Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Bjørg Solem

Applying Unique Program Execution Checking in the development flow of industrial IoT devices to prevent vulnerabilities for side-channel attacks

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Joakim Urdahl & Karine Avagian
June 2023

NTNU
Norwegian University of
Science and Technology

Bjørg Solem

Applying Unique Program Execution Checking in the development flow of industrial IoT devices to prevent vulnerabilities for side-channel attacks

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Joakim Urdahl & Karine Avagian
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Norwegian University of
Science and Technology

Abstract

Recently discovered vulnerabilities in digital designs and computing systems have highlighted the increasing importance of identifying exploitable side channels to ensure secure devices. In the worst scenarios, some vulnerabilities can be maliciously exploited to get hold of passwords and other confidential data.

The Unique Program Execution Checking (UPEC) method is a formal verification technique developed to aid in the discovery of such system vulnerabilities during the early development phase. Previously, this has been tested out on several designs and it has been able to detect potential issues within these.

This master's thesis aims to use the UPEC method to search for vulnerabilities in the newest version of the PULPissimo platform, along with the nRF54L15 design from Nordic Semiconductor. The goal to be accomplished is to assess the scalability of the technique, using the measurements of time-per-proof and memory usage per proof. Additional evaluations will also be conducted, such as how much knowledge is needed beforehand, along with how much manual effort is required to either prove, or disprove, the presence of vulnerabilities in the system.

Although the application onto the nRF54L15 was not accomplished due to issues with the setup in the OneSpin 360 tool, several tests were run on the PULPissimo platform. This led to results regarding the UPEC method's potential scalability, and the discovery of a vulnerability within the system.

The methodology is quite advanced, however, the results accomplished and discussed in this experiment show that this methodology might be a good technique in the future to locate system vulnerabilities. With a bit more work, especially on tutorials to make it more available and understandable, the method could be used on industrial systems.

Sammendrag

Nylige funn av sårbarheter i digitale design og datasystemer har understreket den økende betydningen av å klare å identifisere utnyttbare sidekanaler for å kunne produsere sikre systemer. I de verste tilfellene kan noen sårbarheter bli utnyttet for å få tak i passord og annen konfidensiell data.

Metoden Unique Program Execution Checking (UPEC) er en teknikk innen formell verifikasjon som har blitt utviklet for å hjelpe til med å finne slike systemsårbarheter tidlig i utviklingsfasen. Tidligere har denne blitt testet ut på flere design, og den har greid å identifisere mulige sårbarheter ved disse systemene.

Denne masteroppgaven har som mål å bruke UPEC metoden for å lete etter sårbarheter i den nyeste versjonen av PULPissimo plattformen samt designet nRF54L15 fra Nordic Semiconductor. Hensikten er å vurdere skalerbarheten av teknikken, ved å ta fatt i mål som tidsbruk per bevis som kjøres, samt minnebruk per bevis. Det blir også gjort ytterlige vurderinger, som å se på hvor mye forhåndskunnskap som er nødvendig, samt hvor arbeidskrevende det er å enten bevise, eller motbevise, tilstedeværelsen av sårbarheter.

Selv om implementasjonen av metoden på nRF54L15 ikke ble oppnådd på grunn av problemer med oppsettet av systemet i det benyttede verktøyet OneSpin 360, ble det kjørt flere tester på PULPissimo plattformen. Dette førte til resultater angående metodens potensielle skalerbarhet, samt oppdagelsen av en sårbarhet.

UPEC metoden er ganske avansert. Likevel viser resultatene som har blitt oppnådd og diskutert i dette eksperimentet at det kan være en god fremtidig teknikk for å finne systemsårbarheter. Med litt mer jobb, spesielt i forhold til opplæringsmaterialet som bidrar til å gjøre metoden mer forståelig og tilgjengelig, vil metoden kunne brukes på industrielle systemer.

Acknowledgments

I would like to thank my supervisors at Nordic Semiconductor, Joakim Urdahl and Karine Avagian, for their support and motivating nature throughout. Our talks have always been enriching, and I am grateful they shared their ideas and knowledge with me throughout this process.

Furthermore, I would like to thank my supervisor here at NTNU, Professor Per Gunnar Kjeldsberg, for his encouragement and support during our conversations, and for taking the time to read through my thesis.

An enormous thanks goes to the team at the Technical University of Kaiserslautern, and especially Johannes Müller, for all the information shared about the Unique Program Execution Checking method. I could not have gotten by without all the help and guidance provided by Johannes, who has taken the time to answer all my questions.

I look forward to keeping up with the development and any further publications regarding the UPEC method.

Lastly, I would like to thank my dearest Amund for all his support and love, and my family and friends for their patience and motivation throughout my studies.

Contents

Abstract i

Sammendrag ii

Acknowledgments iii

List of Listings vii

List of Tables viii

List of Figures x

Abbreviations xi

1 Introduction 1
- 1.1 Objectives . . . 1
- 1.2 Motivation . . . 2
  - 1.2.1 Microarchitectural side channels . . . 2
  - 1.2.2 Transient execution side channels . . . 3
  - 1.2.3 Examples of side-channel attacks . . . 3
  - 1.2.4 Some ways to detect vulnerabilities in devices . . . 6
- 1.3 Limitations . . . 6
- 1.4 Research method . . . 7
- 1.5 Thesis structure . . . 7

2 Background 9
- 2.1 General concepts in computer architecture . . . 9
  - 2.1.1 Register-Transfer level . . . 9
  - 2.1.2 Instruction set architecture . . . 9
  - 2.1.3 RISC-V Instruction set architecture . . . 10
  - 2.1.4 Operating system . . . 10
  - 2.1.5 Kernel process . . . 10

2.1.6 Physical memory protection . . . 10
2.1.7 Microarchitectural and architectural states . . . 10
2.2 Introduction to Unique Program Execution Checking . . . 10
2.3 Previous work . . . 11
2.4 Formal verification . . . 11
2.4.1 Logical notations in formal verification . . . 12
2.4.2 Formal property verification . . . 12
2.5 The OneSpin tool . . . 15
2.5.1 Property Checking . . . 15
2.5.2 Black-boxing . . . 15
2.5.3 Counterexamples . . . 16
2.5.4 Hold_bounded(n) result and unreachable counterexamples . . . 16
2.5.5 Vacuous result . . . 16
2.5.6 Approvers and disprovers . . . 16
2.5.7 Timing Diagram Assertion Library . . . 17
2.6 Scalability . . . 18
2.7 The two systems . . . 18
2.7.1 Pulpissimo . . . 18
2.7.2 nRF54L15 . . . 20

3 Unique Program Execution Checking 21
3.1 The Unique Program Execution Checking method . . . 21
3.2 UPEC-SoC . . . 24
3.3 Templates and tutorial . . . 25
3.4 UPEC in practice compared to UPEC in theory . . . 26
3.4.1 Deeper dive into the property . . . 26
3.4.2 The property in relation to the logic equations describing UPEC . . . 28

4 The metrics for the project 30
4.1 Scalability in this experiment . . . 30
4.1.1 Memory and time usage . . . 30
4.1.2 Number of constraints . . . 31
4.1.3 Limitations to scalability in this experiment . . . 31
4.2 Evaluation of the required time and effort . . . 31
4.3 Knowledge beforehand . . . 31
4.3.1 Security expertise . . . 32
4.3.2 About the system . . . 32
4.3.3 About the tool . . . 32

5 Method for implementation 33
5.1 The general approach . . . 33
5.1.1 Black-boxing . . . 34
5.1.2 Analysis . . . 34
5.2 PULPissimo version 7.0 . . . 36
5.2.1 Setup . . . 36
5.2.2 Final property explained . . . 38

5.2.3 Analysis 41
5.3 nRF54L15 44

6 Results **46**
6.1 PULPissimo 46
6.1.1 Alerts discovered 46
6.1.2 Results in regard to scalability 52
6.2 nRF54L15 57

7 Discussion **58**
7.1 Time and effort required 58
7.2 Scalability results for PULPissimo 59
7.2.1 Constraints and states 60
7.2.2 Proof memory-usage with and without optimizations 60
7.2.3 Proof time-usage with and without optimizations 60
7.3 Regarding the alerts 61
7.3.1 P-alerts 61
7.3.2 uDMA L-alerts 62
7.4 Differences in results for the two PULPissimo versions 64
7.4.1 PMP 64
7.4.2 HWPE 65
7.4.3 uDMA 65
7.5 Prior knowledge 65
7.5.1 Need of prior knowledge regarding security and vulnerabilities 65
7.5.2 Need of prior knowledge of the systems 66
7.5.3 Need of prior knowledge regarding the tool 66
7.6 Usage of the UPEC method on industrial IoT devices 66
7.6.1 Experiences with PULPissimo 66
7.6.2 The nRF54L15 67
7.7 Suggestions to ease the usage 69

8 Conclusions **71**
8.1 Future work 71
8.2 Conclusions 72

Bibliography **74**

Appendix **79**
A Property file 79
B Propagation path from uDMA P-alert location 84
C Explanation to the workspace 85

List of Listings

1 Example of a property using an implication operator . . . 13
2 Example of a property using the 'implies' operator . . . 13
3 Example of an assertion . . . 14
4 Example of a property that would give a vacuous result . . . 17
5 UPEC property-shell . . . 27
6 UPEC assertion . . . 27
7 State_equivalence simplified . . . 28
8 Final UPEC property . . . 39
9 Example of a property considering a fanout state in the 'prove' part of the property 43

List of Tables

6.1 Black-boxed modules used in uDMA-propagation test 49
6.2 State_uniqueness size without additional black-boxing 52
6.3 State_uniqueness size with black-boxing used in uDMA-test 53
6.4 Proof runtime experiments 56
6.5 Proof memory-usage experiments 57

List of Figures

1.1 Difference in reload time, victim cache line access vs. no victim cache line access. Inspired by Yarom and Falkner (2014) 4
1.2 Memory access flow in Meltdown susceptible systems. Inspired by Erik Vrielink (2019). 6

2.1 PULPissimo architecture, inspired by The Parallel Ultra Low Power platform (n.d) 19

3.1 UPEC computational model. Inspired by Müller et al. (2021). 23

5.1 Example of how to look for fanout states reached within one clock cycle 42

6.1 All P-alerts collected in PULPissimo 47
6.2 Result from testing the further fanouts for all the six P-alert locations. Tmax=3. . 48
6.3 udma_obs_space function used in 'prove' part of uDMA-property 49
6.4 All the names of the states affected by the propagation of the secret in the first L-alert 50
6.5 All the times at which the states for the first L-alert got different values compared to their counterparts in the other instance 51
6.6 All the names of the states affected by the propagation of the secret in the second L-alert 51
6.7 All the times at which the states for the second L-alert got different values compared to their counterparts in the other instance 52
6.8 Property hold without any black-boxing, using state_uniqueness. Tmax=2. 54
6.9 Property hold with several modules black-boxed, using state_uniqueness. Tmax=2 54
6.10 Property hold with several modules black-boxed, using state_uniqueness. Tmax=3 55
6.11 Property hold with no black-boxing, using uDMA_obs_space. Tmax=3 55
6.12 Property hold with no black-boxing, using uDMA_obs_space. Tmax=4 56

A.1 Property.sv line 2-41 79
A.2 Property.sv line 42-81 80
A.3 Property.sv line 84-115 80
A.4 Property.sv line 116-155 81

A.5 Property.sv line 155-196 82
A.6 Property.sv line 197-222 83
B.1 Whole propagation path from uDMA to sdi_oen_o, shown in fanout view 84

Abbreviations

Abbreviation	Description
BMC	Bounded Model Checking
CTL	Computation Tree Logic
DUT	Device under test
HWPE	Hardware Processing Engines
I2C	Inter-Integrated Circuit
I/O	Input/Output
IoT	Internet of Things
IPC	Interval Property Checking
ISA	Instruction Set Architecture
L-alert	Leakage alert
P-alert	Propagation alert
PMP	Physical Memory Protection
PULP	Parallel Ultra Low Power platform
RTL	Register-Transfer Level
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SVA	SystemVerilog Assertions
Tcl	Tool Command Language
Tidal	The Timing Diagram Assertion Library
UART	Universal Asynchronous Receiver-Transmitter
uDMA	micro-Direct Memory Access
UPEC	Unique Program Execution Checking

1

Introduction

Today, secure electronic computing systems are exceedingly important. Computing systems in today's society range from tiny Internet of Things (IoT) devices, all the way to high-end servers, and can be found pretty much everywhere in day-to-day life (Bhunia and Tehranipoor, 2019b). These computing systems deal with data from a lot of aspects of everyday life, ranging from financial transactions to healthcare records, and more, and this data should not be available to everyone. If an unauthorized party gains access to such data, this could lead to theft of sensitive personal data, or the unauthorized party could potentially be able to damage the computing system. This shows the importance of implementing sufficient security measures and avoiding any vulnerabilities in the design.

Security features are often implemented at the software level, however, without functional hardware mechanics, for instance operational memory isolation, these security features could fall short, all dependent on the system's design. As shown by the discoveries of the infamous Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2019) attacks, properties of the hardware implementation could potentially become a vulnerability that an attacker with malicious intent may exploit. Something such as a small optimization could theoretically present vulnerabilities in a system, while the system still would be regarded as functionally correct. The possible exploitation of these vulnerabilities could potentially lead to the exposure of sensitive information to an unauthorized actor, for instance.

These types of vulnerabilities, whenever found on launched products, can be difficult to fix. The reason for this is that they are rooted in the hardware, which cannot be altered in hindsight. Finding them early on can also possibly be a challenge, as it calls for a lot of knowledge regarding the systems' inner structure and design.

1.1 Objectives

One possible method that could hopefully alleviate the work volume needed to examine and find potential system vulnerabilities is the Unique Program Execution Checking (UPEC) method

(Fadiheh et al., 2019). This is introduced in Chapter 2 and further elaborated in Chapter 3.

Using the concept of the UPEC technique, the desired goal in this thesis is to be able to implement it on an actual industrial design, the nRF54L15 system from Nordic Semiconductor, as well as an open-source system, the PULPissimo platform (Schiavone et al., 2018). The intention behind this is to look at how this method would scale, especially in an industrial setting. This can show how feasible this technique would be in these types of systems, and whether it could be used to help improve security in such devices.

The further intention is to look at how much time and effort is needed to successfully apply the technique, run the tests, and understand the results. It will also be evaluated whether this resulting workload is reasonable. Another purpose behind this is to look at how much knowledge one needs to have beforehand, both regarding security, the tool used, and the system itself. This will, in turn, be compared to how beneficial it ends up being for the systems, in that actual vulnerabilities can be discovered or proven to be non-existent. What metrics will be used to evaluate the scalability of the method, and the effort needed to apply it to the systems, is further detailed in Chapter 4.

Another motive is to look at how the method actually works in practice and to provide an overview as to what the actual implementation of the technique looks like.

This project is done in collaboration with Nordic Semiconductor, a Norwegian semiconductor company that specializes in developing wireless communication technology for IoT solutions (Nordic Semiconductor, n.d.). The nRF54L15 system that will serve as a test case of an industrial design is a design by Nordic Semiconductor.

1.2 Motivation

Below are some potential vulnerabilities and exploitations of vulnerabilities in digital designs described. These show the importance of being able to detect these design problems at an early stage. Because these problems are hardware-related, they are difficult, if not even impossible, to fix on launched products. Thus, the best mitigation to such issues could be to find them early on in development and fix them right away (Bhunia and Tehranipoor, 2019a, p. 18).

1.2.1 Microarchitectural side channels

The descriptions of the logic behavior of a system at the Register-Transfer Level (RTL), which uses registers as building blocks, can be referred to as the microarchitecture of the system (Fadiheh, 2022). These microarchitectures can naturally vary a lot between systems, leaving the designer with a lot of freedom while developing the device. As a consequence, there are a lot of possibilities in regard to implementing optimizations or changing the design otherwise. This can, in some circumstances, cause overlooked functional bugs. Such functional bugs could result in microarchitectural side effects that may be exploitable by someone with malicious intent.

Simple data differences could potentially alter the behavior of the system, and these kinds of behavioral changes could then eventually end up becoming exploitable. If these differences are caused by the confidential data in the system, this could open a ”microarchitectural side channel”, which are channels that accidentally leak private data, without the victim system being aware. The observable or measurable change in behavior could give an attacker the opportunity to decrypt

cipher text, compute cryptographic keys, or otherwise obtain details of the system (Hutle and Kammerstetter, 2015).

There are several different ways sensitive information could leak, and thus be misused by an adversary to find secret data. Some of these ways include execution time differences, power consumption, and electromagnetic leaks (Bhunia and Tehranipoor, 2019c, p. 194).

1.2.2 Transient execution side channels

Transient execution side channels are, as explained in Fadiheh (2022), a phenomenon that occurs whenever processors transiently execute instructions ahead of time, without knowing if the program flow actually reaches these executed instructions. This is also known as speculative execution. In other words, these vulnerabilities originate from the way some processors speculatively execute instructions, in order to improve performance.

In systems using such speculative execution, the system begins executing some instruction before it is known whether this will be needed in the future, based on predictions (Kocher et al., 2019). These guesses are again based on previously executed instructions, with different algorithms for making the predictions. If it turns out that the instruction is actually needed, the system will commit it, and thus have had a performance gain. The issue with this optimization, however, is that it can cause side effects, such as loading the data into the cache. The cache is a temporary storage used to easily and quickly fetch recent, or frequently used, data (Hennessy and Patterson, 2017, p. B-2). Thus, if it turns out that this instruction is not really in the execution path, the data may still be in the cache, even if the instruction is never committed. There are ways to attack a system to get hold of data residing in the cache, such as the Flush+Reload, explained in section 1.2.3.

1.2.3 Examples of side-channel attacks

An attacker could make use of these side channels and potentially be able to analyze the differences in for example timing or voltage levels that these side effects lead to (Hutle and Kammerstetter, 2015). For instance, a timing analysis can be applied to cryptographic systems to extract the secret key. This is because there are some such systems that differ in execution time whenever the guessed keys are partly correct. In such an attack, the adversary can apply different inputs as guesses, and measure the timing difference in the system to determine whether part of the guessed key was correct or not (Bhunia and Tehranipoor, 2019c). Other examples of how side channels can give access to secret data are given directly below, which shows how the Flush+Reload attack and the Meltdown attack might work.

Flush+Reload

The Flush+Reload attack is a way an adversary could get hold of data residing in the cache of a system.

The Flush+Reload attack makes use of the `CLFLUSH` instruction from the Intel x86 architecture. This instruction flushes memory locations from the cache (Yarom and Falkner, 2014).

The way this attack works is that, firstly, the attacker has to map a shared object into address space, which is possible because of the Kernel Shared Memory feature. This is a feature meant to increase memory density, something that is accomplished by merging equal pages (Arcangeli

et al., 2009). Kernel shared memory allows attackers to share a page of data physically with a victim, even if they are different processes. This feature can be used by the attacker to monitor memory line accesses. Because of this, an adversary can make sure that a particular line of memory is evicted from the entirety of the cache (Yarom and Falkner, 2014). Thus, the adversary can flush the monitored memory line.

After the flush, the attacker has to wait and see whether the victim accesses the memory line once again. After the wait time, the attacker reloads this line and checks how long time it takes to load it.

The principle is that if the victim has accessed the line of memory during the wait time, the line is located in the cache. If this is the case, the reload time is therefore shorter than if the line was not brought back into the cache. The reason for this is that a fetch from the cache would be a lot quicker than a fetch from the memory, which would be further away. This is shown in an illustration in Figure 1.1, where a `CLFLUSH` instruction is marked with blue text, the black text represents the waiting period for the attacker and a reload of the line is red. Any other operations performed by the victim in the meantime are marked gray.

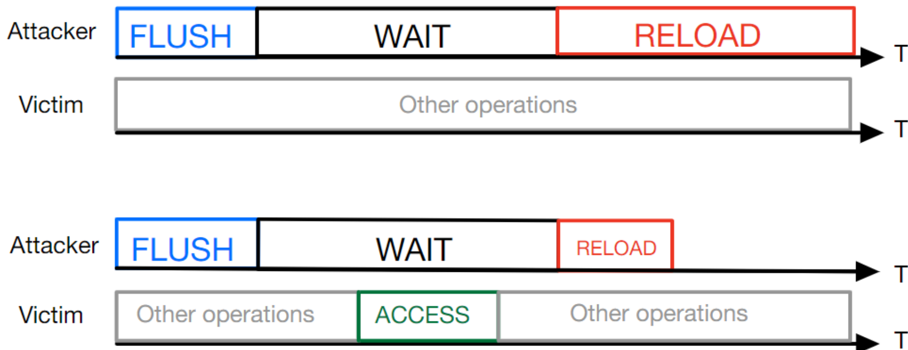


Figure 1.1: Difference in reload time, victim cache line access vs. no victim cache line access. Inspired by Yarom and Falkner (2014)

This timing difference gives the attacker an opportunity to prompt in and make educated guesses of the cache content. Based on the timing difference provided by the system whether the data resides in the cache or not, the attacker could in turn be able to determine whether the prompted in guess is correct or not.

Although the cache data might not be secret data, this attack in conjunction with other attacks, could lead to the leakage of confidential data. Directly below, an attack exploiting the Meltdown vulnerability is further elaborated. This, along with the Flush+Reload attack can lead to such leakage of secret data.

Attacks using Flush+Reload have also already been successfully applied against cryptographic implementations to extract cryptographic keys by trial and error (Benger et al., 2014). While the adversary tries to prompt in guesses of such keys, the system would likely need to fetch the actual key from memory to compare to the guess. In this scenario, it is, therefore, easier to know that the data in the cache is the cryptographic key, which is confidential.

Meltdown

Another example of a very famous and potentially severe vulnerability that can be exploited maliciously is the Meltdown vulnerability. The functionality behind an attack exploiting this vulnerability is thus provided to fully illustrate the weight of how attacks can misuse such vulnerabilities.

This vulnerability could give attackers access to sensitive information stored in a system, such as passwords and encryption keys, because of two optimizations that are put in place in the exposed systems. This vulnerability is quite serious, as it affects nearly every Intel processor produced since 1995 (Graz University of Technology, 2018).

One of these optimizations that can be exploited is out-of-order processing, while the other optimization is an exception handler.

Out-of-order processing is a concept used where the processor can execute instructions in a different order than the original program sequence states (Sarangi, 2021, p. 411). This technique makes use of otherwise unused instruction cycles and hence reduces idle time. Out-of-order processing enables instructions to be executed once their operands are available. This way, the processor can execute instructions that otherwise would be stuck waiting for a previous instruction to finish, as long as these are not dependent on each other. Thus, if one instruction is waiting for resources, another one can be executed if it is ready, and it is independent of the one that is waiting.

Whenever an instruction raises an exception, the control flow should be diverted to the exception handler, and the next instruction is to be prevented from executing. This mechanism is used to manage unexpected events, called exceptions, which are special events that are created when the program executing performs an incorrect action (Sarangi, 2021, p. 379). The execution handler should then try to deal with the exception appropriately.

The issue with using both of these mechanisms simultaneously is that whenever an exception occurs, and the next instruction should be prevented from executing, it might already have been executed because of the usage of out-of-order processing. This can be seen in Figure 1.2, where a permission check would result in an exception if the process does not have the privilege to access the memory. However, the out-of-order execution has caused the memory content to be placed into the cache.

Thus, if an adversary asks to access secret memory, this memory might have been put into the cache to save time. The only thing the adversary then has to do is to make use of the Flush+Reload attack, or any similar attacks, to get hold of the data.

To summarize shortly, an attacker could be able to load the data into the cache, by exploiting the vulnerability of using the exception handler optimization along with the out-of-order processing optimization. By making use of Flush+Reload, an attacker could then try to guess the content of the cache, which now is the secret data. The adversary can be able to tell if the guess is correct or not based on the time it would take for the system to reload the monitored cache line.

There are a lot of specific design details that have to align correctly for such an attack to be feasible. However, this still shows how vulnerabilities potentially could be exploited, and how these exploitations could be quite dangerous.

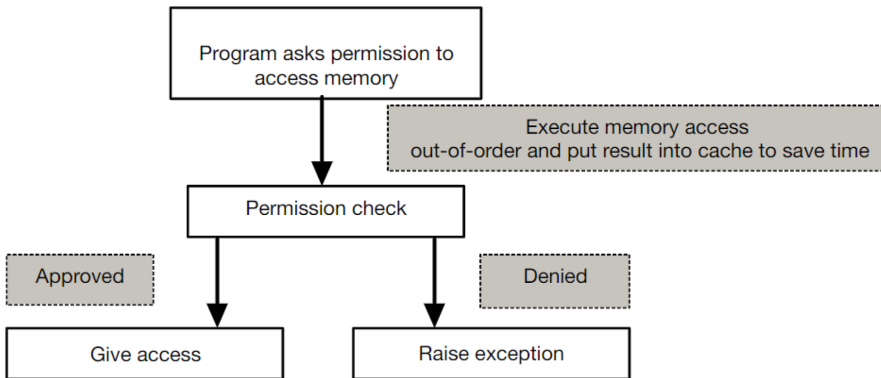


Figure 1.2: Memory access flow in Meltdown susceptible systems. Inspired by Erik Vrielink (2019).

1.2.4 Some ways to detect vulnerabilities in devices

The Meltdown vulnerability and Flush+Reload attack scenarios are just two examples of how various vulnerabilities can be exploited to get hold of secret data. In this case, memory contents of non-accessible memory can be read, which could potentially reveal important secrets. Since these kinds of scenarios are very important to avoid, the subject of security in IoT systems has to be evolving in line with the systems themselves.

Some techniques for finding such vulnerabilities could involve analysis through formal verification, simulation, and analysis of the code (Dessouky et al., 2019). This could require a lot of knowledge about security exploits and how these work to be able to spot any system issues that could be exploited.

There has also been development of design languages for security, such as SecVerilog (Zhang et al., 2015). This implements information flow tracking. The issue with this type of solution, however, is that it calls for changes in the design procedure. For well-established designers with well-defined routines, this can be difficult.

Other possibilities could be to try to break into a system, to see whether it is possible to obtain any confidential data this way. This method would then require a lot of knowledge regarding each system, and possibly also previously known vulnerabilities and attacks.

The reason the UPEC technique might be good for security analysis is that it simplifies the procedure of looking for vulnerabilities, as it is not necessary to have any special security expertise to use this methodology (Müller et al., 2021). The method should be able to detect vulnerabilities that are rooted in observable side effects, and thus also side channels.

1.3 Limitations

As this experiment intends to look into some systems at the product level of whole System-on-Chips (SoCs), the setup and implementation of the method could be a possible challenge.

Working at the product level is quite difficult as it involves a lot of modules and smaller components that have to be taken into consideration. Also, as the experiment is to be performed by someone who is not familiar with the systems beforehand, a lot of time is anticipated to go into research and understanding the environments themselves. This could potentially affect the end result, in that there will be less time to actually implement the method and perform the experiments.

The UPEC technique has already been tested on a previous version of the PULPissimo platform. Even so, an implementation onto a newer version of the platform could still be beneficial, as the environment has gone through changes since then. These differences could have resulted in newly-introduced vulnerabilities in the system, which therefore might yield interesting results.

1.4 Research method

In order to perform this experiment of implementing the UPEC method onto both the PULPissimo and the nRF54L15 system, relevant literature regarding UPEC has been researched. Also, information and documentation surrounding the two systems have been studied. The most relevant information can be found in Chapter 2, which describes the general background for this work, and in Chapter 3, which details the UPEC methodology.

For the nRF54L15 system, which is a project from Nordic Semiconductor, the company's datasheet and information has been used to gain an understanding of the system's inner workings.
For the PULPissimo environment, the GitHub project sites regarding PULPissimo (Schiavone et al., 2018), and the two possible core choices, Ibex (lowRISC team, 2018) and CV32E40P (OpenHW Group, 2023) have been studied, as well as their respective documentation.

The aim to replicate the method on the PULPissimo platform first is intended to be used as a ground measure for comparison when the method is applied to the nRF54L15 system, as well as a method of getting familiar with the technique. Afterward, the results collected from the experiment with PULPissimo are supposed to be used to make decisions regarding how the application onto nRF54L15 should best be performed.

1.5 Thesis structure

The structure of the thesis is as follows:

- Chapter 2 gives the relevant background information regarding common concepts within computer architecture and formal verification. This chapter also presents previous work with the UPEC method, alongside a quick introduction to the method. The two systems, PULPissimo and nRF54L15, along with the OneSpin 360 tool will also be presented.
- Chapter 3 presents all the relevant information regarding the Unique Program Execution Checking method, both the theory behind it and how it is used in practice.
- Chapter 4 introduces the metrics used to evaluate the UPEC method for this experiment.

- Chapter 5 describes the methodology used to perform the experiments conducted in this thesis.
- Chapter 6 presents the results accomplished during the experiments.
- Chapter 7 provides the discussion regarding the accomplished results and the overall experience with the UPEC method.
- Chapter 8 draws conclusions and presents ideas for future work within this subject.

2

Background

This background chapter will give insight into topics that are further explored or otherwise taken into account in this thesis, and thus provide an explanation regarding the relevant literature and subjects. The chapter will also provide a quick introduction to the UPEC method, before it is presented in its entirety in Chapter 3. Previous work on the method will be presented in this chapter as well. Further, the subject of formal verification, as well as the formal verification tool OneSpin 360 is explained, as these subjects are quite fundamental to the UPEC method. Finally, a deeper investigation into the systems used in the performed experiments as well as the subject of scalability is provided.

2.1 General concepts in computer architecture

There are some common concepts in computer architecture that are mentioned briefly throughout this thesis. This section provides the explanation needed to understand these concepts well enough to also comprehend the context in which they are used.

2.1.1 Register-Transfer level

Register-Transfer level abstraction is used to create portrayals of a design, where the building blocks are registers (Vahid, 2010). This gives the opportunity to implement and verify digital circuits in the design flow.

2.1.2 Instruction set architecture

Instruction Set Architecture (ISA) is the way that the hardware is perceived by the software, as it is the definition of all the instructions that are supported by a processor. This also includes the instruction's operands and interfaces with peripheral devices (Sarangi, 2021, p. 17).

2.1.3 RISC-V Instruction set architecture

The RISC-V ISA is an open and simple architecture. It is made to be expandable, in that new user-level extensions can be added (Waterman et al., 2016b).

2.1.4 Operating system

The operating system consists of a set of dedicated programs that administer the device, its peripheral components, and the processes running on the device (Sarangi, 2021, p. 455).

2.1.5 Kernel process

The core component of the operating system is the kernel, and its main role is to administer the processes and their execution. It also has the task of managing the memory (Sarangi, 2021, p. 455). A kernel process is thus a process with control over the rest of the system, and can therefore access the internals of the processor, which should not be available for regular user processes.

2.1.6 Physical memory protection

Physical memory protection (PMP) is an optional RISC-V feature that allows physical memory access privileges to be specified for each physical memory region (Waterman et al., 2016a).

2.1.7 Microarchitectural and architectural states

Microarchitectural state variables include all the state variables, such as registers, buffers, flip-flops, and other components that relate to the logic part of the system's microarchitecture (Fadiheh et al., 2019). These include all the state variables in the system, meaning both those that can be seen by the user, and those that are deep within the system, and thus also invisible to the user.

Architectural state variables are a subset of the microarchitectural states (Müller et al., 2021). These include only those state variables from the microarchitectural states that are program-visible. As stated in Müller et al. (2021), architectural state variables are the registers in the interface of the ISA programmer.

2.2 Introduction to Unique Program Execution Checking

The UPEC technique can possibly be a way to detect vulnerabilities that could eventually be misused in order to conduct attacks on a system. The developers of this method consist of a team from both the Technical University of Kaiserslautern and Stanford University, hereby referred to as the UPEC team.

UPEC is a formal verification-based method, and the idea behind it is to systematically detect vulnerabilities that could lead to possible attacks, all while doing so before product launch. This could therefore be a promising way of reducing vulnerabilities in systems. The underlying concept of the method, briefly explained, is that confidentiality violations occur whenever an attacker

is able to make use of the side effects that are generated by the secret data to uncover secret information (Fadiheh et al., 2019). Such side effects could be timing variations or other visible or measurable differences. Confidentiality in this case relates to the requirement that an untrusted user should not be able to access and read secret and protected data. Therefore, whenever secret data, such as passwords or encryption keys, for example, lead to visible side effects for an attacker, this equates to a confidentiality breach. These side effects can be measured and analyzed in order to gain an understanding of the secret data stored in the system. More details regarding how this approach actually works are presented in Chapter 3.

2.3 Previous work

The UPEC team has given out quite a few publications regarding the UPEC method, in which they detail the inner workings of the method and their own experiments.

A paper (Fadiheh et al., 2019) published by the UPEC team introduces the UPEC concept and explains the methodology. It also details a new kind of attack, called the Orc attack, that is possible to perform on processors with in-order pipelining.

The Ph.D. thesis written by Mohammad Rahmani Fadiheh (Fadiheh, 2022) explains the method and gives in-depth results accomplished by testing it on multiple systems, such as the Berkeley Out-Of-Order Machine (Celio et al., 2017) and Ariane (OpenHWGroup, n.d).

Another published paper (Müller et al., 2021) by the same team details the implementation of the UPEC method onto the PULPissimo platform. This was a similar experiment to the one described in this thesis, but on an older version of the platform.

This was conducted on the PULPissimo version 4.0, and resulted in the detection of vulnerabilities in the Hardware Processing Engines (HWPE) and micro-Direct Memory Access (uDMA) modules, as well as the PMP unit. The issue with the HWPE and the uDMA was that these modules were circumventing the PMP, and an attacker could potentially instruct the modules to read from the protected memory, and write this content onto unprotected outputs. For the PMP, the issue was that there were hardware bugs which resulted in the PMP not behaving according to the ISA.

The same paper also details an experiment on the RocketChip design (Asanović et al., 2016), an SoC without peripheral components which was proven to be secure by the UPEC method.

2.4 Formal verification

Formal verification is the process of checking a design to see if it behaves as expected in comparison to its specification. Formal verification techniques mathematically analyze the possible behaviors of a system, instead of looking only at the results for specific values (Seligman et al., 2015a). Thus, instead of simulating a specific input pattern, and checking the expected behavior for this, a computational model of the design is made and thus evaluated. This gives the developer the opportunity of looking at a larger range of behaviors, which simplifies the process of spotting errors in the design.

One example of a technique is model checking, in which the modeled system is checked to see whether it is possible to reach a state in which the wanted characteristics of the system are violated.

2.4.1 Logical notations in formal verification

In formal verification, there are symbols used to aid the common understanding of the concepts. In addition to a few Boolean (Boole, 1847) logical operators, one Computation Tree Logic (CTL) (Clarke et al., 1986) operator is used as notations in the explanation of the UPEC methodology later in this thesis. Thus, these are explained here.

CTL models time like a tree-like structure. The future can be realized as several different paths, but its outcome is not yet determined (Huth and Ryan, 2004, p. 208).

The notation 'AG' in CTL is essentially used to symbolize that a statement must hold for all the states on all the future paths (Clarke et al., 1986).

The 'AG' notation consists of the operator 'A' which symbolizes 'along All paths' and 'G' which symbolizes 'all future states (Globally)' (Huth and Ryan, 2004, p. 208).

For the two Boolean operator notations used, '$\wedge$' and '$\rightarrow$', these can be explained as follows:

The notation '$\rightarrow$' is the implication operator, which expresses an implication between the right and left sides of the arrow. The notion '$p \rightarrow q$' shows that there is a relationship between these statements in which q is a consequence of p (Huth and Ryan, 2004, p. 4).

Lastly, '$\wedge$' is the conjunction operator, meaning that the notion '$p \wedge q$' would symbolize p **and** q (Huth and Ryan, 2004, p. 4).

2.4.2 Formal property verification

Property checking can be considered a sub-branch of formal verification. The intent behind property checking is to check the behavior of a system against specific properties, to see if they are valid or not. These tests can be performed on blocks or modules to verify their functionality, but also to verify how modules work together at the system level. This type of check inputs the RTL model, along with the set of properties to prove, and the constraints into the tool. The constraints limit the set of possible system behaviors that should be considered.

The output is then represented as a list of the potential proven and disproven properties, as well as a list of the potential inconclusive proofs (Seligman et al., 2015b). This can then be used to find possible design bugs or errors. As a result, designers can identify which parts of the RTL should be redefined to end up with a system that behaves as intended.

An actual example of a property can be seen in Listing 1. Here, the name of the property is ex1, and it begins with the code word 'property' and ends with 'endproperty'. The implication operator, '$\rightarrow$', states that if the left hand of the property is true, the right hand also has to be true on the very same clock edge for the property to hold (Mehta, 2021, p. 434). In order to begin checking the property, the left hand has to go true first, but the right hand should also be true

at the same clock. The notation '##2' represents a time delay of two clock cycles between two events (Institute of Electrical and Electronics Engineers, 2018, p. 342).

The property in Listing 1, therefore, checks whether signal y is set 2 clock cycles after signal x is set. Whenever this happens, the z signal should not be set on the same clock edge.

```
property ex1;
    x ##2 y |-> !z;
endproperty
```

Listing 1: Example of a property using an implication operator

```
property ex2;
    x ##2 y implies !z;
endproperty
```

Listing 2: Example of a property using the 'implies' operator

Instead of using the implication operator, one could also use the 'implies' operator, as seen in Listing 2. Here, the evaluation of both the right-hand side and the left-hand side would begin at the same time. This is in contrast to evaluating the property only when the left-hand side has been proven to hold. Then, the property would fail if either the right-hand side, called the 'prove' part, or the left-hand side, also referred to as the 'assumption' part, fails (Mehta, 2021, p. 510).

Also, there are some built-in functions that can be used within these properties to check specific signal transitions or look at the value of a signal in the past, and more. One such example, that is used in the property describing UPEC in later sections, is the '$past()' function. This checks the value of the signal in the clock cycle previous to the current one (Institute of Electrical and Electronics Engineers, 2018, p. 397).

SystemVerilog Assertions

SystemVerilog Assertions (SVA) is a language that lets designers express what the correct behavior, or the incorrect behavior, of a design is, based on what is desirable to test. An assertion is thus a check against your design specification (Mehta, 2014, p. 9). The expression of the behavior is then verified using a formal verification tool, which simulates the possible states and inputs to determine whether the assertion holds or not.

The example assertion in Listing 3 is the associated assertion to the property 'ex1' presented in Listing 1. Here, the name of the assertion is 'ex1_assertion', and the @(posedge clk) indicates that the property, ex1, should be checked at the positive edge of the clock. The 'assert' keyword

```
ex1_assertion: assert property (@(posedge clk) ex1);
```

Listing 3: Example of an assertion

specifies the property as a requirement for the design that is to be tested (Institute of Electrical and Electronics Engineers, 2018, p. 364). If this were to be swapped with the keyword 'assume', the property would be specified as an assumption for the system (Institute of Electrical and Electronics Engineers, 2018, p. 364). Formal tools would then use this information to restrict the input stimulus.

Constraints

Constraints are used to describe and model the wanted behavior of the system environment (Siemens, 2023a). These are restrictions on the inputs to the device under test (DUT) and are used to ensure that it meets specific requirements and that the design is implemented accurately.

Counterexamples

If the formal verification tool disproves a proof, it also provides a waveform that shows a case of failure (Seligman et al., 2015b). This waveform is called a counterexample. Counterexamples in formal verification thus give examples of inputs that lead to failures in the property tested, and thus show behavior in the system that can witness this claim of failure. This helps with getting an understanding and an insight into the cause of the problem and can thus aid the user in determining a fix to the issue. By analyzing the counterexample, the designer can be able to determine what has to be altered or redesigned for the design to function as intended.

Boolean satisfiability solvers

According to Baek et al. (2021), a Boolean satisfiability solver, or SAT solver, attempts to solve the Boolean satisfiability problem. This is the problem of determining whether there is any input combination of which a Boolean formula, which consists of Boolean variables, can be evaluated to be true. If there is no such combination, the formula or function is unsatisfiable.

Bounded model checking and Interval property checking

Bounded model checking (BMC) (Clarke et al., 2001), is a SAT-based model checking technique in which only sequences of a finite length are investigated. If the property correctness cannot be guaranteed, BMC can still be used for finding counterexamples, which thus proves that a design fails.

As stated in (Fadiheh, 2022), the starting state is thus a known state, often reset. The end of the time window is the max time, Tmax.

Interval property checking (IPC) (Nguyen et al., 2008) is also a SAT-based model checking technique. In contrast to BMC, where the model checking problem is reduced to a finite bounded

time interval, IPC can deliver an unbounded proof. Unbounded proof would mean that the proof is globally valid, and that has no time interval restrictions. Because of this, IPC is used when the property has to hold for the inputs within a range, meaning it is better to guarantee correctness in contrast to BMC.

The difference that enables IPC to deliver an unbounded proof is the starting state of the proof. In IPC, the starting state is an arbitrary starting state, rather than the reset state of the system (Nguyen et al., 2008). Thus, because all possible values are considered for the starting state, this can also include unreachable states of the system.

2.5 The OneSpin tool

The tool used in this experiment to implement the UPEC method is called OneSpin 360. OneSpin is a formal verification tool for digital integrated circuits and systems. It is used to verify that a design follows the specification and to ensure functional correctness. The verification platform thus helps developers with eliminating bugs and errors early on in the process by thoroughly analyzing the RTL (OneSpin Solutions, n.d). Below are explanations of how the formal verification subjects mentioned in the rest of this thesis are used within the OneSpin tool. The concepts themselves might not be OneSpin-specific, but still, they relate to how the tool has been used for these experiments.

2.5.1 Property Checking

As previously mentioned, formal verification is using tools to analyze all possible design behaviors, instead of looking at results for just some specific input values (Seligman et al., 2015a). Property checking in OneSpin is done by specifying the behavior of a module in a formal language and then checking the design against this behavior. A successful check results in a 'hold', while a 'fail' would generate a counterexample that displays a situation that violates the property (Siemens, 2023b). In OneSpin, the design behavior can be specified using SVA, the design intent is captured in properties and assertions, and the behavior of the system can be modeled by using constraints.

2.5.2 Black-boxing

Black-boxing is a technique used within formal verification, in order to simplify complex logic. This can be used to represent parts of the system as a blacked-out box, which could simplify any proofs run on the system, and thus also reduce the time spent analyzing it. Black-boxing results in logic being ignored during the verification because a replacement with no internal logic or implementation details is made in order to simplify the design. The actual internal logic is abstracted away. Signals connected to the black-boxed components inputs are treated as primary outputs of the system, while signals connected to the outputs are seen as primary inputs to the system (Siemens, 2023a). As a result, the proofing process time is shortened, as the tool can thus deal with a simplified design.

In OneSpin, this is implemented using a 'black-box' compile option, which then declares the module in question as a blacked-out box, and the tool then treats the module as such.

2.5.3 Counterexamples

A counterexample is generated by the OneSpin formal verification tool, as a way to show that a property fails, or that it cannot be satisfied. As previously stated, this allows the user to investigate what is the root cause of the property's fail, and how to fix this issue in the design. The counterexample provided by OneSpin shows what input values are the root cause of the property failure, as well as a waveform that shows the design behavior which contradicts the assertion. This also shows values of all other relevant signals in the DUT that help demonstrate why the proof failed (Siemens, 2023a).

2.5.4 Hold_bounded(n) result and unreachable counterexamples

A 'hold_bounded(n)' result would indicate that the assertion tested holds up until a certain point, but no longer. The 'n' represents the specified depth (Siemens, 2023a). This generates a counterexample, which shows behavior indicating what has been violated. However, the tool cannot guarantee that the counterexample provided is reachable.

An unreachable counterexample depicts unrealistic system behavior, meaning that it cannot occur in reality.

Thus, a 'hold_bounded' result is similar to a 'fail', in that the property cannot be proven beyond the point reached during the proof. However, because the counterexample might not show realistic behavior, manual inspection is required to see whether this is the case or not. If the counterexample is unreachable, the issues with the design can be fixed with inspection and correction of any erroneous constraints.

2.5.5 Vacuous result

The result can also be 'vacuous'. This means that the property, its constraints, and the design is contradictory, and thus, it cannot be proven to either fail or hold. This often happens if the left hand side of the implication operator is contradictory. (Siemens, 2023b).

One such simplified contradiction can be seen in Listing 4. Here, the assertion tries to verify that the property 'vacuous_ex_property' is true, but the property itself is contradictory. It states that at the positive edge of the clock, the signal 'sig_X' should both be set and not set, which cannot happen. The right side of the implication operator states that 'sig_Y' should be set in the next positive edge of the clock. Because the left side is contradictory, this property would give a vacuous result.

2.5.6 Approvers and disprovers

By default, OneSpin uses an automatic mode when checking assertions to get the best possible proof results. This automatic mode then combines several proof algorithms. However, in some instances, it can be useful to disable this automatic mode and thus also some proof engines. This could for example be if some proof engines do not help the overall result. In this case, the proof time might be enhanced if the proof engines are disabled.

Approvers, or IPC solvers, in OneSpin, is a feature that can be used to control the strategies used in checking assertions. When using only the approvers, which is the case when running an

```
vacuous_ex_property;
    @(posedge clk) (sig_X && !sig_X) |=> sig_Y;
endproperty


assert property(vacuous_ex_property);
```

Listing 4: Example of a property that would give a vacuous result

IPC proof, a 'hold' can be outputted, but these cannot result in a 'fail', as they are only able to prove an assertion.

When IPC solvers are used, the proof is started in an arbitrary state, which might not be reachable from reset. The reason it might not be reachable is that the starting state itself could violate the property (Siemens, 2023b). Instead of a 'fail', a 'hold_bounded(n)' result can therefore be outputted. This will create a counterexample, which will point toward the violation. However, the counterexample has to be manually inspected to determine whether it is reachable or not.

One can also choose which approvers one wishes to run, as there are differences between the approvers. Some are better for assertions that depend only on a few signals, while some are also able to disprove an assertion (Siemens, 2023b).

Another proof engine type is the BMC disprovers, which also can be used instead of using the automatic mode. These cannot prove an assertion, and can thus only result in a 'fail' or a 'hold_bounded(n)'.

Because this proof engine type makes use of BMC, they will start the property from a known state, often reset. This guarantees that the counterexample provided whenever a 'fail' is outputted is a reachable counterexample (Siemens, 2023b). A 'hold_bounded(n)' result in this case, however, would rather guarantee that the counterexample provided is unreachable. This would therefore show unrealistic system behavior, instead.

2.5.7 Timing Diagram Assertion Library

The Timing Diagram Assertion Library (Tidal) is a language extension made by OneSpin. It is a library of SVA definitions for modeling operations based on generalized timing diagrams. Properties expressed in Tidal are treated as an assertion by the OneSpin tool (Siemens, 2023b).

During_o(T1, O1, T2, O2, SE)

The during_o function is a concept in Tidal that states that the state expression, SE, should hold in the interval from the beginning T1 + O1 to the ending, T2 + O2 (Siemens, 2023b).

2.6 Scalability

Definition

According to Bondi (2000), scalability refers to a system's capacity to adapt and deal with an increasing number of elements or objects, to be able to effectively manage expanding volumes of work, and/or to handle enlargement.

Scalability is then the ability a system or a method has to continue to function as expected whenever its size or volume is altered. If one attempts to apply a method to a system of a larger scale, this also means that the method has to do more work. Scalability thus means that the system or method should be able to function as intended despite having to deal with an increasing amount of work.

The term scalability may vary from system to system, as it could include scalability in terms of size, data traffic, and more.

Limitations

The ability to scale onto a new system could be limited by the size of the new system one tries to apply the method to. As a system becomes larger, and hence also more complex, this also results in a lot more work for the system. Thus, the ability of a system or method to scale depends on whether it has enough resources to deal with this additional workload. An unscalable system would refer to a system in which the additional cost of managing this size increase is too large, or where the system is not able to deal with the growth at all (Bondi, 2000).

2.7 The two systems

2.7.1 Pulpissimo

PULPissimo (Schiavone et al., 2018) is an open-source SoC platform that is part of the ”Parallel Ultra Low Power (PULP) platform”, which is a collaboration between ETH Zurich and the University of Bologna (The Parallel Ultra Low Power platform, n.d). Their aim is, according to The Parallel Ultra Low Power platform (2022), to develop a hardware and software platform that is scalable as well as open. Additionally, they want to meet the demands of current IoT applications, which require flexible processing of data streams.

The PULPissimo SoC is a single-core platform, designed to be very energy-efficient. The architecture includes for instance support for HWPE, which are hardware accelerators that share memory with the core. PULPissimo also supports input/output (I/O) interfaces, some of these are for instance Universal Asynchronous Receiver-transmitter (UART) and Inter-Integrated Circuit (I2C). Both of these are different communication protocols used to transmit data between devices.

The Serial Peripheral Interface (SPI) is also a communication protocol with a master-slave architecture, where the master initiates communication, and the slaves respond to the master's requests (Trivedi et al., 2018).

The architecture of the system can be seen in Figure 2.1, where the most central subsystems for this experiment, along with some Input/Output interfaces, are included. The interconnects are named to avoid any confusion regarding these blocks.

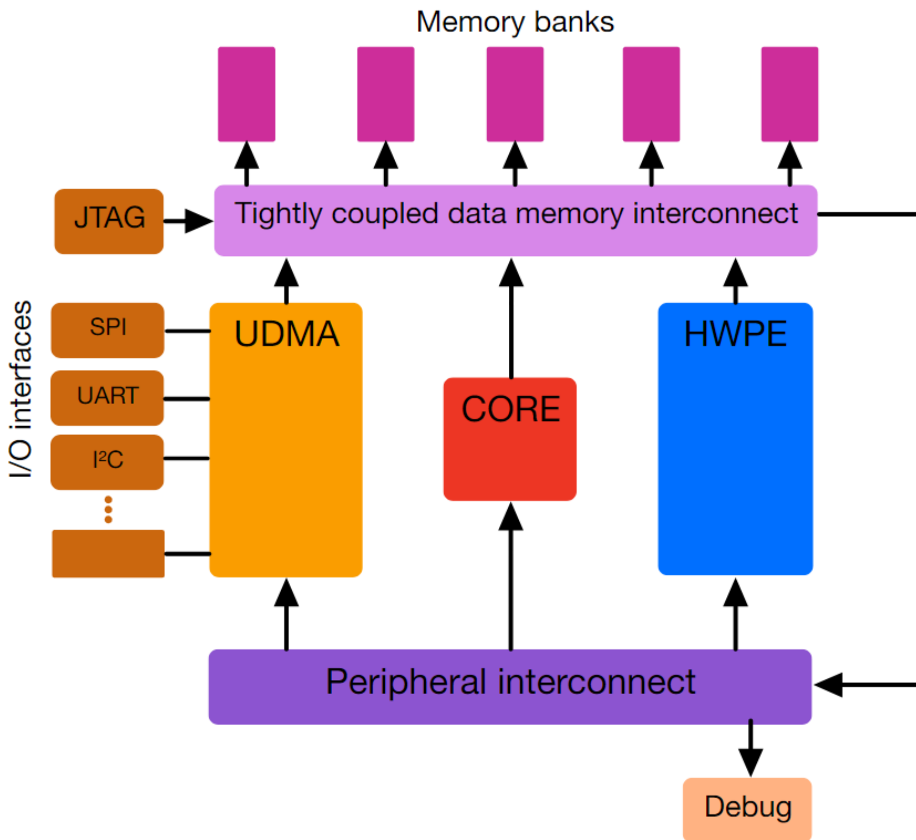


Figure 2.1: PULPissimo architecture, inspired by The Parallel Ultra Low Power platform (n.d)

The uDMA is a subsystem that communicates with the peripherals, such as the SPI and UART, autonomously. It can access memory and thus transfer data from the peripheral to memory or vice versa, without having to go through the processor (Sarangi, 2021, p. 587).

The 'debug' module seen in Figure 2.1 is the subsystem used for debugging during the development of the system.

One can choose to use either the Ibex core or the CV32E40P core, both of which are based on the RISC-V ISA. Both of the cores are also open-source.

CV32E40P core

The CV32E40P is a small, in-order core with a 4-stage pipeline. It is designed for low power consumption, which makes it suitable for many different types of embedded systems, such as the PULPissimo platform. The core was previously known as the RI5CY, and was maintained

by the PULP platform team until February 2020, when it was contributed to OpenHW Group (OpenHWGroup, n.d).

Ibex Core

The Ibex core was initially developed by the PULP team at ETH Zurich, and is now maintained by the lowRISC team, a non-profit company based in Cambridge (lowRISC team, n.d,a). It is a 32-bit CPU core that is highly parameterizable and suitable for embedded control applications (lowRISC team, n.d,b). It is also an in-order core with 2 pipeline stages. It was initially designed to target ultra-low-power and ultra-low-area constraints (The Parallel Ultra Low Power platform, n.d).

2.7.2 nRF54L15

The nRF54L15 is an ultra-low-power SoC with a rich set of peripherals, which provides a range of analog and digital functionality, and advanced security features (Nordic Semiconductor, n.d,). This SoC is a product designed by Nordic Semiconductor, with an Arm Cortex-M33 processor. The processor features a floating-point unit for carrying out operations on floating-point numbers, as well as a digital signal processing extension, which offers signal processing for e.g. voice, audio, and machine learning applications (Arm Developer, n.d,a).

Further, the system has multiple power domains to provide low-power operation. This SoC is thus composed of three domains. The first is the radio domain, which contains the short-range radio and its supporting units. The second is the low-power domain, designed for ultra-low-power operating subsystems. The last one is the peripheral domain, which then contains most of the peripherals (Nordic Semiconductor, n.d,).

nRF54L15 also features Arm TrustZone technology and supporting units to ensure system protection and key management. This enables hardware-enforced separation, which thus reduces the potential for attack by isolating confidential information and critical security firmware from the rest of the application (Arm Developer, n.d,b).

3

Unique Program Execution Checking

The purpose of this chapter is to give insight into the theory behind the UPEC methodology and to look into the intent behind it. The chapter also explains the UPEC-SoC extension to the methodology, which makes it possible to consider peripherals in a system as well. The templates and the tutorial provided by the UPEC team are also introduced.

The last section of this chapter has the intention of helping visualize what the UPEC method and the UPEC-SoC method actually look like in practice. This is to help establish a logical connection between the theoretical statements of the UPEC methods and the actual property as it would be presented in SVA. This is based on experiences gained during the experiments, the usage of the templates provided, and the general theoretic understanding of the methods.

The UPEC-SoC method is built on the same premises as the regular UPEC method. However, it has a few more considerations to be able to consider peripherals and not just the processor core. Therefore, section 3.2 will go on to explain the UPEC-SoC method, to clarify how the method extends onto SoCs as well.

Any further notion of 'the UPEC method' beyond this chapter will regard these two as one methodology. If there is any need to specify which methodology is referred to, this will be done using either 'the regular UPEC method' or 'the UPEC-SoC method', for the remaining chapters.

3.1 The Unique Program Execution Checking method

In attacks making use of microarchitectural side channels, non-trusted user processes do not have the privileges needed to directly read confidential data within a system. However, if there are any observable or measurable differences from a user's point of view, whenever an unprivileged user runs a program, these differences can be misused. An unprivileged user would in this case refer to a user that should not have access to this confidential information.

If the adversary is able to somehow measure the difference in any way, and then work out a way to find the secret based on the difference, the adversary could imaginably be able to manipulate the system and get hold of the secret data.

As stated in Fadiheh et al. (2019), confidentiality in hardware/software systems is the requirement that non-trusted user processes are not able to read confidential data. Therefore, there should be no observable traces of the secret. These traces can be manifested in side effects such as differences in value, timing differences, voltage level changes, and more.

Fadiheh et al. (2019) first described the UPEC method. This approach was introduced as a way to detect hidden side channels. A large motivation behind this methodology is that there seemingly is no need for prior knowledge about previous security exploits (Müller et al., 2021). Therefore, this methodology could really be beneficial for discovering vulnerabilities in systems under development.

The general intent behind the UPEC methodology is to check whether a system could contain any vulnerabilities that could be misused to get hold of secret, or confidential, information. The method thus attempts to test whether the secret information in the system can affect the system's operation in any observable way. If that is the case, then there might be observable traces of confidential information that can be picked up and maliciously exploited. Determining what information should be seen as secret can be done based on the security requirements of the system (Fadiheh, 2022).

The idea behind protecting this confidential data is that a system should behave uniquely, based on the confidential data within. This means that the values assigned to the set of architectural state variables should be independent of the confidential information (Fadiheh, 2022). The system should rather behave predictably, such that there are no user-observable behavior differences due to the secret.

Therefore, if the system executes uniquely in regard to the secret, this secret data remains confidential. This concept is called Unique Program Execution.

UPEC can therefore be applied to check whether this is the case, or not. The way this works, is to set up two of the same SoC, as seen in Figure 3.1. This figure represents two nearly identical instances of the same SoC, in which the only difference between the two is the secret data. This means that the location holding the secret data in the two instances should contain different values.

Then, the purpose is to test whether the rest of the system remains equal throughout the proof, meaning that every state and register in system one should contain the same information and data as its counterpart in system two. If this is not the case, the system reports behavior change, which happens because of the confidential data.

This could be that one register in system instance 1 would obtain a different value than the same register in system instance 2. It could also be that the same value is obtained by the two registers at different time points, indicating a timing difference.

This behavior change can, through regular system operation, propagate throughout the system, from module to module. If ever this change reaches any states or registers that can be viewable for a user that executes any program on this design, meaning any program-visible states, this would count as a vulnerability.

The computational model produced from the design's RTL description must satisfy the following property for a system to be considered protected from side-channel attacks (Müller et al.,

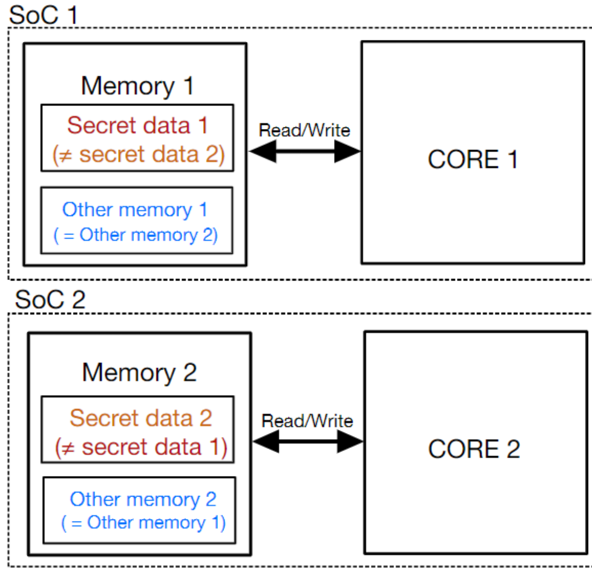


Figure 3.1: UPEC computational model. Inspired by Müller et al. (2021).

2021):

$$\begin{aligned} \text{AG } (&secret_data_protected \\ &\wedge micro_soc_state_1 = micro_soc_state_2 \\ &\rightarrow \text{AG } arch_soc_state_1 = arch_soc_state_2) \end{aligned} \tag{1}$$

In Equation 1, *secret_data_protected* is a constraint that restricts the system to only include the scenarios in which a security mechanism is set up to protect the secret data. This constraint is configured to restrict access to some fixed addresses, in which the location of the secret is restricted to. When looking for transient execution side channels, the hardware can be assumed to be functionally correct. This leads to the implication that all addresses and secure configurations are consistent in their behavior with regard to side channels (Müller et al., 2021).

micro_soc_state represents the set of microarchitectural state variables, while *arch_soc_state* represents the set of architectural state variables (Müller et al., 2021).

In short, the property states that one assumes that there is some security measurement enabled to protect the secret data and that the microarchitectural state variables have the same value in the two instances. If this is the case, this implies that the architectural state variables also have to be equal in the two instances.

Any difference in the value of the user-visible registers or states, also those due to differences in timing, must come from the secret data. Thus, the difference that is visible relates to traces of the secret data, which now can be observable by a potential attacker. This would then be considered a breach of confidentiality (Müller et al., 2021).

3.2 UPEC-SoC

The UPEC-SoC paper (Müller et al., 2021) extends the UPEC method (Fadiheh et al., 2019). This extension also includes any peripherals, meaning that the UPEC-SoC method does not only consider the processor but the whole SoC.

This research can also look at confidentiality violations caused by functional bugs in the hardware.

As stated in Müller et al. (2021), the idea of Unique Program Execution ensures that secret data cannot be seen by an unprivileged attacker running a program. Although all information leakage possibilities in processor cores are covered by this notion, it is not adequate for more advanced SoCs that might feature numerous peripherals. This is due to the fact that the secret can go past the main core and leak straight into unprotected memory locations or memory-mapped outputs, all while the main core program runs uniquely (Müller et al., 2021). Therefore, an extension to Unique Program Execution to also consider spreading to the system's primary outputs, as well as unprotected memory locations, is necessary. As follows, the previously considered architectural state variables, in addition to the added locations, primary outputs and unprotected memory, together make up the observable state variables (Müller et al., 2021).

Because of the extension of the concept of Unique Program Execution, extensions to the property are also in order. This property has to be proven on two identical, separate instances of the SoC (Müller et al., 2021):

$$
\begin{aligned}
\text{AG}\ (&equal_memory_state \\
&\wedge micro_soc_state_1 = micro_soc_state_2 \\
&\rightarrow \text{AG}\ obs_soc_state_1 = obs_soc_state_2)
\end{aligned} \tag{2}
$$

Here, '*micro_soc_state*' is the same as for Equation 1.

The '*obs_soc_state*' represents the previously presented observable set of state variables. If the secret propagates to these, an alert is raised (Müller et al., 2021).

Further, *'equal_memory_state'* is a constraint that requires the memory contents to be equal between the two system instances for all unprotected memory addresses. If this is not the case, there should exist a symbolic protection configuration for these memory locations, so that they cannot be accessed (Müller et al., 2021).

The UPEC-SoC methodology also includes consideration of functional bugs to be the possible cause of security problems in SoCs. Therefore, the previous assumption that the hardware is functionally correct is no longer adequate. This also includes the implication that all addresses and secure configurations are consistent in their behavior with regard to side channels, which now no longer holds.

Because of this, the set of possible secret data locations is now extended to include all memory locations, all primary inputs, and all protected architectural registers (Müller et al., 2021).

Propagation alerts and Leakage alerts

As stated in Fadiheh et al. (2019), secret data may flow to microarchitectural registers. These are not necessarily observable by a user program, which means that they do not result in any program behavior change. This means that no secret data is leaked in this scenario.

There are, however, also cases where the secret data propagates in such a way that it can affect program-visible states. These two cases can be sorted into two separate 'alerts' in the UPEC- and UPEC-SoC methods.

A leakage alert (L-alert) indicates that secret data has influenced architectural states. Therefore, this shows the propagation of secret data into an architectural register (Fadiheh et al., 2019). This means that the propagated secret information has an impact on states that are program-visible and are thus observable or otherwise measurable by a potential attacker, which then would be a breach in confidentiality.

A counterexample produced during the analysis phase of the UPEC-, or UPEC-SoC method, would give insight into such a confidentiality violation. It would give an example of a scenario where a confidential part of memory would be observable, because of the difference in behavior between the two DUTs (Fadiheh, 2022).

A propagation alert (P-alert) shows possible propagation paths of secret data to program-invisible internal states of the system (Fadiheh et al., 2019). The propagated secret, therefore, does not affect architectural states in this case, and is therefore not observable. A P-alert is, however, often a precursor to an L-alert. The reason for this is that the secret frequently travels through internal program-invisible buffers before it is transmitted to an architectural state variable (Fadiheh et al., 2019).

3.3 Templates and tutorial

The UPEC team supplied information as to how the UPEC- and UPEC-SoC methods should be applied in practice, which included templates and a tutorial. This information is a part of their tutorial in the ongoing process of developing the method to make it more accessible and functional for companies and designers for future use. The information provided includes a PowerPoint tutorial, which explained the setup process and analysis process briefly. It also mentioned possible optimizations, without going into further detail regarding these.

They have also developed automated scripts that generate the top-level model which is to represent the two near-identical DUTs. All the microarchitectural states are assumed to be equivalent in the two instances. This is done by setting constraints, which can be autogenerated using a script as well.

Another useful asset that was provided by the team at Kaiserslautern was a Tool Command Language (Tcl)-file that, when ran, collected all the states that were unequal to their counterparts in the counterexamples. This made it possible to actually see the propagation of the secret, and which states it had reached, and at which time these states were reached. This also saved a lot of manual work, as it can take a lot of time just to look through the counterexample, to find states that do not share the same value as their counterpart.

The team also supplied templates to help set up the method on the design to be inspected. The templates include shells showing how to set up the property and its constraints.

3.4 UPEC in practice compared to UPEC in theory

As the methodology behind this way of testing systems for vulnerabilities can be quite complex, this section attempts to make connections between the theory and an actual implementation of the method. Before this can be accomplished, a look into the shell of the UPEC property is necessary. The topics that are introduced here will be further detailed in Chapter 5, which explains the actual analysis approach, and the process used to apply the method.

3.4.1 Deeper dive into the property

The details of all the functions needed to understand the connection between the property in practice, and the theory behind it are provided below. This section thus provides the intention behind the functions in the property, based on the provided material, without going into implementation details.

The actual implementation of the property used in the experiments and further explanations of all the functions are detailed in subsection 5.2.2.

The shell of the property as it would be realized in the OneSpin tool can be seen in Listing 5. In this property, there is an 'implies' operator. The property states that the 'state_equivalence', 'secret_data_protected' and the 'no_pending_secret_access' functions consist of statements that should be assumed to be true at the startup time, hence the t##0.

Further along in the assumption part is 'during_o(t, 0, t, Tmax, unpriv_mode())'. This is a Tidal function, as previously explained in subsection 2.5.7, which in this case specifies that for the duration between the time 0 and Tmax, the function 'unpriv_mode()', introduced below, is assumed to 'hold'. In this case, the 'Tmax' variable should be swapped with whatever clock cycle one wishes to test until.

In the prove part of the property, the Tidal function is used once again. This time, the test is for the time between 1 and Tmax. Thus, the wish is to prove that the 'state_uniqueness()' function holds for the given time interval, given the assumptions before the 'implies' operator.

The associated assertion of the UPEC property in Listing 5 can be seen in Listing 6. This states that the UPEC property should be checked at the positive edge of the clock defined in the top module, which is called 'miter_top', and that the assertion should be disabled if a reset is detected.

State equivalence

The first assumption is that the 'state_equivalence' function should hold at the current time. The state equivalence function is the representation of the constraints stating that all the microarchitectural states have to remain equal to the counterparts found in the other instance. This assumption starts from time zero. A simplified example version of the 'state_equivalence' function can be

```
property UPEC;
    t##0 state_eqivalence() and
    t##0 secret_data_protected() and
    t##0 no_pending_secret_access() and
    during_o(t,0, t,Tmax, unpriv_mode())
implies
    during_o(t,1 t,Tmax, state_uniqueness());
endproperty
```

Listing 5: UPEC property-shell

```
UPEC_assertion: assert property(@(posedge miter_top.clock)
 ↪  disable iff(reset) UPEC);
```

Listing 6: UPEC assertion

seen in Listing 7. This shows how the imagined register 'reg_x' from 'module_x' in the first instance should contain the same value, or be equal to, the same register in the second instance. The same goes for 'register_y' from 'module_y'.

This function should thus contain constraints for all the state variables in the system, stating that these should be equivalent in the two instances.

Secret data protected

The 'secret_data_protected' function that is assumed by the UPEC property ensures that there is some access control set up to protect the secret data from unprivileged users, meaning that they are not accessible for certain user processes. In RISC-V architectures, such as the Ibex core available for the PULPissimo system, this memory protection is often PMP. Thus, the 'secret_data_protected' function is used to set up the PMP in the system to protect the secret addresses used in the proof.

No pending secret access

The 'no_pending_secret_access' macro assumes that, at the startup of the proof, there are no accesses to the secret waiting to happen. The areas containing secret data are inaccessible to user processes. On an envisioned finished and sold product, however, the operating system or kernel process is freely allowed to access these locations. The kernel process could read data from the protected secret location, and then branch into a malicious user-level process. This cannot happen within the actual operation of the SoC, and would therefore be considered an unreachable counterexample (Fadiheh, 2022). Therefore, it is important to assume a scenario where there is

```
function automatic state_equivalence();
    state_equivalence= (
        top_module.inst1.module_x.reg_x ==
        top_module.inst2.module_x.reg_x &&
        top_module.inst1.module_y.register_y ==
        top_module.inst2.module_y.register_y &&
        ....
    );
endfunction
```

Listing 7: State_equivalence simplified

no such access to the secret location at the beginning. Thus, this constraint is put in place to avoid having to analyze these kinds of counterexamples, with the outcome being that it cannot happen in reality (Fadiheh, 2022).

Unpriv mode

The 'unpriv_mode' function is used in the assumption that the program execution is in user mode, and thus, the secret data should not be directly accessible by user processes.

State uniqueness

The 'prove' part of the property is used to express that all the states assumed to be equal previously, plus any program-visible states, such as primary outputs or unprotected memory, should remain equal. Thus, this notion is the same as the 'state_equivalence', plus constraints for primary outputs or unprotected memory. Which states are included in this notion is therefore chosen based on which states in the system can be observed or measured otherwise.

3.4.2 The property in relation to the logic equations describing UPEC

The property shown in Listing 5 can be seen in relation to the logical equations in Equation 1 and Equation 2. Both these logical equations state that for all states forward, it is given that the microarchitectural states within the system should remain equal to their counterparts. For both of the logical equations, this is given by the *'micro_soc_state'* constraints. This can be seen in connection with the 'state_equivalence' function in the UPEC property, which includes all the microarchitectural states, and assumes that these should remain equal from one instance to the other.

'secret_data_protected' from Equation 1 can be seen in relation to the function with the same name used in the UPEC property. However, it also has similarities with the *'equal_memory_state'* from Equation 2. This depicts the need for either constraining the memory of the instances to

equal content, or the usage of a symbolic formulation to protect these memory locations. This means that the data in question should be protected, or otherwise constrained, for the SoC methodology as well.

Lastly, the *'obs_soc_state'*, the *'arch_soc_state'* and the 'state_uniqueness' function can all be seen in connection with each other, as these all depict the states the difference should not propagate to.

In the theory describing the UPEC method, it is stated that the architectural states should be proven to remain equal. In the theory describing the UPEC-SoC method, the observable states should be proven to remain equal. This is the extended notion of the architectural states.

In an actual implementation in SVA, the notion of 'state_uniqueness' involves not only the locations the secret should not flow to but also all the microarchitectural states. This is what gives the possibility of observing P-alerts whenever a difference can be detected in any non-visible state other than the secret. This would thus involve an extension to the 'state_equivalence' function to also include the observable space. For 'state_uniqueness' this is realized by setting constraints on the system, similar to those seen in Listing 7. These are thus set to be proven. If this is not the case, the property is violated.

4

The metrics for the project

As this project intends to look at how well the UPEC method scales onto a new industrial system, some ground rules for how this is to be measured should be presented. The intention is also to look at how much knowledge one should have beforehand and how much time and effort is needed to successfully be able to implement the method. These are not values that can be given in exact measurements, but the intention is to be able to give some understanding of the procedure of applying UPEC on a new system while providing some information about how much work this takes.

4.1 Scalability in this experiment

Scalability in this case would refer to the methods' ability to be used in a range of systems. This means whether it is possible to detect these side channel vulnerabilities, and functional bug issues in rather large SoC systems, as well as in smaller cores.

4.1.1 Memory and time usage

Two of the statistics that are reported by the tool during the proof analysis itself are memory and time usage. These are rather concise measurements that illustrate how the method scales based on which parts of the system are analyzed and how much of the system at a time is analyzed. This also illustrates how large the time frame the system is tested for is. This will thus be taken into consideration, to see how long time the method spends in order to finish one analysis, and how much memory this requires.

An interesting scenario to look at here would be to analyze the whole system, including all the system states, to see how this affects these metrics. This would possibly apply a heavy workload to the tool, as this would contain all the complexity of the system.

It would also be fascinating to see how the increase in time for each proof, also referred to as unrolling, would affect the tool.

4.1.2 Number of constraints

All states in the system are supposed to be constrained to be equal in the system in the 'assume' part of the property. These, along with the observable states, should also be proven to stay equal in the 'prove' part of the property. These constraints are therefore a measure that can be somewhat representative of the size of the system. The larger the system, the more states need to be constrained to be equal. This could give some interesting data to combine with the timing and memory measurement. This could be used to see how long it takes for the proof to run with and without some modules black-boxed, and how much memory this takes. If some modules are black-boxed, the states within these modules do not have to be constrained to be equal between the two instances, meaning there are less states for the proof to consider. This would also give interesting results in regard to the comparison between the PULPissimo and the nRF54L15, and how the method scales from one system to the next.

4.1.3 Limitations to scalability in this experiment

Specifically, in this scenario, the scalability factor could be influenced by the complexity of the complete system and the complexity of the proofs that will be run. Since the method in question requires the setup of two near-identical SoC instances, it calls for a lot of constraints and otherwise complex logic, which could lead to a lot of pressure on the OneSpin tool itself. If the system the method is to be applied to is large, the proof could grow in computational complexity, and it could therefore end up being infeasible to scale the method onto such a large system. This could be shown in the sense that the proofs take too long to run through for them to be reasonable. The proofs could also use too much memory, such that the servers the experiments are run on would be significantly affected. If this is the case, the method could be unscalable onto the system under test.

Another point to be made regarding this topic is that the method calls for analyzing the system over several time points. This is because the possible propagation of the secret information might just end up in a program-visible state after quite some time. This particularly makes the scalability subject interesting. The generation of new states and changes in value over time significantly increases the computational complexity, which could deem the method non-scalable.

4.2 Evaluation of the required time and effort

For this specific implementation, the method will also be evaluated based on the effort needed to successfully detect these vulnerabilities. This will consist of looking at whether the implementation is straightforward, whether there is little need to inspect and discard false vulnerabilities, and how long time is needed by the test engineer to conduct the actual analysis.

4.3 Knowledge beforehand

Another metric that could be interesting to look at is how much knowledge the test engineer needs in order to successfully implement and actually use the method on a system. This would

not include any specific measures, per se, but rather a discussion based on the experiences gained during the experiment.

The UPEC team did provide a tutorial and templates that should aid in the application and usage of the method. Thus, it can be interesting to see whether these provide the prior knowledge needed for testers without previous experience with the method.

4.3.1 Security expertise

The literature describing the UPEC methodology suggests that no expertise regarding security vulnerabilities or attacks is needed to conduct an analysis of a system using the method. If this is the case, the method is highly applicable, as it can be used by a larger group of people. Not everyone has a lot of knowledge regarding intricate security vulnerability details, and methods that call for such knowledge narrow down the possible users. This also saves a lot of time if this is the case because someone conducting such an experiment would not need to spend a lot of time researching such topics.

4.3.2 About the system

Another subject that could be fascinating to investigate is how much knowledge is needed about the system itself in order to successfully perform an analysis. Is it realistic for someone who has not worked on the system beforehand to be able to test it for vulnerabilities, or is this out of the question? This could be an interesting evaluation of the method because if this is the case, there is a lot less time needed to research the system to be able to test it for vulnerabilities. In regards to industry, a design engineer who has worked on developing the system, might not be the one testing it for vulnerabilities.

This information can of course not be provided by the tutorial, as this is system-specific.

4.3.3 About the tool

In the methodology, the OneSpin tool is used. Thus, it can be interesting to see how much one would have to know about the tool itself, or specific formal verification techniques, that can be related to the OneSpin tool. If the method requires a lot of knowledge about the tool, or similar tools, it could be that a potential test engineer with no prior experience with OneSpin could have some issues.

5

Method for implementation

The following section will first explain the general method for implementing the UPEC method onto an SoC design that is to be checked for vulnerabilities. This therefore considers the extension UPEC-SoC, as described in section 3.2. In addition, the explanation will also include information regarding how the analysis should be conducted based on the provided theory, templates, and the insights acquired during the experiment.

Further, the actual implementation of the method onto the PULPissimo platform and the nRF54L15 is explained, which highlights how these procedures went and what considerations were made along the way.

This chapter also gives a more in-depth explanation of the UPEC property implementation, as introduced in subsection 3.4.1, based on experiences and understanding gained throughout the experiments. This details how the functionality actually is set up, and what the functions and constraints actually contain in order to realize the application of the method.

PULPissimo and nRF54L15 are both SoCs, thus, the extension of the UPEC method to also consider peripherals, UPEC-SoC is necessary.

5.1 The general approach

The OneSpin tool is used throughout, first to elaborate and compile the design to be tested, then to actually implement the UPEC method onto the design in question, and finally to analyze the system using the method to look for any potential vulnerabilities. The UPEC method is, as previously mentioned, based around two design instances, which function as a top component.

In practice, this top component is then the same system, just doubled. These two instances share primary outputs, and data, but have individual outputs. In order to set up this, a new top component is made. This can be autogenerated, using the scripts provided by the UPEC team.

The constraints, which make sure that all microarchitectural states remain equal to their counterparts in the second instance, can also be autogenerated using a script provided. The rest of the setup has to be done manually, as it varies from design to design. This includes the setup of the rest of the functions and constraints that surround the property, as introduced in Chapter 3. The idea behind the templates provided is to fill in the macros provided such that the property functions according to the workspace one is verifying the property on.

5.1.1 Black-boxing

As the systems one implements this method on can be rather large, one amendment to this issue is to black-box modules that lead to too much complexity during the proofs, as mentioned in the tutorial provided by the UPEC-team. This can thus help reduce the timing in which the experiments run, and the data usage necessary to search through the whole system. This is done by setting the compile option in OneSpin called '-black_box', and filling in which modules to black-box.

Thus, this procedure can alleviate the computational complexity significantly, as it ignores logic. This process can typically involve black-boxing large memory modules.

The reason this black-boxing of the modules does not affect the functionality of this experiment is that the important part of the system is the secret in question and its potential propagation path. The rest of the system should remain equal between the two instances in order to prove the unique program execution.

Since black-boxed modules are essentially viewed as a black box with inputs and outputs, any module in this method can be black-boxed. This is unless one is able to see the secret propagate onto the module's inputs during analysis. If this happens, the black-boxing of the module would have to be removed, and it would be necessary to analyze this closer. In order to remove the black-boxing of a module, the specific module would then have to be removed from the '-black_box' compile option. Then the entire DUT would have to be re-compiled and re-elaborated.

Which modules to black-box can be chosen based on the complexity of the modules in this case. In these experiments, this included looking at which modules generated the most constraints in the autogenerated constraints file. This is because these generate a lot of states that have to be proven to remain equal for the proof, which generates a lot of complexity for the tool to consider.

Another point regarding black-boxing is that if a module is black-boxed, its outputs have to be constrained to remain equal between the two systems for this methodology. The reason for this is to avoid any newly-introduced differences between the two DUTs to occur. This can happen due to weird behavior from the blacked-out modules.

5.1.2 Analysis

When the setup of the method is finished, the actual analysis of the system is the next step. This then involves reading in the SystemVerilog Assertion file that states the property itself, and then checking this with the appropriate approver. The approver used is approver1, as set up by the templates provided by the UPEC team. This approver is set up to be used eight times for one proof, by setting a 'check' option in the OneSpin tool. The usage of this introduces the possibility of unreachable counterexamples.

The outcome of the proof determines whether the system can be considered secure or not. If the result is a 'hold', the design is secure. However, whenever a 'hold_bounded(n)' is the result from the property check, meaning that the proof is only valid up until this specific point, this could indicate a possible vulnerability in the system. The following approach is to analyze the resulting counterexample to see whether it might be unreachable. If the counterexample is reachable, this indicates an actual P-alert.

If the counterexample is an actual P-alert, this still does not indicate that the propagation of the secret could impact states that are program visible. Unless the counterexample displays that a program-visible state, such as a primary output or an unprotected memory location, shows any difference, it might be just a P-alert. If the counterexample shows a difference in such a program-visible state, this is considered an L-alert, and thus also a vulnerability.

Procedure for unreachable counterexamples

The approach whenever the tool provides a counterexample is to examine this thoroughly and see whether the provided scenario actually can occur in the design, or if it should be regarded as unreachable. This is to see whether it seems like the counterexample could be caused by an unreachable starting state or not. This includes thoroughly analyzing the counterexample waveform to look for any strange behavior from any part of the system.

Another option could be to replicate the system behavior in a simulation environment. This would need a setup of a simulation environment and code to try to replicate the behavior leading to the counterexample, to see whether it can actually occur in reality.

Disprovers can also help with determining whether a counterexample is reachable or not, as they start the proof in a reachable state. However, they might also result in a vacuous result, which then would not give any additional information.

If the counterexample is indeed unreachable, this can be solved by fixing the constraints, which are most likely causing issues and thus also the reason for the generation of the false counterexample. These thus have to be examined and tested to see whether any of them could be the issue or not. This can be done by removing some constraints, or changing some, and then running the same proof to look for any differences or clues otherwise as to what the issue might be.

Unrolling to investigate a P-alert

If the check results in a P-alert, this should be examined further. One example could be that there is a difference between register 'x' in the two instances, in other words, `inst1.reg_x != inst2.reg_x`. If this counterexample is given at a time t=2, the next step is to check this further, by unrolling. Thus, the aim is to increase the time step, to look at whether this difference could propagate through the system, and end up in a register that can be visible or accessible by a user or a program.

5.2 PULPissimo version 7.0

The PULPissimo platform, as described in Chapter 2, is an open-source design. This system has previously been tested for vulnerabilities using the UPEC method, as detailed in Müller et al. (2021). This is also explained in section 2.3 in this thesis.

As a method of getting familiar with the UPEC technique, the intent was to implement it onto the PULPissimo platform. The objective behind this was to gain an understanding of the setup and usage of the method, so that the process of implementing it onto a new system would be somewhat simplified.

As the previous implementation of UPEC on the PULPissimo platform was done in 2020, on the 4.0 version, the platform has gone through some changes up until the 7.0 version that this experiment is conducted on.

One of the main differences in the platform was that the PMP unit had been removed from the CV32E40P core.

Other than this, there have also been some other changes in the individual modules, such as which signals should be regarded as states, name changes in these signals as well as general optimizations of the subsystems. Since there have been changes, there could potentially also be newly-introduced vulnerabilities, as these theoretically could occur due to simple changes in the RTL.

This section will give insight into the implementation process and how this was accomplished. It will then give details surrounding what considerations had to be made for the design, and how the actual testing is performed. This section also shows how the theoretical knowledge surrounding the UPEC method was tied together with and used alongside the tutorial provided by the UPEC team at Kaiserslautern for the PULPissimo system.

5.2.1 Setup

The approach used for this experiment was to use the provided templates as a startup point, and then fill these in to fit with the PULPissimo system. The setup also included a lot of trial and error to get the system itself up and running, and then continuing on to the application of the method.

The final set of files used in the experiments is provided as a .zip folder. This is further detailed in Appendix C. This set of files includes the experiment conducted to further investigate the uDMA P-alert, as detailed in section 6.1.1. This does not, however, include the automated scripts, as these are not efforts achieved from these experiments.

Issues encountered

During the setup of the system, there were several issues that were encountered and debugged.

The first issue encountered was actually an issue with the PULPissimo system itself. The problem was that during the setup of the system, and the work to implement the method, a fatal error was reported in the message log. This error reported that 'Internal error 62 in MSL has occurred', and to notify support. This was fixed by trying to black-box some modules to look at where the issue could reside. The issue was a few lines in one module named `axi_to_axi_lite`,

where the 'assume' keyword was used instead of 'assert'. This was thus fixed by swapping this out.

Another issue was the PMP setup. Initially, the CV32E40P core was used, and even though this core does not feature a PMP, this was not explicitly explained anywhere on the GitHub project site. The PMP was removed when the core was updated from RI5CY to CV32E40P. The system still included variables to enable and use the PMP but without the functional PMP. This was discovered, and thus the core was swapped in favor of the Ibex core instead, which has a PMP unit. This gave the opportunity to set up a secure environment in the system, which is necessary for the UPEC methodology.

Further along, during the analysis phase of the process, the system ended up with a 'vacuous' result for all the proofs that were run. This was attempted debugged by removing some constraints, changing the proof itself, and generally investigating different modules. Since the result was vacuous, there is not a lot of information given about what could be the issue, nor how this can be debugged.

The issue turned out to be a problem with the autogenerated state constraints. Some signals used in the state constraint functions had child signals that were not used or initialized. This could maybe have resulted in OneSpin listing them as unknown signals. Since these are constraints saying that 'signal 1' in 'instance 1' should be equal to 'signal 2' in 'instance 2', a situation where an unknown should be equal to an unknown arises. Since these are unknowns, there is no way to tell the value for sure. Therefore, setting the two unknowns to be equal would be considered false, and could therefore be the issue resulting in the vacuousness. This was then fixed by adding the missing signals, which in turn also helped the problem.

Black-boxing the memory modules

For the setup and compilation of the system, the two memory modules `tc_sram` and `generic_rom` were black-boxed. The reason for this was that the tutorial provided suggested that large memory arrays could be black-boxed to ease system complexity. The reason this could be done was that the memory content should be equivalent between the two instances. This can still be accomplished by making sure that the outputs of the memory remain equal, by setting constraints. What happens within the memory modules is not interesting in the methodology, as the memory instances should be equivalent regardless.

Changes in top module

The top module file in the setup was changed away from the autogenerated top level. The reason for this is that the generated file is based on the top-level module in the design, which has 'inouts', which can pass information in both directions. Since the inputs to the system are supposed to be constrained to be equal, the autogenerated file using inouts as basis would need two signals per input pair, and then have these constrained to be equal.

An alternative, which was used for this setup, is to go down one level, and use the 'soc_domain' in the PULPissimo as a basis for the new top module instead, this does not need the extra constraints. There is no functional difference between the two choices, and thus the latter option was

chosen in this setup.

5.2.2 Final property explained

This section details how the functionality behind the property actually is set up, and what the functions have to contain to be able to use the UPEC method. This section, therefore, gives a more thorough explanation as to what the functions actually contain in a real setup. This goes a bit further than subsection 3.4.1, which introduced what the functions represent, without showing the inner workings of the property.

The final property that ended up being used for testing can be seen in Listing 8, with a Tmax of 3. A Tmax of 3 is used for the first test to further investigate any P-alert that has been detected in T=2. This listing also shows the constraints, which will be explained as well.

All functions and constraints used in the main property can be studied in Appendix A. This appendix shows the final 'property.sv' which was used in the further testing of the uDMA, elaborated in section 6.1.1. The property itself is slightly changed for the figures seen in Appendix A in comparison to Listing 8. However, the surrounding constraints and functions as they are detailed below remain the same.

Looking further into Listing 8, from the top, the 'instr_constraint', seen in detail in Figure A.4, is used to exclude any additional P-alerts that could enter by the instruction interface. This is done by setting the inputs of the module to be equal between the instances. The reason for this is that it is arguable whether the path to instruction memory should be checked or not, as what instructions are executed is not really secret information, and could thus likely not give any interesting results.

The 'unpriv_constraint' can be seen in conjunction with the 'unpriv_mode' in Listing 5. The difference in Listing 8 is that it is set up as a constraint and thus lasts through the whole proof. This is used to make sure that the privilege level of the system is set to 'user mode'. This is because the vulnerabilities that are interesting to look at are those that can occur in this scenario. The 'unpriv_mode' function can be seen in Figure A.1. It simply specifies the value of two variables such that the system mode is set to 'unprivileged'.

Another constraint is the 'mem_constraint', which assumes the property 'mem_data_array_blackboxing_constraint'. This can be seen in Figure A.4, and here, there is an evaluation of where the secret address resides. Based on this, the input address of the module that includes the address of the secret is changed in one of the instances. This creates a difference within the secret address, and thus also a possibility for the propagation of this difference.

As previously explained, the memory was set up in such a way that the two modules `tc_sram` and `generic_rom` were black-boxed. The rest of the 'mem_data_array_blackboxing_constraint' property also assumes the other memory location outputs from the black-boxed modules, which are not considered for the secret, to stay equal throughout the proof. This is done by constraining them to be equal. The reason for this is that because these memory locations are black-boxed, they would result in more differences between instances, which is not desirable. Therefore, this is done to avoid any additional, non-realistic, P-alerts from occurring. The potential secret data locations, which should differ, are therefore commented out of the code. These should not be

```
instr_constraint: assume property (@(posedge miter_top.clock)
 ↪  instruction_input_equal);

unpriv_constraint: assume property (@(posedge miter_top.clock)
 ↪  unpriv_mode_constraint);

mem_constraint: assume property (@(posedge miter_top.clock)
 ↪  mem_data_array_blackboxing_constraint);

symbolic_address_constraint: assume property (@(posedge
 ↪  miter_top.clock) symbolic_secret_address);

property UPEC;
    t ## 0 state_eqivalence() and
    t ## 0 no_secret() and
    during_o(t,2 t,3, no_secret()) and
    during_o(t,0 t,3, pmp_memory_protection_state_static()) and
    t ## 3 p_alerts_blocking_clause() and
implies
    t ## 3 state_uniqueness());
endproperty


UPEC_assertion: assert property (@(posedge miter_top.clock)
 ↪  disable iff(reset) UPEC);
```

Listing 8: Final UPEC property

constrained to be equal in the two instances.

This evaluation of the secret address location leads to the 'symbolic_address_constraint', which actually sets up the secret address. This is set up to be selected from either one of two possible addresses, as can be seen in Figure A.3, which shows the property assumed. These two are therefore chosen as the possible secret data locations in this setup. The two addresses are either one in the 'Private bank 1' memory or in the 'Tightly Coupled Data Memory bank 0'.

Both are instances of the memory module `tc_sram`. This constraint also makes sure that for each positive edge of the clock, which the property triggers on, the previous secret address is equal to the secret address on the current clock flank. The reason for this is that the secret address should not change halfway through the proof. This is done with the usage of the '$past()'-keyword, and thus also removes the possibility for new, unrealistic alerts to occur during one single proof.

'State_equivalence' is the same here as in Listing 5, which states that, at startup, all microar-

chitectural states should remain equal. This is done with constraints for each state between the two system instances. An example displaying this type of constraint can be seen in Listing 7.

'No_secret', as can be seen in Figure A.2, can look a lot like the 'mem_data_array_blackboxing_constraint'. The difference is that the 'mem_data_array_blackboxing_constraint' is assumed for the entirety of the proof. The constraints in this property do not include setting the chosen secret data location to be equal between instances, because this content should differ. The 'no_secret' function, however, constrains all the memory location outputs to be equivalent for the two instances, and it is used twice in the property itself.

The first usage of 'no_secret', at t##0, is to avoid any secret propagation at time 0, the very beginning of the proof. The reason for this is that this type of scenario is not realistic. For this to happen, the secret would have been accessed before the proof started, and this situation would thus lead to non-realistic P-alerts. This can be seen in conjunction with the 'no_pending_secret_access' function in subsection 3.4.1.

After this time point, there is a pause in the usage of 'no_secret'. Here, the rest of the memory location outputs are still constrained to be equal in the 'mem_data_array_blackboxing_constraint', but the secret location itself is not, meaning this can cause propagation.

The other usage, starting at T=2, is after the P-alerts have occurred, and been collected. This, therefore, avoids that after T=2 and all the way to Tmax=3, there is no possibility for multiple secret propagations.

The 'no_secret' function, therefore, makes sure that the outputs of all the memory modules are constrained to be equal, except for at T=1. They have now become new primary inputs to the system. This way, the new inputs to the system are guaranteed to be equal, and this avoids any extra state differences stemming from weird behavior from the black-boxed memory.

The access control is to simulate an environment where an ordinary user-level process is not able to access the secret data locations and get hold of any confidential information. This is done by setting up the PMP in the environment, which is done in the 'pmp_memory_protection_state_static' function. The PMP is configured such that the PMP entry does not permit read, write, or instruction execution access to the two chosen potential secret data locations. The setup is shown in Figure A.1. The addresses the PMP should protect are thus set according to the secret addresses. Therefore, the PMP protection locations line up with the options for the secret data location, which is the 'Private bank 1' address or the 'Tightly Coupled Data Memory bank 0' address.

The 'p_alerts_blocking_clause' is used to avoid any additional P-alerts, other than the one currently looked into, from interfering. This can be seen together with the 'instruction_input_equal' function in Figure A.4. For this particular test, all the P-alerts stemming from the HWPE module have been commented out. This is due to the fact that this module was abstracted away during the alternative route to test the uDMA. The P-alert stemming from the uDMA has also been commented out, as this is the one that was further investigated in this particular proof.

Finally, the 'prove' part of the property, after the implies keyword, shows the usage of the 'state_uniqueness' function. This function also sets states to be equal in the two instances, as in the example in Listing 7. The 'state_uniqueness' function includes all the microarchitectural states of the system, from the 'state_equivalence', along with the observable space. The observable

space includes the outputs of the system, and the unprotected memory. This is set to be proven at Tmax=3. If it cannot be proven, propagation onto other states than the secret location has been detected at this time point.

The 'prove' part can also be changed to prove equivalence for only specific observable states, for instance.

5.2.3 Analysis

The analysis part then consisted of running the proof and looking for any possible P-alerts, as indicated by any 'hold_bounded(0)' result given by the tool. If there were any, these had to be investigated further to look for any migration into user-visible space.

As the PULPissimo is an SoC, the process considers the UPEC-SoC methodology. Therefore, user-visible space in this scenario would include any primary outputs and unprotected memory regions.

Finding the P-alerts

The P-alerts were thus gathered by running the proof with the 'state_uniqueness' macro in the 'prove' part of the property. Then, the P-alerts were all collected into a 'blocking_clause', which prevented them from returning while looking for new ones. This process was continued until the proof returned a 'hold'. This indicated that there were no new P-alerts, and these could consequently be investigated further.

Differentiating the P-alerts from the L-alerts

Whenever all the P-alerts had been detected, these had to be investigated further, to see whether they could lead to an L-alert or not. This process consisted of unrolling the property further and looking at whether this difference between the two instances propagated to somewhere that was considered observable. To analyze a specific P-alert, this was removed from the blocking clause.

The way the unrolling was done was by running the proof for a longer time frame every time the proof returned a 'hold_bounded(n)' result. Each time there was a 'hold_bounded(n)' result, the counterexample was then investigated to see whether the P-alert had propagated onto visible states, if so, it was to be considered an L-alert.

There are two possible outcomes to increasing the time window to investigate a P-alert closer. Either the property is unrolled until the propagated difference affects any location that can be regarded as observable, or there are no new P-alerts to be detected, meaning that there is no further propagation.

Fanout optimization

During the process of locating all the P-alerts, the 'state_uniqueness' macro was used, meaning that all the microarchitectural states and also the observable states in the system were considered. This increases the size of the proof significantly, and thus also puts quite a strain on the tool. It also took a lot of time for the proof to finish.

The unrolling done to look at later clock cycles is essentially generating multiple copies of the design instances, which also requires a lot of resources.

Therefore, as discussed with the UPEC team, one possible remedy was to not include all of these states in each proof, but rather just look at the states in the immediate possible propagation paths for the P-alerts. These are also referred to as fanouts. This would thus mean only testing for, and trying to prove the property for those states the difference could propagate to. This excludes all the states that are not in this immediate propagation path, which in turn saves a lot of computation, and also time, while running the test.

If the difference is not able to propagate onto any of the P-alert location's fanouts in the next clock cycle, the P-alert is therefore not able to propagate any further at all.

This method consisted of adding the possible propagation path within the next clock cycle to another macro, and then considering this one instead of the complete 'state_uniqueness' macro when unrolling the property further. Then, when this analysis resulted in a new 'hold_bounded(n)' and thus also provided a new counterexample, the process would be repeated to look at the fanouts for the next clock cycle. This was unless the counterexample showed that there was a difference in any observable states. If that was the case, then this should be considered an L-alert, lest the counterexample could be proven to be unreachable.

The process of finding the possible propagation path can be seen in Figure 5.1. This shows the 'fanout view' in the OneSpin tool for a state called 'buffer'. The two states highlighted in blue show the propagation paths of 'buffer' that could occur one clock cycle later, as indicated by the timestamps. These are 'data_q' and the same state 'buffer', again, but at a later time. The value of 'buffer' is captured at time 7, while the value of 'data_q' and the new value of 'buffer' is captured at time 8, meaning that the alert would need one clock cycle to propagate from 'buffer' to 'data_q', or back to 'buffer'.

Fanout Nets	Value	Kind	Module	Hierarchical	Time
buffer	{f000_0...	Net [State]	io_generi...	inst2/pul...	7
buffer	{f000_0...	Net [State]	io_generi...	inst2/pul...	8
data_o	f000_0008	Output	io_generi...	inst2/pul...	7
data_o	f000_0008	Output	io_tx_fifo...	inst2/pul...	7
s_spi_cmd	f000_0008	Net	udma_sp...	inst2/pul...	7
src_data_i	f000_0008	Input	udma_dc...	inst2/pul...	7
src_data_i	f000_0008	Input	cdc_fifo_...	inst2/pul...	7
src_data_i	f000_0008	Input	cdc_fifo_...	inst2/pul...	7
data_q	{6000_0...	Net [State]	cdc_fifo_...	inst2/pul...	8

Figure 5.1: Example of how to look for fanout states reached within one clock cycle

The 'prove' part of the property presented in Listing 8 should now instead declare that these states should remain equal to their counterpart in the other instance when analyzing the next clock cycle.

Such an example can be seen in Listing 9. This shows a scenario where an imagined P-alert has been detected in a state at Tmax=2.

The fanout to be tested is named `inst1.pulp_soc_i.soc_peripherals_i.i_udma.i_spim_gen[0].i_spim.u_fifo.i_fifo.buffer`, which is the same state as in Figure 5.1. The property is thus set up to prove that this state remains equal to its counterpart in

'inst2' for the next clock cycle. If this is not the case, a 'hold_bounded(n)' will be given by the tool, along with a counterexample.

```
property UPEC;
    t ## 0 state_eqivalence() and
    t ## 0 no_secret() and
    during_o(t,2, t, 3, no_secret()) and
    during_o(t,0, t, 3, pmp_memory_protection_state_static())
 ↪  and
    t ## 3 p_alerts_blocking_clause()
implies
    t ## 3 (inst1.pulp_soc_i.soc_peripherals_i.i_udma.i_spim_g
 ↪  en[0].i_spim.u_fifo.i_fifo.buffer ==
 ↪  inst2.pulp_soc_i.soc_peripherals_i.i_udma.i_spim_gen[0].i_
 ↪  spim.u_fifo.i_fifo.buffer);
```

Listing 9: Example of a property considering a fanout state in the 'prove' part of the property

Alternative route

Other processes to find any potential propagation to any observable location were also discussed during conversations with the team at Kaiserslautern, and thus also tested. One example of such an alternative route could be to swap the 'prove' part in the property to no longer include 'state_uniqueness' or any fanouts reachable within one clock cycle, but rather just try to prove that the observable states remain equal throughout.

For the first test when running this type of proof, this would probably return a 'hold', as the difference in behavior has not yet been able to affect any other states. Thus it has not yet been able to propagate throughout to reach observable space. However, as the system is unrolled, and the timeframe is then increased, the difference might be able to propagate throughout and eventually end up in any of the states that should be proven to be equal for the system to remain confidential. This scenario would therefore result in a 'hold_bounded(n)', and also provide a counterexample that can be analyzed.

Black-boxing to remove complexity

While the alternative technique was attempted, it was discovered while running at a Tmax=4 that this was computationally expensive for the tool. This was because the proof took a long time to finish. In order to reduce the time spent, it was attempted to black-box further modules in the system. This was done by looking at which modules had created a lot of state constraints in the 'state_uniqueness' or 'state_equivalence' functions. The thought behind this was that if a lot of these states were abstracted away from the systems, the proof would have fewer constraints to consider.

The method was then to try to black-box modules that seemed to be causing a lot of complexity, and to remove this black-boxing if the secret propagated to the input of the black-boxed module. Further, the process was to attempt to add more modules to the black-box compile option and re-compile the system each time. This was done multiple times in order to reduce the test time for Tmax=3.

The reason Tmax=4 was not chosen was that Tmax=3 already took less time, meaning it required a shorter wait time for the results after each run. It was desirable to reduce this test time, such that the runtime for the next test with Tmax=4, and tests with even larger max times, would also be reduced. The final black-boxing that was used included the modules seen in Table 6.1. This table also shows how much the proof time was reduced for each new black-boxed module.

Test using disprover

A third option that also was attempted, besides the alternative route and the fanout method, was to use disprovers instead of approvers. These would 'reverse' the proof in that instead of getting a 'hold' or a 'hold_bounded(n)' this returns either a 'fail' or a 'hold_bounded(n)', as explained in subsection 2.5.6. This is because the disprovers are not able to prove an assertion, but rather disprove it. Thus, these are able to prove that the counterexample is reachable whenever they produce a 'fail'.

The disprovers were used by setting an option in OneSpin to use the disprovers instead of the approvers. This test then also included just the observable states in the 'prove' part. Whenever the tool was not able to prove that these remained equivalent between the two instances, the proof would result in a 'fail' instead.

Because the disprovers are able to provide a reachable counterexample, these can be used to prove that the propagation of the L-alert is realistic behavior from the system. This is thus a good alternative to having to manually analyze the counterexample.

However, it is also possible that these could return a 'vacuous' result, which would not tell anything about whether the counterexample is reachable or not. This could be due to a false counterexample, but it could also be caused by issues with the constraints in the system. In this case, manual inspection is again required.

5.3 nRF54L15

The work on nRF54L15 ended up being delayed for quite some time, as the startup point with the setup and analysis of PULPissimo ended up being more extensive than first anticipated. When the timing was right to begin implementing the UPEC method onto this environment, the setup of the platform, without the UPEC setup, in OneSpin also took an extended amount of time. This meant that this experiment had to be let go of.

The intention was to be able to use the UPEC method to analyze this system as well, so the process of setting up the nRF54L15 in OneSpin was begun. However, this is a system still under development, and it is a complex SoC with many analog blocks, a fine-grained clock, and power gating. Thus, there are a lot of technicalities that need to be considered in order to successfully be able to set up such an SoC system in OneSpin.

The issues with getting the setup up and running resulted in the nRF54L15 using around 16 hours to go through elaboration and compilation in the OneSpin tool, which was not feasible for running experiments on. Also, it only passed elaborate and compile after setting some specific compile options that ended up turning unstable signals to unknown signals instead.

The new top module needed for UPEC analysis of this system was autogenerated, but before trying to generate the constraints for the setup, the intention was to work towards reducing the elaborate and compile time. The reason for this, is that the state constraints have to be based on the setup with the new top module. Therefore, the system would have to be re-compiled and re-elaborated again in order to generate the state constraints.

Working towards trying to reduce the setup time was attempted by black-boxing some modules, to try to test the method on a smaller part of the system. However, this testing did not go any further, because it was soon discovered that there were some issues with the clock and reset generation in the system. This would then also have to be fixed before being able to test the system using the UPEC method, as this could lead to unexpected system behavior.

This resulted in the conclusion that it was best to shelf this experiment for now, and rather focus on the PULPissimo instead.

6

Results

Provided in this chapter are the concrete results that the experiments led to. This shows the results of the further investigation of the P-alerts. Additionally, an L-alert was detected, which is explained further in this section.

Otherwise, the results presented in this chapter include statistics regarding the scalability of the PULPissimo system. This will consider how many constraints are needed to run the proof, as this is dependent on the number of states in the system, and thus also its size. Other scalability results that are interesting to report are how long some of the proof run times turned out to be. It is also relevant to look at how proofs can be affected by black-boxing, or by the usage of fanout registers in the prove part. The memory usage for the proofs, and how this differs based on the optimizations, is also reported in this section.

6.1 PULPissimo

The implementation of the UPEC method onto the PULPissimo was accomplished, and thus also resulted in some specific data for both vulnerabilities and for scalability of the method onto the newest version of the system.

6.1.1 Alerts discovered

The system was analyzed thoroughly using the UPEC method, which resulted in the discovery of P-alerts. These were then further investigated and thus also resulted in an L-alert.

P-alerts

The P-alerts, that were given by the proof throughout the analysis of the system, can be seen in Figure 6.1. These are gathered in the 'p_alerts_blocking_clause', which is used to avoid the same alerts from being generated again when re-running the proof to look for new ones. These were found by running the property found in Listing 8, just with a Tmax=2, and without the 'p_alerts_blocking_clause'. A Tmax=1 did not give any P-alerts, as this is the window where the

secret addresses are unequal, and the propagation starts.

These were alerts that could possibly propagate further and were thus also investigated. The six P-alerts stemming from the HWPE module can be grouped together, and looked at as one, as they originate from the same module. This concept is the same for the two P-alerts from `axi64_2_lint32`. Thus, propagation from one of the locations is enough to consider the module as vulnerable.

```
55 function automatic p_alerts_blocking_clause();
56    p_alerts_blocking_clause = (
57         // HWPE
58         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.i_fifo_incoming.fifo_registers ==
59         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.i_fifo_incoming.fifo_registers &&
60         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.i_fifo_incoming.fifo_registers ==
61         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.i_fifo_incoming.fifo_registers &&
62         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.tcdm_master_r_data_q ==
63         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.tcdm_master_r_data_q &&
64         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.i_fifo_incoming.fifo_registers ==
65         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.i_fifo_incoming.fifo_registers &&
66         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.tcdm_master_r_data_q ==
67         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.tcdm_master_r_data_q &&
68         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.tcdm_master_r_data_q ==
69         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.tcdm_master_r_data_q &&
70
71         miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.load_store_unit_i.rdata_q ==
72         miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.load_store_unit_i.rdata_q &&
73         miter_top.inst1.pulp_soc_i.i_lint_jtag.dbg_module_i.i_dbg_lint.lint_biu_i.data_out_reg ==
74         miter_top.inst2.pulp_soc_i.i_lint_jtag.dbg_module_i.i_dbg_lint.lint_biu_i.data_out_reg &&
75         miter_top.inst1.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_write.data_r_rdata_q ==
76         miter_top.inst2.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_write.data_r_rdata_q &&
77         miter_top.inst1.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_read.data_r_rdata_q ==
78         miter_top.inst2.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_read.data_r_rdata_q &&
79         miter_top.inst1.pulp_soc_i.i_dm_top.i_dm_csrs.sbdata_q ==
80         miter_top.inst2.pulp_soc_i.i_dm_top.i_dm_csrs.sbdata_q &&
81         miter_top.inst1.pulp_soc_i.soc_peripherals_i.i_udma.i_udmacore.u_tx_channels.r_data ==
82         miter_top.inst2.pulp_soc_i.soc_peripherals_i.i_udma.i_udmacore.u_tx_channels.r_data
83
84         );
85 endfunction
```

Figure 6.1: All P-alerts collected in PULPissimo

All the P-alerts listed in the blocking clause were further investigated by trying to prove no further propagation onto their fanouts within the next clock cycle, Tmax=3. All these tests were done without any additional black-boxing, other than the black-boxing of the module `tc_sram` and the `generic_rom`. These two modules were also black-boxed for the setup of the system, as this eased the system complexity.

The P-alert detected in the core, `miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.load_store_unit_i.rdata_q`, in Figure 6.1, only had two states that could be possibilities of further propagation. These two were tested to see if they remained equal to their counterpart in the clock cycle after the one where the alert was detected. The test resulted in a 'hold', proving that there is no additional propagation of this alert onto new states, meaning this alert does not evolve into an L-alert, and therefore it is not an issue with the design.

The same tests were also conducted for all the other modules that generated P-alerts as well and their respective fanouts. Most of the locations only had one or two possible further propagation paths. All the tests ended up holding, except for the test for the uDMA. This module also had 14 possible further propagation paths for the P-alert.

The result of these tests can be seen in Figure 6.2, showing how only the uDMA P-alert propagated further, as indicated by the 'hold_bounded(0)' result from the fanout test.

Name	Proof Status	Runtime
!	! <any status>	
Assertions		
Constraints		
Properties		
sva/checker_bind/ops/UPEC_core_assertion	hold	01:06:49
sva/checker_bind/ops/UPEC_hwpe_assertion	hold	00:07:32
sva/checker_bind/ops/UPEC_i_dm_assertion	hold	00:10:03
sva/checker_bind/ops/UPEC_interconn_assertion	hold	00:05:13
sva/checker_bind/ops/UPEC_jtag_assertion	hold	00:04:07
sva/checker_bind/ops/UPEC_udma_assertion	hold_bounded (0)	00:03:53
SVA Functions		
SVA Named Properties		
SVA Sequences		

Figure 6.2: Result from testing the further fanouts for all the six P-alert locations. Tmax=3.

L-alert in uDMA

After the detection of the P-alert from the uDMA module, `inst1.pulp_soc_i.soc_peripherals_i.i_udma.i_udmacore.u_tx_channels.r_data`, found at Tmax=2, this was tested further using the fanout method and Tmax=3. This resulted in the discovery that the alert can propagate even further onto new states. Thus, this P-alert was explored to a greater degree using the alternative strategy. This method includes trying to prove that the observable states remain equal between the instances, as elaborated in section 5.2.3.

The function 'udma_obs_space' which was used in the 'prove' part for this test can be seen in Figure 6.3. This shows how the unprotected memory space locations, and the outputs for the top module, should remain equal for the whole duration of the test in order to prove confidentiality. If this is not the case, however, and a difference can be detected in any of the locations for this function, this could be an L-alert and hence also a vulnerability.

For the unprotected memory, the inputs of the memory are considered, as these modules are black-boxed. However, if the alert propagates onto these inputs, the black-boxing of the module in question has to be removed to further investigate.

The function shown in Figure 6.3 does not contain all the outputs from the top module, but rather a selection based on analysis of the fanout of the P-alert location and where this could potentially propagate. This could have included all the observable states in the system, but only a selection was chosen in order to reduce the proof complexity.

At first, this P-alert was researched further without any additional black-boxing, and increasing Tmax to 4. Without additional black-boxing, this test took 10 hours and 27 minutes. Thus, additional black-boxing was set up to reduce proof-complexity.

The black-boxed modules included those seen in Table 6.1. This shows what modules were added to the black-box compile setting for each run and how long time each proof ran for. Thus, the first test included only `fc_hwpe` as the additional black-boxed module. The second test still included `fc_hwpe`, but also `apb_adv_timer` as well as `apb_fll_if_i` and so forth. All these

```
function automatic udma_obs_space();
        udma_obs_space =(
                //unprotected memory space
                miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.wdata_i      == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.wdata_i &&
                miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.wdata_i      == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.wdata_i &&
                inst1.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.wdata_i                == inst2.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.wdata_i &&
                inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.wdata_i                == inst2.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.wdata_i &&
                inst1.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.wdata_i              == inst2.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.wdata_i &&
                inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.wdata_i              == inst2.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.wdata_i &&

                //Outputs in top module
                inst1.i2c_scl_o                                                 == inst2.i2c_scl_o &&
                inst1.hyper_ck_no                                               == inst2.hyper_ck_no &&
                inst1.uart_tx_o                                                 == inst2.uart_tx_o &&
                inst1.spi_sdo_o                                                 == inst2.spi_sdo_o &&
                inst1.sdio_clk_o                                                == inst2.sdio_clk_o &&
                inst1.i2c_sda_o                                                 == inst2.i2c_sda_o &&
                inst1.spi_oen_o                                                 == inst2.spi_oen_o &&
                inst1.spi_csn_o                                                 == inst2.spi_csn_o
);
endfunction
```

Figure 6.3: udma_obs_space function used in 'prove' part of uDMA-property

	Name of the added black-boxed modules	Time-per-proof
1st test	fc_hwpe	1 hour and 8 minutes
2nd test	apb_adv_timer, apb_fll_if_i	1 hour and 5 minutes
3rd test	axi_xbar_intf, jtag_tap_top	15 minutes
4th test	apb_gpio, apb_soc_ctrl, apb_timer_unit	11 minutes
5th test	axi_cdc_src, axi_cdc_dst, axi_lite_to_apb_intf	8 minutes

Table 6.1: Black-boxed modules used in uDMA-propagation test

tests were run for Tmax=3, with the intention of getting this proof to run in under 10 minutes. This was because if this proof time was reduced, then the time to run the same proof, with Tmax=4 or higher would hopefully also reduce.

The reason behind these choices in black-boxing was that these modules did not seem to be able to interfere with the possible propagation path from uDMA to any observable states. This was investigated by looking at the possible fanouts over a larger period of time. Another reason was that this seemed as if it would lessen the complexity significantly, as these modules generated a lot of constraints for the 'state_constraints' file.

Because the run with Tmax=3 took 8 minutes with the final modules added to the black-box compile setting, the proof was set up to run for Tmax=4. This took just under one hour, and thus a proof for Tmax=5 was set up to run, to check whether this would take long.
The latter test resulted in a 'hold_bounded(0)' and took approximately 30 minutes.

The 'hold_bounded(0)' result thus indicates that one of these observable states no longer remains equal to its counterpart when investigating for time=5. This equates to an L-alert. Before

this time, the same test resulted in a 'hold', indicating that the issue had not propagated far enough for it to be considered an L-alert yet.

Part of the counterexample that was provided by the tool can be seen in Figure 6.4 and in Figure 6.5. The first figure shows the paths of all the states that have been affected by the propagation, while the latter shows how these change values over the analyzed time window. The states which were affected by the secret were gathered using the '.tcl' script provided by the UPEC team. The two figures illustrate that the P-alert had propagated from the secret memory location output, `inst.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o`, which relates to the secret address having been chosen to be the one from the 'Tightly Coupled Data Memory bank 0'. This is because the path relates to this module.

From here, it further affects the uDMA module, which gave a difference in the two instances at t##2, to finally end up in `spi_oen_o`, at t##5, which is marked with the yellow vertical line in Figure 6.5. `Spi_oen_o` is an enable signal, and an output in the top module, and thus also an observable state.

The propagation path from the P-alert can also be observed in the fanout of `inst1.pulp_soc_i.soc_peripherals_i.i_udma.i_udmacore.u_tx_channels.r_data`. This can be seen in full in Appendix B in Figure B.1, which shows all the states which are also given in Figure 6.4. This illustrates how the difference can propagate through these states clock cycle by clock cycle. The states that can be found in the counterexample Figure 6.4, are highlighted in blue in the fanout view of Figure B.1.

Secret Propagation
inst1/pulp_soc_i/l2_ram_i/CUTS[0]/bank_i/rdata_o
inst2/pulp_soc_i/l2_ram_i/CUTS[0]/bank_i/rdata_o
inst1/pulp_soc_i/soc_peripherals_i/i_udma/i_udmacore/u_tx_channels/r_data
inst2/pulp_soc_i/soc_peripherals_i/i_udma/i_udmacore/u_tx_channels/r_data
inst1/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_cmd_fifo/i_fifo/buffer
inst2/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_cmd_fifo/i_fifo/buffer
inst1/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_dc_cmd/i_cdc_fifo/i_src/data_q
inst2/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_dc_cmd/i_cdc_fifo/i_src/data_q
inst1/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_dc_cmd/i_cdc_fifo/i_dst/i_spill_register/spill_register_flushable_i/gen_spill_reg/a_data_q
inst2/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_dc_cmd/i_cdc_fifo/i_dst/i_spill_register/spill_register_flushable_i/gen_spill_reg/a_data_q
inst1/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_txrx/r_spi_mode
inst2/pulp_soc_i/soc_peripherals_i/i_udma/i_spim_gen[0]/i_spim/u_txrx/r_spi_mode
inst1/spi_oen_o
inst2/spi_oen_o

Figure 6.4: All the names of the states affected by the propagation of the secret in the first L-alert

This L-alert was also tested using a disprover, which resulted in a 'vacuous_hold'.

Because of this, it was also attempted to debug the constraints by removing some and changing some, as these most likely could be causing issues within the proof. This was done to look for any inconsistencies in the system. However, this gave no indication of what any possible error with the constraints could be.

Also, the provided counterexample was investigated to look for any invalid or otherwise strange behavior in the system, without being able to find any traces of such demeanor. If this had been found, it could indicate that the counterexample was unreachable.

The P-alert from uDMA also propagated to another observable state, `Spi_sdo_o`, another output from the SPI module in the system. `Spi_sdo_o` is the 'slave data output' that transmits the requested data back to the master in the SPI module. This is also an output in the top mod-

Figure 6.5: All the times at which the states for the first L-alert got different values compared to their counterparts in the other instance

ule, and is therefore also categorized as an observable state. This L-alert was detected using the fanout method. It was possible to see early on in the test that it could propagate onto this output, by looking at the fanouts. Thus, the 'prove' part of the property for each time increase was set to try to prove that the fanout states which eventually led to `Spi_sdo_o` would remain equal for each time increase. This was not the case. The result for T##7 was a 'hold_bounded(0)', which indicated propagation onto the `Spi_sdo_o` state.

The counterexample provided, and the secret propagation can be seen in Figure 6.6 and in Figure 6.7. The first figure shows the names of the states affected, while the second figure shows which states were affected at which times.

Figure 6.6: All the names of the states affected by the propagation of the secret in the second L-alert

Here, it can be seen that the secret propagated from the output `inst.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o`, which relates to the secret address having been set up to be in 'Private bank 1', due to the `'pri1'` part of the path name. Thus, the output from this address is unequal in the two instances, and this propagates throughout the system, and eventually ends up in `Spi_sdo_o`, which is visible to a user. As Figure 6.7 shows, this took 7 unrollings,

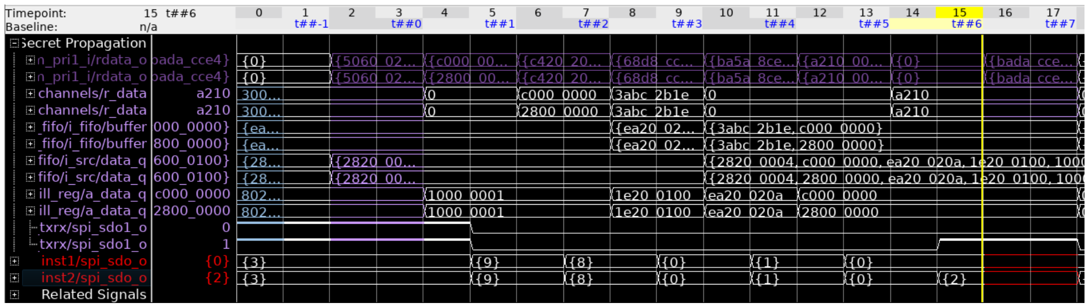


Figure 6.7: All the times at which the states for the second L-alert got different values compared to their counterparts in the other instance

What types of states	Number of states	Number of states per instance
Autogenerated states	7834	7834/2 = 3917
Unprotected memory states	12	12/2 = 6
Top-module outputs	132	132/2 = 66
States to consider in total	7978	7978/2 = 3989

Table 6.2: State_uniqueness size without additional black-boxing

as the last propagation happens at the beginning of time point T## 7. The first proof that was run after the detection of the P-alert, at Tmax=3, took 7 minutes before getting a 'hold_bounded(0)', while the last proof, for T##7, lasted for 3 hours and 10 minutes. This was done without any additional black-boxing.

6.1.2 Results in regard to scalability

Constraints and states

The number of constraints generated for the 'state_constraints' file, which includes the 'state_equivalence' and the 'state_uniqueness' macros, can show how large the system is.

The 'state_uniqueness' function includes the microarchitectural states as well as the observable states, and these have to be proven to remain equal throughout the proof. Each line of the function 'state_uniqueness' looks like the lines in the example seen in Listing 7. Thus, for each line in this function, there is one state that is considered in the proof, and which has to be proven to remain equal to its counterpart in the other instance. Therefore, each line of this function correlates to one state.

The total amount of states from the 'state_uniqueness' that have to be considered in the 'prove' part of the proof, without any additional black-boxing can be seen in Table 6.2. These include the autogenerated microarchitectural states, which are the same for the 'state_uniqneness' function and the 'state_equivalence' function, as well as the observable states. The observable states include the unprotected memory states and the top-module outputs.

What types of states	Number of states	Number of states per instance
Autogenerated states	5118	5118/2 = 2559
Unprotected memory states	12	12/2 = 6
Top-module outputs	132	132/2 = 66
States to consider in total	5262	5262/2 = 2631

Table 6.3: State_uniqueness size with black-boxing used in uDMA-test

As can be seen in Table 6.2, there are 3989 states per system instance that have to be considered in the proof, without any additional black-boxing.

For the analysis of the uDMA, several modules were black-boxed. Then, the system had to be re-compiled, and the 'state_constraints.sv' file had to be autogenerated. Hence, this black-boxing also affected the number of states that were constrained to be equal. This can be seen in Table 6.3, where the number of states per instance that have to be considered has lessened. The new number is 2631.

With the uDMA module black-boxed as well, the number of states per instance was reduced to 1520.

Proof runtime

For the runtime metric, the total runtime to go through the whole system, and check whether the states remain equal in both instances was measured. This was without additional black-boxing or any fanout macros to ease the proof complexity.

This includes all the states in the 'state_uniqueness' function, meaning all the autodetected microarchitectural states that are also considered in the 'state_equivalence' function. It also includes the top-module outputs and the states for the unprotected memory.

This proof was tested during the initial search for the P-alerts, in which some were discovered after a few minutes, while the last P-alert that was found took 2 hours for the tool to discover. Each time a new P-alert is detected, it had to be manually inserted into the blocking clause, and the proof had to be run again.

Because there was no way to know whether the last P-alert found really was the last P-alert, the same proof was run until a 'hold' was given as a result. This would indicate that there were no additional P-alerts to find. Thus, this proof also tested the total run time of a proof considering the entire system. This proof was set up with a Tmax=2.

Because there were so many states to analyze for the tool, this experiment took close to 41 hours to result in a 'hold', as can be seen in Figure 6.8.

Another interesting thing about this is that several of the approver1 solvers had given up.

A similar proof was also tested, just with quite a lot of black-boxing this time. This included black-boxing the most complex subsystems, such as the uDMA and the HWPE, but also the modules that were also black-boxed for the testing of the uDMA. Thus, all the same modules as

```
-R- Property 'sva/checker_bind/ops/UPEC_assertion' holds (checked in 40 h 51 min 28 sec, 57282 MB used)
    Prover          Group Status       Steps/Max.         Real Time    Mem. Peak  Result / Reason
    ---------------------------------------------------------------------------------------------------
    approver1:0    0      done           1/1     20 h 55 min 33 sec 57282.60 MB  open / given up
    approver1:1    0      done           0/1     40 h 50 min 43 sec 57282.60 MB  open
    approver1:2    0      done           1/1     4 h 16 min 53 sec 57282.60 MB  open / given up
    approver1:3    0      done           1/1     21 h 19 min 26 sec 57282.60 MB  open / given up
    approver1:4    0      done           1/1     40 h 50 min 43 sec 57282.60 MB  hold
    approver1:5    0      done           1/1     3 h 59 min 02 sec 57282.60 MB  open / given up
    approver1:6    0      done           1/1     21 h 14 min 08 sec 57282.60 MB  open / given up
    approver1:7    0      done           0/1     40 h 50 min 43 sec 57282.60 MB  open
    approver1:8    0      done           1/1     4 h 01 min 43 sec 57282.60 MB  open / given up

-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 3)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 2)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 1)
```

Figure 6.8: Property hold without any black-boxing, using state_uniqueness. Tmax=2.

in Table 6.1, plus `udma_subsystem` were black-boxed for this test.

This proof was run to have a comparison to look at how long this test would take with black-boxing versus without. The results in Figure 6.9 show how this affected the proof timing.

```
-R- Property 'sva/checker_bind/ops/UPEC_assertion' holds (checked in 14 min 54 sec, 15923 MB used)
    Prover          Group Status       Steps/Max.         Real Time    Mem. Peak  Result / Reason
    ---------------------------------------------------------------------------------------------------
    approver1:0    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:1    0      done           1/1       14 min 48 sec 15923.30 MB  hold
    approver1:2    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:3    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:4    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:5    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:6    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:7    0      done           0/1       14 min 48 sec 15923.30 MB  open
    approver1:8    0      done           0/1       14 min 48 sec 15923.30 MB  open

-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 3)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 2)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 1)
mv>
```

Figure 6.9: Property hold with several modules black-boxed, using state_uniqueness. Tmax=2

Similarly, it was tested to see whether there could also be a difference in proof time due to analyzing a larger time window. Nearly the same test is run in both Figure 6.9 and Figure 6.10. The only difference is that the first picture shows the result for a test with Tmax=2, and the second gives the result for a test with Tmax=3. Both of these tests have the same black-boxing as in Table 6.1, plus `udma_subsystem`. Both tests also use the 'state_uniqueness' function in the 'prove' part of the property.

The alternative process, in which the proof is set up to look for L-alerts by trying to prove that the observable states remain equal, was also evaluated in regard to runtime. The 'prove' part thus only included the states used in the 'udma_obs_space', as seen in Figure 6.3. The proof max time in this experiment was set to 3, and no extra modules were black-boxed.

This scenario had already been tested with regard to actual propagation, by checking just the fanouts in the 'prove' part of the property, as explained in section 6.1.1. Thus, it was already known that there would be no propagation onto observable states at this stage in the proof, since the propagation was still only proven to have reached internal states yet. However, this specific

```
-R- Property 'sva/checker_bind/ops/UPEC_assertion' holds (checked in 47 min 33 sec, 22019 MB used)
    Prover        Group Status      Steps/Max.        Real Time    Mem. Peak  Result / Reason
    ----------------------------------------------------------------------------------------------
    approver1:0   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:1   0     done           1/1       47 min 24 sec 22019.10 MB  hold
    approver1:2   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:3   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:4   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:5   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:6   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:7   0     done           0/1       47 min 24 sec 22019.10 MB  open
    approver1:8   0     done           0/1       47 min 24 sec 22019.10 MB  open

-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 3)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 2)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 1)
mv>
```

Figure 6.10: Property hold with several modules black-boxed, using state_uniqueness. Tmax=3

test was set up to look at how the different experiments with various proof setups hold up against each other.

```
-R- Property 'sva/checker_bind/ops/UPEC_assertion' holds (checked in 2 h 01 min 30 sec, 40221 MB used)
    Prover         Group Status      Steps/Max.        Real Time    Mem. Peak  Result / Reason
    ----------------------------------------------------------------------------------------------
    approver1:0   0     done           1/1      2 h 00 min 41 sec 40221.90 MB  hold
    approver1:1   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:2   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:3   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:4   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:5   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:6   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:7   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open
    approver1:8   0     done           0/1      2 h 00 min 41 sec 40221.90 MB  open

-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 3)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 2)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 1)
mv>
```

Figure 6.11: Property hold with no black-boxing, using uDMA_obs_space. Tmax=3

This proof resulted in a ‘hold’ after two hours of runtime, as can be seen in Figure 6.11, which again proves that there is no secret propagation onto the user-observable states at this time. However, for the alternative route, this does not prove that there is no propagation at later clock cycles, as this proof only concentrates on the observable states. For this type of proof, the time window would thus have to be increased to look further.

Another unrolling using the alternative strategy, and no additional black-boxing, was therefore tested in regard to proof-time. With the proof max time increased to 4, it took 10 hours and 27 minutes before getting a ’hold’, as seen in Figure 6.12.

With the same black-boxing as described in Table 6.1, the same test with Tmax=4, took approximately an hour to achieve a ’hold’.

As described in section 6.1.1, the last test for uDMA using the alternative strategy used the black-boxing seen in Table 6.1, and resulted in a ’hold_bounded(0) in around 30 minutes.

All the tests that were run to check the runtime can be seen collectively in Table 6.4. This presents what Tmax and function were used in the ’prove’ part of the property, along with what

modules were black-boxed in addition to the two memory modules `tc_sram` and `generic_rom`. This also shows the runtime of the proof, and what results the proof ended up with.

All the fanout tests used to investigate the second uDMA L-alert were not researched with proof time in mind. However, the first fanout proof took 7 minutes, and the last took 3 hours and 10 minutes, as reported in section 6.1.1. The last proof was unrolled 7 times.

```
-R- Property 'sva/checker_bind/ops/UPEC_assertion' holds (checked in 10 h 28 min 05 sec, 49410 MB used)
    Prover        Group Status      Steps/Max.        Real Time    Mem. Peak   Result / Reason
    ---------------------------------------------------------------------------------------------
    approver1:0   0     done           1/1     7 h 50 min 32 sec 49410.20 MB  open / given up
    approver1:1   0     done           0/1     10 h 27 min 18 sec 49410.20 MB  open
    approver1:2   0     done           0/1     10 h 27 min 18 sec 49410.20 MB  open
    approver1:3   0     done           1/1     8 h 35 min 57 sec 49410.20 MB  open / given up
    approver1:4   0     done           0/1     10 h 27 min 18 sec 49410.20 MB  open
    approver1:5   0     done           0/1     10 h 27 min 18 sec 49410.20 MB  open
    approver1:6   0     done           1/1     8 h 00 min 35 sec 49410.20 MB  open / given up
    approver1:7   0     done           0/1     10 h 27 min 18 sec 49410.20 MB  open
    approver1:8   0     done           1/1     10 h 27 min 18 sec 49410.20 MB  hold

-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 3)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 2)
-I- Finished property 'sva/checker_bind/ops/UPEC_assertion' (Stage 1)
mv>
```

Figure 6.12: Property hold with no black-boxing, using uDMA_obs_space. Tmax=4

Tmax	'Prove part'	Black-boxed modules	Proof runtime	Proof result
2	state_uniqueness	No black-boxing	2 hours	hold_bounded(0)
2	state_uniqueness	No black-boxing	40 hours 50 min	hold
2	state_uniqueness	Modules seen in Table 6.1 plus `udma_subsystem`	14 min 48 sec	hold
3	state_uniqueness	Modules seen in Table 6.1 plus `udma_subsystem`	47 min 24 sec	hold
3	udma_obs_space	No black-boxing	2 hours	hold
4	udma_obs_space	No black-boxing	10 hours 27 min	hold
4	udma_obs_space	Modules seen in Table 6.1	1 hour	hold
5	udma_obs_space	Modules seen in Table 6.1	30 min	hold_bounded(0)

Table 6.4: Proof runtime experiments

Proof memory-usage

The peak memory usage of each proof was also reported by the tool. The proof which had no extra black-boxing, Tmax=2, and wanted to prove for no propagation onto all the states provided

in 'state_uniqueness' used 57.3 GB of memory. This can be seen in Figure 6.8. The same proof, but with substantial black-boxing, in Figure 6.9, used 16 GB in comparison. If this proof were to be unrolled further, until Tmax=3, the memory usage would increase to 22 GB, as seen in Figure 6.10.

The experiment which used the 'udma_obs_space' function in the 'prove' part was also looked at for memory usage. One experiment for Tmax=3 and one for Tmax=4 was conducted and these resulted in 40,2 GB and 49,4 GB, respectively. Neither of these two experiments had any additional modules black-boxed.

All the memory-usage results can also be seen collectively in Table 6.5.

Tmax	'Prove part'	Black-boxed modules	Memory-usage
2	state_uniqueness	No black-boxing	57.3 GB
2	state_uniqueness	Modules seen in Table 6.1 plus `udma_subsystem`	16 GB
3	state_uniqueness	Modules seen in Table 6.1 plus `udma_subsystem`	22 GB
3	udma_obs_space	No black-boxing	40.2 GB
4	udma_obs_space	No black-boxing	49.4 GB

Table 6.5: Proof memory-usage experiments

6.2 nRF54L15

The UPEC method application onto nRF54L15 was not accomplished within the time of this project. Setting up the nRF54L15 at an SoC level, which is necessary for the UPEC method, turned out to be challenging.

This was because of the issues with the setup of the system itself in OneSpin, which thus ended up taking too much time, leaving it unfeasible to continue. This means that there were no opportunities to analyze for, and potentially detect, vulnerabilities in this system.

There are also no specific results showing scalability onto this system, as there were no possibilities to run any tests to look at memory usage or how much time each test would take. There were also no opportunities to look at how intricate the setup of the system would have to be, how many states would have to be considered in the constraints, nor whether it would have to be black-boxed to be able to run tests.

However, the fact that it was not possible to set up the nRF54L15 system at an SoC level is discussed further in subsection 7.6.2. This is because it can be used to evaluate the scalability of the method, without considering specific metrics.

7

Discussion

The following sections will provide an in-depth discussion regarding how the process to implement the UPEC method went, and thus discuss the time and effort needed. How much prior knowledge is required to be able to do such an experiment will also be looked further into. The results provided in the previous chapter will also be discussed. This will include what vulnerabilities were discovered in the PULPissimo and how this would affect the system, but also how the optimizations affect the proofs. The general scalability of the UPEC method onto IoT devices will also be discussed, based on the experiences for the PULPissimo, but also for the abandoned experiment for the nRF54L15. Finally, potential ways to ease the usage of the UPEC method are elaborated.

7.1 Time and effort required

The setup of the method took a substantial amount of time, as one does need some knowledge regarding the system, the inner workings of the method itself, and the tool. The methodology can prove to be quite complex, even though the theory behind it is well-documented. In total, the setup and being able to begin the actual analysis of the PULPissimo system took somewhere around 125-175 hours to accomplish, even with help from the UPEC team. This could have been affected by the reduced prior knowledge regarding the system and OneSpin as well. The approximate number of hours thus also included a lot of research regarding the system itself, and OneSpin-related topics, as well as trial and error regarding the method setup.

After this, the analysis also took quite a lot of time, as a lot of the proofs took a long time, as reported in Chapter 6. Also, in order to black-box modules, it is necessary to re-compile and elaborate the setup again, before running a new proof. In total, this can also take a long time.

Some proofs can take a lot of time, resulting in long periods having to wait for a result. The proof which checked the whole system using 'state_uniqueness' in the 'prove' part took close to 41 hours, without any additional black-boxing of the system. This is shown in Figure 6.8. This includes all the states that are assumed to be, and should be proven to stay, equal to their counterpart, plus any observable states. Naturally, this is quite a complex proof, however, it does

not need any additional effort, other than having to investigate the counterexamples and increasing the time window.

Normally, really time-consuming proofs would most likely be interrupted, and the system would be simplified using black-boxing. However, this proof was used to look for P-alerts. The last P-alert was detected after running the proof for around 2 hours, as reported in section 6.1.2. Because of this, it is not difficult to imagine that a new P-alert could be detected at around 20 hours, or maybe even deeper into the proof, for instance. Thus, in order to be sure that there would not be any more P-alerts occurring, the wait to achieve a 'hold' felt necessary.

When investigating the P-alerts further, there is another option, which requires less time for the proof, but a bit more manual effort and understanding. This is the method using the fanout optimization, instead of using 'state_uniqueness' in the proof. These tests only include a few states to prove as equal to their equivalent states in the other instance. This results in the proofs taking far less time to run.

This, of course, needs a tad bit more manual work to find the fanouts to be checked. Once the method to do this is fully comprehended, however, this does not necessarily take long. It could take as little as just a couple of minutes.

There is still also the potential that these types of proofs could end up taking longer, not proof time, but in total. If there are several paths possible for propagation, there could be some paths that lead to an observable state eventually, while others might not. This means that the process of checking all the paths that one P-alert could potentially propagate to could take a lot of time in total. Once one path has been checked, the others also would have to be investigated in new proofs, if the first one does not result in an L-alert.

The analysis of counterexamples can also be time-consuming. These are large waveforms, potentially going over a long time frame if the proof has been unrolled. Also, there can be a lot of logic and states to consider if only a few modules have been black-boxed. This has to be done manually in order to discard the possibility of false counterexamples. Thus, it can take a long time in order to be completely sure that no significant information has been missed, which could point towards any unrealistic behavior. A measure of how time-consuming this is can be difficult to estimate exactly. The unrolling factor, what states are considered in the 'prove' part, and the black-boxing factor can all heavily influence the size of each counterexample.

7.2 Scalability results for PULPissimo

Both the PULPissimo platform and the nRF54L15 are complex systems, as they are product-level, meaning that they include all the modules and subsystems to make up a complete and functional system. This means that there could be issues with getting the UPEC method to scale. Even if this experiment failed to test the method onto two separate systems, a lot of different types of tests were run on PULPissimo. There are a few ways to simplify the proofs so that they will not end up being too complex for the tool, and these have been tested during the course of this experiment.

7.2.1 Constraints and states

As presented in section 6.1.2, black-boxing modules reduces the number of states that have to be considered in the system. When further analyzing specific modules due to observed P-alerts, this can thus greatly help the proofs while using the 'state_uniqueness' macro to prove for no differences between the two instances.

During the testing of the uDMA module, there were quite a few modules that were black-boxed, and this reduced the number of states significantly. This also correlates to ease of the system's complexity, and thus also proofs that are less computationally heavy for the tool.

This is, however, not the procedure to follow while looking for the P-alerts, as this could lead to P-alerts not getting detected. However, this can be used while investigating the specific P-alerts, by black-boxing modules that likely will not be affected by any further propagation.

For further unrolling, there is an increase in states that have to be re-evaluated during the course of the proof, as the simulations also lead to value changes in registers over time. This further complicates the system. This can therefore be positively affected by ignoring modules, and thus also alleviating the number of states that have to be considered.

7.2.2 Proof memory-usage with and without optimizations

The peak memory usage of the proofs increases whenever the proof is unrolled to a longer time window, or if there is no black-boxing, as can be seen in Chapter 6.

Unrolling to a longer time window is more or less inevitable, as the method is based on looking at the propagation path over time. This leads to the proof having to look at the system changes over a time window, which naturally leads to larger memory usage.

This can, however, therefore be improved by black-boxing away parts of the system. This is because the proof would not need to consider the extra states within the subsystems that are black-boxed. Therefore, it seems as though the memory usage would, in most cases, be considered reasonable.

If this is not the case, and the method is to be implemented on a significantly larger system, with a lot more states, more subsystems would have to be black-boxed. As long as the black-box is removed and these are investigated further if the propagation path leads to the input of any black-boxed systems, there is nothing standing in the way of using this as an optimization.

7.2.3 Proof time-usage with and without optimizations

Using the 'state_uniqueness' macro when running a proof, without any additional modules black-boxed, resulted in taking 41 hours to finish at Tmax=2. This resulted in a 'hold,' because all the P-alerts detected at this max time had already been collected. The same test, but with a significant number of modules black-boxed, gave a far better result regarding timing, as it took only approximately 15 minutes to get a hold.

One test using 41 hours is not feasible at all, but this could simply be solved by black-boxing some of the more complex subsystems, at least for the proofs set up to look at further P-alert propagation.

The alternative method, using just the observable states in the 'prove' part of the property, also takes a lot of time. This has to result in a 'hold' for all time-frames where the P-alert has not

yet propagated onto observable states. Thus, an increase in Tmax from 3 to 4 went from 2 hours to 10 hours and 27 minutes to accomplish a 'hold', as found in section 6.1.2. This was without any additional black-boxing. If this proof was to be run again, with yet another increase in Tmax, it would naturally take even longer. Here, black-boxing also helps quite a lot, but whenever the proof needs to be analyzed for a large time window, the proof still spends a long time before it is finished, due to the complexity of the proof.

If a 'hold_bounded(n)' was to be outputted by any proofs, these could be detected far earlier in the proof. Thus, the rest of the system would not have to be tested, leading to a decrease in proof run time. The reason for the potential decrease in the run time is that a proof violation can be detected without having to look at all the states. There is, however, no way of knowing for certain. The alert could be detected very deep into the proof, meaning that it could take nearly the same amount of time to get the 'hold_bounded(n)' as well.

For the possibility of following the fanouts, the proofs themselves do not take long, at least not for the proofs with a somewhat low Tmax. However, if a P-alert location has a lot of possible further propagation paths, the manual inspection could take a long time to investigate the possible path.

As expected, the runtime of the proof increases whenever the Tmax is increased. This can be seen in the fanout test to detect the second L-alert. The first proof only took 7 minutes, while the last took just over 3 hours, as reported in section 6.1.2. Therefore, it is imaginable that this type of proof could end up being quite complex if the system was large enough. Especially if any P-alerts would have a long possible propagation path before reaching observable states, as this would result in having to increase the Tmax a lot.

Even with the optimizations that could help the proof runtime, it could still take some time to analyze just one P-alert. It could be necessary to increase the Tmax, or to look at multiple propagation paths when using the fanout method. This naturally indicates that looking at and analyzing all the P-alerts is quite a demanding process in regard to time.

Some proofs can run for a long time, even with a small Tmax if the proof considers a lot of states. Because of this, a few hours to run one proof is shown to be realistic, especially for longer time windows. There could therefore potentially be long turnaround times just to find out that the proof has to be changed to look at another path or to increase the time window further.

An alternative could be to have several setups to check each P-alert, and then run these tests in parallel. This does however require using multiple OneSpin tool licenses, which for this experiment, and most scenarios in general, is not feasible.

7.3 Regarding the alerts

7.3.1 P-alerts

All the P-alerts that were detected at Tmax=2 were further investigated, and most of these turned out to not propagate further. This would mean that these are not vulnerabilities within the design, as they do not propagate onto observable states, and a user or attacker would not be able to see or measure any difference.

The P-alert stemming from the debug module, line 79-80 in Figure 6.1, was not actually

really necessary to check further. This module can of course read the memory. However, if this was a system that would be sent into production and sold, this module would be either password protected or removed completely. This is because the module is just meant to be used for design debugging during production. Without access to the system, a user cannot tell the module to read the memory, and then also pass information along to observable space. Still, the P-alert was tested for further propagation onto its fanout states at T=3, without any further propagation.

7.3.2 uDMA L-alerts

For the uDMA, the P-alert propagated further and reached observable states, meaning it is to be regarded as an L-alert. This would then be a vulnerability, given that the counterexample provided is reachable. This is not a certainty, given that the UPEC method uses approvers. If the two L-alerts detected are actual L-alerts and thus also a vulnerability in the uDMA subsystem, this could be misused by users, as this is a breach in confidentiality. Because the uDMA is able to read the memory, and also communicate this onto user-observable output ports, this could be quite a serious issue.

A similar, if not the same, vulnerability was detected during the previous experiment on the PULPissimo platform as well, as informed in section 2.3. This could be the same one, only that it has not been patched and fixed in the newer version. This discussion will be picked up in section 7.4.

Because of the uDMA module, which is able to read the memory, it seems as if traces of the secret, if not all of the secret, can be found in this observable system output. This can be seen in the fact that the two instances that are compared end up with a difference in value in this specific state. The only thing that was set to be different in the two instances was the secret data, which means that the secret in question has affected several states throughout, and eventually the output. Thus, the uDMA is able to read the secret, and also transmit traces of it onto outputs.

The first L-alert that was found indicated that the `Spi_oen_o`, which is an enable signal and also an output in the top module, was affected by the secret. Although this could potentially carry traces of the secret, it would most likely not be able to carry a lot of information regarding the secret data, because it is simply an enable signal. Thus, getting hold of the secret data in its entirety would likely require a significant amount of time, testing various inputs, and comparing the outputs gained in order to decipher the information.

The second L-alert that was detected was on the output `Spi_sdo_o`, which is a slave data output. This carries the data requested by the master in the SPI interface, back to the master. This output goes all the way back to the top module of the PULPissimo system, and is thus also observable. This might be able to carry even more information about the secret data, if not all of the secret itself. This cannot be said for sure, without actually testing the hypothesis, but it is imaginable that the uDMA could be asked to just fetch the secret data and then end up writing it onto the `Spi_sdo_o` output. If this is the case, the second L-alert might be a more serious vulnerability than the first one discovered.

The exact cause of these issues was not determined during this experiment. The reason was that a lot of the time was spent on the setup and analysis of the system, as well as trying to

determine whether the counterexamples were reachable or not. Therefore, there was no remaining time in the end left to accurately determine the cause of the secret data leakage.

However, the uDMA is somehow able to read the secret and leak it out onto user-observable outputs, despite the usage of the PMP unit which is set up to protect this secret data location.

True counterexample or not

Regarding the possibility that these counterexamples might not be reachable, because of the usage of approvers in the UPEC method, this was further investigated. This included manually inspecting the counterexamples, to look for any irregularities in the system, or otherwise strange behavior.

By using the method which consists of manual inspection of the counterexample, it could be difficult to determine whether the counterexample is unreachable or not. The counterexample provided during the detection of the first L-alert, for instance, was quite large, as it was spread over a timeframe between start and Tmax=5. The states seen in Figure 6.4 are just a small section of all the total states that are listed in the counterexample. This would mean that there are several states and signals to consider, which all also change values over the duration of the test. Therefore, trying to determine the whole picture and looking for any inconsistencies in regard to true system behavior can be challenging. This method was thus attempted for the uDMA L-alerts, but this technique requires a lot of knowledge about how the system should behave. It also requires a lot of attention to detail regarding any strange changes of value in all the states the counterexample provides.

This analysis therefore did not give any indication of anything that could be deemed unrealistic, however, because of the complexity of the counterexample, even further analysis could possibly disprove this conclusion.

Also, a disprover was tested on the proof that gave the first L-alert, propagation to `Spi_oen_o`. This is because disprover proofs could result in a ‘fail’, which would prove that the counterexample is reachable. However, this resulted in a ‘vacuous’ result, meaning that this test did not give any further answers.

Another method to determine whether the counterexample is reachable or not could be to test the constraints and look for any issues with these. If any of the constraints are the reason for the inability to disprove the property, these can be fixed, and thus also give an indication of whether the counterexample is true or false.

The constraints were thus also investigated, both by black-boxing different modules to see whether this could give any indication of any issues, and also by running simple properties to see what results these would give. Some of the simpler proofs ended up being unreachable when they should not have been, which again could point towards issues with the constraints. This could still mean that the alert is a valid L-alert, just that the constraints might be too restrictive or otherwise contradicting.

This was attempted debugged further, to find out which constraints were possibly causing the issue, but due to limitations in time, this did not give any fruitful results.

A third option could be to set up a simulation environment for this, and try to replicate the behavior of the uDMA, to see whether the difference actually does propagate or not. This method

would be a lot more time-consuming, and thus also not feasible for this type of experiment. Therefore, this method was not attempted.

This would require a lot more work, for quite a low gain. If the behavior is realistic, then the L-alert is proven to be true, however, if it proves not to be realistic, the issue causing the problem would not have been found.

This is, however, possibly a better way than analyzing the counterexample to actually determine for certain whether the behavior is realistic or not. If those performing the UPEC test method have limited knowledge regarding the systems, this could especially be the case.

The possible fix to the uDMA L-alert

The uDMA can access the memory, and also fetch and deliver this memory content to peripherals. If there is a possibility for this module to access the secret information it should not be able to pass it onto I/O ports. If this is the case, an adversary could easily get hold of the information.

As seen during these tests, the module is able to access the secret, as the secret address is the only location in the whole system with a difference in value in comparison to its counterpart at the beginning of the proof. This difference also propagates throughout the system onto observable states, via the uDMA, proving that the uDMA is not secure in regards to misuse by an adversary.

Because the exact issue causing the secret data leakage was found, an exact fix to the issue is also difficult to imagine. However, one possible solution to this potential vulnerability would be to block the user from accessing this particular module. This way, there would be no way for an adversary to use the uDMA to read out secret information from the memory onto the I/O ports.

This would, however, also punish those with no bad intentions, as they would not be able to access the module either.

7.4 Differences in results for the two PULPissimo versions

The previous experiment on the PULPissimo platform by Müller et al. (2021) was performed on the PULPissimo version 4.0, while this was performed on the 7.0 version. Naturally, this has led to some different results regarding the systems' vulnerabilities than what was found in the last experiment. As described in section 2.3, the previous tests resulted in the detection of vulnerabilities in the PMP, HWPE and the uDMA.

7.4.1 PMP

As the two experiments on the PULPissimo platforms were performed using different cores, this also means that the PMP unit was different, as this is part of the core's architecture. There was a P-alert that did occur in the core for this experiment as well, but it was not an issue with the PMP unit. Either way, the P-alert did not propagate any further, and was thus also ruled out as a vulnerability.

7.4.2 HWPE

The HWPE was reported as having issues in the previous PULPissimo experiment using this method. For this experiment, the HWPE was tested using the fanout method and resulted in no further propagation, which indicates that there are no issues with this module. The reason for this difference could be that this module, or higher-level modules, have since been changed, which could have resulted in a fix for the previous vulnerability.

In the previous experiment, this was explained as an issue with the HWPE circumventing the PMP, meaning that the issue could have been resolved because of the change of core in this experiment. The core is different, and thus the PMP is also implemented differently, which could have positively affected this vulnerability.

The path of the HWPE P-alert location is dependent on the `fc_subsystem` module, which is used to set up the core, and which has been changed code-wise since the previous experiment. This could, potentially, have affected the HWPE issue that was previously detected.

7.4.3 uDMA

In the previous experiment, there were also reported issues with the uDMA, as with this experiment. This could thus be the same problem as was detected in the previous PULPissimo version as well.

This issue was also a problem with the uDMA circumventing the PMP. This looks a lot like the same issue as the one detected in this experiment. The PMP is supposed to protect the secret memory location, but the uDMA module is still able to read it and leak it out onto user-observable outputs.

The fact that this issue with the uDMA and PMP still seems to remain, while the HWPE issue does not, disproves the theory that the HWPE issue had been resolved due to the change of core. Even though the PMP unit has been changed along with the core, it is still not able to protect the secret data such that the uDMA can leak it.

This does however strengthen the theory surrounding the `fc_subsystem` module. The uDMA issue has not been resolved, while the HWPE did not have any further propagation of the P-alert. This might be because the uDMA module is not dependent on the `fc_subsystem` in the platform. It is, instead, implemented and instantiated under the `soc_peripherals_i` module, as this subsystem is used to communicate with the peripherals in the system.

7.5 Prior knowledge

7.5.1 Need of prior knowledge regarding security and vulnerabilities

When it comes to whether it is necessary to have any sort of security expertise beforehand, the literature describing the method claims that there is no need for any such knowledge. During this experiment, there has not been any indication of anything that could prove otherwise.

The method gives a clear indication of what modules could be vulnerable, based on the propagation of the secret onto visible states or registers via the vulnerable module. Thus, there seems to be no need to research and understand intricate details regarding vulnerabilities or ways to exploit these in order to analyze a system using the UPEC method.

7.5.2 Need of prior knowledge of the systems

In order to set up the method, some prior knowledge regarding the system seems to be necessary, which is also most often the case when products under development are to be tested. The reason that this knowledge is needed to be able to implement the method is that some of the setup of the method revolves around the inner workings of the modules of the system. Also, it is necessary to know which outputs and other system registers can be regarded as observable from the user's point of view.

To thoroughly understand the detected vulnerabilities, what causes the issue, and how this potentially could be misused also requires system knowledge. This could be based on really intricate design details, such as the way some modules work or how they communicate with the other system parts. Thus, in order to see what the actual issue is, plus how this could be fixed, comprehension regarding the design is necessary.

Lastly, when it comes to the topic of unreachable counterexamples, these counterexamples have to be investigated thoroughly in order to spot any irregularities. If any such weird behavior is spotted, the counterexample is unreachable and thus shows a false P-alert or L-alert. This calls for quite some knowledge regarding how the system might behave, as the counterexamples could provide quite a lot of information that has to be investigated.

7.5.3 Need of prior knowledge regarding the tool

In the described methodology, and the tutorial provided by the UPEC team, the tool OneSpin is to be used to perform the analysis. The team provided a tutorial that helped with the setup of the method, but for someone with limited knowledge regarding OneSpin or similar verification tools, it could still be difficult.

Also, the usage of black-boxing modules to reduce complexity, along with the usage of the 'fanout method', can be difficult to understand for someone not familiar with the tool.

7.6 Usage of the UPEC method on industrial IoT devices

During the experiments on the PULPissimo platform and the attempt to set up the nRF54L15, a lot of experiences were gathered regarding how this method could scale onto industrial devices.

7.6.1 Experiences with PULPissimo

During the implementation of the UPEC method onto the PULPissimo platform, a lot of different tests were run. The PULPissimo SoC is already quite a complex system, with a lot of possibilities for extra peripherals. There are, however, several ways to go forth during the analysis part of the UPEC method. For the PULPissimo platform, there seem to be no issues with abstracting away subsystems to lessen the number of states that each proof must consider. Thus, any system could theoretically be reduced a lot in regard to the number of modules to be analyzed per proof, by the usage of black-boxing.

However, a potential issue with this theory is that there could be a scenario where a lot of modules are set to communicate or otherwise be dependent on each other. This could mean that

the alert could propagate through a lot of modules before eventually reaching, or perhaps not reaching, an observable state. Thus, all the modules which the alert propagates through, could not be black-boxed during this analysis, which again would complicate the proof. This would all be dependent on the design of the system.

Another issue is the initial search for P-alerts. Because the initial P-alert search uses 'state_uniqueness', which includes all the system states, this could take a lot of time, as reported in section 6.1.2. When this was performed on the PULPissimo, it took a long time to achieve a 'hold', as previously discussed. This signifies that there are no additional P-alerts to detect.

Furthermore, the P-alerts could be discovered very early in a proof, perhaps after only a few minutes. However, it is also possible that the tool discovers these differences between states very deep into the system, and therefore takes several hours to output a 'hold_bounded(n)'.

A big positive with the UPEC method, however, is that once someone who wishes to use this technique to evaluate systems has used it once, the process seems to ease a lot for further usage. The approach of using the method does not seem to change much after having tried it once. Although there likely must be some slight changes to the setup and analysis, it seems like the next usage on another system would be more understandable and manageable. Once the procedure is fully comprehended it would therefore be easier and easier for each try.

These are just theories, as the original thought to actually test the method onto the nRF54L15 had to be let go of. However, the theory can be backed up by the fact that the principle behind the method is straightforward each time.

If, for instance, one were to try to apply attacks onto systems and try to penetrate the system to check whether it is secure, this would likely be much more advanced. This method would likely change a lot from system to system, dependent on what safety measures have been set up, and the overall system design. Thus, this would require a lot more knowledge regarding the system itself. It would also require a lot more knowledge and understanding of how attacks work, and how vulnerabilities can be exploited.

7.6.2 The nRF54L15

The setup

The setup of the nRF54L15 did end up taking a long time, with elaboration and compilation of the system in OneSpin taking 16 hours. This could of course most likely have been affected by the other issues discovered that prevented the analysis of the system. If this is not the case, however, the generation of the state constraints would also take long. The system would have to be elaborated and compiled again, after setting up the new top module. Most likely, there would also thus have been quite a lot of state constraints for the tool to consider as well, which in turn would affect the proof time greatly.

The issue with the setup of the nRF54L15 was not only the time it took to elaborate and compile, but also that there were issues with getting the clocks and resets to function correctly.

Another issue with the nRF54L15 setup was that it only passed elaboration and compilation after setting some specific compile options, which thus ended up setting all unstable signals in the system to 'unknown'. This is likely not feasible for the UPEC methodology. A similar issue

was encountered in the PULPissimo system. The issue was that OneSpin might have listed some uninitialized states as unknown, which resulted in a vacuous result. The same would thus probably happen for the nRF54L15, because of the unknown signals.

For the nRF54L14, the issue was therefore not the implementation of the UPEC method, but rather the setup of the system itself in OneSpin. This is not a UPEC-specific issue, but it could still reflect negatively on the method.

The UPEC method calls for property checking at an SoC level, which of course complicates the setup. Often, property checking is done at the block level, meaning that smaller parts of the system are verified at a time. The UPEC method instead requires the entire system to work together in OneSpin.

If the systems to be tested are not developed using the OneSpin tool, this could result in the need for having to set up the whole system in the tool for this specific test. As seen in this experiment, setting up SoC-level systems can be a tedious task. This could then affect a company's time frame to get the product onto market. Also, if the designers are not familiar with the tool beforehand, this task could be even more time-consuming and difficult.

Unless the whole system is to be implemented in OneSpin to run other tests and the testers are familiar with the tool, the usage and application of the method can be rather complicated.

However, troubling modules could imaginably be black-boxed to ease the setup of the system or to narrow down the possible difficult areas. Thus, this could be a way of easing the setup and finding the issues that complicate the setup. It could be that some modules are not an issue individually, but that they cause issues in conjunction with others.

It might also be that the black-boxing technique could be used for the P-alert search as well. Still, the individual black-boxed modules would have to be searched through eventually to ensure a secure system regardless. However, it might be that they do not have to be searched in conjunction with the rest of the system. If this is the case, the setup would be less complicated. This is, however, just a theory, and was not properly tested.

The initial P-alert search

In the initial search for P-alerts, the whole system would likely have to be considered. Even if it might be possible to black-box some modules during this search, they would still have to be considered individually. Thus, this would not result in a shorter total time for the initial search, as several proofs would have to be run. All modules would thus have to be searched through to properly ensure that the system could be regarded as safe from vulnerabilities. This is therefore possibly the limiting factor, in regard to proof runtime.

With possible run times of 41 hours for the PULPissimo platform, as reported in Chapter 6, it is not unimaginable that the nRF54L15 could take even longer to fully analyze for P-alerts. This is because the PULPissimo platform only took a few minutes to finish elaborate and compile in comparison to the nRF54L15.

If one were able to set up the whole system, and then search for P-alerts using the 'state_uniqueness' function, this could however be done in conjunction with other work. This way, the tester would not have to just wait for the proof to finish. Still, it would have to be checked regularly, and how often these checks should be done is difficult to say.

Another disadvantage with this is that it would take up a OneSpin license for a, potentially, long time.

It is also a possibility that for quite large proofs, the approver would give up without finding any result for the proof, which happened for several of the approver1 instances in Figure 6.8. This could be based on resource limits, which would then be the limiting factor. It could also have been that they simply ran for too long without any actual result.

The further investigation of the P-alerts

After the initial search for P-alerts, the system can again be simplified by black-boxing modules, during the further investigation of the P-alerts' propagation path. This eases the proof complexity significantly.

Another option is to look at the fanouts, which also reduces time-per-proof. Still, there is a chance that this further analysis could take a long time as well, based on the number of fanouts each P-alert location has. There could, imaginably, be quite a lot of them, and the analysis could take a long time, in total. This includes both the proof run times, as well as the work to find the fanouts and set up new proofs to run.

An alternative would be to only look at whether the observable states remain equal or not for each Tmax, to alleviate this issue of having to analyze all the fanouts. In this method, however, a lot of subsystems would have to be abstracted away to ensure that the proofs themselves do not have unreasonably long run times. For this type of analysis, there seems to be no issue in doing so.

Thus, after the initial setup is finished, and the P-alerts have been detected, there are several good optimizations to the UPEC method which reduce the proof time considerably. This could indicate that with a bit more time for the setup of the system, the method would likely be scalable onto the nRF54L15 as well.

7.7 Suggestions to ease the usage

The tutorials provided by the UPEC team helped with the setup on the PULPissimo system, and the theory behind the methodology is well described with several papers and a Ph.D. thesis on the subject. Still, the methodology is quite difficult to understand, especially for someone without prior knowledge.

The team answered questions throughout, but adding more information to the tutorial instead would aid the users understanding, and also help reduce the UPEC team's workload in the long run.

Things that could be helpful would be to add some examples and small experiments into the tutorial. These could give information regarding both the setup and the analysis, and how to ease the proof complexity. It would also help if there were pictures that showed how to use the specific OneSpin features, as these could be unfamiliar to some users of the method.

Understanding how to use the optimizations, such as how to black-box modules, and how to locate the fanouts of a location can be difficult for someone who has not done it before. In the tutorial, these optimizations are only briefly mentioned as options. It would thus also aid

comprehension of the analysis part of the method if some explanations regarding these approaches were provided in the tutorial.

Another point is that the method requires the tester to be able to tell whether the counterexamples are reachable or not, because of the usage of approvers. This can be difficult to determine for someone who has not done this before, as the counterexamples consider a lot of logic. The counterexamples are of course system-specific, and thus exact information regarding the counterexamples cannot be provided in any tutorials. However, adding examples of how a reachable counterexample should look, and how an unreachable counterexample might look, could still help. This could provide information about what to look for in a counterexample, which could aid someone without previous experience.

When it comes to the complexity of the initial P-alert search, and the setup of whole SoC systems in OneSpin, this also has to be worked on as well. It might be that the UPEC team has some methods for dealing with this, and if so, they could also be clearly communicated in the tutorial.

It might be that some modules could be black-boxed for the setup and the initial P-alert search, which would ease the setup. This was not tested for this experiment, however.

8

Conclusions

This chapter provides potential ideas for future work that could be done to continue the research done in this experiment. Further, the key conclusions drawn from the results obtained and the information gathered throughout this study will be presented.

8.1 Future work

For the next step in this research, the obvious path would be to continue with the implementation onto the nRF54L15 once the setup of the system in the OneSpin tool ends up being successful. This could be interesting to do, in order to test whether it would actually be possible to scale the UPEC method onto this, and similar devices. There could be new, undiscovered types of vulnerabilities in such systems, that could be beneficial to locate and thus also fix. This would also aid in testing the scalability of the method.

Another interesting experiment would be to try to exploit the vulnerability in the uDMA module in the PULPissimo system. It could be fascinating to see whether it would be possible to exploit this vulnerability to get hold of any secret information that should be protected by the PMP unit. This can also aid in mapping out the severity of these types of vulnerabilities.

As discussed in subsection 7.3.2, the first L-alert that was found would most likely give a lot less information about the secret than the second L-alert. This is because of the type of outputs the L-alerts ended up propagating onto. The first L-alert propagated onto `Spi_oen_o`, which is an enable signal, and therefore might not be able to give a lot of information regarding the secret. The second L-alert propagated onto `Spi_sdo_o`. This is the slave output of the SPI module, meaning that this signal carries the fetched data that the SPI master requested back to the master. This can therefore, possibly, leak some more information regarding the secret data.

This could also be an interesting theory to test out, to see whether it would be possible to misuse both of these outputs. Thus, it could be an idea to look at what kind of information about the secret could be gathered from both of these two and whether one would provide more secret data information or not. It can also be interesting to look at how intricate this procedure would have to be to get hold of any data of significance.

In regards to how to improve the UPEC methodology to ease the application and analysis process for later users, adding some examples and smaller experiments to the tutorial could be a good start. This could help ease the understanding, as the methodology is quite advanced.

There might be someone with limited prior knowledge about the OneSpin tool seeking to test the method, as well. Because of this, adding more explanations regarding the specific OneSpin features used could also aid in making the method more available.

This would not only aid the users of the method, as it could be far easier to understand the usage, but it could also aid the UPEC team. If more information is provided beforehand, the team would not have to spend a lot of time helping each user. This could also aid in testing the scalability of the method, as new users would be able to test it, which would result in the method being tested on several different designs.

8.2 Conclusions

The need for secure IoT devices, and techniques to ensure these, increases alongside the complexity of these systems. There are already several reported discoveries of system vulnerabilities and possible exploitations of these that could lead to the theft of passwords, cryptographic keys, personal data, and other similar information. As this experiment shows, the Unique Program Execution Checking method is a well-thought-out technique that could aid designers and testers in locating such vulnerabilities.

However, it is quite an advanced methodology that could be difficult to understand fully. During these experiments, the UPEC team provided templates and a tutorial on how to apply the method onto a system. Still, there was quite a bit of confusion regarding the setup and how the analysis should be performed, which led to conversations with the UPEC team to find solutions. This could have been avoided if the tutorial included examples and small experiments that could help enhance the understanding.

During this process, the UPEC-SoC method was re-applied to a newer version of the PULPissimo platform and ended up locating a vulnerability within the uDMA subsystem, which could result in secret data being written out to observable outputs. A similar, if not the same, vulnerability was also found in the previous application onto this platform by the UPEC team. However, some vulnerabilities found during the prior experiment were not found this time around, most likely due to design changes in some of the system modules in the newer version.

This experiment also aimed to test the scalability of the UPEC method by applying this onto the nRF54L15. Due to issues with the setup of the device, the application of the method onto this design was not accomplished. As the UPEC methodology requires property checking at the SoC level, the setup can be difficult to manage.

Still, several results regarding scalability could be obtained due to different tests of the method, on the PULPissimo, using different changes to the initial property. These scalability tests could indicate that, dependent on how long time one is willing to allocate for running the proofs, a much larger system could be more difficult to test.

One of the UPEC proofs that was run on the PULPissimo, which considered all the system states, spent 41 hours before resulting in a 'hold'. Thus, a longer run time is not unlikely for an

even more complex system.

There are, however, optimizations to the method that can reduce the proof complexity while looking at the P-alerts' propagation path. Thus, these optimizations also alleviate the proof run-times significantly.

The method is well documented, but it is still a difficult concept to comprehend. These experiments show that someone without prior experience with the method could be able to apply it to a system to look for vulnerabilities, which strengthens the usefulness of the method. Still, the process would have been easier if more concepts were explained in detail in the tutorial. All in all, with some additional effort, especially in improving the tutorial to help new users, the methodology could potentially be used in industrial systems in the future.

Bibliography

Arcangeli, A., Eidus, I., Wright, C., 2009. Increasing memory density by using KSM, in: Proceedings of the Linux Symposium, pp. 19–28.

Arm Developer, n.d,a. DSP Overview. URL: https://developer.arm.com/Architectures/Digital%20Signal%20Processing. Last accessed: 2023-05-11.

Arm Developer, n.d,b. System-Wide Security for IoT Devices. URL: https://www.arm.com/technologies/trustzone-for-cortex-m. Last accessed: 2023-05-11.

Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D.A., Richards, B., Schmidt, C., Twigg, S., Vo, H., Waterman, A., 2016. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

Baek, S., Carneiro, M., Heule, M.J.H., 2021. A Flexible Proof Format for SAT Solver-Elaborator Communication, in: Groote, J.F., Larsen, K.G. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham. pp. 59–75.

Benger, N., van de Pol, J., Smart, N.P., Yarom, Y., 2014. “Ooh Aah... Just a Little Bit” : A Small Amount of Side Channel Can Go a Long Way, in: Batina, L., Robshaw, M. (Eds.), Cryptographic Hardware and Embedded Systems – CHES 2014, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 75–92.

Bhunia, S., Tehranipoor, M., 2019a. Chapter 1 - Introduction to Hardware Security, in: Bhunia, S., Tehranipoor, M. (Eds.), Hardware Security. Morgan Kaufmann, pp. 1–20. doi:10.1016/B978-0-12-812477-2.00006-X.

Bhunia, S., Tehranipoor, M., 2019b. Chapter 2 - A Quick Overview of Electronic Hardware, in: Bhunia, S., Tehranipoor, M. (Eds.), Hardware Security. Morgan Kaufmann, pp. 23–45. doi:10.1016/B978-0-12-812477-2.00007-1.

Bhunia, S., Tehranipoor, M., 2019c. Chapter 8 - Side-Channel Attacks, in: Bhunia, S., Tehranipoor, M. (Eds.), Hardware Security. Morgan Kaufmann, pp. 193–218. doi:10.1016/B978-0-12-812477-2.00013-7.

Bondi, A.B., 2000. Characteristics of Scalability and Their Impact on Performance, in: Proceedings of the 2nd International Workshop on Software and Performance, Association for Computing Machinery, New York, NY, USA. p. 195–203. doi:10.1145/350391.350432.

Boole, G., 1847. The mathematical analysis of logic: being an essay towards a calculus of deductive reasoning. Macmillan, Barclay, & Macmillan, Cambridge.

Celio, C., Chiu, P.F., Nikolic, B., Patterson, D.A., Asanović, K., 2017. BOOM v2: an open-source out-of-order RISC-V core. Technical Report UCB/EECS-2017-157. EECS Department, University of California, Berkeley.

Clarke, E., Biere, A., Raimi, R., Zhu, Y., 2001. Bounded Model Checking Using Satisfiability Solving. Formal Methods in System Design 19, 7–34. doi:10.1023/A:1011276507260.

Clarke, E.M., Emerson, E.A., Sistla, A.P., 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Trans. Program. Lang. Syst. 8, 244–263. doi:10.1145/5397.5399.

Dessouky, G., Gens, D., Haney, P., Persyn, G., Kanuparthi, A., Khattri, H., Fung, J.M., Sadeghi, A.R., Rajendran, J., 2019. HardFails: Insights into Software-Exploitable hardware bugs, in: 28th USENIX Security Symposium (USENIX Security 19), USENIX Association, Santa Clara, CA. pp. 213–230.

Erik Vrielink, 2019. Meltdown. URL: https://spectrum.ieee.org/how-the-spectre-and-meltdown-hacks-really-worked. Last accessed: 2023-05-07.

Fadiheh, M.R., 2022. Unique Program Execution Checking: A Novel Approach for Formal Security Analysis of Hardware. Doctoral Thesis. Technische Universität Kaiserslautern. doi:10.26204/KLUEDO/6930.

Fadiheh, M.R., Stoffel, D., Barrett, C., Mitra, S., Kunz, W., 2019. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking, in: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 994–999. doi:10.23919/DATE.2019.8715004.

Graz University of Technology, 2018. Meltdown and Spectre: Vulnerabilities in modern computers leak passwords and sensitive data. URL: https://meltdownattack.com. Last accessed: 2023-05-01.

Hennessy, J.L., Patterson, D.A., 2017. Computer Architecture, Sixth Edition: A Quantitative Approach. 6th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Huth, M., Ryan, M., 2004. Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press.

Hutle, M., Kammerstetter, M., 2015. Chapter 4 - Resilience Against Physical Attacks, in: Skopik, F., Smith, P. (Eds.), Smart Grid Security. Syngress, Boston, pp. 79–112. doi:10.1016/B978-0-12-802122-4.00004-3.

Institute of Electrical and Electronics Engineers, 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012) , 1–1315doi:10.1109/IEEESTD.2018.8299595.

Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., 2019. Spectre Attacks: Exploiting Speculative Execution, in: 40th IEEE Symposium on Security and Privacy (S&P'19).

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M., 2018. Meltdown: Reading Kernel Memory from User Space, in: 27th USENIX Security Symposium (USENIX Security 18).

lowRISC team, 2018. Ibex: An embedded 32 bit RISC-V CPU core. URL: https://ibex-core.readthedocs.io/en/latest/. Last accessed: 2023-05-31.

Mehta, A.B., 2014. SystemVerilog Assertions and Functional Coverage: Guide to Language, Methodology and Applications. Springer New York, New York, NY. doi:10.1007/978-1-4614-7324-4_2.

Mehta, A.B., 2021. Introduction to SystemVerilog. Springer International Publishing, Cham. doi:10.1007/978-3-030-71319-5_14.

Müller, J., Fadiheh, M.R., Antón, A.L.D., Eisenbarth, T., Stoffel, D., Kunz, W., 2021. A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level, in: 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 991–996. doi:10.1109/DAC18074.2021.9586248.

Nguyen, M.D., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., Kunz, W., 2008. Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27, 2068–2082. doi:10.1109/TCAD.2008.2006092.

Nordic Semiconductor, n.d. About Nordic Semiconductor. URL: https://www.nordicsemi.com/About-us. Last accessed: 2023-06-09.

Nordic Semiconductor, n.d,. Product Specification. URL: https://projecttools.nordicsemi.no/infocenter/index.jsp. Internal info center of Nordic Semiconductor, information used with permission. Not accessible without permission. Last accessed: 2023-05-26.

OneSpin Solutions, n.d. Automatic and exhaustive analysis of a code base for classic implementation problems. URL: https://www.onespin.com/products/360-dv-inspect. Last accessed: 2023-05-11.

OpenHW Group, 2023. OpenHW Group CV32E40P User Manual. URL: https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/. Last accessed: 2023-05-31.

OpenHWGroup, n.d. Ariane. URL: https://github.com/lowRISC/ariane. Last accessed: 2023-05-18.

OpenHWGroup, n.d. CV32E40P. URL: https://github.com/openhwgroup/cv32e40p. Last accessed: 2023-05-21.

Sarangi, S.R., 2021. Basic Computer Architecture. 2nd ed., White Falcon Publishing.

Schiavone, P.D., Rossi, D., Pullini, A., Di Mauro, A., Conti, F., Benini, L., 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX, in: 2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), pp. 1–3. doi:10.1109/S3S.2018.8640145.

Seligman, E., Schubert, T., Kumar, M.V.A.K., 2015a. Chapter 1 - Formal verification: From dreams to reality, in: Seligman, E., Schubert, T., Kumar, M.V.A.K. (Eds.), Formal Verification. Morgan Kaufmann, Boston, pp. 1–22. doi:10.1016/B978-0-12-800727-3.00001-0.

Seligman, E., Schubert, T., Kumar, M.V.A.K., 2015b. Chapter 4 - Formal property verification, in: Seligman, E., Schubert, T., Kumar, M.V.A.K. (Eds.), Formal Verification. Morgan Kaufmann, Boston, pp. 87–117. doi:10.1016/B978-0-12-800727-3.00004-6.

Siemens, 2023a. Reference Manual: OneSpin 360TM Version 2023.1. Siemens. OneSpin Manuals are only accessible for licensed users. Information used and cited with permission.

Siemens, 2023b. User Manual: OneSpin 360TM Version 2023.1. Siemens. OneSpin Manuals are only accessible for licensed users. Information used and cited with permission.

lowRISC team, n.d,a. About lowRISC. URL: https://lowrisc.org/about/. Last accessed: 2023-03-20.

lowRISC team, n.d,b. Ibex. URL: https://github.com/lowRISC/ibex. Last accessed: 2023-03-17.

The Parallel Ultra Low Power platform, 2022. About PULP. URL: https://pulp-platform.org/projectinfo.html. Last accessed: 2023-05-11.

The Parallel Ultra Low Power platform, n.d. PULPissimo. URL: https://github.com/pulp-platform/pulpissimo. Last accessed: 2023-03-17.

Trivedi, D., Khade, A., Jain, K., Jadhav, R., 2018. SPI to I2C Protocol Conversion Using Verilog, in: 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), pp. 1–4. doi:10.1109/ICCUBEA.2018.8697415.

Vahid, F., 2010. Digital Design with RTL Design, Verilog and VHDL. 2nd ed., John Wiley and Sons.

Waterman, A., Lee, Y., Avizienis, R., Patterson, D.A., Asanović, K., 2016a. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1. Technical Report. EECS Department, University of California, Berkeley.

Waterman, A., Lee, Y., Patterson, D.A., Asanović, K., 2016b. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report. EECS Department, University of California, Berkeley.

Yarom, Y., Falkner, K., 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack, in: 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, San Diego, CA. pp. 719–732.

Zhang, D., Wang, Y., Suh, G.E., Myers, A.C., 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security, in: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, New York, NY, USA. p. 503–516.

Appendix

A Property file

```
2   module property_checker(
3           //`include "io.sv"
4           input reset,
5           input clock
6
7   );
8
9   `include "tidal.sv"
10
11  `include "/state_constraints/state_constraints_udma_check.sv"
12
13  //`include "/generate_state_constraint/state_constraints.sv"
14
15  `include "/state_constraints/udma_bb_constraints.sv"
16
17  `begin_tda(ops)
18
19
20  function automatic pmp_memory_protection_state_static();
21          pmp_memory_protection_state_static = (
22                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_cfg_rdata[0]  == 8'b00010000  &&
23                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_addr_rdata[0] == 32'h07002030 &&
24                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_cfg_rdata[1]  == 8'b00010000  &&
25                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_addr_rdata[1] == 32'h070040C0 &&
26                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_cfg_rdata[2]  == 8'b00010000  &&
27                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_addr_rdata[2] == 32'h00002030 &&
28                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_cfg_rdata[3]  == 8'b00010000  &&
29                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.pmp_addr_rdata[3] == 32'h000040C0);
30  endfunction
31
32  function automatic unpriv_mode();
33          unpriv_mode = (
34                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.priv_lvl_q == 2'b00 &&
35                  miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.priv_lvl_q == 2'b00 &&
36
37                  miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.mstatus_q.mprv == 1'b0 &&
38                  miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.cs_registers_i.mstatus_q.mprv == 1'b0
39           );
40  endfunction
41
```

Figure A.1: Property.sv line 2-41

```
42 function automatic no_secret();
43         no_secret =
44                 miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o &&
45                 miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.rdata_o &&
46                 miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.rdata_o &&
47                 miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.rdata_o &&
48                 miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.rdata_o &&
49                 miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o;
50 
51 endfunction
52 
53 function automatic p_alerts_blocking_clause();
54 p_alerts_blocking_clause = (
55         /*miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.i_fifo_incoming.fifo_registers ==
56         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.i_fifo_incoming.fifo_registers &&
57         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.i_fifo_incoming.fifo_registers ==
58         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.i_fifo_incoming.fifo_registers &&
59         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.tcdm_master_r_data_q ==
60         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.tcdm_master_r_data_q &&
61         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.i_fifo_incoming.fifo_registers ==
62         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_b_tcdm_fifo_load.i_fifo_incoming.fifo_registers &&
63         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.tcdm_master_r_data_q ==
64         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_a_tcdm_fifo_load.tcdm_master_r_data_q &&
65         miter_top.inst1.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.tcdm_master_r_data_q ==
66         miter_top.inst2.pulp_soc_i.fc_subsystem_i.fc_hwpe_gen.i_fc_hwpe.i_mac_top_wrap.i_mac_top.i_streamer.i_c_tcdm_fifo_load.tcdm_master_r_data_q &&*/
67         miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.load_store_unit_i.rdata_q ==
68         miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.load_store_unit_i.rdata_q &&
69         miter_top.inst1.pulp_soc_i.i_lint_jtag.dbg_module_i.i_dbg_lint.lint_biu_i.data_out_reg ==
70         miter_top.inst2.pulp_soc_i.i_lint_jtag.dbg_module_i.i_dbg_lint.lint_biu_i.data_out_reg &&
71         miter_top.inst1.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_write.data_r_rdata_q ==
72         miter_top.inst2.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_write.data_r_rdata_q &&
73         miter_top.inst1.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_read.data_r_rdata_q ==
74         miter_top.inst2.pulp_soc_i.i_soc_interconnect_wrap.i_axi64_to_lint32.axi64_2_lint32_i.parallel_lint_read.data_r_rdata_q &&
75         miter_top.inst1.pulp_soc_i.i_dm_top.i_dm_csrs.sbdata_q ==
76         miter_top.inst2.pulp_soc_i.i_dm_top.i_dm_csrs.sbdata_q &&
77         //miter_top.inst1.pulp_soc_i.soc_peripherals_i.i_udma.i_udmacore.u_tx_channels.r_data ==
78         //miter_top.inst2.pulp_soc_i.soc_peripherals_i.i_udma.i_udmacore.u_tx_channels.r_data
79         );
80 endfunction
81 
```

Figure A.2: Property.sv line 42-81

```
83 
84 function automatic udma_obs_space();
85     udma_obs_space =(
86         //unprotected memory space
87         miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.wdata_i      == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.wdata_i &&
88         miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.wdata_i      == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.wdata_i &&
89         inst1.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.wdata_i                == inst2.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.wdata_i &&
90         inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.wdata_i                == inst2.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.wdata_i &&
91         inst1.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.wdata_i              == inst2.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.wdata_i &&
92         inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.wdata_i              == inst2.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.wdata_i &&
93 
94         //Outputs in top module
95         inst1.i2c_scl_o                                                 == inst2.i2c_scl_o &&
96         inst1.hyper_ck_no                                               == inst2.hyper_ck_no &&
97         inst1.uart_tx_o                                                 == inst2.uart_tx_o &&
98         inst1.spi_sdo_o                                                 == inst2.spi_sdo_o &&
99         inst1.sdio_clk_o                                                == inst2.sdio_clk_o &&
100        inst1.i2c_sda_o                                                 == inst2.i2c_sda_o &&
101        inst1.spi_oen_o                                                 == inst2.spi_oen_o &&
102        inst1.spi_csn_o                                                 == inst2.spi_csn_o
103 );
104 endfunction
105 
106 
107 
108 property symbolic_secret_address;
109 //address ranges:
110 //pri1: 32'h1C00_8000 to 32'h1C01_0000
111 //TCDM bank0: 32'h1C01_0000 to 32'h1C08_2000
112 //static secret: 32'h1c00_80c0 || 32'h1C01_0300
113         ( $past(miter_top.secret_address) == miter_top.secret_address) &&
114         (miter_top.secret_address == 32'h1C00_80C0 || miter_top.secret_address == 32'h1C01_0300);
115 endproperty
```

Figure A.3: Property.sv line 84-115

```
116
117 property mem_data_array_blackboxing_constraint;
118         //( miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o ) &&
119         ( miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[1].bank_i.rdata_o ) &&
120         ( miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[2].bank_i.rdata_o ) &&
121         ( miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[3].bank_i.rdata_o ) &&
122         ( miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.bank_sram_pri0_i.rdata_o ) &&
123         //( miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o ) &&
124         miter_top.inst1.pulp_soc_i.boot_rom_i.rom_mem_i.Q == miter_top.inst2.pulp_soc_i.boot_rom_i.rom_mem_i.Q &&
125
126         //If secret address = bank0, change input address in one instance. Else: constrain memory output to be equal due to new PI
127         ( ( ( miter_top.secret_address >= 32'h1C01_0000 ) &&
128         ( miter_top.secret_address <  32'h1C08_2000 ) &&
129         (miter_top.secret_address[3:2] == 2'b00) ) ?
130         ( ($past(miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.addr_i) == (miter_top.secret_address[18:4] - 16'h1000)) ||
131         ((miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o) == (miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o)) ) :
132         ( miter_top.inst1.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.CUTS[0].bank_i.rdata_o ) ) &&
133
134         //If secret address = pri1, change input address in one instance. Else: constrain memory output to be equal due to new PI
135         ( ( ( miter_top.secret_address >= 32'h1C00_8000 ) &&
136         ( miter_top.secret_address <  32'h1C01_0000 ) ) ?
137         ( ($past(miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.addr_i) == miter_top.secret_address[14:2]) ||
138         ((miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o) == (miter_top.inst2.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o)) ) :
139         ( miter_top.inst1.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o == miter_top.inst2.pulp_soc_i.l2_ram_i.bank_sram_pri1_i.rdata_o ) );
140 endproperty
141
142 property unpriv_mode_constraint;
143         unpriv_mode();
144 endproperty
145
146 property instruction_input_equal;
147         miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_gnt_i ==
148         miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_gnt_i &&
149         miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_rvalid_i ==
150         miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_rvalid_i &&
151         miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_rdata_i ==
152         miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_rdata_i &&
153         miter_top.inst1.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_err_i ==
154         miter_top.inst2.pulp_soc_i.fc_subsystem_i.FC_CORE.lFC_CORE.instr_err_i;
155 endproperty
```

Figure A.4: Property.sv line 116-155

```
158 //constraints used for the shortcut-testing of udma. Outputs from these modules are now regarded as
159 //Primary inputs to the system, and thus have to be constrained to be equal.
160 //Found in udma_bb_constraints.sv
161
162 hwpe_constraint: assume property (@(posedge miter_top.clock) hwpe_bb_constraint);
163
164 i_apb_adv_timer_constraint: assume property ( @(posedge miter_top.clock) i_apb_adv_timer_bb_constraint);
165
166 apb_fll_if_i_constraint: assume property ( @(posedge miter_top.clock) apb_fll_if_i_bb_constraint);
167
168 axi_xbar_constraint: assume property (@(posedge miter_top.clock) axi_xbar_bb_constraint);
169
170 jtag_top_i_constraint: assume property (@(posedge miter_top.clock) jtag_top_i_bb_constraint);
171
172 apb_gpio_constraint: assume property ( @(posedge miter_top.clock) apb_gpio_bb_constraint);
173
174 i_apb_soc_ctrl_constraint: assume property ( @(posedge miter_top.clock) i_apb_soc_ctrl_bb_constraint);
175
176 i_apb_timer_unit_constraint: assume property ( @(posedge miter_top.clock) i_apb_timer_unit_bb_constraint);
177
178 axi_master_cdc_i_constraint: assume property ( @(posedge miter_top.clock) axi_master_cdc_i_bb_constraint);
179
180 axi_slave_cdc_i_constraint: assume property ( @(posedge miter_top.clock) axi_slave_cdc_i_bb_constraint);
181
182 i_axi_lite_to_apb_constraint: assume property ( @(posedge miter_top.clock) i_axi_lite_to_apb_bb_constraint);
183
184
185 //no additional p-alerts from instruction interface
186 instr_constraint: assume property ( @(posedge miter_top.clock) instruction_input_equal);
187
188 //mode set to unprivileged
189 unpriv_constraint: assume property ( @(posedge miter_top.clock) unpriv_mode_constraint);
190
191 //Change in secret address + memory blackbox constraints
192 mem_constraint: assume property ( @(posedge miter_top.clock) mem_data_array_blackboxing_constraint);
193
194  //Secret address setup
195 symbolic_address_constraint: assume property ( @(posedge miter_top.clock) symbolic_secret_address);
196
```

Figure A.5: Property.sv line 155-196

```
198  property UPEC;
199          t ## 0 state_equivalence() and //All microarchitectural states assumed to be equal
200          during_o(t,2, t, 5, no_secret()) and //no multiple secret propagation
201          during_o(t,0, t, 5, pmp_memory_protection_state_static()) and //PMP setup. Secret address is protected.
202          t ## 0 no_secret() and //no access to secret at startup
203          /*during_o(t,0, t, 5, unpriv_mode()) and*/
204          t ## 5 p_alerts_blocking_clause() //no other P-alerts interfering
205  implies
206          //t ## 3 state_uniqueness();
207          t ##  5 udma_obs_space();
208          //t ##5 inst1.spi_oen_o == inst2.spi_oen_o;
209
210  endproperty
211
212
213  UPEC_assertion: assert property ( @(posedge miter_top.clock) disable iff(reset) UPEC);
214
215  `end_tda
216
217  endmodule
218
219  bind miter_top property_checker checker_bind(
220          .reset(reset),
221          .clock(clock));
222
```

Figure A.6: Property.sv line 197-222

B Propagation path from uDMA P-alert location

Path: [top]/inst1/pulp_soc_i/soc_peripherals_i/i_udma/i_udmacore/u_tx_channels TimePoint: 1

Fanout Nets	Value	Kind	Modu	Hierarchic	Time
r_data	5800_02ec	Net [State]	ud...	inst1/p...	1
r_data	5800_02ec	Net [State]	ud...	inst1/p...	2
s_data	0	Net	ud...	inst1/p...	1
ext_data_o	{dead_b...	Output	ud...	inst1/p...	1
lin_data_o	{dead_b...	Output	ud...	inst1/p...	1
tx_lin_data_o	{dead_b...	Output	ud...	inst1/p...	1
s_tx_ch_data	{dead_b...	Net	ud...	inst1/p...	1
data_tx_i	dead_beef	Input	ud...	inst1/p...	1
data_tx_i	dead_beef	Input	ud...	inst1/p...	1
data_tx_i	5800_02ec	Input	ud...	inst1/p...	1
data_tx_i	dead_beef	Input	ud...	inst1/p...	1
cmd_i	dead_beef	Input	ud...	inst1/p...	1
data_i	dead_beef	Input	io_...	inst1/p...	1
data_i	dead_beef	Input	io_...	inst1/p...	1
buffer	{2000_0...	Net [State]	io_...	inst1/p...	2
buffer	{2000_0...	Net [State]	io_...	inst1/p...	3
data_o	f000_0000	Output	io_...	inst1/p...	2
data_o	f000_0000	Output	io_...	inst1/p...	2
s_spi_cmd	f000_0000	Net	ud...	inst1/p...	2
src_data_i	f000_0000	Input	ud...	inst1/p...	2
src_data_i	f000_0000	Input	cd...	inst1/p...	2
src_data_i	f000_0000	Input	cd...	inst1/p...	2
data_q	{c081_0...	Net [State]	cd...	inst1/p...	3
data_q	{c081_0...	Net [State]	cd...	inst1/p...	4
async_data_o	{c081_0...	Output	cd...	inst1/p...	3
async_data	{c081_0...	Net	cd...	inst1/p...	3
async_data_i	{c081_0...	Input	cd...	inst1/p...	3
dst_data	b081_0000	Net	cd...	inst1/p...	3
data_i	b081_0000	Input	spi...	inst1/p...	3
data_i	b081_0000	Input	spi...	inst1/p...	3
a_data_q	a801_0002	Net [State]	spi...	inst1/p...	4
data_o	a801_0002	Output	spi...	inst1/p...	4
data_o	a801_0002	Output	spi...	inst1/p...	4
dst_data_o	a801_0002	Output	cd...	inst1/p...	4
dst_data_o	a801_0002	Output	cd...	inst1/p...	4
dst_data_o	a801_0002	Output	ud...	inst1/p...	4
s_udma_cmd	a801_0002	Net	ud...	inst1/p...	4
udma_cmd_i	a801_0002	Input	ud...	inst1/p...	4
s_cd_cfg_check	301	Net	ud...	inst1/p...	4
s_cd_cfg_chk_type	0	Net	ud...	inst1/p...	4
s_cd_cfg_clkdiv	1	Net	ud...	inst1/p...	4
s_cd_cfg_cpha	1	Net	ud...	inst1/p...	4
s_cd_cfg_cpol	1	Net	ud...	inst1/p...	4
s_cd_cfg_lsb	0	Net	ud...	inst1/p...	4
s_cd_cfg_qpi	1	Net	ud...	inst1/p...	4
s_qpi	1	Net	ud...	inst1/p...	4
rx_qpi_o	1	Output	ud...	inst1/p...	4
s_rx_qpi	1	Net	ud...	inst1/p...	4
rx_qpi_i	1	Input	ud...	inst1/p...	4
s_data_rx	fbe5_0000	Net	ud...	inst1/p...	4
s_rx_counter_bits	3	Net	ud...	inst1/p...	4
s_rx_mode	3	Net	ud...	inst1/p...	4
s_spi_mode	0	Net	ud...	inst1/p...	4
r_spi_mode	2	Net [State]	ud...	inst1/p...	5
r_spi_mode	2	Net [State]	ud...	inst1/p...	6
spi_oen0_o	0	Output	ud...	inst1/p...	5
spi_oen0_o	0	Output	ud...	inst1/p...	5
spi_oen	{0}	Output	ud...	inst1/p...	5
spi_oen_o	{0}	Output	so...	inst1/p...	5
spi_oen_o	{0}	Output	pu...	inst1/p...	5
spi_oen_o	{0}	Output	so...	inst1/s...	5
spi_oen1_o	0	Output	ud...	inst1/p...	5

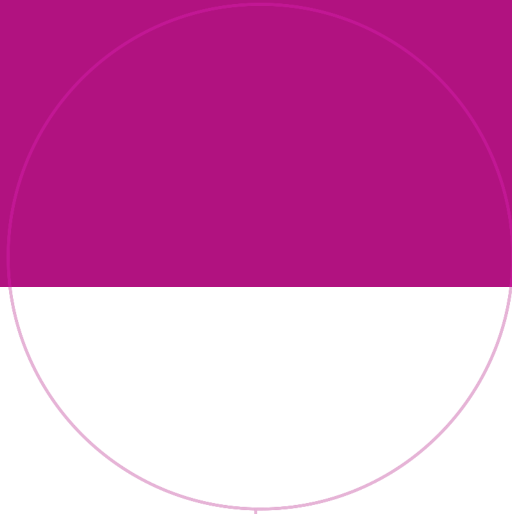
Figure B.1: Whole propagation path from uDMA to sdi_oen_o, shown in fanout view

C Explanation to the workspace

The workspace provided includes the final workspace as it was during the experiment that was used to find the first uDMA L-alert. This experiment used the alternative route, and thus also black-boxed more modules.

The experiment is in the folder 'uDMA_propagation_and_bb', which is detailed below. The rest of the folders include the RTL of the PULPissimo platform and the intellectual properties for all the modules within the system. It also contains the document file of the PULPissimo platform from the PULPissimo GitHub repository (The Parallel Ultra Low Power platform, n.d).

- The folder 'uDMA_propagation_and_bb' involves the '.tcl' file to run the proof, as well as the property file. This property is set up to test the uDMA against the states as seen in Figure 6.3. The folder also contains the subfolders 'miter_top' and 'state_constraints', as detailed below.
- This test black-boxes some modules, as described in Chapter 5, and therefore also includes the black-boxing constraints in the 'udma_bb_constraints.sv' file, under the sub-folder 'state_constraints'. This sets the outputs of the black-boxed modules to be equal, to avoid any new counterexamples occurring because of these modules behaving differently in the two instances.
- Also under 'state_constraints' is the 'state_constraints.sv' which includes all the state constraints of the system if no extra modules are black-boxed. This file includes the 'state_equivalence' function, the 'state_uniqueness' function, and the 'soc_obs_states' function. This file has been used when looking for the P-alerts, and looking at further propagation without any black-boxed modules.
- The 'state_constraints_udma_check.sv' in the subfolder 'state_constraints' is used for the uDMA experiment. This is the same file as the one described above, just with fewer constraints, as several modules have been black-boxed.
- The top module, 'miter_top.sv' can be found under the 'miter_top' sub-folder.



Norwegian University of
Science and Technology