results into the register file. As a result, there is no scenario in which an ALU operation is stalled in the middle of execution. Considering the ALU is fully pipelined, it can receive a new operation at every clock cycle. In the patched design, however, this is modified. An ALU operation may lose the arbitration

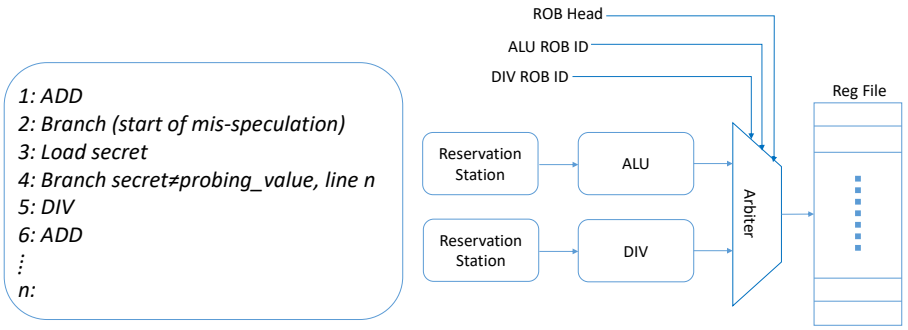


Figure 8.5: Initial patch against Spectre-STC and its security flaw

to an older DIV operation and thus, the ALU may get stalled and may not be available for a new operation in the following clock cycle. This new stalling scenario can create contention for using the ALU between different operations.

Fig 8.5 shows the scenario in which the new resource contention can leak secret information. Let us assume that in this situation, while the first ADD instruction (line 1) is still waiting for the operands, the processor speculatively executes the second branch (line 4), which, depending on the value of the secret, may or may not execute a sequence of DIV and ADD instructions. Since the DIV instruction is older, it wins the arbitration over the ADD instruction and it can cause stalling in the ALU, which, in turn, delays the first ADD instruction (line 1). The first ADD should stay in the reservation station until the contention is resolved and the younger speculative ADD writes back its result. This delay may only happen if the secret is equal to the probing value, which creates a TES attack scenario similar to Spectre-STC.

The new side channel is very intricate and hard to detect and it is only introduced to the system through the design modification aiming to patch another side channel. This demonstrates that implementing fine-grained design patches can be challenging and requires exhaustive verification to avoid such scenarios.

Fig 8.6 shows a patch to Spectre-STC which does not create a new side channel. The FIFO between the ALU and register file must be selected big enough so that it may never be full. Considering the latency of ALU and DIV operations, computing the size of the FIFO is straightforward.

8.3.2 Patching against Meltdown

BOOM was advertised by the design team to be secure against the Meltdown category of TES attacks. As a matter of fact, our UPEC case study on BOOM v2.0.1 confirmed this statement (the case study is reported in Chapter 9).

In BOOM, security against Meltdown attacks is achieved by separating the address translation from the cache accesses and using a physically-indexed

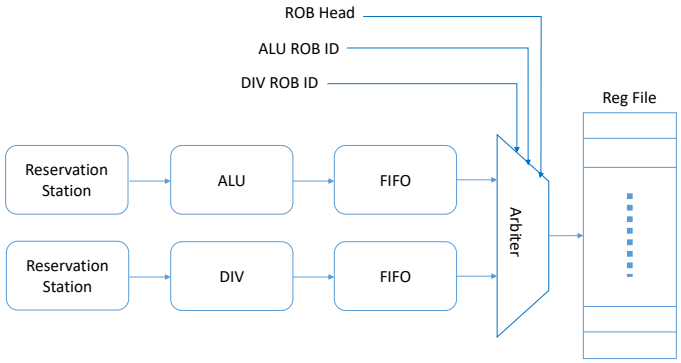


Figure 8.6: Patch against Spectre-STC

physically-tagged cache design. It means that the address translation happens within the Load/Store Unit (LSU) and then the physical address is sent to the data cache. As a result, the exception is known to the LSU before the cache is accessed and thus the propagation of the secret can be blocked in case of an illegal load before the exception is raised by the ROB.

Our case study on BOOM v3.0.0 led to a surprising observation: Although the LSU still uses the same physically-indexed physically-tagged cache design, the design failed the UPEC proof and was proven to be vulnerable to the Meltdown attack.

BOOM v3.0.0 still implements the fundamental fix against Meltdown, i.e., checking the validity of the address before issuing the load. However, a design update on the LSU removed the state bits that marked illegal loads in the load queue, presumably for area or power optimization. This design update did not corrupt the functionality since the exception is still properly sent to the ROB, which raises the exception when the illegal load reaches the head. However, the new LSU does not check the validity of the address when issuing the load to the data cache. As a result, an illegal load is executed and the data is returned to the pipeline. This enables a Meltdown attack.

The vulnerability was reported to the design team in charge of the BOOM design. Considering the severity of the threat, a patch was deployed. The patch was based on adding a new bit to every entry in the load queue, called *failure* bit, to mark whether or not the address translation phase determined the address to be valid. This bit was later used to block illegal loads from being issued to the data cache.

The patched design was re-verified by UPEC, which, to our surprise, proved the design to be still vulnerable to Meltdown, in a more subtle way. In the case when there is no contention over the TLB and data cache, the load is immediately issued to the data cache. The failure bit, however, is updated in the following clock cycle. As a result, the content of the failure bit is not up to date when the

load is issued and thus, in this particular scenario, an illegal load can still get executed. If the load access is a cache hit, then the data is sent back to the core, which can be leaked through side channels similar to a Meltdown attack.

The vulnerability was, again, reported to and confirmed by the design team and a UPEC-verified fix was proposed. However, officially deploying the fix to the BOOM design is postponed to BOOM v4.

This case study is a real-world example of the threat of TES attacks, even when they are well understood by the designers. Design teams consistently check the functional correctness of the design updates w.r.t. the functional specification, but vulnerabilities to side channels can easily escape this verification process. The complexity of modern pipelines makes it necessary to work on fine-grained optimization to improve performance, while each decision can have drastic security implications. It is also extremely challenging for designers to think in terms of side channels and anticipate all the side channel scenarios that can occur due to a simple design decision. So it is naturally foreseeable to have TES vulnerabilities being introduced to the design in the process of design optimizations. This observation further attests to the need for hardware security verification techniques such as UPEC to be part of the sign-off process.

Chapter 9

Experimental Results

The proposed UPEC approach has been evaluated by verifying three different SoC designs: RocketChip (v1.2.0) [46], the Berkeley Out-Of-Order Machine (BOOM v2.0.1) [48] and Ariane (v4.1.2) [47]. All results were obtained using the commercial property checker OneSpin 360 DV running on an Intel Core i7 at 3.4 GHz, with 32 GB of RAM.

9.1 UPEC for In-order Pipelines

We evaluated the effectiveness of UPEC for capturing vulnerabilities in in-order pipelines by targeting different design variants of RocketChip and the Ariane design. The considered RocketChip design variants included the original design as well as two modified design variants vulnerable to two different versions of the Orc attack. In the first modified variant, we conditionally bypassed one buffer in the cache-to-core interface pipeline to make the design vulnerable to the attack described in Sec. 8.1.1. For the second vulnerable design, RocketChip was modified such that a cache line refill is not canceled in case of an invalid access (similar to the vulnerability depicted in Fig. 8.1). While the illegal access itself is not successful but raises an exception, the cache content is modified and can be analyzed by an attacker.

It should be noted that these design modifications are minor (compared to the overall design size) and they represent a realistic design optimization. The modified designs successfully pass all tests in the test suite provided by the RISC-V framework.

In the following experiments, the secret data is assumed to be in a protected location, A, in the main memory, possibly with a copy in the data/instruction cache. In our RocketChip experiments, memory protection was implemented using the *Physical Memory Protection (PMP)* scheme of the RISC-V ISA by configuring the memory region holding the location A of the secret data as inaccessible in user mode. In our Ariane experiment, secret protection is achieved through virtual address translation, i.e., the memory page holding the location A

is marked as unreadable and unexecutable by the page table. In our experiments, we assumed correctness of the page table walking mechanism, i.e., the content of the Translation Lookaside Buffer (TLB) is always correct w.r.t. the page table.

For RocketChip and Ariane the proof methodology is based on the properties of Figures 6.1 and 6.2 and does not employ *microequivalence*. Consequently, these experiments target information leakage through both functional bugs and TES attacks. UPEC successfully captured all vulnerabilities to Orc Attacks in the RocketChip design variants. In addition, UPEC found a design bug introducing an ISA incompliance in the locking mechanism of the *Physical Memory Protection* (PMP) unit of the original RocketChip design.

PMP in the RISC-V ISA is managed by having pairs of address and configuration registers (PMP entry). There are 16 registers for 32-bit addresses, each of them associated with an 8-bit configuration register for storing access attributes. There is also a locking mechanism by which the software can lock PMP entries from being updated, i.e., only a system reboot can change the contents of the locked PMP entries. According to the ISA specification, if a memory range and the PMP entry for the end address is locked, the PMP entry of the start address of the range must be automatically locked, regardless of the contents of the associated configuration register. This mechanism has not been correctly implemented in the RocketChip, enabling a modification of the start address of a locked memory range in privileged mode, without requiring a reboot. This can compromise security [6] and is therefore forbidden by the RISC-V ISA. The vulnerability has been removed in the current version of the design at the time of this writing.

A 2-core design variant of RocketChip is also verified using UPEC, based on a sound blackboxing approach as described in Sec. 7.4. In this experiment, the second core is blackboxed and the attacker is assumed to have control over both cores to run user-level processes. Comparing the proof runtime of this experiment with the experiments on the single core RocketChip design shows the fact that the employed abstraction approach described is effective in mitigating complexity of multicore systems.

UPEC proved the Ariane design to be free of vulnerabilities to TES attacks. However, UPEC found an invalid information flow which has relevant security implications. (This was reported to the Ariane development team.)

In Ariane, in every fetch cycle, the address is translated by the memory management unit and then the physical address is sent to the instruction cache (I-Cache). The I-cache fetches the instruction and sends it to the decode unit. In case of an invalid/illegal address, the memory management unit sets the exception bit. This bit is forwarded by the I-cache to the decode unit. If the exception bit is set, the decode unit discards the instruction and proceeds with exception handling. From a functional point of view this does not violate the ISA specification. The I-cache, however, does not take the exception into account. If the invalid/illegal address causes a cache miss, the refill is done by the I-cache controller. This allows a user-level process to refill a cache line with an inaccessible address. This may further expand the ability of the attacker to control the content of the I-Cache prior to the victim process and launch classical

Table 9.1: Computational Effort for Verifying RocketChip and Ariane

Design variant	Proof status	CPU	Mem.	k	# P-loc.
RocketChip modif. 1	detected	3 min	2.7 GB	4	
RocketChip modif. 2	detected	18 min	4.6 GB	9	
RocketChip original	detected	5 min	2.7 GB	5	
RocketChip patched	verified	5 min	2.7 GB	4	14
RocketChip 2-core	verified	7 min	3.3 GB	4	14
Ariane Original	detected	6 min	7.2 GB	2	
Ariane Patched	verified	1 min	4 GB	2	2
Ariane (page table conf.)	verified	22 min	8.8 GB	1	0

side channel attacks, even in cases where there is no page sharing between the attacker and victim. We patched the design and re-verified with it UPEC.

In an additional experiment for Ariane, we verified the confidentiality of the page table, as outlined in Sec. 7.3. Protecting physical addresses that are stored in the page table is important to limit the impact of attack scenarios such as RowHammer [119]. We verified the absence of any vulnerability that reveals the physical address to a user-level process using a modified experimental setup in which the confidential information of the miter model is an arbitrary physical page number stored in the TLB.

Tab. 9.1 shows the proof complexity of the experiments. The designs are verified using the iterative UPEC methodology, described in Sec. 6.3. In case of the detected vulnerabilities, the CPU time, memory consumption and property time window (k) of the proof that returned the corresponding L-alert is reported. All vulnerabilities were found within the *UPEC_Induction_Base* algorithm, before proceeding to *UPEC_Induction_Step*. In case of the designs verified to be secure, the reported CPU time, memory consumption and property time window (k) corresponds to the most complex proof in terms of run time within the *UPEC_Induction_Base* and *UPEC_Induction_Step* algorithms. It should be noted that in all these experiments, the most complex proof was always within the last iteration of *UPEC_Induction_Base*. For the secure designs, the last column in Tab. 9.1 shows the number of P-locations that have to be proven secure in *UPEC_Induction_Step*.

In these experiments, it took approximately 10 person-days of manual work to verify each design, which was predominantly spent in analyzing the provided counterexamples and debugging the design. Considering the complexity of the designs, the incurred manual effort is small compared to the efforts required for the functional verification of these processors.

Tab. 9.2 evaluates the effect of different factors on the proof complexity, based on the UPEC experiment on RocketChip. Both 32-bit and 64-bit versions of the design are verified and the highest proof runtime is reported, which shows

Table 9.2: Proof Complexity in Different Settings

	CPU	Mem.
32-bit datapath	5 min	2.7 GB
64-bit datapath	16 min	6 GB
UPEC w. cone-of-influence reduction	5 min	2.7 GB
UPEC w/o. cone-of-influence reduction	61 min	6 GB

the robustness of our approach w.r.t. datapath complexity. The table also shows the significant improvements that can be achieved by simplifying the prove part of the property based on cone-of-influence reduction (cf. Sec 6.2).

9.2 UPEC for Out-of-Order Pipelines

The BOOM (v2.0.1) processor is of particular interest in our experiments since it is known to be vulnerable to Spectre-style attacks while deemed secure with respect to Meltdown. It should be noted that the experiments reported in this section must not be confused with the case study reported in Sec. 8.3, which was done on BOOM v3.0.0. Due to similarities between both case studies in terms of proof complexity and solver runtimes, we report, in this section, only on the case study done on BOOM v2.0.1.

BOOM is a full-grown SoC with an out-of-order core which features a branch prediction unit with support for nested branches, virtual memory translation with a TLB, a non-blocking data cache with miss status handling registers (MSHR), a page table walker, a physical register file with dynamic register mapping and other features typically employed with out-of-order cores. BOOM's performance is comparable to ARM cores between Cortex A9 and A15, depending on its configuration. The verified BOOM design (single core and peripherals) consists of more than 650 k state bits. The only difference with the previous experiments is that the UPEC proofs were constrained using the concept of *microequivalence* (Sec. 7.1).

As a demonstration that we can focus separately on different classes of attacks, in our experiments, we decomposed the proofs into checks for Meltdown versus Spectre, i.e., as defined in Sec. 4.3, we assume that the instruction accessing the secret either has the proper privilege to do so (Spectre class) or it does not (Meltdown class). Furthermore, the secret is assumed to reside in the main memory, with the possibility of a copy in the data cache.

Tab. 9.3 has three sections, describing our experiments with the checks for the Spectre and Meltdown classes, as well as for page table confidentiality. Similar to Tab. 9.1, it shows the computational effort for the most complex proofs within *UPEC_Induction_Base* and *UPEC_Induction_Step* for our patch-and-verify flow, as described below. It should be noted that the vulnerabilities were detected by L-alerts in *UPEC_Induction_Base*. For the secure designs, the last column

Table 9.3: Computational Effort for Verifying BOOM

Check	Patch no.	Proof result	CPU (min)	Mem. (GB)	k	# P-loc.
Spectre	0	Spectre-STC v1	2	9	5	
	1	Spectre-STC v2	16	14	7	
	2	Spectre (cache)	14	14	7	
	3	secure	5	6	2	0
Meltdown	0	secure	5	6	2	3
Page table conf.	0	secure	5	5.9	1	3

Table 9.4: Required Manual Effort to Develop and Prove UPEC Property

Task	Manual effort
Microequivalence	4 person-weeks
Invariants	4 person-days
Property and miter	2 person-hours

in the table shows the number of P-locations that had to be proven secure in *UPEC_Induction_Step*.

Tab. 9.4 shows the amount of manual work required to develop and prove the UPEC property for BOOM.

For the class of Spectre attacks, our approach generated counterexamples to the UPEC property in terms of L-alerts (cf. Def. 5 in Sec. 5.3) demonstrating that the considered BOOM design is vulnerable to Spectre-STC, a so far unknown variant of Spectre, described in Sec. 8.2. We employed an iterative design procedure where we iteratively patch the vulnerability detected in a selected UPEC counterexample, and then re-verify the design using UPEC. After fixing the Spectre-STC vulnerability through a minor fix in the first iteration, UPEC identified (by L-alert) a second version of the Spectre-STC vulnerability. This version is similar to the first one except that port contention now happens on the TLB rather than on the write port to the register file. The patch-and-verify flow can be repeated until all the vulnerabilities have been removed.

In the third iteration of our flow, we picked a UPEC counterexample pointing to the original Spectre attack, which uses the cache as side channel [24]. We applied a simple and rather conservative patch for this vulnerability. Our fix for Spectre prevents speculative load instructions to execute before all preceding branch instructions are resolved (a variant of *eager-delay* approach proposed in [77]). The patched design coming out of this third iteration ("secure design variant" in Tab. 9.3) was then formally verified to be secure with respect to TES attacks. Although the fix incurs performance penalties, it still allows for

speculative execution of the majority of instructions.

In a separate experiment, we also examined the original design for the class of Meltdown attacks, as defined in Sec. 4.3. The computational effort for this experiment, which proves security of the original BOOM v2.0.1 w.r.t. Meltdown, is also listed in Tab. 9.3.

To the best of our knowledge, presently no other method is capable of solving the above RTL verification tasks. As a result of our experiments, the patched BOOM design is the first formally proven RTL model of a secure speculative execution processor.

Chapter 10

2-Safety Models in Hardware Verification

This chapter explores the application of UPEC and its underlying computational model for different verification targets. Extending UPEC for verifying confidentiality in SoCs is described in Sec. 10.1. Sec. 10.2 discusses verifying data-oblivious computation at the hardware level using UPEC. In addition, a new methodology for complete processor verification is proposed in Sec. 10.3.

10.1 Confidentiality Verification in SoCs

The emerging threat of hardware security vulnerabilities caused many microcontroller-based systems to employ low-level security and access control mechanisms to protect confidential resources from attacker-controlled firmware. In embedded systems with a more complex software stack, the concept of Trusted Execution Environments (TEEs) has been established for this purpose. Well known examples include ARM Trustzone [127] and RISC-V Keystone [128] All these approaches rely on low-level security features in the hardware, which are often referred to as the "hardware root of trust" and serve as the foundation for the trustworthiness of embedded systems.

In order to establish the trust in hardware, the functional correctness of these features has to be verified. In today's industrial practice, formal property checking is often employed to hunt for security-relevant hardware bugs. Substantial effort is required, however, to cover all security mechanisms by a set of properties. It is not sufficient to restrict this exercise to only the processor—also all peripherals must be covered. Even then, verification gaps may still remain. Since properties are usually formulated locally for individual modules, security issues related to the communication between modules or to the interaction between hardware and firmware are easily missed [37].

In this section, we describe a formal verification methodology to detect security-critical bugs in the hardware and in the hardware/firmware interface of

SoCs. This approach extends UPEC to detect all *functional* design bugs that cause confidentiality violations and to cover not only the processor but also its peripherals. The proposed methodology is particularly effective in capturing hard to detect security vulnerabilities, including the ones that are introduced based on cross-modular effects (integration and communication issues) or poorly understood hardware/firmware interaction. By merit of the UPEC flow, the proposed methodology requires neither any prior knowledge about possible vulnerabilities nor special security expertise.

This section also describes a *compositional* verification approach for hardware and firmware. It allows for developing and integrating security features at the firmware level while avoiding computationally expensive co-verification models.

10.1.1 UPEC for SoCs

The notion of Unique Program Execution as a security requirement assures that secret data is not observable by an unprivileged attacker executing a program. While this notion covers all information leakage scenarios in processor cores, it is not sufficient for complex SoCs featuring many peripheral components and accelerators. As demonstrated in our experiments, in complex SoCs, the secret can bypass the main core and leak directly to unprotected regions of data memory or memory-mapped outputs, while the program on the main core executes uniquely. A representative case that we found involves a system component featuring a memory bus master interface which can circumvent memory protection and copy secret data into unprotected memory space, or pass it on to primary outputs. In this scenario, an attacker can retrieve the secret without involving the program-visible state in the processor. A related case is the attack in [129], which used a DMA component to transfer secret data from the memory to LEDs independently of the processor.

This impacts the threat model. In addition to an adversary running a program in an unprivileged mode on the system's core, we assume an attacker to be able to monitor all primary outputs. Therefore, we extend Unique Program Execution to consider not only secret propagation to the program-visible registers as a security alarm, but also propagation to the primary outputs of the system. Note that this covers also the scenario in which the secret propagates directly, without involving the core, to unprotected memory regions (e.g., by DMA). From there, an attacker program can subsequently load and observe it in the program-visible registers of the core.

The set of state variables for which we raise an alarm (i.e., an L-alert) whenever the secret has propagated to them, i.e., architectural state variables, primary outputs and unprotected memory locations, is called the set of *observable state variables*. It should be noted that the set of observable state variables in a design is not selected based on previously known attacks but rather based on the general structure of SoCs and their deployment as embedded system in the vicinity of the attacker in a zero-trust environment.

In the UPEC verification methodology of Chapter 6, the location of the secret is limited to addresses within the data memory. Consequently, the protection

mechanisms, as assumed by the macro *secret_data_protected*, are configured to restrict access to only these specific addresses. Therefore, any guarantee provided by the proof applies only to scenarios in which secrets are stored in the assumed locations. In addition, programs have to implement the configuration as it is assumed in the proof. We emphasize that, when targeting TESs, such restrictions to the proof are legitimate. When we are only interested in side channels, we may assume that the hardware is functionally correct which implies that all addresses and all secure configurations behave in the same way w.r.t. side channels. These assumptions, however, are no longer adequate for the purposes of the approach proposed here where functional bugs are considered as the root cause for security issues in SoCs.

For this reason, we extend the set of possible locations for the secret. We include all memory locations of the system, all primary inputs as well as all architectural registers that are protected from an unprivileged user program. Instead of statically protecting single addresses, we use a *symbolic* formulation for the protected addresses and the corresponding configuration of protection mechanisms, which can be refined to any concrete implementation.

Typically, in memory protection systems the address space is partitioned into regions where all addresses share the same access rights (read/write/execute/etc.). Regions and access rights are specified as entries in a *configuration table*. We do not make any assumptions about the details of the configuration other than that it defines for each address in the address space whether it is protected or not w.r.t. confidentiality.

Definition 10 (Symbolic Protection Configuration)**.** *Let C be the set of all possible configurations, i.e., all possible values of all entries in the configuration table, and A the set of all addresses. The set $P \subseteq C \times A$ is the set of all pairs (c, a) of a configuration c and an address a that is protected under this configuration. We model P by its characteristic function $p : C \times A \mapsto \mathbb{B}$ and call it* symbolic protection configuration. □

In practice, this function $p(c, a)$ is specified in terms of the registers in the programming interface of the hardware memory protection mechanism. A reusable verification implementing the function p for the case of RISC-V Physical Memory Protection (PMP) is provided in [130].

With these concepts, the property for Unique Program Execution extended for SoCs (*UPEC-SoC*) can be formulated. The goal is to extend UPEC to prove confidentiality of a secret stored in any possible address in the address space and also explore every possible valid protection configuration. In other words, instead of assuming that the secret resides in a single protected memory location, we must prove the UPEC property under the following *equal_memory_state* constraint:

$$\begin{aligned} &\mathit{equal\_memory\_state} \Leftrightarrow \\ &\forall a, c : (\mathtt{mem}_1[a] = \mathtt{mem}_2[a] \ \vee \ p(c, a)) \end{aligned} \tag{10.1}$$

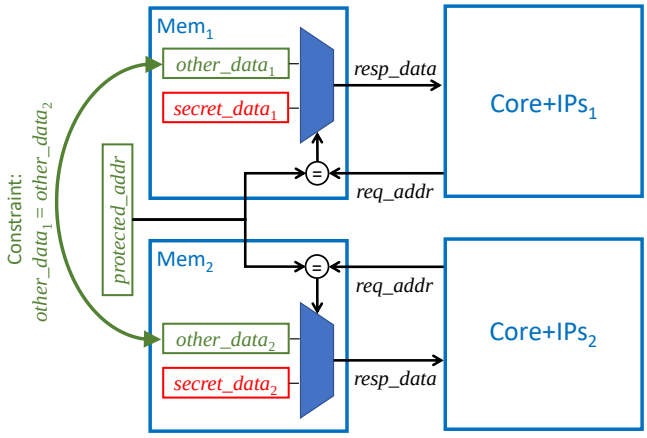


Figure 10.1: Computational model for UPEC-SoC: The memory returns the same data, except for the protected address.

Evaluating *equal_ memory_ state* calls for an enumeration of the SoC's entire address space. The complexity associated with this may seem like a show stopper for proving the presented property. However, we explain in the following that this extension to UPEC can be handled without a blow-up of complexity.

Similar to the UPEC proof methodology of Chapter 5, we run UPEC-SoC on a bounded model. We derive the bounded model from an extension to the UPEC computational model, as shown in Fig. 10.1. Instead of explicitly enumerating all addresses in the property, we introduce the address of the secret as a new state variable, *protected_ addr*, to the computational model.

The content of the SoC's memory in Fig. 10.1 is constrained to be equal in both instances for all addresses except the one specified by *protected_ addr*. Note that it is sufficient to consider only one word of secret data by *protected_ addr*. All other locations are treated symbolically (but, equally, in instance 1 and 2) and may assume arbitrary protection status. If the proof succeeds, i.e., it is not possible to leak a single word (at any value of *protected_ addr*), then also larger blocks of secret data composed of several words cannot be leaked.

While this solution still implicitly enumerates all addresses, it allows us to leverage synergies between our model and the characteristics of a typical SAT solver: During the proof procedure, the SAT solver learns that only differences between both design instances can result in a security alarm. And, consequently, it considers only those addresses that can cause such differences. This greatly reduces the complexity of our proof. From the solver's point of view, it is only one more register in the system that is functionally constrained by its surrounding components.

Fig. 10.2 shows the extended version of UPEC base property for UPEC-SoC. The property must be employed in the UPEC_Base flow and is proven on an unrolling of the computational model in Fig. 10.1. In this property, *protection_ config_ regs* is the set of registers responsible for configuring the hardware

UPEC_Base (P, k, blocking_clause):
  assume:
    at t: $micro\_soc\_state_1 = micro\_soc\_state_2$;
    at t: $p(protection\_config\_regs, protected\_addr)$;
    at $t+k-1$: $\Lambda_P = \mathit{true}$;
    at $t+k$: blocking_clause = true;
  prove:
    at $t+k$: $Alert\_candidate_{P_1} = Alert\_candidate_{P_2}$;

Figure 10.2: UPEC base property extended for UPEC-SoC.

protection mechanism. The property and the proof for UPEC_Step must be extended in a similar way.

By exploring all possible values for *protected_addr* in the *any-state proof* and, consequently, considering all locations for the secret, the proof is complete for all confidentiality violations, provided that the firmware configures the memory protection mechanism with a configuration that complies with the protection configuration p. This motivates the compositional approach of the next section.

10.1.2 Compositional Hardware/Firmware Verification

Counterexamples to our property point to confidentiality violations. These security violations are caused either by hardware bugs, which can be fixed directly in the RTL model, or by some unspecified behavior in the hardware/firmware interface. Although these violations could also be addressed at the RTL, e.g., by introducing new hardware security features to the system, fixing them in the firmware is often preferable.

Therefore, we propose a *compositional* verification flow. In this flow, UPEC-SoC exhaustively verifies the hardware and captures all behaviors in the hardware/-firmware interface that can cause security violations. This information is then passed on to the firmware development and verification process in the form of constraints that specify the security-critical aspects of the hardware/firmware interface and disallow the captured security-violating behaviors.

The flow is illustrated in Fig. 10.3. The central reasoning on the RTL model is done by the UPEC-SoC proof. If the UPEC-SoC procedure succeeds, the design is proven to be secure w.r.t. confidentiality violations. In case of a violation, the counterexample has to be inspected to determine whether it indicates a hardware bug or an issue to be fixed in firmware. In case of a hardware bug, the RTL design must be patched and then be re-verified using UPEC-SoC in the next iteration. For firmware issues, two steps are taken: First, formal constraints on the hardware/firmware interface are created based on the captured security violation. Second, the UPEC-SoC proof is re-run under the newly created constraints, i.e., it now considers only firmware complying with these constraints and excludes the problematic firmware behavior.

The constraints for the firmware are then translated into standard C assertions

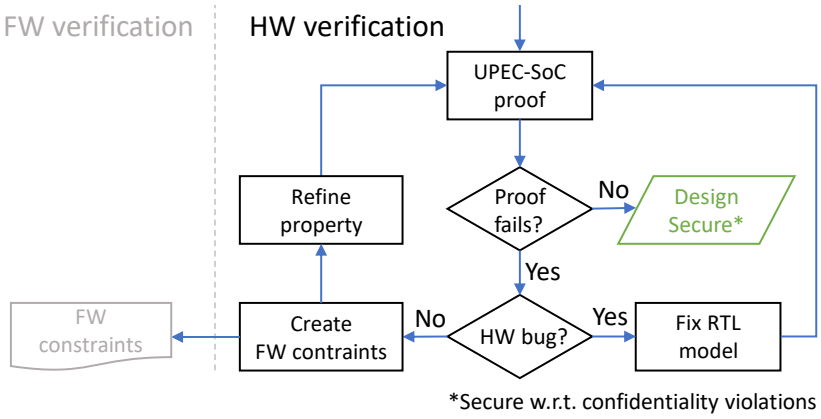


Figure 10.3: Compositional flow

which only depend on registers and addresses in the programmer's interface and are therefore straightforward to be checked in firmware. This makes the flow compatible with most firmware development and verification procedures.

Verifying an RTL design using the UPEC-SoC flow captures the security-critical behaviors of the RTL and makes them visible at the ISA level. Consequently, as long as the firmware complies with the constrained hardware/firmware interface, security at the ISA level automatically leads to security of firmware execution on the RTL implementation. This relieves the firmware developer from complex co-verification of firmware and hardware and enables the use of conventional ISA-level firmware verification tools to verify security objectives of the system without leaving a gap.

10.1.3 Experiments

The effectiveness of the proposed methodology is demonstrated on Pulpissimo [131] which is a RISC-V-based SoC design. All experiments were conducted using the commercial property checker OneSpin 360 DV on an Intel i7-8700 @ 3,20 GHz with 64 GB RAM, running Linux.

Pulpissimo is a microcontroller-style SoC featuring a large number of peripherals and I/O capabilities including SPI, I2S, Camera Interface (CPI), I2C, UART and JTAG.

The configuration we use includes the single-issue 4-stage pipeline core RI5CY which implements PMP. Two components in the design turned out to be of special interest in our experiments: the Micro Direct Memory Access (uDMA), a DMA for interfacing with peripheral components, and a Hardware Processing Engine (HWPE) which is a generic hardware accelerator performing multiply-accumulate operations on data streams. Both components have a memory bus master interface for accessing the memory independently of the core and they have a slave interface to the peripheral bus to be controlled by the software.

Table 10.1: Results for Experiments on Pulpissimo v.4

Vulnerability	Vulnerability type	Fix
PMP priorities	ISA incompliance	RTL fix
PMP address ranges	ISA incompliance	RTL fix
HWPE	circumventing PMP	Firmware constraints
uDMA	circumventing PMP	Firmware constraints
Address duplication	missing documentation	documentation/RTL fix

We verified security w.r.t. confidentiality by applying the iterative flow of Sec. 10.1.2 to the entire SoC. As a result, five major security violations were detected. The results are summarized in Tab. 10.1.

Two of the detected vulnerabilities come from hardware bugs in the PMP producing behavior which is not compliant with the ISA specification.

The next encountered vulnerability relates to the HWPE accelerator in the SoC. A program that uses the HWPE provides a set of input and output pointers. Then, the accelerator loads its inputs from the memory, computes the results and stores them wherever the output pointers point to. The load and store operations are not restricted by the protection mechanisms in the core (PMP). Our methodology discovered a scenario in which an attacker can circumvent the memory protection in the core by instructing the HWPE to read its inputs from a protected region and write its outputs to an unprotected area. As a countermeasure, we created FW constraints and restricted the proof to limit the HWPE's access to only unprotected memory addresses. For systems running untrusted user-level software we envision a setup which blocks direct user-level access to the accelerator and instead provides system calls that ensure that the input and output pointers stay within the unprotected address regions. These system functions can be checked for correct access handling using the created firmware constraints.

We detected a similar vulnerability involving the uDMA. A user-level program can trigger leakage of confidential data by instructing the uDMA to read the secret from the memory and pass it on to attacker-controlled I/O ports. This issue was also resolved by creating firmware constraints.

Another vulnerability is caused by the fact that the system memory is mapped twice into the address space of the core's data memory interface. Although this optimizes the memory bus performance, it further complicates the PMP setup. In addition, the lack of documentation for this feature may lead to a false configuration of the PMP. This issue has also been reported to the design team. It can be fixed by an RTL patch or by updating the documentation.

We also observed security alarms indicating that memory contents can be read by the debugging interfaces in the SoC. We did not include them in our list of security issues, since in a real application the interfaces would most likely be excluded from the design or would receive further protection, e.g., password protection.

After resolving all issues, UPEC-SoC certifies security w.r.t. confidentiality under the condition of firmware compliance with the formal constraints specified for the hardware/firmware interface.

10.2 Verifying Hardware Root-of-Trust for Data-Oblivious Computing

The importance of preventing microarchitectural timing side channels in security-critical applications has surged immensely over the last several years. Constant-time programming [132–134] has emerged as a best-practice technique to prevent leaking out secret information through timing. It builds on the assumption that certain basic machine instructions execute timing-independently w.r.t. their input data. However, whether an instruction fulfills this data-independent timing criterion varies strongly from architecture to architecture. Subtle optimizations of the microarchitecture can render these assumptions invalid.

In this section, we propose a UPEC-based methodology to formally verify data-oblivious behavior in hardware using standard property checking techniques. Each successfully verified instruction/operation represents a trusted hardware primitive for developing data-oblivious algorithms. A counterexample, on the other hand, represents a restriction that must be communicated to the software developer. We evaluate the proposed methodology in multiple case studies, ranging from small arithmetic units to medium-sized processors. One case study uncovered a data-dependent timing violation in the extensively verified and highly secure Ibex RISC-V core [135].

In the following, we propose a methodology for verifying data-independent execution timing, first for individual functional units (FUs), then for processor cores.

10.2.1 UPEC-DIT for Functional Units

In this section, the goal is to modify UPEC to prove that the hardware operates independently of data. Since the proposed methodology is used to prove *data-independent timing*, we will further refer to it as *UPEC-DIT*.

We focus on verifying data-obliviousness for individual FUs, e.g., cryptographic units. We model such an FU as a standard finite state machine (FSM) of Mealy type, $M = (I, S, S_0, O, \delta, \lambda)$, with finite sets of input symbols I, output symbols O, states S, initial states $S_0 \subseteq S$, transition function $\delta : S \times I \mapsto S$ and output function $\lambda : S \times I \mapsto O$. The sets I, O and S are encoded in (binary-valued) input variables X, output variables Y and state variables Z, respectively.

During a timing side channel attack, the attacker measures the operation time in order to deduce confidential information. The operation time of an FU can be measured by observing its control interface, e.g., its interaction with a bus. The basic idea of UPEC-DIT is to prove that this control interface which is responsible for timing does not depend on data. In our threat model, an attacker

can observe all control outputs of the device but not its data outputs, i.e., the attacker sees when an operation or instruction finishes, but not the result of the computation. This threat model is conservative in the sense that it covers also all indirect observations of control variables by timing measurements in the environment of the FU, as, e.g., by measuring instruction execution time in the pipeline.

For our analysis we, therefore, partition the output variables Y into control outputs Y_C and data outputs Y_D such that $Y = Y_C \cup Y_D$ and $Y_C \cap Y_D = \emptyset$. We do the same with the inputs, i.e., $X = X_C \cup X_D$ and $X_C \cap X_D = \emptyset$. For example, in a hardware multiplier, X_D comprises the operands and X_C represents handshaking and operation mode signals. Similarly, Y_D covers computation results while Y_C represents handshaking and operation flags. Partitioning the I/O variables is a manual step in our methodology (the only one) and can be easily performed with the help of the specification. Note that also related approaches require manual action, e.g., annotations of the RTL design code [136]. Dividing I/O signals into data and control splits the encoded I/O values (symbols) into data and control components: it is $I \subseteq I_C \times I_D$ with control inputs I_C and data operands I_D, and $O \subseteq O_C \times O_D$ with control outputs O_C and computation results O_D. Correspondingly, the output function λ is split into $\lambda_D : S \times I \mapsto O_D$ and $\lambda_C : S \times I \mapsto O_C$.

Definition 11 (Data-Independent Operation)**.** *A finite state machine M operates data-independently if, for two instances M_1 and M_2 of M, an* arbitrary *time point t, a maximum latency k, an arbitrary starting state $S_1^t = S_2^t = S^t \in S$, a sequence of control inputs I_C^τ, and sequences of arbitrary data inputs $I_{D,1}^\tau$ and $I_{D,2}^\tau$:*

$$\forall \tau \in [t, t+k] : \lambda_C(S_1^\tau, I_C^\tau, I_{D,1}^\tau) = \lambda_C(S_2^\tau, I_C^\tau, I_{D,2}^\tau)$$

The definition verifies whether there exists any sequence of data inputs to M that can affect the attacker-visible control outputs of M. Note that in this definition, the same control inputs are applied to each instance of M, however the data inputs may differ. We choose k to be greater than or equal to the maximum latency of the FU so that any timing variation during the full duration of the operation is detected. This latency can be taken from the FU's specification. We now map this definition to a formal interval property which is proven on the RTL design.

Fig. 10.4 depicts the computational model used in the approach. Similar to UPEC, all formal properties are proven on a *2-safety model* which instantiates the DUV twice. Instead of proving that the DUV produces specified values at specific time points, as in conventional functional property checking, UPEC-DIT searches for discrepancies between the two instances. Whether the computed values are correct according to a specification does not matter. This is key for the scalability of the approach and allows us to run UPEC-DIT without a formalization of the ISA model.

Figure 10.4: UPEC-DIT for functional units

assume:
  at t: $Z_1 == Z_2$;
  during $t..t+k$: $X^{pub}_{D,1} == X^{pub}_{D,2}$;
prove:
  during $t..t+k$: $Y_{C,1} == Y_{C,2}$;

Figure 10.5: UPEC property for a functional unit

Fig. 10.5 shows the interval property to be proven on the computational model. In the shown pseudo-code, X, Y and Z refer to vectors of input, output and state variables, respectively, of the two design instances. Note that the control inputs X_C are already constrained by the 2-safety model itself (Fig. 10.4).

A key benefit of UPEC-DIT is its flexibility in creating proofs tailored to application-specific security requirements. In practice, not all data inputs are relevant for verifying data-oblivious behavior. For example, the timing of an encryption unit might allow for a variability w.r.t. the plain text, but not the secret key. In this case, the verification engineer can exclude specific inputs to restrict the verification space. This is reflected by an optional assumption, $X^{pub}_{D,1} == X^{pub}_{D,2}$, which ensures equality on *public* data. During diagnosis of counterexamples, the verification engineer can run the analysis at even finer granularity, by running a separate property check for each confidential X^{secret}_{D} individually while assuming equality on all other X_D.

10.2.2 UPEC-DIT for Processor Cores

Modern processors implement complex microarchitectures with pipelining and, often, several FUs. We extend the idea from Sec. 10.2.1 by considering the execution of a single instruction in a processor pipeline from instruction decode to write-back. By verifying the instructions of an ISA individually (or in groups expected to have the same timing behavior), we create a set of formally verified hardware primitives for data-oblivious programming at a fine level of granularity.

A key observation is that an individual instruction does not need to execute in "constant time", but, rather, *independently of its operands*. In data-oblivious

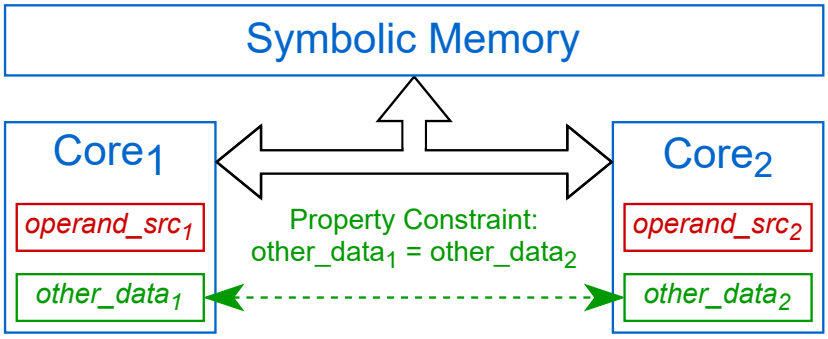


Figure 10.6: UPEC-DIT for processor cores

programming, the program itself is considered public information. This means that, for example, any timing variation originating from a different immediate field is acceptable and does not compromise data-obliviousness of the hardware. As an even more compelling example, consider a Read-After-Write (RAW) hazard in a pipelined processor causing a stall. The resulting change in instruction timing is due to the specific set of registers used by the instruction sequence in the pipeline, which is considered public information.

On the other hand, any variation in timing which manifests itself as a dependency on the *contents* of registers or forwarding buffers needs to be detected. Well known such examples are *fast-paths* in integer multiplication or division, or long execution times in subnormal floating-point operations. Similarly, more elaborate optimization mechanisms in modern processors, such as *computation reuse* or *value prediction* [137] may lead to data-dependent behavior. The goal of UPEC-DIT is to systematically uncover such variations, without producing false counterexamples caused by permissible timing variations.

In order to prove data independence of an individual instruction, we first need to define its lifetime. Generally speaking, it begins when the instruction is being fetched and ends when the instruction leaves the processor. While the starting point is marked by the first pipeline stage, there are multiple scenarios for the end of an instruction's execution. In most cases, an instruction lives until it has reached the last stage in the pipeline, usually called *Write-Back (WB)* or *Commit* stage. However, a pipeline flush, caused by an older instruction, can lead to an early squash of the *instruction under verification (IUV)*. In order to avoid false counterexamples that result from uncommitted instructions, UPEC-DIT considers only those timing variations that are caused by the particular IUV, as elaborated below.

Fig. 10.6 depicts the 2-safety computational model for cores. Memory is modeled as an abstract, "symbolic" memory that provides arbitrary but equal responses to the memory ports of the two core instances. The operand sources are unconstrained so that they can induce timing variations. Similar to the original UPEC approach, the proof begins in an arbitrary but equal (symbolic) starting

state, as the green arrow in Fig. 10.6 insinuates.

```
assume:
    at t:                  State_Equivalence();
    at t:                  IUV_in_Decode_Stage(type);
prove:
    during t..t_wb:        No_Control_Discrepancy();
```

Figure 10.7: UPEC-DIT property template for processor cores

We can now detail the UPEC-DIT property, as shown in Fig. 10.7. The notation introduced in Sec. 10.2.1 is extended to split state variables Z into control variables Z_C and data variables Z_D, such that $Z = Z_C \cup Z_D$ and $Z_C \cap Z_D = \emptyset$.

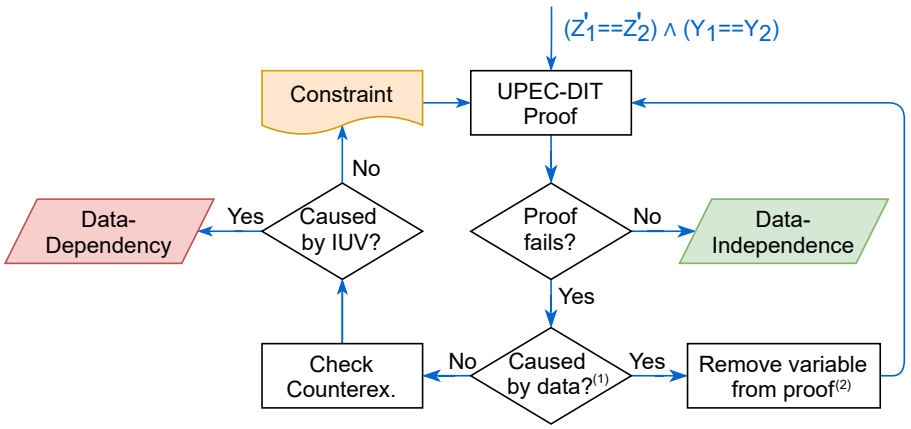
As a first step, we need to define a time point, t_{wb}, which marks the time when the IUV is in the last pipeline stage. It is determined dynamically as an *SVA sequence* based on logic pipeline signals. If the IUV does not reach the end of the pipeline, e.g., due to a flush, t_{wb} is set to the commit time of the next younger instruction. How to describe this time point varies between processor implementations. In most architectures, instruction identifiers, such as the corresponding program counter, are carried along the pipeline and can be used to identify when instructions commit.

When specifying the UPEC-DIT property, the user needs to inspect the ISA to determine all operands to the instructions. For these operands the user needs to identify the corresponding RTL state variables $Z_{op} \subseteq Z_D$ in the design. By proving data independence of execution times with respect to Z_{op} we establish the data-obliviousness of the ISA for the proven instructions. In all examined cores, Z_{op} is easy to identify and is simply given by the register file and the forwarding buffers. We now explain the elements of the UPEC-DIT property of Fig. 10.7 in more detail.

State_Equivalence() denotes a macro that constrains every state variable in $Z' = Z \setminus Z_{op}$ to equal values in the two instances, i.e., $Z'_1 == Z'_2$. This macro can be generated fully automatically exploiting features of commercial tools to identify all state-holding elements.

IUV_in_Decode_Stage(*type*) assumes that the IUV is being decoded at time point t. The parameter *type* specifies a group of instructions, e.g., R-type arithmetic instructions, expected to have equal timing behavior. If needed, also individual instructions can be considered for the proof. We begin the observation of instruction behavior in the Instruction Decode (ID) stage, as the Instruction Fetch (IF) stage does not contribute to data-dependent behavior.

No_Control_Discrepancy() in the proof statement extends the prove part of Fig. 10.5. Instead of only proving equality of the control outputs, Y_C, we also consider all control signals, Z_C, present inside the design, i.e., we check that $(Z_{C,1} == Z_{C,2}) \wedge (Y_{C,1} == Y_{C,2})$. This ensures the soundness of the overall approach. If the IUV causes a timing discrepancy, i.e., a difference in the control



(1) The differing output $y_i \in Y$ or state $z_i \in Z'$ is considered data Y_D or Z_D, respectively. **(2)** The proof target is weakened, such that $Y^{next} = Y^{prev} \setminus \{y_i\}$ or $Z^{next} = Z^{prev} \setminus \{z_i\}$.

Figure 10.8: UPEC-DIT flow for processors

flow of the system, UPEC-DIT will find it. The starting state of the proof is *symbolic*, i.e., it implicitly covers all possible pipeline contexts of the IUV. Also, because the pipeline state is equal at the start of the proof, any timing variation caused by the (public) sequence of instructions, e.g., RAW hazards, do not cause the proof to fail.

In practice, almost all timing variations can be found by only considering Y_C and a subset of Z_C responsible for pipeline stalls and flushes. However, for guaranteeing an exhaustive verification, *all* control signals Z_C need to be covered. In Fig. 10.8, we propose a systematic approach avoiding any gaps in the setup of the property. In the beginning of our procedure, we formulate the proof target as $(Z'_1 == Z'_2) \wedge (Y_1 == Y_2)$. After receiving a counterexample, the verification engineer analyzes the differing signals and decides whether the detected timing variation is acceptable or not. If the corresponding signal is not considered to be control-related, i.e., it is considered unobservable for the attacker in the given threat model, it is removed from the proof obligation in further iterations. An internal pipeline buffer propagating data is a typical example for causing this situation. In the other case, the counterexample is investigated.

In some cases it is not the IUV that causes a timing violation leading to a counterexample, but some other instruction in the pipeline, e.g., when a younger instruction receives data forwarded by the IUV. Such a counterexample should be ignored because it is also produced when checking the property for the other instruction. We avoid such a redundant counterexample by adding a constraint to the property, as shown in Fig. 10.8, such that this scenario is excluded.

This procedure is continued until either the proof holds or a data-dependent timing is shown in the counterexample.

It is important to note that our approach does not miss any timing variations

Table 10.2: Results for Functional Units

Design	DIT	#States	Time (s)	Mem (MB)
BasicRSA-T100	✗	532	<1	589
SHA1	✓	911	<1	306
SHA256	✓	1103	<1	296
SHA512	✓	2162	<1	329
AES1	✓	2472	4	994
AES2	✓	554	<1	819
FWRISCV M/D/S-Unit	⚠	331	<1	596
ZipCPU Div-Unit	✗	142	11	1347
CVA6 Div-Unit	✗	209	<1	580

For each entry, we list whether the FU has data-independent timing (DIT), number of state bits, total proof time and peak memory usage.

due to the interdependence between the IUV and older instructions. Interdependence through data is covered by completely capturing all operand sources, Z_{op}, of the IUV. Interdependence by control is covered by detecting the control variation of the older instructions, using the flow of Fig. 10.8. Such older instructions must be either repaired or avoided in data-oblivious programs.

UPEC-DIT can soundly black-box any component by treating the cut signals at the interface to the black-boxed components as pseudo-inputs/outputs, considering them in the proof. If a discrepancy can travel to one of these signals, UPEC-DIT will return a counterexample and the verification engineer can decide to investigate this suspicious information flow by unblackboxing the corresponding component.

The verification engineer does not require extensive knowledge about timing attacks or microarchitectural optimizations. Counterexamples can be traced back to the IUV simply by following the diverging paths in the two instances. This guides the engineer systematically towards possible security violations.

10.2.3 Experiments

In this chapter, we evaluate the proposed methodology on various designs. All experiments were performed using the commercial property checker OneSpin 360 DV on an Intel Core i7-6700 @ 3,4 GHz with 32 GB of RAM.

Functional Units

Tab. 10.2 shows the results for FUs. The first design, *BasicRSA-T100*, is a benchmark taken from *Trust-Hub* [138]. It implements RSA encryption and is meant to be a demonstrator for hardware trojans. Using UPEC-DIT we detected an unmentioned timing side channel in the design. The FU computes the modular

exponentiation needed for the encryption algorithm in a *square-and-multiply* fashion. While this approach is very efficient, it makes the encryption time directly dependent on the size of the secret key.

SHA1, *SHA256* and *SHA512* [139] implement different variants of the *Secure Hash Algorithm*. *AES1* and *AES2* are different implementations of the *Advanced Encryption Standard*. All of these FUs were proven by UPEC-DIT to have data-independent timing.

The *Multiplication-Division-Shifting-Unit* is part of the *Featherweight RISC-V* project [140]. Its goal is to build a resource-efficient implementation for FPGAs. All of its operations take multiple clock cycles, shifting only one bit in each cycle. In this FU, multiplication and division were proven to execute data-independently. However, UPEC-DIT produced a counterexample for shift operations, as the timing is directly dependent on the shift amount. This can be dangerous because shift operations are considered data-independent in almost all applications.

The last two designs are *Serial Division Units* taken from the *ZipCPU* [141] and *CVA6* [142] open-source projects. Both FUs showed various data-dependent timing variations. *ZipCPU* implements an early termination when dividing by zero. Also, if the signs of both operands are different, the division takes an additional cycle. The *CVA6* FU implements certain optimizations for hardware division which entail that the division time is determined by the difference in leading zeros of the operands. This creates a data dependency.

In all case studies, UPEC-DIT was applied without a-priori knowledge of the designs. It systematically guided the user to the points of interest. Even though the designs had operations taking up to 192 cycles, proof time and memory requirements remained insignificant.

Processors

We investigated four different open-source RISC-V processors from low to medium complexity, as shown in Tab. 10.3. The *Ibex* processor is listed twice, as it comes with a *data-independent timing (DIT)* security feature, which we examined separately.

The first design we investigated is the sequential *Featherweight RISC-V* processor [140] which aims at balancing performance with FPGA resource utilization. As our results show, most instructions execute independently of their input data. However, there was one big exception, namely, R-Type shift instructions. The shifting unit can only shift one bit in each cycle (cf. 10.2.3), which results in data-dependent timing depending on the shift amount (rs2). We singled out the shift instructions in a separate proof and showed that other R-Type instructions like addition do, in fact, preserve data-independent timing. I-Type shift instructions also execute with dependence on the shift amount. The shift amount, however, is specified in the (public) immediate field of the instruction. Load, Store and Jump (JALR) can cause an exception in case of a misaligned address, while Branches incur a penalty if a branch is not taken.

The *Ibex RISC-V Core* [135] is an extensively verified, production-quality open-source 32-bit RISC-V CPU. It is maintained by the *lowRISC* not-for-profit

Table 10.3: Results for Processors

	FWRISCV	**IBEX**	**IBEX (DIT)**	**SCARV**	**CVA6**
I-Type	✓	✓	✓	✓	✓
R-Type	✓/✗	✓/✓	✓/✓	✓/✓	✓/✓
Mult	✓/✓	✓/✗	✓/✓	✓/✓	✓/✓
Div	✓/✓	✓/✗	✓/✓	✓/✓	✗/✗
Load	⚠	⚠	⚠	✗	✗
Store	⚠/✓	⚠/✓	⚠/✓	✗/✗	✗/✓
Jump	⚠	✓	✓	⚠	✗
Branch	✗/✗	✗/✗	✓/✓	✗/✗	✗/✗
#States	3086	1019	1021	2334	682849
AT	3 sec	2 min	4 min	3 min	1.5 hour
WCT	4 sec	5 min	6 min	8 min	3 hour
Mem	1.7 GB	4.5 GB	4.3 GB	2.1 GB	11.9 GB

In each experiment (separate proof), UPEC-DIT proved that the instruction class executes data-independently either always (✓), only under certain software restrictions (⚠), or not at all, i.e, it depends on its operands (✗). Multi-operand instructions denote rs1/rs2 separately, as they can have different impact on timing. We also report the number of state bits in the original design, the *average time (AT)* and *worst-case time (WCT)* of a single proof as well as the peak memory requirements (GB).

company and deployed in the OpenTitan platform. It is highly configurable and comes with a variety of security features, including a *data-independent timing (DIT)* mode. When activating this mode during runtime, execution times of all instructions are supposed to be independent of input data. In our experiments, we apply UPEC-DIT for both inactive and active DIT mode and use the default *"small"* configuration, with the *"slow"* option for multiplication.

When the DIT mode is turned off, we found three cases of data-dependent execution time: Division and (slow) multiplication implement fast paths for certain scenarios. Taken branches cause a timing penalty, as the prefetch buffer has to be flushed. Misaligned loads and stores are split into two aligned memory accesses.

The first two issues are solved when DIT mode is active, as seen in Tab. 10.3. However, the timing violation for misaligned memory accesses is not addressed.

When running Ibex in DIT mode, data-oblivious memory accesses require special measures, such as the integration of the core with a data-oblivious memory sub-system. For example, an oblivious RAM controller [143] makes any memory access pattern computationally indistinguishable from any other access pattern of the same length. However, our experiments with UPEC-DIT reveal that even with such strong countermeasures in place, Ibex still suffers from a side channel in the case of memory accesses that are misaligned. This is because the core creates a different number of memory requests for aligned and misaligned accesses. We reported this issue to the lowRISC team and suggested to disable the misaligned access feature for DIT mode. With this fix, the hardware would remain secure even in case that a faulty/malicious software introduces a misaligned access. The lowRISC team refined the documentation and will consider the proposed fix for future updates of the core.

The *SCARV* [144] is a 5-stage single-issue in-order CPU core, implementing the RISC-V RV32IMC instruction sets. We run UPEC-DIT on the core to prove that most instructions do not leak information through timing. Branches, however, do take longer if they are taken due to a pipeline flush. Memory accesses can cause exceptions if they are misaligned. Furthermore, any loads and stores to a specific address region are interpreted as memory-mapped I/O accesses which do not issue a memory request.

CVA6 [142] is a 6-stage, single-issue 64-bit CPU. Besides a data-dependent division and timing variations by mispredicted branches UPEC-DIT also found an interesting case in which a load can be delayed. To prevent RAW hazards, whenever a load is issued, the store buffer is checked for any outstanding stores to the same offset. If any exist, the load is, conservatively, stalled until the stores have been committed. However, this can cause a timing delay in case of a matching offset, even if both memory accesses go to different addresses.

Tab. 10.3 also shows that time and memory requirements are moderate, even in the case of a medium-sized processor.

10.3 Gap-free Processor Verification based on 2-Safety Models

The required manual effort and verification expertise are among the main hurdles for adopting formal verification in processor design flows. Developing a set of properties that fully covers all instruction behaviors is a laborious and challenging task. Although there exist techniques that propose complete formal verification of processor cores [43, 145], they require high manual effort and are hard to integrate in many industrial verification flows. Other approaches [146]which generate properties and verification IPs from an executable specification of the ISA, also fail to fully cover corner cases of instruction execution due to relying on bounded proofs.

In this section, we propose a systematic and "complete" processor verification approach based on a 2-safety verification model which requires considerably less manual effort and expertise compared to the state of the art. The approach is based on Symbolic initial state Symbolic Quick Error Detection (S^2QED) [1], which originally covered a subset of design bugs in processor pipelines. The approach presented here augments S^2QED with additional checks to reach a complete processor verification.

The properties in our approach are simple and can be automatically generated from an ISA model with small manual effort. Furthermore, unlike in conventional property checking, the verification engineer does not need to explicitly specify the processor's behavior in different special scenarios, such as stalling, exception, or speculation, since these scenarios are taken care of implicitly by the 2-safety computational model. The great promise of the approach is shown by an industrial case study with a 5-stage RISC-V processor.

10.3.1 Background: Complete Property Set

In Interval Property Checking (IPC), the design behaviors can be specified in a special property format called *operation properties* [43, 111]. Each operation property describes a certain design behavior in a finite time interval in which the design starts and ends in a so called *important state* or *conceptual state*. An operation is defined as a set of finite sequences of state transitions between two important states such that only unimportant states are visited in between. An important state is an abstract state and corresponds to one or more concrete states. Each concrete state can only belong to one important state. Operation properties are supposed to be chained in the sense that the end state of one operation is the start state of the succeeding operation. These operations (transitions between conceptual states) can be viewed as forming a conceptual state machine (CSM). The CSM is a finite automaton describing the sequencing of operations and is close to the specification. The goal is to completely describe every input/output behavior by a sequence of operation properties without any gaps [43, 111]. Interval Property Checking fulfilling the following completeness criterion is called *Complete Interval Property Checking (C-IPC)*.

Definition 12. *(Complete Property Set): A property set is complete w.r.t. a Design Under Verification (DUV), if the sequence of output signal values over a period of time is uniquely defined by the property set according to a set of* determination requirements*. The* determination requirements *are specifications of signal behavior in the design, describing which output and state variables of the design are to be uniquely determined in each operation [43, 111].* □

The determination requirements define the input/output space considered by the property set.

Corollary 1. *A property set $V = \{P_1, P_2, \ldots, P_n\}$ is complete if two arbitrary state machines satisfying all properties in the set are sequentially equivalent with respect to the determination requirements.* □

The verification engineer makes a property set complete by ensuring that every possible operation is covered by an operation property and that all outputs and other signals referred to in the determination requirements are uniquely specified at every time point by the operation properties. Completeness of a set of properties can be checked automatically, and independently of a design. A complex sequential equivalence check, as in [147], is not needed. Instead, completeness can be established inductively by considering all pairs (P_i, P_j) of properties describing an operation P_i and a direct successor operation P_j and by performing a set of *completeness tests* [43, 111] described below.

A) **Case Split Test**: The case split test checks that all paths between the important states are covered by at least one property. In other words, it checks that at the ending important state of each operation, for every possible input combination, there exists at least one operation property which determines the next important state. This ensures that there is no input scenario missed in the property set.

B) **Successor Test**: The successor test checks for every operation whether the successor operation is uniquely determined. For every pair of predecessor/successor operations (P_i, P_j), the execution of P_j must be uniquely determined by the predecessor P_i. Passing successor and case split tests ensures that there exists a unique chain of operations for every input trace.

C) **Determination Test**: The determination test checks whether a set of operation properties uniquely determine the outputs of a circuit (or other signals in determination requirements, e.g., general purpose registers in processors) at all time points.

D) **Reset Test**: The above three tests form an inductive proof stating that if an operation determines its ending important state and output, then there always exists a successor operation that uniquely determines the next important state and output. The validity of this inductive reasoning depends on the reset state (i.e., the induction base). The reset test checks whether the reset input sequence initializes the system deterministically to a unique important state and fulfills all determination requirements.

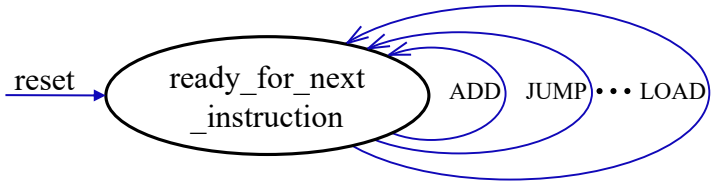


Figure 10.9: Conceptual state machine of a pipelined processor

Complete Processor Verification

Pipelined processors execute several operations simultaneously such that different operations overlap each other by one or more clock cycles. Logically, the executions of instructions are independent of each other, even though, physically, they overlap each other in time as they pass through the pipeline. Based on this insight, operation properties can be written in such a fashion that the specified computations and the corresponding determination requirements overlap temporally. In this way, the operation properties provide an abstract view on instruction execution that is close to the ISA-level programmer's view on the processor.

A CSM to capture the operations in a pipeline is shown in Fig. 10.9. Each operation starts and ends in an important state called *ready_for_next_instruction*. The state transition graph of this finite state machine is of a special, "degenerate" form: it consists of a single state and many transitions beginning and ending at that state. Each transition is labeled with a different opcode and represents the execution of a specific instruction. The special structure of this state transition graph has implications for checking the completeness of the property set, as discussed for our new approach in Sec. 10.3.3.

10.3.2 S^{2}QED Processor Verification

Design errors in hardware, often called "bugs" or "logic bugs", lead to incorrect behavior of the implementation in certain scenarios.

Definition 13. *(Error Scenario): In the context of processor design, an* error *is a deviation of the implementation's behavior from its specification, in a certain* error scenario*. An error scenario consists of (1) an instruction in which the error becomes observable, (2) the instruction's operands, and, (3) its* program context*, i.e., the sequence of previously executed instructions.* □

We say, an error scenario *activates* a logic bug. Logic bugs can be categorized into *single-instruction bugs* or *multiple-instruction bugs*.

Definition 14. *(Single-instruction bug): A bug is a* single-instruction bug *if there exists (1) an instruction opcode and (2) a set of operands such that the execution of the instruction leads to an error* in all program contexts*, i.e., independently of all previously executed instructions.* □

Definition 15. *(Multiple-instruction bug): A bug is a* multiple-instruction bug *if it is not a single-instruction bug and if there exists (1) an instruction opcode, (2) a set of operands, and, (3) a program context such that the execution of the instruction leads to an error.* □

A multiple-instruction bug requires error scenarios consisting of specific instruction sequences that set up the microarchitecture of the processor such that the bug is activated. In contrast to a single-instruction bug, there are program contexts in which a multiple-instruction bug is not activated.

S²QED Computational Model

S²QED [1], originally inspired by post-silicon validation technques such as Quick Error Detection (QED) [148], is a formal processor verification approach targeting complex instruction pipelines. S²QED proves that every instruction executes independently of the previous pending instructions in the pipeline, i.e., independently of its program context. The computational model of S²QED, similar to the model employed by UPEC, consists of two identical and independent instances of the processor under verification which are constrained to execute the same instruction, at an arbitrary time point. Fig. 10.10 shows the computational model in which the two CPU instances of the same processor are unrolled for a time window as large as the upper bound of the execution time of an instruction in the pipeline.

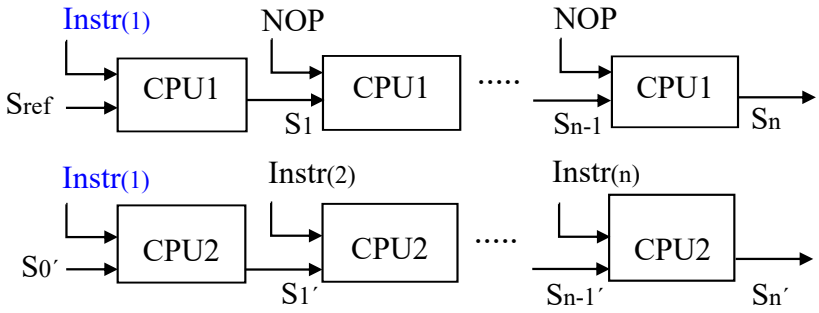


Figure 10.10: S²QED computational model

Definition 16. *(QED consistency): In the S²QED computational model, the two CPU instances are* QED-consistent *at a time point t, if the corresponding architectural state elements of both instances at time point t hold the same values.* □

For a processor with N registers a QED-consistent register state is characterized by the named logic expression:

$$qed\_consistent\_registers := \bigwedge_{i=0}^{N-1} \left(R^i_{cpu1} = R^i_{cpu2}\right) \tag{10.2}$$

assume:
  at t_{IF}: $cpu2\_fetched\_instr() = cpu1\_fetched\_instr()$;
  during $[t_{\text{IF}}+1, t_{\text{WB}}]$: $cpu1\_fetched\_instr() = \text{NOP}$;
  at t_{IF}: $cpu1\_state() = S_{\text{ref}}$;
  at t_{WB}: $qed\_consistent\_registers()$;
prove:
  at $t_{\text{WB}}+1$: $qed\_consistent\_registers()$;

Figure 10.11: Original S^{2}QED property

This expression is a Boolean predicate that is implemented as a *macro* in the property language of the verification tool. It represents architectural states in which the register files of both CPU instances have identical contents.

Consider an S^{2}QED computational model, in which R_{cpu1} and R_{cpu2} represent the general purpose register files of CPU 1 and CPU 2, respectively. Fig. 10.11 shows the S^{2}QED property that is to be proven on this model. The property specifies that if two independent CPU instances fetch the same instruction and the register files are consistent with each other before the write-back (the macro *qed_ consistent_ registers*() specifies this consistency), then the two CPU instances must be QED-consistent also after the write-back, independently of the pipeline context. The CPU 1 instance is constrained to start from a flushed-pipeline state S_{ref} and fetches only NOPs in the time frames for $t > 1$. A flushed-pipeline state S_{ref} is forced on the CPU 1 instance by letting it execute only NOPs for as many time frames before time point t as there are pipeline stages. This results in a significant reduction of proof complexity and excludes any false counterexample to the property that can result from an inconsistent pipeline register. The CPU 2 instance is left unconstrained to start from a symbolic initial state and is allowed to execute an arbitrary sequence of instructions for the time frames $t > 1$. In this computational model, the SAT solver compares the scenario 1, where the *Instruction Under Verification* (IUV) is executed in a flushed-pipeline context, with all scenarios 2 where the IUV is executed in an arbitrary context including the ones where bugs are activated and propagated.

Extending S^{2}QED for single-instruction bugs

The S^{2}QED property shown in Fig. 10.11 can detect all logic bugs resulting in a QED-inconsistent state [1]. In other words, the instruction execution is verified to be independent of the program context or previously fetched and executed instructions. As a result, the property covers multiple-instruction bugs. However, single-instruction bugs can be masked due to the fact that "common-mode" bugs like a bug in the data path of the ALU have the same effect on both CPU instances and may not lead to a QED-inconsistent state.

We extend the original S^{2}QED approach such that it detects all logic bugs in a processor including single-instruction bugs. With our new approach, called Complete S^{2}QED or C-S^{2}QED, a complete processor verification can be achieved

assume:
at t_{IF}: *cpu2_fetched_instr*() = *cpu1_fetched_instr*();
during [t_{IF}+1, t_{WB}]: *cpu1_fetched_instr*() = NOP;
at t_{IF}: *cpu1_state*() = S_{ref};
at t_{ID}: *ready_for_next_instruction*();
at t_{ID}: *instr_register_type*();
at t_{WB}: *qed_consistent_registers*();
prove:
at t_{EX}: *ready_for_next_instruction*();
at $t_{\text{WB}} + 1$: *qed_consistent_registers*();
at $t_{\text{WB}} + 1$: *cpu1_reg_value*(*reg_addr* @ t_{ID}) = *expected_value*(*funct_type* @ t_{ID});

Figure 10.12: Extended S^{2}QED property for Register-type instructions

with substantially less manual effort when compared to traditional C-IPC. This is possible since S^{2}QED "automatically" explores all possible program contexts so that only a much simplified property set is needed that covers just the single-instruction bugs.

Instead of one S^{2}QED property for the entire ISA, we develop an extended S^{2}QED property for each of the different instruction classes such as "register-type", "memory", "control flow". The extended S^{2}QED property is shown in Fig. 10.12 for register-type instructions in a 5-stage RISC processor. This is denoted by the macro *instr_register_type*() in the assumption. At time point t_{IF}, the same instruction is fetched by both CPU instances, 1 and 2. Just like in the original S^{2}QED property we assume that (i) CPU 1 starts from a flushed pipeline state, (ii) it fetches only NOPs after the time point t_{IF} and (iii) the previous instruction execution has resulted in a QED-consistent state. The macro *ready_for_next_instruction*() describes the state (cf. Fig. 10.12) of the CPU 1 and CPU 2 pipelines, when they are ready for the next instruction.

At time point t_{ID}, both pipelines begin processing the IUV. At time point t_{WB}+1, one clock cycle after the results of an instruction execution are committed, the QED consistency (macro *qed_consistent_registers*()) and the expected values of the instruction execution (macro *expected_value*(*funct_type* @ t_{ID})) are checked. These checks ensure that any logic bug in a processor is found by the new extended S^{2}QED property.

The macro *expected_value* (*funct_type* @ t_{ID}) checks the correctness of results of the instruction execution in the CPU 1 instance, for the following reason: Since the CPU 1 instance fetches NOPs before and after the time point t_{IF}, the complex instruction interleaving scenarios such as forwarding or control transfers do not need to be considered. These scenarios are, instead, covered by checking QED consistency between the CPU instances. This leads to a simplification of the property and makes it possible to generate such property set from an ISA model. Also, checking QED consistency ensures that the CPU 2 instance completes with the same, expected results. The property shown detects all logic bugs with

respect to the execution of register type instructions. Similarly, an extended S^{2}QED property is developed for each instruction class of the ISA.

10.3.3 Completeness Check for C-S^{2}QED

Since the extended S^{2}QED property set of the previous section does not concern itself with analyzing all special contexts in which an instruction can be running, the conventional completeness check of C-IPC is bound to fail. In particular, it is the case split test that fails. On the other hand, the reset test, successor test and determination test do hold for the proposed S^{2}QED version of the property set if each property uniquely determines the outputs and architectural state variables of the processor. This must be ensured by the set of macros *expected_value*(*func_type*). Since these checks can be applied to our new approach in their conventional form [111], we do not further detail them in this chapter.

Next, we present an adaptation of the case split test to the reasoning of the new formulation. We then prove that checking the extended S^{2}QED properties under the adapted completeness check is equivalent to C-IPC in terms of covering all logic bugs.

Case split test

An operation property P is a property written in the form of an implication $A \implies C$, where the antecedent A is a set of *assumptions* and the consequent C is a set of *commitments*. An assumption or a commitment is an LTL formula where the only temporal operator allowed is X. Such a formula can be mapped to a finite unrolling of a transition structure. In the following, we use the notation $next(A, l)$ to denote a "temporal shift" of the formula A by l clock cycles, i.e., $next(A, l) := \mathsf{X}^l A$. This adds a temporal offset l to all time points referred to in the formula A.

We first consider the case split test for the property suite of a general hardware design and then look at simplifications for a processor pipeline. Let a set of important states be given by the commitments $\{C_i\}$ of the properties $\{P_i\}$ for an arbitrary design. Then, for every important state (given by a commitment C_P) reached in an operation P it is checked whether the disjunction of the assumptions $\{A_{Q_j}\}$ of all successor properties Q_j completely covers the commitment C_P, i.e., for every path starting in a substate of the important state C_P there exists an operation property Q_j whose assumption A_{Q_j} describes the path. Let $\{A_{Q_1}, A_{Q_2}, \ldots\}$ be the set of assumptions of the successor properties, then the case split test checks whether

$$C_P \rightarrow next((A_{Q_1} \vee A_{Q_2} \vee A_{Q_3} \vee ....), l_P) \tag{10.3}$$

where l_P is the length of the property P, i.e., the number of clock cycles between the starting and the ending state of P. The *next* operator aligns the starting state of property Q_j with the ending state of property P_i.

In a processor pipeline, the operation properties all begin and end in the same conceptual state, referred to as *ready_for_next_instruction*() in Fig. 10.9. In our

formulation, we have grouped instructions into properties according to instruction classes, i.e., we have a total of n properties to consider when the processor core supports n instruction classes. Since there exists only a single conceptual state that is reached in every operation, we only need to verify for every property P_i that its commitment C_{P_i} is completely covered by the assumptions of the possible successor properties. Let $\{A^2_{Q_1}, A^2_{Q_2}, A^2_{Q_3}, \cdots\}$ and $\{A^1_{Q_1}, A^1_{Q_2}, A^1_{Q_3}, \cdots\}$ be the assumptions of the successor properties on the CPU 2 and CPU 1 instances, respectively. We need to prove:

$$C_{P_i} \rightarrow next(\ (\ (A^1_{Q_1} \wedge A^2_{Q_1}) \vee (A^1_{Q_2} \wedge A^2_{Q_2}) \\ \vee (A^1_{Q_3} \wedge A^2_{Q_3}) \vee \cdots), l_{P_i}) \tag{10.4}$$

Assuming that the extended S^{2}QED property set holds on the computational model, it is proven that the result of every operation is consistent between the two CPU instances, regardless of the predecessor operations. Therefore, it is sufficient to prove the case split test on one of the CPU instances. In other words, in order to pass the case split test, every possible instruction sequence must be implicitly considered on the CPU 2 instance. The case split test for S^{2}QED is reduced to:

$$C_{P_i} \rightarrow next((A^2_{Q_1} \vee A^2_{Q_2} \vee A^2_{Q_3} \vee \cdots), l_{P_i}) \tag{10.5}$$

If this test passes, it means that for every possible instruction sequence of the processor there exists a chain of properties that is executed. In other words, every execution trace is "covered" by a sequence of operation properties.

The case split test is easy to satisfy for a set of S^{2}QED properties. For a processor that implements n instruction classes, we have n S^{2}QED properties. By ensuring the correctness of the macros that capture the opcode of the instruction classes, such as *instr_ register_ type*() in Fig. 10.12, the case split can be easily proven.

Based on completeness checking with the S^{2}QED-adapted case split test, we can formulate the following theorem.

Theorem 1. *Given an extended S^{2}QED property set $V = \{P_1, P_2, \cdots, P_n\}$ in which P_i is a property for an instruction class, created for a given ISA, as described in Sec. 10.3.2. If the property set fulfills the completeness checks for S^{2}QED, then it detects all logic bugs in the ISA implementation of a processor core.*

Proof. Assume that there is a bug in the processor. If the bug is a single-instruction bug, it means that there is an erroneous instruction that produces a wrong result (even) if it is executed in a flushed pipeline. Because the property set passes the case split test and the successor test, there exists a single S^{2}QED property targeting the instruction. Since the property set passes the determination test, there exists a macro *expected_ value*(*funct_ type*) which is called by the commitment of the property and which verifies the computation result of the instruction. According to this macro, the property fails for the expected architectural state in the CPU 1 instance.

If the bug is a multiple-instruction bug, two cases have to be distinguished: (1) The bug is activated in a flushed-pipeline context. Then, the bug will be detected in the same way as described above for the single-instruction bug. (2) The bug is not activated in a flushed-pipeline context. Then, there exists another program context that activates the bug. Because the property set passes the case split test and the successor test, there exists a unique sequence of operation properties covering the program context, i.e., the sequence of instructions in the pipeline. Hence, there exists an S^{2}QED property that covers the program context in the CPU 2 instance of the model and has the instruction exposing the bug as the IUV. Because the property set passes the determination test, the macro *qed_consistent_registers*() determines the full architectural state including the state variables exposing the bug. Since the bug is not activated in the CPU 1 instance, it is detected as an inconsistency between CPU 1 and CPU 2 architectural states. □

It should be noted that the theorem is based on the assumption that the C-S^{2}QED property set fulfills the determination requirement on every clock cycle of every operation of the system. Therefore, the property set must also prove that the content of architectural registers do not change during clock cycles in which the pipeline is stalled and there is no instruction write-back.

In the following, an extended S^{2}QED property set fulfilling the completeness checks is referred to as *Complete S^{2}QED* (C-S^{2}QED).

10.3.4 Experiments

The effectiveness of C-S^{2}QED is demonstrated by verifying a RISC-V processor implemented in an industrial SoC for a safety-critical application. The RISC-V core implements a Harvard architecture and supports the RV32I base integer instruction set from the RISC-V ISA. The 5-stage pipelined core has an in-order fetch unit, a 32-bit ALU, a configurable multiply/division unit and an exception handler.

The processor core has been previously verified with a complete set of properties based on the conventional C-IPC [111, 145]. 14 logic bugs comprising both single-instruction bugs and multiple-instruction bugs were found during this verification process. In our experiments for C-S^{2}QED, all these logic bug scenarios were injected into the RTL. For property checking, we used the commercial tool OneSpin 360 DV-Verify on an Intel® Xeon® E5-2690 v3 @2.6GHz with 32 GB RAM.

All 14 logic bugs that were previously detected by the conventional C-IPC approach are also detected by C-S^{2}QED properties within 30 s of computation time. Additionally, two error scenarios were detected by the C-S^{2}QED properties. The reported errors are activated in a flushed-pipeline context, in which the failing properties identified the unnecessary stalling of the pipeline. As the CPU 1 instance is fetching NOPs before and after the time point t_{IF}, the results of the instruction (fetched at t_{IF}) execution are expected at specific time points. Because of this, any unnecessary stalls (which result in performance loss) associated with

Table 10.4: Bug Detection Results for C-S^{2}QED

Effort for base instr. set	5(person days)
Effort for new extension	1(person days)
Runtime (with bugs)	< 30 s
Runtime (without bugs)	18 min
CEX length ([min, max])	[1, 5] instructions

any instruction class are identified by the C-S^{2}QED property set.

Tab. 10.4 summarizes the results from the C-S^{2}QED experiments on verifying the RISC-V core. Rows 1 and 2 show the manual effort required to develop the verification setup and the properties for the base integer instruction set and to support the new ISA extension (e.g., mult/div instructions), respectively. "Runtime" refers to the computation time spent to detect a bug (row 3) or to prove its absence (row 4). In case of a bug being reported by the formal tool, the length of the counterexample (CEX) is also reported.

The reported experiment has shown that C-S^{2}QED can detect all logic bugs in a processor irrespective of their context in the program within a reasonable amount of time. By merit of the proposed computational model, the verification engineer is relieved from analyzing microarchitectural details of instruction execution for formulating the properties. This resulted in a significant improvement on the required manual effort for verifying the processor pipeline.

Chapter 11

Conclusion and Future Work

This thesis proposes Unique Program Execution Checking or UPEC as a formal hardware security verification technique at the register transfer level (RTL). UPEC exhaustively verifies a confidentiality requirement of the design, proving security against both explicit information leakages as well as implicit flows through TES. UPEC advances the adoption of formal methods in the field of hardware security and it is the first formal RTL verification technique that can exhaustively verify security against the class of TES vulnerabilities in microarchitectures, without relying on a priori knowledge of the user about the attacks. As a result, previously unknown TES attacks can be detected by an automated technique rather than by the clever thinking of offensive security researchers.

In this thesis, we explored various optimization techniques to maximize the scalability of the UPEC approach. By utilizing these optimizations, a well-defined verification methodology is proposed that divides the verification problem into efficient and feasible sub-problems. The proposed verification methodology employs inductive reasoning to establish the security guarantee for the design. The UPEC verification methodology is proven to be scalable even to complex out-of-order processors, such as BOOM with more than 650 k state bits.

UPEC employs SAT-based proofs on unrolled circuit models, which is which is a well-accepted formal proof technique applicable to industrial-scale designs that can be found in many commercial verification tools. Besides being tool-agnostic, UPEC has the advantage of benefiting from the power of the modern solvers, which contributes to the scalability of the proposed verification flow.

UPEC does not require changing the standard design flow or using security-driven HDLs, and, therefore, can be easily adopted by industrial design flows. It does not demand extensive security knowledge from the user and the required manual work is small compared to the total SoC design and verification effort at the RTL. The bulk of the effort is spent on analyzing the provided counterexamples and security violations, which, in the end, contributes to understanding the issue and fixing the design.

The proposed approach, due to its exhaustiveness, has promise for an efficient

solution regarding legacy hardware. By analyzing the RTL design of legacy systems the UPEC user can collect all possible attack scenarios feasible in the design. The collected information can be passed on to the software domain to enforce software fixes only for the necessary cases and relevant gadgets. Therefore, UPEC can contribute to methodologies employing contracts between hardware and software [68] such that sound compositionality can be achieved between measures at the software level and the RTL.

UPEC and the verification experiments conducted within this thesis have contributed to a better understanding of the threat of TES attacks. This thesis has shown that TES can occur also in simple processors. They do not only result from certain architectural features, such as out-of-order execution and speculative execution, but can also be inserted by low-level RTL design modifications. Such vulnerabilities may be introduced inadvertently by seemingly innocuous design decisions and, even worse, they may also be deliberately created by a malicious software/hardware provider to create an invisible "backdoor". Such a new kind of Trojan can be extremely hard to find, since it does not conform to the conventional Trojan models (trigger and payload), and, more importantly, it neither corrupts the functionality of the design nor does it add any redundant logic. With the increasing role of shared hardware and software infrastructures involving components from numerous providers as well as open-source domains, such a risk must be given appropriate attention.

UPEC and the underlying 2-safety computational model contribute to other verification targets within the processor and SoC design. In this thesis, we have described UPEC-based methodologies for exhaustively verifying SoC designs against security-critical bugs as well as for verifying data-oblivious computing hardware primitives. An approach for functional verification of processor pipelines based on a computation model similar to the UPEC computational model has also been proposed, which significantly reduces the required manual work in formal processor verification.

All in all, this thesis sheds light on the importance of hardware security sign-off verification at the RTL and takes the first steps towards a unified formal verification methodology for detecting microarchitectural security vulnerabilities. By providing formally proven security guarantees, UPEC contributes to establishing hardware as a root of trust and relieves software developers from considering microarchitectural details.

11.1 Future Work

The work of this thesis covers the class of TES attacks, but does not address the class of general side channels, such as, e.g., cache side channel vulnerabilities of encryption software. Although classical side channel vulnerabilities should be addressed at the software level, a systematic solution against them requires a hardware analysis to collect information about all possible channels. Therefore, an extension of UPEC to detect the general class of side channel vulnerabilities can contribute to preventing side channels at the software level.

Considering different threat models and security targets is another direction for future work. Verifying the security of a design at the presence of untrusted third party IPs is a relevant security target, given the complexity of modern SoCs and the need for using 3rd party IPs. Another example is verifying the security at the presence of fault injection attacks, which is a prevailing threat for many embedded system devices deployed in zero-trust environments.

Although many different hardware design patches against the Spectre attack have been proposed in the literature, a secure implementation of such a fix is still a challenging task. UPEC, as an exhaustive verification technique, can assist designers to only implement design fixes that are absolutely necessary and avoid overconservative solutions. In addition, each UPEC security alert provides a detailed scenario for a possible TES attack, which guides the designer to the right location in the design to implement a proper patch. Future research should explore the possibility of a systematic design methodology based on UPEC with the objective to achieve a formally proven secure design with minimum overhead. UPEC counterexamples can be a great asset to the designers and therefore, future works should investigate the possibility of interleaving the design and verification steps in a UPEC-based design flow to minimizing the manual effort of developing secure microarchitectures.

Summary

Hardware security issues have been emerging at an alarming rate in recent years. In particular, transient execution attacks, such as Spectre and Meltdown, pose a genuine threat to the security of modern computing systems. The recent surge in hardware vulnerabilities has also been acknowledged by the *Common Weakness Enumeration* database (CWE), which now includes a separate category for hardware vulnerabilities. Despite recent advances, understanding the intricate implications of microarchitectural design decisions on hardware security remains a great challenge and has caused a number of update cycles in the past. The hardware security verification challenge was mostly addressed by path analysis (taint analysis) techniques. Path analysis techniques check whether or not an illegal information flow can happen through a certain suspicious path between two points in the design. These techniques have pioneered information flow analysis in the hardware domain. However, they usually suffer from scalability issues when considering complex designs, limiting their applicability in practice. Furthermore, in order to select a suspicious path, the user must rely on a priori knowledge about potential vulnerabilities. As a result, these techniques may not be exhaustive in finding previously unknown vulnerabilities.

This thesis addresses the need for a new approach to hardware sign-off verification which guarantees the security of processors at the Register Transfer Level (RTL). To this end, we introduce a formal definition of security with respect to microarchitectural vulnerabilities, formulated as a hardware property. We present a formal proof methodology based on Unique Program Execution Checking (UPEC) which can be used to systematically detect all vulnerabilities to transient execution attacks in RTL designs. UPEC does not exploit any a priori knowledge on known attacks and can therefore detect also vulnerabilities based on new, so far unknown, types of channels. This is demonstrated by the new attack scenarios discovered in our experiments with UPEC. UPEC operates on a verification model consisting of two identical instances of the SoC design under verification. The SoC instances in the model execute the same program. The only difference between the two instances is the content of the protected part of the memory, i.e., the secret.

The UPEC verification methodology is based on a property that is implemented in terms of constraints and equality checks on the two SoC instances. The basic idea is to exhaustively consider all possible programs and check whether there exists any program for which its execution on one SoC instance is different

from its execution on the other SoC instance. Since the only difference between the SoC instances is the value of the secret, any discrepancy in the program execution must originate from the secret and shows a confidentiality violation.

UPEC verification methodology is based on an unrolled circuit model with a symbolic starting state. The symbolic starting state is a contributing factor to the scalability of the proof and enables the solver to consider program execution starting from any possible pipeline state/pipeline context. The symbolic pipeline state implicitly represents any possible program history. As a result, any sequence of instructions, with arbitrary length, necessary to prime the system prior to an attack, can be implicitly represented by the starting state. This enables the solver to fast-forward to the exact time point when the secret is starting to propagate into the system. Therefore, UPEC can detect vulnerability to complex attacks, such as Spectre, with temporally short counterexamples.

The other contributing factor to the scalability and exhaustiveness of UPEC is our novel iterative verification methodology. Our proposed methodology is based on inductive reasoning that automatically decomposes the proof problem along the possible propagation scenarios. This divide-and-conquer technique enables UPEC to deliver security guarantees even for complex designs.

UPEC provides a unified approach to hardware security analysis. In addition to TES channels, it also detects those functional bugs in a design that cause security issues, for example, related to the communication between the core and its peripherals.

Many sources believe that TES channels are intrinsic to highly advanced processor architectures based on speculation and out-of-order execution, suggesting that such security risks can be avoided by staying away from high-end processors. However, our case study on the RocketChip RISC-V design showed that similar leakage scenarios are possible also in in-order processors. Such a vulnerability can be introduced to the system by detailed RTL design decisions and thus must be addressed during RTL design time.

Our case study on the BOOM RISC-V design led to finding a previously unknown variant of the Spectre attack, which can exploit port contention even in single-threaded processors. Similar vulnerabilities have also been confirmed in Intel processors, which further attest to the power of UPEC in capturing relevant security threats in RTL designs.

We have shown the effectiveness of UPEC on multiple case studies using RISC-V designs, including in-order processors (RocketChip), pipelines with out-of-order write-back (Ariane), microcontroller platforms with a large set of peripherals (Pulpissimo) and processors with a deep out-of-order speculative execution pipeline (BOOM). These experiments have resulted in finding multiple cases of previously unknown vulnerabilities. To the best of our knowledge, UPEC is the first RTL verification technique that exhaustively covers transient execution side channels in processors of realistic complexity. Our case study on BOOM, with more than 650 k state bits, proves the scalability of our technique. UPEC also provides the basis for other security verification targets. Preliminary research results have been shown on applying UPEC to detect security-critical design bugs in access control mechanisms of SoCs, as well as to verify data-oblivious

computing at the hardware level.

Kurzfassung

In den vergangenen Jahren wurde eine Vielzahl neuer Sicherheitslücken aufgespürt, die Hardwaredesignern nach wie vor große Probleme bereiten. Insbesondere sogenannte "Transient Execution Attacks" wie Spectre und Meltdown stellen eine große Bedrohung für die Sicherheit moderner Computersysteme dar. Diese Angriffsklasse basiert auf der vorübergehenden Ausführung spekulativer Befehle, deren Ergebnisse im Falle einer unzutreffenden Spekulation oder bei Erkennung einer Verletzung von Zugriffsrechten wieder zurückgesetzt werden und daher nicht im Programmablauf sichtbar sind. Dennoch können diese ganz oder teilweise ausgeführten Befehle Spuren in der Mikroarchitektur der Hardware hinterlassen, welche mithilfe der genannten Angriffsmethoden über Seitenkanäle ausgelesen werden können. Die jüngste Zunahme von Schwachstellen in der Hardware wurde auch von der Datenbank *Common Weakness Enumeration* (CWE) erkannt, die inzwischen eine eigene Kategorie für diese Sicherheitslücken führt.

Trotz der jüngsten Fortschritte ist es nach wie vor eine große Herausforderung, die komplizierten Auswirkungen der jeweiligen Mikroarchitektur auf die Hardwaresicherheit zu verstehen, was in der Vergangenheit eine ganze Reihe von Updates und Patches in den verschiedensten Systemen nach sich zog. Zur Aufspürung potentieller Angriffskanäle wurde die Hardware bislang hauptsächlich mithilfe von Techniken zur Pfadanalyse (Taint-Analyse) auf ihre Sicherheitslücken hin untersucht. Pfadanalysen prüfen, ob ein unerwünschter Informationsfluss über einen bestimmten verdächtigen Pfad zwischen zwei Punkten im Design erfolgen kann. Diese Techniken haben bei der Analyse des Informationsflusses im Hardware-Bereich eine Vorreiterrolle eingenommen, leiden bei komplexen Designs jedoch in der Regel unter Problemen bei der Skalierbarkeit, was ihre Anwendbarkeit in der Praxis einschränkt. Um einen verdächtigen Pfad zur Überprüfung auszuwählen, muss sich der Benutzer außerdem auf bereits vorhandenes Wissen über potenzielle Schwachstellen verlassen. Somit ist mit diesen Techniken nur schwer eine lückenlose Abdeckung aller vorhandenen Schwachstellen möglich.

Diese Arbeit befasst sich mit der Notwendigkeit eines neuen Ansatzes zur Hardwareverifikation, der die Sicherheit von Prozessoren auf dem Register Transfer Level (RTL) untersucht. Zu diesem Zweck führen wir eine formale Definition von Sicherheit in Bezug auf Schwachstellen in der Mikroarchitektur ein, die sich als Hardware-Property formulieren lässt. Desweiteren präsentieren wir eine formale Beweismethodik, die auf der eindeutigen Ausführung von Programmen (Unique Program Execution Checking (UPEC)) basiert und die dazu verwendet

werden kann, alle Schwachstellen in Bezug auf Transient Execution Attacks in RTL-Entwürfen systematisch zu finden. UPEC benötigt kein Vorwissen über bekannte Angriffe und kann daher auch Schwachstellen erkennen, die auf neuen, bisher unbekannten Arten von Seitenkanälen basieren. Dies wird durch zwei neue Angriffsszenarien demonstriert, die in unseren Experimenten mit UPEC entdeckt wurden. UPEC benutzt ein Verifikationsmodell, das aus zwei identischen Instanzen des zu verifizierenden SoC-Designs besteht. Die Instanzen des SoC im Modell führen dabei das gleiche Programm aus. Der einzige Unterschied zwischen beiden Instanzen ist der Inhalt des geschützten Speicherbereichs und somit der geheimen Daten.

Die Verifizierungsmethodik von UPEC basiert auf einer Property, die die Gleichheit der beiden SoC-Instanzen unter der Bedingung eines realistischen, aber beliebigen Programmablaufs prüft. Die Grundidee besteht darin, alle möglichen Programme vollständig zu betrachten und zu prüfen, ob es irgendein Programm gibt, dessen Ausführung sich auf beiden SoC-Instanzen unterscheidet. Da die geheimen Daten der einzige Unterschied zwischen den Instanzen sind, muss jede Abweichung in der Programmausführung aus dem Geheimnis stammen und stellt somit eine Verletzung der Datensicherheit dar. Die Verifizierungsmethodik von UPEC basiert auf einem abgerollten Schaltkreismodell des Prozessors mit einem symbolischen Anfangszustand. Der symbolische Anfangszustand dient der Skalierbarkeit des Nachweises und ermöglicht dem Solver die Programmausführung von jedem möglichen Pipelinezustand/Pipelinekontext aus. Der symbolische Pipelinezustand repräsentiert dadurch implizit jede mögliche Befehlshistorie. Infolgedessen kann jede beliebige Folge von Instruktionen in beliebiger Länge, die notwendig ist, um das System für eine Attacke vorzubereiten, implizit durch diesen Ausgangszustand dargestellt werden. Dies ermöglicht es dem Solver, genau zu dem Zeitpunkt "vorzuspulen", an dem sich das Geheimnis im System auszubreiten beginnt. UPEC kann dadurch die Anfälligkeit für komplexe Angriffe, wie z. B. Spectre, mit zeitlich kurzen Gegenbeispielen offenlegen. Ein weiterer Faktor, der zur Skalierbarkeit und Vollständigkeit von UPEC beiträgt, ist unsere neuartige iterative Verifikationsmethodik. Die von uns vorgeschlagene Methode basiert auf induktiven Verfahren, die das Beweisproblem automatisch entlang der möglichen Ausbreitungsszenarien zerlegen. Dieses "Divide-and-Conquer" Verfahren ermöglicht es UPEC, selbst für komplexe Designs Sicherheitsgarantien zu liefern. UPEC bietet einen vereinheitlichten Ansatz zur Hardware-Sicherheitsanalyse. Zusätzlich zu den TES-Kanälen erkennt es auch die funktionalen Fehler in einem Design, die Sicherheitsprobleme verursachen können. Ein Beispiel hierfür sind Sicherheitslücken, die mit der Kommunikation zwischen einem Prozessorkern und seinen Peripheriegeräten einhergehen.

Viele Hardwareentwickler gehen davon aus, dass TES-Kanäle in der Natur von hochentwickelten Prozessorarchitekturen liegen, die auf spekulativer Ausführung und Out-of-Order Execution basieren. Demnach könnten solche Sicherheitsrisiken vermieden werden, indem man sich von High-End Prozessoren fernhält. Unsere Fallstudie zum RocketChip RISC-V-Design hat jedoch gezeigt, dass ähnliche Datenlecks auch bei In-Order Prozessoren möglich sind. Eine solche Schwachstelle kann durch sehr spezifische RTL-Designentscheidungen

in das System eingeführt werden und muss daher bereits während der RTL Entwurfszeit bedacht werden. Unsere Fallstudie zum BOOM RISC-V-Design führte zur Entdeckung einer bisher unbekannte Variante des Spectre-Angriffs, die die Verzögerungen von um Ressourcen konkurrierenden Prozessen sogar in Single-Thread-Prozessoren ausnutzen kann. Andere Forschungsgruppen bestätigten unabhängig von unserer Arbeit ähnliche Schwachstellen in Intel-Prozessoren, was ein weiterer Beweis für die Leistungsfähigkeit von UPEC beim Erfassen relevanter Sicherheitsbedrohungen in RTL-Designs ist.

Die Wirksamkeit von UPEC wurde in mehreren Fallstudien mit RISC-V-Prozessoren gezeigt, einschließlich In-Order Prozessoren (RocketChip), Pipelines mit Out-of-Order Write-Back (Ariane), Mikrocontroller Plattformen mit einer großen Anzahl von Peripheriegeräten (Pulpissimo) und Prozessoren mit einer tiefen spekulativen Out-of-Order-Pipeline (BOOM). Diese Experimente führten zur Entdeckung mehrerer bisher unbekannter Schwachstellen. Nach unserem besten Wissen ist UPEC die erste RTL-Verifikationstechnik, die Transient Execution Seitenkanäle in Prozessoren von realistischer Komplexität aufdecken kann. Unsere Fallstudie zu BOOM, ein Prozessor mit mehr als 650.000 Zustandsbits, beweist außerdem die Skalierbarkeit unserer Technik. UPEC bietet darüber hinaus die Grundlage für weitere Verifikationsansätze. Es wurden bereits vorläufige Forschungsergebnisse zur Anwendung von UPEC bei der Erkennung von sicherheitskritischen Designfehler in Zugriffskontrollmechanismen bei SoCs gezeigt und Erweiterungen von UPEC zur Verifikation von datenunabhängigen Berechnungen auf Hardwareebene veröffentlicht.

Bibliography

[1] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz. "Symbolic Quick Error Detection using Symbolic Initial State for Pre-silicon Verification". In: *Design, Automation & Test in Europe Conference (DATE)*. IEEE. 2018, pp. 55–60.

[2] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *Design, Automation & Test in Europe Conf. (DATE)*. 2019, pp. 994–999.

[3] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz. "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors". In: *IEEE/ACM Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[4] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz. "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors". In: *IEEE Transactions on Computers* (2022), pp. 1–1. DOI: 10.1109/TC.2022.3152666.

[5] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz. "Gap-free Processor Verification with S²QED and Property Generation". In: *2020 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France*. Mar. 2020.

[6] J. Müller, M. R. Fadiheh, A. L. Duque Anton, T. Eisenbarth, D. Stoffel, and W. Kunz. "A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2021, pp. 991–996.

[7] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz. "Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing". In: *2022 59th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2022.

[8] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *arXiv preprint arXiv:1812.04975* (2018).

[9] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. R. Fadiheh, D. Stoffel, W. Kunz, C. Barrett, W. Ecker, et al. "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1000–1005.

[10] K. Ganesan, F. Lonsing, S. S. Nuthakki, E. Singh, M. R. Fadiheh, W. Kunz, D. Stoffel, C. Barrett, and S. Mitra. "Effective Pre-Silicon Verification of Processor Cores by Breaking the Bounds of Symbolic Quick Error Detection". In: *arXiv preprint arXiv:2106.10392* (2021).

[11] A. Greenberg. *This Bluetooth Attack Can Steal a Tesla Model X in Minutes*. Wired.com. URL: `https://www.wired.com/story/tesla-model-x-hack-bluetooth/`.

[12] R. Waugh. *Could a vulnerable computer chip allow hackers to down a boeing 787? back door could allow cyber-criminals a way in*. 2012. URL: `http://www.dailymail.co.uk/sciencetech/article-2152284`.

[13] J. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. Fletcher, and D. Kohlbrenner. "Augury: Using data memory-dependent prefetchers to leak data at rest". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 1518–1518.

[14] *Common Weakness Enumeration*. `https://cwe.mitre.org/`.

[15] Y. Yarom and K. Falkner. "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *USENIX Security Symposium*. Vol. 1. 2014, pp. 22–25.

[16] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu. "On code execution tracking via power side-channel". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 1019–1031.

[17] Y. Yarom, D. Genkin, and N. Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". In: *Journal of Cryptographic Engineering* 7.2 (2017), pp. 99–112.

[18] C. Percival. "Cache missing for fun and profit". In: *BSDCan*. 2005. URL: `http://www.daemonology.net/papers/htt.pdf`.

[19] D. Gullasch, E. Bangerter, and S. Krenn. "Cache games–Bringing access-based cache attacks on AES to practice". In: *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2011, pp. 490–505.

[20] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks". In: *25th USENIX security symposium (USENIX security 16)*. 2016, pp. 565–581.

[21] O. Aciicmez and J.-P. Seifert. "Cheap hardware parallelism implies cheap security". In: *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE. 2007, pp. 80–91.

[22] D. Jayasinghe, R. Ragel, and D. Elkaduwe. "Constant time encryption as a countermeasure against remote cache timing attacks". In: *IEEE 6th International Conference on Information and Automation for Sustainability (ICIAfS)*. IEEE. 2012, pp. 129–134.

[23] J. Kong, O. Aciicmez, 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*. ACM. 2008, pp. 25–34.

[24] P. Kocher, 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: *arXiv preprint arXiv:1801.01203* (2018).

[25] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. "Meltdown". In: *arXiv preprint arXiv:1801.01207* (2018).

[26] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, et al. "Fallout: Leaking Data on Meltdown-resistant CPUs". In: *Proc. ACM Conference on Computer and Communications Security (CCS)*. 2019.

[27] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *arXiv preprint arXiv:1905.05726* (2019).

[28] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. "RIDL: Rogue In-Flight Data Load". In: *IEEE Symposium on Security and Privacy (S&P)*. 2019.

[29] V. Kiriansky and C. Waldspurger. "Speculative buffer overflows: Attacks and defenses". In: *arXiv preprint arXiv:1807.03757* (2018).

[30] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, et al. "Speculative interference attacks: Breaking invisible speculation schemes". In: *arXiv preprint arXiv:2007.11818* (2020).

[31] E. J. Ojogbo, M. Thottethodi, and T. Vijaykumar. "Secure automatic bounds checking: prevention is simpler than cure". In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 43–55.

[32] L. Bowen and C. Lupo. "The Performance Cost of Software-based Security Mitigations". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 210–217.

[33] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Stefan, T. Rezk, and G. Barthe. "Towards Constant-Time Foundations for the New Spectre Era". In: *arXiv preprint arXiv:1910.01755* (2019).

[34] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. "A formal approach to secure speculation". In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE. 2019, pp. 288–304.

[35] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. "SPECTECTOR: Principled detection of speculative information flows". In: *arXiv preprint arXiv:1812.08639* (2018).

[36] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. "Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 954–968.

[37] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran. "Hardfails: Insights into software-exploitable hardware bugs". In: *USENIX Security Symposium*. 2019, pp. 213–230.

[38] *Cadence. Incisive Enterprise Simulator*. 2014. URL: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html.

[39] *Averant. Solidify*. 2018. URL: http://www.averant.com/storage/documents/Solidify.pdf.

[40] *Mentor. Questa Verification Solution*. 2018. URL: https://www.mentor.com/products/fv/questa-verification-platform.

[41] Onespin Solutions GmbH. *OneSpin 360 DV-Verify*. URL: https://www.onespin.com/products/360-dv-verify/.

[42] *Cadence. JasperGold Formal Verification Platform*. 2014. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[43] J. Bormann and H. Busch. *Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties)*. European Patent Application, Publication Number EP1764715. Sept. 2005.

[44] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz. "Unbounded Protocol Compliance Verification using Interval Property Checking with Invariants". In: *IEEE Transactions on Computer-Aided Design* 27.11 (Nov. 2008), pp. 2068–2082.

[45] J. Urdahl, D. Stoffel, J. Bormann, M. Wedler, and W. Kunz. "Path Predicate Abstraction by Complete Interval Property Checking". In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2010, pp. 207–215.

[46] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. *The rocket chip generator*. Tech. rep. UCB/EECS-2016-17. Univ. of California at Berkeley, CA, USA, 2016.

[47] *Ariane*. https://github.com/lowRISC/ariane.

[48] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic. "BOOMv2: an open-source out-of-order RISC-V core". In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2017.

[49] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Tech. rep. California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 2014.

[50] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Fourth Edition. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 1-55860-596-7.

[51] R. M. Tomasulo. "An efficient algorithm for exploiting multiple arithmetic units". In: *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[52] D. M. Gordon. "A survey of fast exponentiation methods". In: *Journal of algorithms* 27.1 (1998), pp. 129–146.

[53] H. Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007, pp. 552–561.

[54] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic Model Checking Without BDDs". In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 193–207. ISBN: 3-540-65703-7.

[55] M. R. Prasad, A. Biere, and A. Gupta. "A Survey of Recent Advances in SAT-based Formal Verification". In: *International Journal on Software Tools for Technology* 7.2 (Apr. 2005), pp. 156–173.

[56] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz. "A Normalization Method for Arithmetic Data-Path Verification". In: *IEEE Transactions on Computer-Aided Design* 26.11 (Nov. 2007), pp. 1909–1922.

[57] M. Wedler, D. Stoffel, and W. Kunz. "Normalization at the Arithmetic Bit Level". In: *Proc. International Design Automation Conference (DAC-05)*. June 2005.

[58] W. Kunz, J. Marques-Silva, and S. Malik. "SAT and ATPG: Algorithms for boolean decision problems". In: *Logic synthesis and Verification*. Springer, 2002, pp. 309–341.

[59] M. Thalmaier, M. Nguyen, M. Wedler, D. Stoffel, J. Bormann, and W. Kunz. "Analyzing k-Step Induction to Compute Invariants for SAT-Based Property Checking". In: *Proc. International Design Automation Conference (DAC)*. 2010, pp. 176–181.

[60] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. "Spectre is here to stay: An analysis of side-channels and speculative execution". In: *arXiv preprint arXiv:1902.05178* (2019).

[61] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury. "KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution". In: *arXiv preprint arXiv:1909.00647* (2019).

[62] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. "oo7: Low-overhead defense against spectre attacks via binary analysis". In: *arXiv preprint arXiv:1807.05843* (2018).

[63] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu, and Z. Zuo. "SPECUSYM: Speculative symbolic execution for cache timing leak detection". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1235–1247.

[64] R. J. Colvin and K. Winter. "An abstract semantics of speculative execution for reasoning about security vulnerabilities". In: *International Symposium on Formal Methods*. Springer. 2019, pp. 323–341.

[65] M. Vassena, C. Disselkoen, K. v. 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 ACM on Programming Languages*. ACM New York, NY, USA, 2021, pp. 1–30.

[66] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala. "IODINE: Verifying Constant-Time Execution of Hardware". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1411–1428.

[67] M. Patrignani and M. Guarnieri. "Exorcising Spectres with Secure Compilers". In: *arXiv preprint arXiv:1910.08607* (2019).

[68] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. "Hardware-Software Contracts for Secure Speculation". In: *arXiv preprint arXiv:2006.03841* (2020).

[69] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer. "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface". In: *29th USENIX Security Symposium ({USENIX} Security 20)*. 2020, pp. 1481–1498.

[70] M. C. Tol, K. Yurtseven, B. Gulmezoglu, and B. Sunar. "Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings". In: *arXiv preprint arXiv:2006.14147* (2020).

[71] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. "Invisispec: Making speculative execution invisible in the cache hierarchy". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 428–441.

[72] A. Gonzalez, B. Korpan, J. Zhao, E. Younis, and K. Asanović. "Replicating and Mitigating Spectre Attacks on an Open Source RISC-V Microarchitecture". In: *Workshop on Computer Architecture Research with RISC-V*. 2019.

[73] S. Ainsworth and T. M. Jones. "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 132–144.

[74] S. Kim, F. Mahmud, J. Huang, P. Majumder, N. Christou, A. Muzahid, C.-C. Tsai, and E. J. Kim. "ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage". In: *2020 IEEE Secure Development (SecDev)*. IEEE. 2020, pp. 96–107.

[75] G. Saileshwar and M. K. Qureshi. "CleanupSpec: An Undo Approach to Safe Speculation". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2019, pp. 73–86.

[76] Y. Wu and X. Qian. "ReversiSpec: Reversible Coherence Protocol for Defending Transient Attacks". In: *arXiv preprint arXiv:2006.16535* (2020).

[77] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander. "Efficient invisible speculative execution through selective delay and value prediction". In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2019, pp. 723–735.

[78] C. Sakalis, Z. I. Chowdhury, S. Wadle, I. Akturk, A. Ros, M. Själander, S. Kaxiras, and U. R. Karpuzcu. "On Value Recomputation to Accelerate Invisible Speculation". In: *arXiv preprint arXiv:2102.10932* (2021).

[79] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander. "Understanding Selective Delay as a Method for Efficient Secure Speculative Execution". In: *IEEE Transactions on Computers* 69.11 (2020), pp. 1584–1595.

[80] C. Sakalis, M. Alipour, A. Ros, A. Jimborean, S. Kaxiras, and M. Själander. "Ghost loads: What is the cost of invisible speculation?" In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 2019, pp. 153–163.

[81] L. Zhao, P. Li, R. Hou, M. Huang, P. Liu, L. Zhang, and D. Meng. "Exploiting Security Dependence for Conditional Speculation against Spectre Attacks". In: *IEEE Transactions on Computers* (2020).

[82] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci. "NDA: Preventing Speculative Execution Attacks at Their Source". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2019, pp. 572–586.

[83] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu. "Specshield: Shielding speculative data from microarchitectural covert channels". In: *International Conference on Parallel Architectures and Compilation Techniques (PACT). ACM*. 2019.

[84] J. Fustos, F. Farshchi, and H. Yun. "Spectreguard: An efficient data-centric defense mechanism against spectre attacks". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.

[85] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. "Context: A generic approach for mitigating spectre". In: *Proc. Network and Distributed System Security Symposium*. Vol. 10. 2020.

[86] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher. "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 707–720.

[87] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci. "DOLMA: Securing Speculation with the Principle of Transient Non-Observability". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.

[88] Z. N. Zhao, H. Ji, M. Yan, J. Yu, C. W. Fletcher, A. Morrison, D. Marinov, and J. Torrellas. "Speculation invariance (invarspec): Faster safe execution through program analysis". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 1138–1152.

[89] K.-A. Tran, C. Sakalis, M. Själander, A. Ros, S. Kaxiras, and A. Jimborean. "Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design". In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 241–254.

[90] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. "The gem5 simulator". In: *ACM SIGARCH computer architecture news* 39.2 (2011), pp. 1–7.

[91] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh. "HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security". In: *ACM SIGSAC Conf. on Computer & Communications Security*. ACM. 2018, pp. 1583–1600.

[92] G. Cabodi, P. Camurati, F. Finocchiaro, and D. Vendraminetto. "Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification". In: *Intl. Conf. on Codes, Cryptology, & Information Security*. Springer. 2019, pp. 462–479.

[93] G. Cabodi, P. Camurati, S. F. Finocchiaro, F. Savarese, and D. Vendraminetto. "Embedded Systems Secure Path Verification at the HW/SW Interface". In: *IEEE Design & Test* 34.5 (2017), pp. 38–46.

[94] P. Subramanyan and D. Arora. "Formal verification of taint propagation security properties in a commercial SoC design". In: *Design, Automation & Test in Europe Conf. (DATE)*. IEEE. 2014, pp. 313–314.

[95] W. Hu, L. Wu, Y. Tai, J. Tan, and J. Zhang. "A Unified Formal Model for Proving Security and Reliability Properties". In: *IEEE 29th Asian Test Symposium (ATS)*. IEEE. 2020, pp. 1–6.

[96] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton. "End-to-End Automated Exploit Generation for Processor Security Validation". In: *IEEE Design & Test* 38.3 (2021), pp. 22–30.

[97] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[98] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. "Caisson: a hardware description language for secure information flow". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 109–120.

[99] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. "Sapper: A language for hardware-level security policy enforcement". In: *ACM SIGARCH Computer Architecture News* 42.1 (2014), pp. 97–112.

[100] M.-M. Bidmeshki and Y. Makris. "Toward automatic proof generation for information flow policies in third-party hardware IP". In: *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*. IEEE. 2015, pp. 163–168.

[101] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[102] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. "A hardware design language for timing-sensitive information-flow security". In: *ACM SIGPLAN Notices* 50.4 (2015), pp. 503–516.

[103] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: constructing hardware in a scala embedded language". In: *DAC Design automation conference 2012*. IEEE. 2012, pp. 1212–1221.

[104] R. Guanciale, M. Balliu, and M. Dam. "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 1853–1869.

[105] S. A. Seshia and P. Subramanyan. "UCLID5: Integrating modeling, verification, synthesis and learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE. 2018, pp. 1–10.

[106] C. Trippel, D. Lustig, and M. Martonosi. "Checkmate: Automated synthesis of hardware exploits and security litmus tests". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 947–960.

[107] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu. "INTROSPECTRE: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 874–887.

[108] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. "Osiris: Automated Discovery of Microarchitectural Side Channels". In: *arXiv preprint arXiv:2106.03470* (2021).

[109] S. K. Muduli, G. Takhar, and P. Subramanyan. "Hyperfuzzing for SoC security validation". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–9.

[110] Z. He, G. Hu, and R. Lee. "New Models for Understanding and Reasoning about Speculative Execution Attacks". In: *arXiv preprint arXiv:2009.07998* (2020).

[111] J. Urdahl, D. Stoffel, and W. Kunz. "Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs". In: *IEEE Trans. on Comp.-Aided Design of Integrated Circuits & Systems* 33.2 (Feb. 2014), pp. 291–304.

[112] M. R. Clarkson and F. B. Schneider. "Hyperproperties". In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[113] W. Hu, A. Ardeshiricham, and R. Kastner. "Identifying and Measuring Security Critical Path for Uncovering Circuit Vulnerabilities". In: *International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE. 2017.

[114] D. Brand. "Verification of Large Synthesized Designs". In: *Proc. International Conference on Computer-Aided Design (ICCAD)*. 1993, pp. 534–537.

[115] D. Stoffel, M. Wedler, P. Warkentin, and W. Kunz. "Structural FSM-Traversal". In: *IEEE Transactions on Computer-Aided Design* 23.5 (May 2004), pp. 598–619.

[116] *UPEC BOOM Verification Suite*. `https://github.com/TUK-EIS/upec-boom-verification-suite`.

[117] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. "Verifying Safety Properties Of A Power PC-TM Microprocessor Using Symbolic Model Checking Without BDDs". In: *Proc. International Conference on Computer Aided Verification (CAV)*. 1999, pp. 60–71.

[118] M. Sheeran, S. Singh, and G. Stalmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 2000.

[119] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 361–372.

[120] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. "SPOILER: Speculative load hazards boost rowhammer and cache attacks". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 621–637.

[121] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124077269, 9780124077263.

[122] H. Nemati, R. Guanciale, P. Buiras, and A. Lindner. "Speculative Leakage in ARM Cortex-A53". In: *arXiv preprint arXiv:2007.06865* (2020).

[123] S. Rokicki. "GhostBusters: mitigating spectre attacks on a DBT-based processor". In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 927–932.

[124] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. Garcia, and N. Tuveri. "Port contention for fun and profit". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 870–887.

[125] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention". In: *arXiv preprint arXiv:1903.01843* (2019).

[126] J. Fustos, M. Bechtel, and H. Yun. "SpectreRewind: Leaking Secrets to Past Instructions". In: *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*. 2020, pp. 117–126.

[127] *ARM TrustZone Technology*. `https://developer.arm.com/ip-products/security-ip/trustzone`.

[128] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *European Conf. on Computer Systems (EuroSys)*. 2020, pp. 1–16.

[129] D. Tychalas, A. Keliris, and M. Maniatakos. "Stealthy Information Leakage Through Peripheral Exploitation in Modern Embedded Systems". In: *IEEE Transactions on Device and Materials Reliability* 20.2 (2020), pp. 308–318.

[130] *Symbolic PMP*. `https://github.com/kangoojim/symbolic-pmp`.

[131] *PULPissimo*. `https://github.com/pulp-platform/pulpissimo`.

[132] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. "Practical mitigations for timing-based side-channel attacks on modern x86 processors". In: *2009 30th IEEE symposium on security and privacy*. IEEE. 2009, pp. 45–60.

[133] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. "Fact: A flexible, constant-time programming language". In: *2017 IEEE Cybersecurity Development (SecDev)*. IEEE. 2017, pp. 69–76.

[134] *BearSSL*. https://www.bearssl.org/.

[135] *Ibex RISC-V Core*. https://github.com/lowRISC/ibex.

[136] R. G. Kici, K. v. Gleissenthall, D. Stefan, and R. Jhala. "Solver-Aided Constant-Time Circuit Verification". In: *arXiv preprint arXiv:2104.00461* (2021).

[137] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher. "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data". In: *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021.

[138] *Trust-Hub*. `https://trust-hub.org`.

[139] *SHA cores*. `https://opencores.org/projects/sha_core`.

[140] *Featherweight RISC-V*. `https://github.com/Featherweight-IP/fwrisc`.

[141] *Zip CPU*. `https://github.com/ZipCPU/zipcpu`.

[142] F. Zaruba and L. Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (Nov. 2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[143] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D. Serpanos, and S. Devadas. "A low-latency, low-area hardware oblivious RAM controller". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 215–222.

[144] *SCARV*. `https://github.com/scarv/scarv-cpu`.

[145] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno. "Complete Formal Verification of TriCore2 and Other Processors". In: *Design & Verification Conference & Exhibition (DVCon)*. 2007.

[146] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi. "End-to-End Verification of ARM ® Processors with ISA-Formal". In: *Proceedings of 28th International Conference on Computer Aided Verification*. 2016.

[147] K. Claessen. "A Coverage Analysis for Safety Property Lists". In: *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE Computer Society, 2007, pp. 139–145. ISBN: 0-7695-3023-0.

[148] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra. "QED: Quick error detection tests for effective post-silicon validation". In: *2010 IEEE International Test Conference*. IEEE. 2010, pp. 1–10.

Acronyms

C-S²QED Complete S²QED. 110, 113, 114

S²QED Symbolic initial state Symbolic Quick Error Detection. 105, 108–113

BMC Bounded Model Checking. 19–21, 52, 127

C-IPC Complete Interval Property Checking. 106, 110, 111, 113, 114

CSM Conceptual State Machine. 105, 107

CWE Common Weakness Enumeration. 1, 118, 121

DIT Data-Independent Timing. 102, 104

DMA Direct Memory Access. 89, 93

DUV Design Under Verification. 106

FIFO First In First Out. 79

FU Functional Unit. 57, 61, 62, 95–97, 101, 102

HDL Hardware Description Language. 8, 27, 28

HWPE Hardware Processing Engine. 93, 94

IP Intellectual Property. 117

IPC Interval Property Checking. 8, 10, 13, 19, 21, 22, 42, 52, 105, 127

ISA Instruction Set Architecture. 2, 18, 23, 31, 32, 35, 37, 39, 41, 55, 64, 69, 74, 75, 93, 94, 96, 97, 99, 105, 107, 110–114

IUV Instruction Under Verification. 98–101, 109, 110, 113

LSU Load/Store Unit. 80

LTL Linear Temporal Logic. 111

MSHR Miss Status Handling Register. 85

OOO Out-of-Order. 7

OS Operating System. 41, 73

PMP Physical Memory Protection. 82, 83, 90, 93, 94

PU Prediction Unit. 57

QED Quick Error Detection. 108

RAW Read-After-Write. 71–74, 98, 100

ROB Re-Order Buffer. 13, 56–61, 64, 80, 127

RSA Rivest–Shamir–Adleman public key cryptosystem. 15, 16, 101

RTL Register Transfer Level. 2, 7, 24–27, 29, 30, 41, 43, 46, 68, 74, 75, 77, 78, 87, 92–94, 96, 99, 113, 115, 116, 118, 119, 121–123

SMT Simultaneous Multithreading. 75

SoC System-on-Chip. 1, 5, 7, 25, 27, 38, 42–44, 46–48, 53, 54, 57, 61–67, 74, 82, 85, 88–91, 93, 94, 113, 115–119, 122, 123

SPI Speculation Initiating Instruction. 58–62

STT Speculative Taint Tracking. 26

TEE Trusted Execution Environment. 88

TES Transient Execution Side channel. 3, 4, 6–10, 13, 14, 16, 17, 23–28, 30, 32–35, 38, 55, 56, 64, 65, 68, 69, 71, 74, 75, 77–79, 81, 83, 87, 90, 115–117, 119, 122, 127

TLB Translation Lookaside Buffer. 65, 81, 83–86

UPEC Unique Program Execution Checking. 7–10, 13, 19, 27, 28, 30, 32–34, 37–39, 41–48, 50, 52–56, 59, 61, 63–66, 68, 74–77, 79–86, 88–91, 95, 96, 98, 108, 115–119, 121–123, 127

UPEC-DIT UPEC for Data-Independent Timing. 95–102, 104

List of Figures

List of Tables

Curriculum Vitae

Name: Mohammad Rahmani Fadiheh
Place of birth: Mashhad, Iran

Career

2017 - 2022	:	Research Assistant Technische Universität Kaiserslautern, Germany
2015 - 2017	:	Electronics Engineer Liebler Messtechnik GmbH, Germany
2015 - 2017	:	M.Sc. in Electrical and Computer Engineering Technische Universität Kaiserslautern, Germany
2011 - 2014	:	B.Sc. in Electerical Engineering Amirkabir University of Technology, Iran
2009 - 2013	:	B.Sc. in Biomedical Engineering Amirkabir University of Technology, Iran