SAARLAND UNIVERSITY

BACHELOR THESIS

Automatic Inference of Hardware-Software Contracts for Open-Source Processors

---

Submitted by:
Gideon MOHR

Supervisor:
Prof. Jan REINEKE

Second Reviewer:
Prof. Marco GUARNIERI

Faculty of Mathematics and Computer Science
Department of Computer Science
Real-Time and Embedded Systems Lab

17.03.2023

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ______________________ ______________________

(Datum/Date) (Unterschrift/Signature)

Abstract

Security of both hardware and software has always been an important concern, especially for applications handling sensitive information. In recent years, hardware has been threatened more and more by side-channel attacks.

At this point, hardware-software contracts could be a major improvement for microarchitectural security. Such contracts allow to specify possible information leakage through microarchitectural side channels on the level of the instruction set architecture (ISA), thus software developers, given such a contract, can ensure that sensitive data is protected by avoiding certain instructions or operations.

However, for most processors available today, the respective hardware-software contract has not been specified yet. While in the future it could be possible to incorporate hardware-software contracts in the design process of new hardware, finding a valid and meaningful contract for existing hardware is hard, even for simple designs.

This work presents an algorithm and its implementation that automatically generates a contract candidate for a given microarchitecture by analyzing execution traces to determine which executions can be distinguished by an adversary and to extract architectural differences that could be identified by a possible contract. These observations are collected and allow to eventually compute a contract candidate.

Contents

Abstract **iii**

1 Introduction **1**

2 Background **3**
2.1 Side Channels and Side-Channel Attacks 3
2.2 Hardware-Software Contracts 4
2.2.1 Instruction Set Architectures and Microarchitectures 4
2.2.2 Contract Satisfaction 6
2.2.3 Instruction-Level Contracts 7
2.3 The RISC-V ISA Family 8

3 The Contract Generation Problem **9**
3.1 General Definition 9
3.2 Ensuring the Existence of a Contract 10
3.3 A Template for Contracts 11
3.4 Limiting the Set of Test Cases 14

4 An Algorithm to Generate Contracts **17**
4.1 Abstract Overview 17
4.2 Simulation 18
4.3 Extraction of Possible Observations 18
4.4 Computing Contract Candidates 19
4.5 Test Case Generation 21

5 Implementation Details **23**
5.1 The Testbench Design 24
5.1.1 Synchronizing the Two Cores 25
5.1.2 Extracting the Architectural State 26
5.1.3 Determining Adversary Distinguishability 26
5.2 Extraction of Possible Contract Observations 26
5.3 Test Case Generation 27
5.4 Integrating the Ibex and CVA6 Processors 28
5.4.1 The Ibex Core 28
5.4.2 The CVA6 Core 29

6 Evaluation **31**
6.1 The Generated Contract 31
6.2 The Evolution of the Contract during Generation 32
6.2.1 The Size of the Computed Contract Candidate 32
6.2.2 The Quality of the Generated Contract Candidate 33
6.3 Computation Time 36

7 Related Work **37**
7.1 Side-Channel Attacks 37
7.2 Hardware-Software Contracts 37

8 Summary and Conclusion **39**

Bibliography **41**

1 Introduction

Innovations in computer science introduced digital elements to many different parts of our daily lives and increasingly also involve processing sensitive information that is required to be kept confidential. At the same time, many companies rely on cloud computing to process and store this information. Among other factors, this demands for a higher level of security of both hardware and software. Unfortunately, even if there were no bugs in software and if every piece of hardware behaved according to its specification, there are still possible attacks.

The abstraction layer between hardware and software, the instruction set architecture, hides implementation details of the hardware on which the software is running. This makes it easy to build software that runs on many different processors, but also allows for the extraction of information through unintended communication channels, called side channels. The information obtained through such a channel can be used to gain information about sensitive data. An attack that uses side channels to extract confidential information is called a side-channel attack. Many of these side-channel attacks rely on timing differences introduced by microarchitectural optimizations such as caches.

Over the last decades, side-channel attacks gained a lot of attention as those attacks affected many devices and preventing these attacks significantly decreased the performance of affected processors. Researchers have since then tried to find ways to build processors that do not expose any sensitive information or prevent leakage on existing processors with a smaller performance penalty.

While it is clear that side channels cannot be removed completely, e.g. every computation will consume time, it can be possible to reduce or remove data leakage through a side channel, e.g. by making sure that every possible computation consumes exactly the same amount of time or every execution accesses the same sequence of memory addresses. However, this means that any optimization that only improves a certain subset of computations will not be able to have any positive impact on performance as this performance improvement would leak information about the processed data. Without these optimizations, current hardware would be significantly slower, therefore, it is not feasible to remove leakage through side channels altogether.

Hardware-software contracts are one approach to formally capture the leakage that can be observed through a given side channel. These contracts can be formulated independently of the concrete hardware and thus provide an interface to the software

level. The information contained in a hardware-software contract can thus be used by higher abstraction levels such as the operating system or other software applications to protect certain secrets from leaking through side channels. This could significantly improve the security of systems processing sensitive information in untrusted environments. However, as of today, hardware-software contracts have rarely been specified and thus their usage in many applications is limited.

This work presents an algorithm that allows to automatically generate hardware-software contracts for a given microarchitecture. The focus lies on instruction-level contracts which are introduced in the next chapter. This type of contract aims to associate leakages through side channels directly with the executed instructions.

Chapter 3 formally captures the problem of contract generation and shows why it is infeasible to obtain the optimal hardware-software contract. Most importantly, as of today, it is not possible to prove the validity of a certain-hardware software contract, thus this work focuses on generating a hardware-software contract based on a limited set of execution traces. Furthermore, efficient usage of hardware-software contracts is only possible if they can be specified in a predefined structure. Thus, Section 3.3 introduces a template that restricts the space of possible contracts, but allows for better analysis and guides the contract generation. While such a contract candidate is not a complete solution, it can provide a starting point for further analysis and indicates vulnerable parts of the given microarchitecture.

With those limitations it is possible to formulate an algorithm in Chapter 4 that computes the (or one of the) optimal contract candidates based on the evaluated execution traces. This includes generating the execution traces using simulation, analyzing the obtained traces for possible additions to the contract, and eventually computing the optimal solution based on the results. In order to evaluate the presented approach, two open-source processors, the Ibex and CVA6 core, are integrated into this workflow. More details on the necessary steps are presented in Chapter 5.

Chapter 6 then has a closer look at the generated contract candidates and tries to estimate the accuracy and precision of the results based on further measurements on larger test sets.

2 Background

2.1 Side Channels and Side-Channel Attacks

Generally speaking, a side channel provides the possibility to obtain certain information about a system indirectly, i.e. not through an intended channel of communication.

The concept is not limited to computer science, for example when cracking a safe in some cases it is possible to hear the internal mechanics of the lock providing you feedback on your current input. While the intended "communication" channel would only give the information "correct" or "incorrect", the sounds can be interpreted to obtain more details such as "first digit correct". This information can be abused to efficiently infer the correct combination to open the safe. Such an attack is called a side-channel attack.

In the area of computer science, microarchitectural side channels have been used for a multitude of different attacks over the last few years. While this type of attack is not new, it turns out to be very difficult to prevent data leakage through these side channels.

There also can be side channels in the software itself, but these tend to be fixed more easily as software can usually be updated easily after deployment. Furthermore, there are formal methods that can help to avoid at least certain types of side channels in software. Hardware side channels usually arise from implementation details in the so-called microarchitecture and are thus not detectable at higher abstraction layers such as at the level of the instruction set architecture (ISA). A formal definition of these two terms will be given in the next section, at the moment, it is important to know that the ISA provides the functional semantics of a system. The microarchitecture implements the functionality requested by the ISA in hardware. Different types of side-channel attacks use different implementation details of the microarchitecture such as the timing, the power consumption, or certain effects of optimizations intended to improve performance.

As hardware operates in the physical world, time- and power-based side channels inherently exist. The only option to eliminate side-channel attacks would be to eliminate information leakage through this channel, however, this would require a system to always exhibit the worst-case or even to repeat the computation for any possible input which would drastically decrease performance and make obsolete most optimizations implemented in the hardware.

2.2 Hardware-Software Contracts

As has just been explained, in many cases, it is not feasible to close side channels in a way that no data can be leaked through a side channel. Therefore, hardware-software contracts do not try to eliminate data leaking through side channels but aim to capture what information could potentially be leaked through a specific side channel. This information then can be used to protect sensitive information at a higher level making sure it cannot be obtained through side channels.

Before defining hardware-software contracts, a few preliminary definitions will be required. Hardware-software contracts are tightly connected to the differences between architectural and microarchitectural semantics, thus the following paragraphs introduce formalizations of these concepts that are sufficient for the scope of this work.

2.2.1 Instruction Set Architectures and Microarchitectures

The architectural semantics are given by the instruction set architecture (ISA) and describe the functional behavior at the level of instructions.

Definition 1 (Instruction Set Architecture)**.** The semantics of an instruction set architecture can formally be seen as a function $\texttt{ISA} : \Sigma \to \Sigma$ that takes an architectural state $\sigma \in \Sigma$ and produces the subsequent state according to the semantics of the instruction set architecture.

While the most common ISAs such as x86, ARMv8, or RISC-V are not explicitly given as formalized functions but rather via textual descriptions, this definition implicitly also applies to those. For all of these ISAs, the semantics are specified step-wise, i.e. by describing how the current state determines the instruction executed next and how this instruction alters the architectural state yielding a function mapping one architectural state to its successor.

The definition of an ISA is usually kept as abstract as possible by only defining the sequence of architectural states while the concrete implementation of the functionality is left to the hardware designer. This allows for different design variations and various optimizations depending on the intended use case for this specific design, leading to various microarchitectures implementing the same ISA. While microcontrollers or battery-powered devices need to be as efficient as possible, server CPUs have to meet completely different requirements. Therefore, hardware designers can freely implement optimizations such as caching, pipelining, or out-of-order execution. The formal definition of a microarchitectural semantic seems very similar to the definition of the ISA from above, however, one step in the microarchitectural semantic represents one clock cycle instead of one instruction as seen in the definition of the ISA.

Definition 2 (Microarchitecture)**.** The semantics of a microarchitecture can formally be defined as a function $\texttt{MARCH} : M \to M$ where M denotes the set of microarchitectural states.

Actually, if the microarchitectural state only contains the architectural state, the definitions are equivalent. However, in most cases to achieve the optimizations mentioned above, additional state is required. In all cases, it is possible to extract the current architectural state given a microarchitectural state using a function $\texttt{ARCH} : M \rightarrow \Sigma$. In the context of hardware-software contracts, it is actually important to also compare the contents of the microarchitectural state that is not reflected in the architectural state. For two microarchitectural states μ_0 and μ_1, let $\mu_0 \equiv_{\texttt{MARCH}} \mu_1$ denote, that the microarchitectural part of the state is equal, i.e. that any difference between μ_0 and μ_1 must be reflected in the architectural state they represent or formally:

$$\forall \mu_0, \mu_1 \in M.\ \mu_0 \neq \mu_1 \wedge \texttt{ARCH}(\mu_0) = \texttt{ARCH}(\mu_1) \Rightarrow \mu_0 \not\equiv_{\texttt{MARCH}} \mu_1$$

It is also possible that two subsequent microarchitectural states can be mapped to the same architectural state, e.g. when a complex operation takes more than one clock cycle. Therefore, it is important to not only consider the states individually but rather the whole trace of the execution:

Definition 3 (Execution Traces)**.** In f^*, the operator * is a shorthand for the repeated application of a function $f : X \rightarrow X$ for any set X, i.e. for $x_i \in X, i \in \mathbb{N}$ the following holds:

$$f^*(x_0) := [x_0, f(x_0), f(f(x_0)), ...] := [x_0, x_1, x_2, ...]$$

Furthermore, let $g(f^*(x))$ with $g : X \rightarrow Y$ for a given set Y and $y_i \in Y, i \in \mathbb{N}$ be defined as:

$$g(f^*(x_0)) := [g(x_0), g(x_1), g(x_2), ...] := [y_0, y_1, y_2, ...]$$

The sequences of values such as $[x_0, x_1, x_2, ...]$ or $[y_0, y_1, y_2, ...]$ are called execution traces of f or the composition $g \circ f$ respectively.

One could now assume that a microarchitecture implements an ISA if there is one matching architectural trace for every microarchitectural trace. However, as pointed out above, in some cases multiple cycles could be needed for one architectural step. The following definition allows to filter out these parts of a trace:

Definition 4 (Asymmetric Weak Trace Equivalence)**.** One trace $y = [y_0, y_1, ...]$ is weakly equivalent to a trace $x = [x_0, x_1, ...]$ (written $x \sim y$) iff there is a sequence $i_n \in \mathbb{N}, n \in \mathbb{N}$ with $i_n < i_{n+1}$, such that

$$[x_0, x_1, ...] = [y_{i_0}, y_{i_1}, ...]$$

and

$$\forall n \in \mathbb{N}.\ \forall j \in \mathbb{N}.\ i_n < j < i_{n+1} \Rightarrow y_{i_n} = y_j$$

Using this definition, it is finally possible to define what it means to correctly implement an ISA:

Definition 5 (Implementing an ISA)**.** A microarchitecture $\texttt{MARCH}$ implements a given $\texttt{ISA}$ if and only if the following property holds:

$$\forall \sigma_0 \in \Sigma.\ \forall \mu_0 \in M.\ \texttt{ARCH}(\mu_0) = \sigma_0 \Rightarrow \texttt{ISA}^*(\sigma_0) \sim \texttt{ARCH}(\texttt{MARCH}^*(\mu_0))$$

Intuitively, for any given state, the microarchitectural trace must continue as defined by the ISA with the possibility of repeating the same architectural state multiple times. This definition excludes some valid microarchitectures if the ISA allows that e.g. two instructions can be completed in one cycle. This means that one microarchitectural step corresponds to two architectural steps, but as the microarchitectures considered in this work retire at most one instruction per cycle, this definition will be sufficient in the following.

2.2.2 Contract Satisfaction

In the context of hardware-software contracts, specifying an adversary model is crucial. Depending on the setup, an adversary might be able to infer different kinds of information about the execution trace of the given microarchitecture. However, in almost every case, an observation can be linked to a specific part of the microarchitectural state, e.g. a specific register. Nevertheless, adversaries can be defined in a more general way by not restricting the set of possible adversary observations.

Definition 6 (Adversary)**.** The observations an adversary can make in a single step of the microarchitectural execution are described by a function $\texttt{ADV}_{\texttt{SS}} : M \rightarrow A$ where A is the set of possible adversary observations. The trace of observations starting in a given state $\mu \in M$ can be seen as a function $\texttt{ADV} : M \rightarrow \mathcal{L}(A)$ defined as $\texttt{ADV}(\mu) := \texttt{ADV}_{\texttt{SS}}(\texttt{MARCH}^*(\mu))$.

The idea of hardware-software contracts is to obtain information about the leakage the adversary can get based on the architectural state or more precisely which two executions the adversary can distinguish based on the leakage traces. The contract is a function that extracts a sequence of architectural observations:

Definition 7 (Hardware-Software Contract)**.** The architectural observations for a specific contract in a single step of the architectural execution are described by a function $\texttt{CTR}_{\texttt{SS}} : \Sigma \rightarrow O$. The trace of observations starting in a given state $\sigma \in \sigma$ can be seen as a function $\texttt{CTR} : M \rightarrow \mathcal{L}(O)$ defined as $\texttt{CTR}(\sigma) := \texttt{CTR}_{\texttt{SS}}(\texttt{ISA}^*(\sigma))$. Such a function is called hardware-software contract.

These definitions allow to determine what it means to satisfy a hardware-software contract. A microarchitecture satisfies a contract for a given adversary model iff every difference visible to the adversary is explained by the architectural observations, given that the microarchitectural part of the state is initially equal. This restriction is important, as otherwise there might be adversary observations which cannot be explained by the contract. For example, let μ_0 and μ_1 be two different states that

represent the same architectural state. Thus, the trace of architectural states will be the same throughout the execution, thus the contract observations cannot be different. However, the adversary might be able to extract the initial differences by observing the respective part of the microarchitectural state. Therefore, this restriction is a prerequisite in the following definition:

Definition 8 (Contract Satisfaction)**.** Microarchitecture MARCH satisfies contract CTR under the adversary ADV, written $\mathtt{MARCH} \vDash_{\mathtt{ADV}} \mathtt{CTR}$ iff the following holds:

$$\begin{aligned}&\forall \mu_0, \mu_1 \in M.\\ &\quad \mu_0 \equiv_{\mathtt{MARCH}} \mu_1 \land \mathtt{CTR}(\mathtt{ARCH}(\mu_0)) = \mathtt{CTR}(\mathtt{ARCH}(\mu_1))\\ &\quad \Rightarrow \mathtt{ADV}(\mu_0) = \mathtt{ADV}(\mu_1)\end{aligned}$$

This definition of hardware-software contracts has the advantage that the contract purely depends on the architectural state. Thus, the information can be processed independently from the concrete microarchitecture. Furthermore, contracts are allowed to over-approximate the leakage, thus it is possible to find a contract that is satisfied by multiple microarchitectures. This allows for various use cases for these hardware-software contracts. For example, compilers could take a contract as input and generate code that makes specified secrets not observable for an adversary on any microarchitecture implementing the given contract by ensuring the contract observations are independent of the value of the secrets.

2.2.3 Instruction-Level Contracts

This work considers a special type of contracts – instruction-level contracts. For all major ISAs, programs are defined as a sequence of instructions whose format is specified by the ISA. An execution is therefore a sequence of executed instructions possibly altering the architectural state. The idea of instruction-level contracts is to associate observations with one of the executed instructions. To formalize this, the first step is to extract the sequence of instructions:

Definition 9 (Instruction-Level Contract)**.** Let I be the set of instructions defined by ISA and let CTX be a set of contexts that contain the relevant part of the architectural state needed to execute an instruction.

Then $\mathtt{INSTR} : \Sigma \rightarrow (I \times \mathtt{CTX})$ defined as

$$\mathtt{INSTR}(\sigma) := (i, ctx)$$

for $\sigma \in \Sigma$ extracts the first instruction of the (remaining) program alongside the context needed to execute the instruction.

Now define a function $\mathtt{CTR}_{\mathtt{INSTR}} : (I \times \mathtt{CTX}) \rightarrow O$ which maps an instruction in its context to the corresponding architectural observations. The composition of these

two functions can be seen as a contract for a single step in the architectural execution:

$$\texttt{CTR}_{\texttt{SS}} := \texttt{CTR}_{\texttt{INSTR}} \circ \texttt{INSTR}$$

A contract CTR for the ISA ISA that can be specified as a composition of the ISA and the two aforementioned functions, i.e. $\texttt{CTR}(\sigma) = \texttt{CTR}_{\texttt{INSTR}}(\texttt{INSTR}(\texttt{ISA}^*(\sigma)))$ is called instruction-level contract.

Note that the contents of a context are not defined further at this point to allow to include any information needed such as the content of registers or the content of memory at certain addresses.

2.3 The RISC-V ISA Family

This work focuses on processors implementing an ISA from the RISC-V family [18]. RISC-V was originally developed at the University of California, Berkeley and is now maintained by the RISC-V Foundation. RISC stands for "Reduced Instruction Set Computer" which means that the ISA aims to provide fewer and more basic instructions compared to CISC (Complex Instruction Set Computer) architectures and thus might require more instructions to execute a given task, however, the provided instruction can potentially be executed more efficiently, allowing e.g. more efficient and shorter pipelines.

RISC-V provides multiple base instruction sets for different instruction lengths. Furthermore, there is a multitude of ISA extensions providing more specialized or advanced functionality. Thus there is not one RISC-V ISA but rather every processor implements a certain subset of the RISC-V ISA. The capabilities of a core are specified in the exact name of the ISA it implements. The Ibex [19] core for example implements RV32I[M]C[B]. This means that it implements the 32-bit base integer instruction set (I), the extension for compressed instructions (C), and optionally the extensions for integer multiplication and division (M) as well as for bit manipulation (B).

There are different formats of instructions to efficiently encode the required attributes of each individual instruction and to accelerate decoding of instructions. In order to identify instructions, all of them include the OPCODE and some instructions contain additionally the FUNCT7 and FUNCT3 to further determine the exact operation. Instructions can include references to the registers RS1, RS2 and RD where RS1 and RS2 are source registers which will be read and the result will be saved in the register RD. Some instructions contain an immediate value IMM which can have various lengths depending on the exact type of the instruction.

3 The Contract Generation Problem

As explained in the previous chapter, hardware-software contracts could be a major improvement to mitigate side-channel attacks as they provide a well-defined interface between hardware and software such that application developers can make sure sensitive data is protected well enough against a given adversary model.

In the future, processors could be designed with a given hardware-software contract in mind, thus continuous testing during development could ensure that the resulting product is compliant with the given contract. However, as of today, hardware-software contracts have not been formulated for almost any processors available today, thus it is important to derive possible contracts for an existing microarchitecture as well.

This work focuses on the latter and explores how candidates for hardware-software contracts can be inferred automatically for a given microarchitecture implementing a subset of the RISC-V ISA.

3.1 General Definition

While it is trivial to specify a contract that satisfies contract satisfaction by leaking the complete architectural state, i.e. by making any two different executions contract distinguishable, such a contract is generally not desired. A precise contract classifies two executions as distinguishable only if the adversary is able to distinguish those two executions as well. Thus, a "good" contract minimizes the executions it classifies as distinguishable. In the best case, it only classifies two executions as distinguishable if and only if the adversary is able to distinguish the two cases. However, this is not always achievable as leakages might only occur in certain microarchitectural configurations which are invisible to the contract as it only takes the architectural state into account.

In order to define what the "best" contract looks like, an order on contracts is required:

Definition 10 (Preorder on contracts)**.** Let $\mathtt{CTR}$ and $\mathtt{CTR}'$ be two contracts, then

$$\mathtt{CTR} \leq \mathtt{CTR}' :\Leftrightarrow \forall \sigma, \sigma' \in \Sigma.\ \mathtt{CTR}(\sigma) \neq \mathtt{CTR}(\sigma') \Rightarrow \mathtt{CTR}'(\sigma) \neq \mathtt{CTR}'(\sigma')$$

Intuitively, the least or one of the least elements of this preorder corresponds to the best contract. Using the notation introduced in the previous chapter, the problem of contract generation can be formalized as follows:

Definition 11 (Optimal Hardware-Software Contract Generation)**.** Given an instruction set architecture ISA, a microarchitecture MARCH that implements ISA, and an adversary model ADV suitable for MARCH, find a hardware-software contract CTR that satisfies the following:

1. Contract satisfaction (c.f. Definition 8):

$$\mathtt{MARCH} \vDash_{\mathtt{ADV}} \mathtt{CTR}$$

2. Least contract:

$$\begin{aligned} &\forall \mathtt{CTR}'.\ \mathtt{MARCH} \vDash_{\mathtt{ADV}} \mathtt{CTR}' \\ &\Rightarrow \mathtt{CTR} \leq \mathtt{CTR}' \end{aligned}$$

Once generated and proven to be correct, such a contract could be used to precisely predict any leakage for a given program to ensure the protection of sensitive data. However, the next three sections illustrate several challenges that restrict the generation in reality.

3.2 Ensuring the Existence of a Contract

Definition 11 requires the optimal contract to be a least element of the preordered set of contracts. However, the existence of such an element cannot be guaranteed, thus an optimal solution might not always exist. Nevertheless, it is still possible to specify a contract that satisfies Definition 8, it might just be incomparable to other valid contracts, thus it is not the least element by definition.

One possibility to ensure the existence of a contract is to require the contract to be a minimal element of the preordered set of contracts. This ensures that if there is a least contract, this contract is chosen, and if there is not, any minimal contract can be chosen as contract.

Experiments have shown, that there might be multiple minimal contracts with very different properties. There could be a contract that classifies almost every execution as distinguishable except for one which any other valid contract classifies as distinguishable. This contract is a minimal element even if intuitively most other valid contracts would be better because they classify fewer executions as distinguishable. Thus, it makes sense to classify contracts according to another performance measurement while making sure that the chosen contract is still minimal (or even least, if a least contract exists). In order to be a valid contract, a contract needs to classify

every pair of executions as distinguishable which the adversary can distinguish. A better contract intuitively classifies fewer executions falsely as distinguishable, thus optimizes the precision of the resulting contract.

As there is by definition no execution that the adversary can distinguish but the contract does not classify as such, the precision corresponds to the number of executions falsely classified as distinguishable:

$$p(\mathtt{CTR}) := |\{(\mu, \mu') \in M^2 \mid \mathtt{ADV}(\mu) = \mathtt{ADV}(\mu') \wedge \mathtt{CTR}(\mathtt{ARCH}(\mu)) \neq \mathtt{CTR}(\mathtt{ARCH}(\mu'))\}|$$

Given such a function p, the second requirement of Definition 11 becomes:

2. Most precise contract:

$$\forall \mathtt{CTR}'.\ \mathtt{MARCH} \vDash_{\mathtt{ADV}} \mathtt{CTR}' \Rightarrow p(\mathtt{CTR}) \leq p(\mathtt{CTR}')$$

This definition of p minimizes the number of executions falsely classified as distinguishable and together with the requirement of contract satisfaction, it minimizes the number of executions classified as distinguishable in general. Such a contract is a minimal element according to the preorder from Definition 10 and if a least element exists, by definition, any minimal element is a least element, thus if there is a least contract, such a contract will be selected.

3.3 A Template for Contracts

In Definition 11, the contract can be an arbitrary function. However, this may make contracts very hard to specify or interpret for any further application. This work uses a template that defines which information from the architectural state can be used and combined to determine whether two executions are distinguishable. Unfortunately, this also means that certain contracts cannot be specified in this template, thus, the generated contract might not be an optimal solution according to Definition 11, but only the optimal solution within the set of possible contracts.

To allow to restrict the space of contracts, the problem can be reformulated as follows:

Definition 12 (Hardware-Software Contract Generation from a Restricted Contract Space)**.** Given an instruction set architecture ISA, a microarchitecture MARCH that implements ISA, an adversary model ADV suitable for MARCH, a space of possible contracts C, and a function $p : C \to \mathbb{N}$ as presented in Section 3.2, find a hardware-software contract $\mathtt{CTR} \in C$ that satisfies the following:

1. Contract satisfaction (c.f. Definition 8):

$$\mathtt{MARCH} \vDash_{\mathtt{ADV}} \mathtt{CTR}$$

2. Most precise contract:

$$\forall \texttt{CTR}' \in C.\ \texttt{MARCH} \vDash_{\texttt{ADV}} \texttt{CTR}' \\ \Rightarrow p(\texttt{CTR}) \leq p(\texttt{CTR}')$$

Given this definition, the next task is to fix a suitable set C of contracts. As this work considers instruction-level contracts, the contract template needs to be specified according to Definition 9. Thus, the contract observations must be extracted from the instruction and its corresponding context which can include the values of registers or the content of memory.

As a first step possible leakages have to be identified, i.e. the set of architectural observations O needs to be characterized. For a given ISA, there are different possibilities on what O can look like, however, it needs to be powerful enough to capture all considered leakages. One way to specify $\texttt{CTR}_{\texttt{INSTR}}$ is to make it extract certain parts of the context as an observation, thus for a given instruction that e.g. reads the register `RS1`, the value of `RS1` included in the context could be an observation.

In the case of RISC-V, there are different types of instructions whose effect depends on various parts of the architectural state. The following promising candidates of values that could be considered by the contract and thus need to be included in the context can be identified:

- `TYPE`: The type of the executed instruction, composed of the `OPCODE`, `FUNCT7` and `FUNCT3`.
- `RS1`, `RS2`: The source register numbers `RS1` and `RS2` that contain the operands of the instruction.
- `RD`: The destination register number in which the results will be stored.
- `IMM`: The immediate value, which is directly encoded into the instruction.
- `reg[RS1]`, `reg[RS2]`: The values stored in the registers `RS1` and `RS2`.
- `mem[reg[RS1]+IMM]`, `mem[reg[RS2]+IMM]`: The content of the memory which is possibly accessed by the instruction.

It might not be obvious that all of these components could cause a difference, however, one can imagine cases where it e.g. takes longer to access a certain subset of registers if they are located further away from the core. Furthermore, not every instruction has an immediate value or accesses memory, thus not all of those possible observations may be applicable to a given instruction.

The next important step is to determine the granularity of the contracts. While it is possible to specify very fine-grained contracts (e.g. if an instruction is of a specific type, the sum of the two operands is 42 and the results are written back to an odd register number, then the immediate value is observable) or very coarse-grained

contracts (e.g. every instruction leaks the immediate value), none of those extremes is suitable in this context. If the contract is very fine-grained, many test cases will be needed to obtain a reasonably accurate contract, while a coarse-grained contract very rapidly becomes meaningless as it starts to leak everything.

Experiments have shown that reasonable contracts can be obtained if the type of an instruction is used to determine architectural observations as instructions of the same type usually behave similarly.

Thus, contracts considered in this work can be expressed as a set of pairs that include the type of the instruction and the observed architectural value, e.g. if the immediate value of a load instruction influences the adversary observations, the pair $(\texttt{load}, \texttt{IMM})$ should be included in the contract. Generally, these pairs are composed of an instruction type $t_i \in T_{\texttt{INSTR}}$ and an observation type $t_o \in \{$TYPE, RS1, RS2, RD, IMM, reg[RS1], reg[RS2], mem[reg[RS1]+IMM], mem[reg[RS2]+IMM]$\}$. Formally, Definition 9 requires an instruction-level contract to be a function producing the architectural observations given the instruction and its context. The following shows that it is possible to transform a set CTR of pairs as described above into a function $\texttt{CTR}_{\texttt{INSTR}}$ as required by the definition:

$$\texttt{CTR}_{\texttt{INSTR}}(i, \texttt{ctx}) := \{(t_o, v_o) \mid (t_i, t_o) \in \texttt{CTR} \wedge \texttt{ctx}[t_o] = v_o\}$$

In this equation, t_i is the instruction type corresponding to instruction i, and $\texttt{ctx}[t_o]$ extracts the value v_o indicated by the observation type t_o from the context.

To simplify notation, in the following, contracts constructed using this template are used interchangeably as a function or as a set of pairs, depending on the context. The set C of contracts includes all contracts that can be expressed using the above notation.

One important question now is how restrictive this definition of C is in practice. The fact that C can only consider the observations listed above, makes clear that any leakage caused by a difference of a value not included in this list cannot be explained by a contract $\texttt{CTR} \in C$. In theory, the adversary could be able to observe the entire architectural state or any property about the values stored in the architectural state, thus any difference in the architectural state could cause different adversary observations. However, the idea of instruction-level contracts is that adversary observations can be associated with a specific instruction, thus these cases are out of scope for this work. The list of possible observations presented above includes the entire architectural state that is directly associated with a specific instruction, thus many instruction-specific leakages can be expressed by contracts in C.

In the future, it might make sense to allow for more specific values to be considered by the contract, e.g. to distinguish aligned and unaligned memory accesses, as this allows to specify leakages at a higher granularity. A more advanced analysis could

also include e.g. adjacent memory addresses to allow specifying leakage e.g. caused by a prefetcher. However, the directly associated architectural state as listed above turned out to be sufficient for the processors analyzed in this work.

3.4 Limiting the Set of Test Cases

The second challenge is that currently there is no feasible method to prove contract satisfaction, thus the above problem cannot be solved completely. This work, therefore, tries to approximate the optimal contract by simulating a limited set of test cases, analyzing the adversary observations, and afterwards computing the optimal contract candidate according to the test cases considered.

Each of these test cases is specified as a pair of two architectural states (note that an architectural state implicitly includes a sequence of instructions to be executed) which has the advantage that the same set of test cases can be reused for various microarchitectures implementing the same ISA. Consequently, there must be a mapping from architectural to microarchitectural states to determine the initial state used for the simulation. Thus, there is a function $\texttt{INIT} : \Sigma \to M$ such that

$$\forall \sigma \in \Sigma.\ \texttt{ARCH}(\texttt{INIT}(\sigma)) = \sigma$$

or in other words, INIT maps an architectural state to a valid microarchitectural state that represents the intended architectural state. The resulting microarchitectural states must be microarchitecturally equivalent to ensure that a $\texttt{CTR} \in C$ that satisfies Definition 8 also satisfies contract candidate satisfaction as defined below. This means, that INIT must satisfy the following:

$$\forall \sigma_0, \sigma_1 \in \Sigma.\ \texttt{INIT}(\sigma_0) \equiv_{\texttt{MARCH}} \texttt{INIT}(\sigma_1)$$

This allows to define contract candidate satisfaction:

Definition 13 (Contract Candidate Satisfaction)**.** Given a set $\texttt{TC} \subseteq \Sigma^2$ of pairs of architectural states, a microarchitecture MARCH, an adversary ADV and the set of possible contracts C, a contract $\texttt{CTR} \in C$ is called contract candidate, written $\texttt{MARCH} \vDash^{\texttt{TC}}_{\texttt{ADV}} \texttt{CTR}$ iff the following holds:

$$\begin{aligned}
&\forall (\sigma_0, \sigma_1) \in \texttt{TC}.\\
&\quad \texttt{CTR}(\sigma_0) = \texttt{CTR}(\sigma_1)\\
&\quad \Rightarrow \texttt{ADV}(\texttt{INIT}(\sigma_0)) = \texttt{ADV}(\texttt{INIT}(\sigma_1))
\end{aligned}$$

Using this definition, the problem solved in this work can be defined as follows:

Definition 14 (Optimal Hardware-Software Contract Candidate Generation)**.** Given an instruction set architecture ISA, a microarchitecture MARCH that implements ISA, an adversary model ADV suitable for MARCH, a space of possible contracts C, a function $p : C \to \mathbb{N}$ as presented in Section 3.2, and a set of test cases TC, find a hardware-software contract candidate CTR that satisfies the following:

1. Contract candidate satisfaction (c.f. Definition 13):

$$\texttt{MARCH} \vDash^{\texttt{TC}}_{\texttt{ADV}} \texttt{CTR}$$

2. Most precise contract:

$$\forall \texttt{CTR}' \in C.\ \texttt{MARCH} \vDash^{\texttt{TC}}_{\texttt{ADV}} \texttt{CTR}' \Rightarrow p(\texttt{CTR}) \leq p(\texttt{CTR}')$$

In order to be able to evaluate p, its scope must also be restricted to the test cases being evaluated:

$$p(\texttt{CTR}) := |\{(\mu, \mu') \in M_{\texttt{TC}} \mid \texttt{ADV}(\mu) = \texttt{ADV}(\mu') \land \texttt{CTR}(\texttt{ARCH}(\mu)) \neq \texttt{CTR}(\texttt{ARCH}(\mu'))\}|$$

where $M_{\texttt{TC}} = \{(\texttt{INIT}(\sigma_0), \texttt{INIT}(\sigma_1)) \mid (\sigma_0, \sigma_1) \in \texttt{TC}\}$.

Once proving contract satisfaction is possible, it will be possible to check whether the contract candidate is complete, i.e. if it actually explains any leakage observed by the given adversary.

4 An Algorithm to Generate Contracts

With the restrictions presented in the last chapter in mind, it is possible to formulate an algorithm that computes the optimal contract candidate.

4.1 Abstract Overview

A high-level overview of the algorithm used in this work is given in Algorithm 1. The algorithm is parameterized with the set of test cases. As outlined in the last chapter, each test case consists of two architectural states. The architectural state defines the current valuations of registers and memory as well as the sequence of instructions to be executed (stored in a certain location in memory).

Each of these test cases is simulated using two instances of the given microarchitecture (one for each architectural state included in the test case). The simulation determines whether the adversary is able to distinguish the two executions. Furthermore, the simulation produces a trace of microarchitectural states. More details on the simulation will be given in Section 4.2.

This trace is then analyzed to extract contract observations that would allow a contract to differentiate the two executions. The analysis collects these observations and returns a set `OBS` of possible observations. For every adversary-distinguishable test case, there must be at least one possible observation to allow to compute a contract that satisfies Definition 14. For a more detailed discussion on what is included in the set `OBS`, refer to Section 4.3.

Depending on whether the adversary was able to distinguish the two executions, the set `OBS` is added to the list `Distinguishable` or the list `Indistinguishable`. After evaluating all the test cases, the contract is computed using these two lists. This method needs to ensure contract candidate satisfaction and then optimize the solution according to the function p from Definition 14. The implementation of this method is discussed in Section 4.4.

Algorithm 1 Contract Candidate Generation as described in Section 4.1

```
Input
    MARCH   the processor design
    ADV     the adversary model
    TC[]    the set of test cases
Output
    CTR     the contract candidate

Distinguishable ← EmptyList( )
Indistinguishable ← EmptyList( )
for all TC in TC[] do
    TRACE, ADVDistinguishable ← SIMULATE(MARCH, ADV, TC)
    OBS ← ANALYZE(TRACE)
    if ADVDistinguishable then
        Distinguishable.APPEND(OBS)
    else
        Indistinguishable.APPEND(OBS)
    end if
end for
CTR ← COMPUTE(Distinguishable, Indistinguishable)
```

4.2 Simulation

The goal of the simulation is to determine whether the given adversary is able to distinguish the two architectural states on a running system. This means that both architectural states need to be simulated and the corresponding adversary observations must be compared. To coordinate this simulation, the processor is embedded in a testbench which is responsible to initialize the processor to the given architectural state, obtain the adversary observations and terminate the simulation once the relevant instructions have been executed completely.

Furthermore, the simulation collects the trace of pairs of microarchitectural states which will be used in the next step to extract architectural differences which a contract could use to determine whether these two executions are classified as distinguishable.

4.3 Extraction of Possible Observations

The contract template presented in the last chapter requires contracts to be composed of a set of pairs of an instruction type and the respective leakage, e.g. if the adversary is able to obtain the immediate value of a load operation (or at least parts of it), the contract should include the pair (`load`, `IMM`). The traces obtained in the simulation allow to compute all observations that would make the given executions contract distinguishable.

From each microarchitectural state in the given trace, the corresponding architectural state is computed. The possible observations presented in the last chapter can all be mapped to a specific part of the architectural state, e.g. the immediate can be extracted by looking at the memory location indicated by the current program counter. For every pair of architectural states, these observations are compared. If there is any difference, the corresponding observation is added to the set `OBS` of observations that would distinguish these two executions. This process is repeated for every pair of architectural states contained in the trace.

The approach described above is not the only possibility to determine possible additions to the contract but in practice turned out to be very flexible and efficient.

It would also be possible to start analyzing the last architectural state in the sequence and stop the analysis as soon as an observable difference has been found. This would ensure that there exists at least one contract generated with the extracted observations that can distinguish the two executions. However, it is not always clear if this would be the ideal addition to the contract as the leakage might depend on a certain interaction between two instructions. Imagine for example a store and a subsequent load which might be faster due to caching. The main question now is whether the store that causes the value to be cached or the load that causes the value to be read from the cache should be blamed for an observable difference in execution time. By collecting all possible additions to the contract, the question which of these candidates is included in the final contract candidate is delayed to the contract computation. At this point, the results from other test cases may already force a certain observation to be in the contract, thus the question can be resolved more easily in some cases.

4.4 Computing Contract Candidates

The problem of contract candidate generation formulated in Definition 14 can be translated into an Integer Linear Programming (ILP) [6] problem as follows:

Given the set of distinguishable test cases $D \subseteq \texttt{TC}$, the set of test cases that are equal from the perspective of the adversary $E := \texttt{TC} \backslash D$ and a set of possible observations O that can be added to a contract. Let the function $\texttt{analyze} : \texttt{TC} \to \mathcal{P}(O)$ extract the architectural observations from a given test case, as described in Section 4.3.

Let s_o, $o \in O$ be variables representing whether observation o is part of the contract and c_e, $e \in E$ be variables representing whether the test case e is falsely classified as contract distinguishable. The following constraints ensure contract candidate

satisfaction:

$\forall o \in O.\ 0 \leq s_o \leq 1$	s_o is a boolean variable
$\forall d \in D.\ (\sum_{o \in \texttt{analyze}(d)} s_o) \geq 1$	at least one observation explaining the distinguishability must be selected

In order to allow counting the number of test cases falsely classified as distinguishable, the following constraints are added:

$\forall e \in E.\ 0 \leq c_e \leq 1$	c_e is a boolean variable
$\forall e \in E.\ (\sum_{o \in \texttt{analyze}(e)} s_o) \leq c_e$	c_e is 1 if at least one of the observations explaining e is selected, i.e. if it is falsely classified as distinguishable

The task of the ILP solver is now to minimize the resulting contract's precision (c.f. the function p from Chapter 3). This means that the number of test cases falsely classified as distinguishable needs to be minimized, i.e.

$$\min(\sum_{e \in E} c_e)$$

In practice, instead of the sets D and E of test cases, the method receives the lists `distinguishable` and `indistinguishable` as inputs. Each element of these lists is the set of possible observations corresponding to one test case, thus the function `analyze` is not needed to extract the observations. Consequently, the two lists are sufficient to implement the above constraints to solve the ILP problem and thus to compute a valid contract candidate according to Definition 14.

If there are multiple optimal solutions, the solver is free to choose any of them. This means for the example presented in Section 4.3 that if there are many test cases in which there are stores to different addresses but which the adversary cannot distinguish, the observation from the load operation is likely included in the contract and vice versa. If there is no leakage if only one of the two instructions is present, the solver can select any of the two possible additions to the contract.

It would be possible to compute a contract after each addition, however, as the current contract candidate is not needed for the simulation or the trace analysis, computing the contract once after evaluating all the test cases is sufficient. Thus, the observations provided by the analysis are just stored during computation. The implementation of the function p minimizing the number of executions the contract falsely classifies as distinguishable as suggested in Chapter 3 requires reasoning about all test cases. Thus, also the results of test cases the adversary cannot distinguish are stored.

4.5 Test Case Generation

One remaining question is how the set of test cases is obtained. As explained above, every test case consists of two architectural states. This includes the registers and the memory including the program code.

There are many different options for test case generation, thus it is helpful to discuss which properties a good test set should have.

First of all, a leakage that cannot be observed in this set of test cases, will most probably not be included in the generated contract. Thus, one goal is to include as many different leakages as possible. Having many different adversary-distinguishable test cases increases the probability of obtaining a contract candidate that comes close to a correct contract, i.e. a contract that identifies nearly all possible sources of leakage. However, there should also be test cases that the adversary cannot distinguish to guide the contract generation towards the actual leakage. This allows to decide which observations actually caused a leakage and thus helps decreasing the number of test cases falsely classified as adversary distinguishable. This results in a higher precision of the resulting contract.

Another important consideration is whether the pairs of architectural states should be generally the same and only feature small differences or if there should be various differences. While some leakages might only appear if there are multiple differences, it does not help in contract generation if multiple leakages are contained in one execution, as a contract that only detects one of these leakages would already cause the executions to be contract distinguishable and thus further test cases without this specific leakage are needed to reliably detect the other leakages.

While it is definitively interesting to look at some explicitly designed test cases testing e.g. caches or certain types of loops, it is not feasible to design all the required tests by hand. Thus, there has to be some automatic generation of test cases.

One approach would be to simply start in an initial state in which all registers are zero and to generate random instructions for each of the two programs. However, the adversary will likely be able to distinguish these executions as the instructions being executed will most likely be completely different. Furthermore, if the initial state of the registers does not differ, some kinds of leakages, e.g. the content of `RS2` leaks, are very unlikely to be covered in the set of test cases as this would require that both random programs first load a different value into this register and afterwards both execute the leaking instruction. This could be solved by generating a random valuation for every register, however, the executed instructions will most probably still be completely different, thus it would be very difficult to identify the actual source of a leakage.

Therefore, the program generation should be more guided. A more promising approach is to generate two random valuations for every register and then execute the

same sequence of instructions with those initial states. This would ensure that most kinds of leakages can be detected as the operands of the instructions are likely to differ and two instructions of the same type are executed at the same time allowing to conclude that certain operands of this instruction leak if the adversary is able to distinguish the two executions. However, for example, the leakage of an immediate value cannot be covered with this approach as the immediate values are directly encoded into the instructions and thus are the same in both executions. Therefore, it might make sense to make both executions use the same type of instructions but randomly generate the operands for each program individually. In any case it is important to keep in mind that a small number of possible observations that would make two executions distinguishable is desirable to guide the computation of the contract.

A more involved approach could further fix templates for certain structures, e.g. loops, branches, or load-store sequences. This would allow to detect leakages that only occur under certain conditions which are very unlikely to appear in randomly generated test cases that do not take these structures into account. However, this would require a significant effort in identifying possible templates that might be useful.

Overall, it is clear that many different test cases are required to obtain a meaningful contract. This includes both, adversary-indistinguishable and adversary-distinguishable test cases. However, at the moment, it is not clear what the ideal test case generation looks like and which structures it should actively incorporate into the test cases to trigger specific observations. A more detailed description on how test cases were generated for the evaluation of this work is given in Section 5.3.

Once the test cases have been generated, they can be provided to Algorithm 1 in order to be simulated and evaluated.

5 Implementation Details

The implementation of the algorithm presented in the last chapter combines multiple tools and libraries to accomplish the desired behavior. The main logic implementing contract candidate generation is written in Java and, due to its modular structure, only a few additions are required to support a new processor or even another ISA.

One important part of the algorithm is the simulation which does not happen directly in Java. All the processors considered in this work are written in the hardware-description language Verilog [11] or more specifically SystemVerilog [10] which is a variant of Verilog with support for more advanced language features. There is a variety of tools that have support for analyzing and simulating Verilog structures, among the most popular free tools are Icarus Verilog [24], Verilator [21], and Yosys [27]. However, all of these tools only support a certain subset of Verilog and only certain elements from SystemVerilog. Luckily, there is a tool called sv2v [20] which allows to translate SystemVerilog to standard Verilog using features that are (mostly) supported by all three mentioned tools.

Initially, Yosys and SymbiYosys [26] have been used for the simulation. Yosys can, among other things, build a SMT [3] model for a given Verilog module and include formal properties to be proven. SymbiYosys then invokes a bounded model check to check the specified properties. By making sure it runs long enough, the bounded model check is actually sufficient to detect adversary distinguishability and to collect a sufficiently long trace for contract candidate generation, however, for slightly more complex designs such as the cores examined in this work, the bounded model check becomes very slow compared to other alternatives. Furthermore, this work does not require symbolic variables which means a bounded model check has no advantage over a standard event-based simulation of the Verilog structure.

Therefore, the current implementation uses Icarus Verilog for the simulation. Verilator was also considered, but it turned out that Icarus Verilog was easier to integrate into the existing project. Icarus Verilog compiles a given Verilog structure that can afterwards be simulated using an integrated tool. It allows for memory contents to be read from a file which means that multiple test cases can be executed using the same binary, significantly reducing the overall time required for evaluation. Test cases as presented in Section 4.5 consist of the register contents and the memory content. However, the registers can easily be initialized by prepending a sequence of instructions loading immediate values into the registers to the actual program.

This means that the compiled binary is independent of the test case and the entire architectural state can be established by reading the memory contents from a file.

Currently, the contract computation is implemented using Integer Linear Programming (ILP) [6] with the constraints presented in the last chapter. The OR-Tools [22] library is easy to integrate and provides a good performance so far. This allows to very efficiently generate an optimal solution for the contract candidate generation problem. In the implementation phase it turned out that under certain conditions, namely when a contract observation is included only in test cases that are adversary distinguishable, the solver can include this observation at no additional cost even if the contract already covers this test case with another observation that is included in the contract. This means that the resulting contract may become unnecessarily large, especially when few test cases are adversary indistinguishable. In order to circumvent this problem, after computing any optimal solution, the solver is invoked again to compute the minimal solution that is optimal w.r.t. to the number of false positives.

5.1 The Testbench Design

To integrate the processor design into the remaining project, another abstraction layer, the testbench, is required which coordinates the two executions and allows easier access to important values. The testbench is a crucial part of this project as it provides the interface between the processor design to be tested and higher abstraction levels that invoke the simulation and evaluate the results.

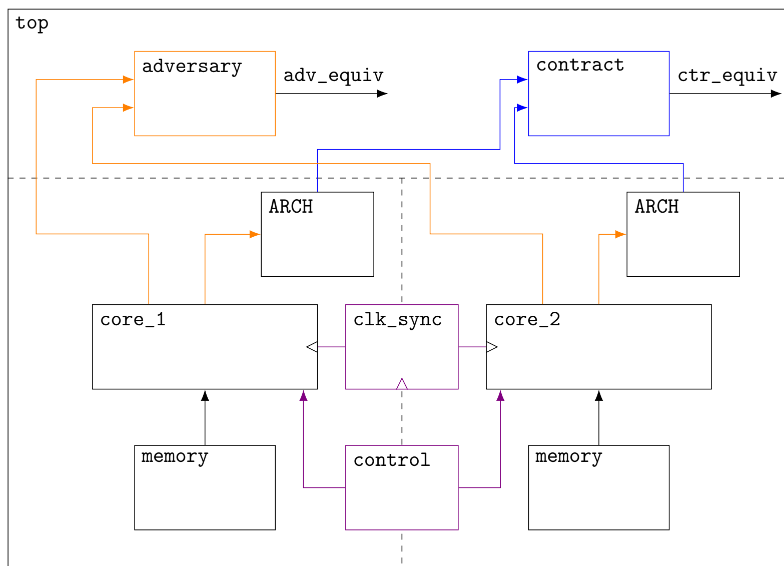


FIGURE 5.1: Simplified module structure used for simulation. Wires transmitting microarchitectural state are depicted in orange, architectural ones in blue and control signals are violet.

The overall structure is shown in Figure 5.1 and can be divided into three sections, one for each of the two instances of the processor core and one containing modules that allow to interact with the testbench and to extract important information. The control modules are furthermore responsible to coordinate the two cores, i.e. to initialize, synchronize and eventually terminate the simulation once the desired number of instructions has been executed.

Almost every processor design provides a different interface to integrate the core, but usually, the interface requires a clock signal, a connection to memory, and some inputs for external interrupts or devices. More details on how the clock signals are provided to the two cores are presented in Section 5.1.1. As interrupts are not considered in this work, static inputs representing no pending interrupts are connected here. The memory is connected to the memory module which dynamically loads the sequence of instructions to be tested from a file. The remaining part of the memory is initialized with default values. To increase simulation speed, the memory is not represented as a large array of data but rather as a buffer storing only the last n stores. By choosing n high enough, i.e. higher than the absolute number of executed instructions, correct behavior of the memory can be ensured.

5.1.1 Synchronizing the Two Cores

The clock inputs require some more effort than the memory interface. As mentioned in Section 2.2.1, the two executions are not necessarily synchronized, i.e. an instruction may take longer on one of the two cores due to e.g. different input values. However, this will lead to problems if the two states which are compared for the contract evaluation are reached at different points in time. One solution would be to buffer the states produced by one core until the other core has reached the respective state as well. Unfortunately, the cores can drift further away from each other over time which makes it hard to fix a reasonable size for this buffer.

The approach used in this work uses the clock input to force the cores to be synchronized. It takes advantage of the fact that all cores considered only operate on clock edges as there are no external interrupts that could trigger the state of the core to change. Thus, a core can be "paused" by not changing the clock signal until the other core has caught up. In order to obtain the correct architectural state, the core has to be paused precisely after one instruction has retired. How this information can be obtained varies by core but usually, there is already some internal signaling indicating a retirement that needs to be made visible to the `clk_sync` module. However, altering the clock input also changes the sequence of microarchitectural states and thus alters the observations of the adversary. This means that as soon as the clock signals are altered by the `clk_sync` module, the two executions need to be adversary distinguishable as well. This essentially translates to an adversary which is able to observe retirements. This assumption, however, does not seem unreasonable

as timing information of instructions is usually among the first things visible to an adversary.

5.1.2 Extracting the Architectural State

Once the simulation is working, the contract candidate generation needs to be able to access (relevant parts of) the architectural state. The synchronization approach presented above simplifies this process. The module `ARCH` is responsible for transforming a microarchitectural state into its architectural counterpart. The exact approach varies depending on the core, however, in any case the instruction which has just been retired as well as relevant registers and memory entries have to be extracted. The memory contents are easily obtained as the memory module is not included in the core, thus additional outputs can be added easily. Getting access to the instruction is a bit more difficult as the complete instruction is usually no longer required after decoding the relevant parts of it. Thus, the instruction may need to be stored in additional registers and passed along the pipeline of the core until retirement. The architectural registers can be implemented very differently depending on the actual design, especially regarding how updating register values works. If the core implements the RISC-V Formal Interface (RVFI) [25], the extraction process can be simplified as this interface already provides relevant data directly upon retirement. This data includes the retired instruction, the corresponding program counter, the values read from the source registers and written to the destination register, as well as the value read from or written to memory alongside the respective address.

5.1.3 Determining Adversary Distinguishability

Apart from the architectural state, adversary observations need to be made visible at the top level. If the adversary model is able to observe certain registers from the microarchitectural state, it is as simple as adding a new wire from the core to the adversary module and asserting within this module that the values from both cores are always equivalent. As mentioned above, retirements always need to be visible to the adversary, thus the adjusted clocks are always compared in the adversary model. Once a difference has been detected, `adv_equiv` becomes zero and will not change anymore.

In this project, the adversary model always only includes the retirements, i.e. the adversary can only see when each of the cores retires an instruction. However, other adversary models would be very easy to integrate into the existing project.

5.2 Extraction of Possible Contract Observations

The last section demonstrated how adversary distinguishability is constantly evaluated, thus, one might ask why it is necessary to continue the evaluation once it is clear that the adversary is able to distinguish the two executions. It turns out that it is

possible that the adversary already observed something that is not yet visible at the architectural level. Figure 5.2 shows a three-stage pipeline. While only the shaded instructions `A`, `B` and `C` have been retired yet and thus have affected the architectural state, the instructions `D`, `E` and `F` are already in the pipeline and thus already affected the microarchitectural state. This means that one of these instructions could also be responsible for a difference in the adversary observations. It is possible to extract the upcoming instructions given an architectural state by simulating the architectural behavior, however, it is much easier to make sure every instruction in the pipeline is retired before aborting the simulation to avoid this computation.

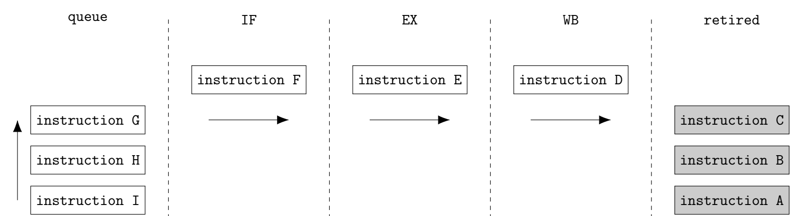


FIGURE 5.2: Conceptual model of a three-stage pipeline. Only `A`, `B`, and `C` are architecturally visible, while `D`, `E`, and `F` are already being executed and thus may already have affected the microarchitectural state.

Once it is ensured that all instructions are retired, the next step will be to extract possible explanations for the observed leakage. The simulation generates a value change dump (VCD), a standard format defined alongside Verilog for logic simulations [11]. This VCD file includes the sequence of valuations of the entire simulation. Specifically, the registers of the `ARCH` module are now interesting. They allow to obtain the executed instructions and their respective context which is thereafter used to find differences according to the contract template as specified in Section 3.3.

5.3 Test Case Generation

The effectiveness of the contract candidate generation largely depends on the size and quality of the set of test cases.

Some different approaches for randomly generated sets of test cases have been presented in the last chapter. For this project, it turned out that the two architectural states should be quite similar to allow to generate a relatively accurate contract with a rather small set of test cases. In order to be able to observe every type of observation presented in Section 3.3, test cases are explicitly generated to lead to differences in these observations.

In practice, this means that for every type of instruction, one random instance of this instruction is generated, i.e. register numbers and immediate values are generated using a pseudo-random number generator. To introduce a difference either a prefix is calculated or the instruction is slightly altered to lead to a specific observation.

In order to check whether e.g. the immediate value of a `load` instruction leaks, a random instance of a `load` instruction is generated as explained above. To introduce an observable difference, this instruction serves as a base for two different instructions that are the same as the base except for the immediate value that is replaced by one of two newly generated random numbers. If during the simulation of the test case it turns out that the adversary was able to distinguish the two executions, one possible addition to the contract would be (`load`, `IMM`) which could be extracted from the trace of the simulation.

On the other hand, if the goal is to check whether the value of `RS1` in the `load` instruction leaks, the value of `RS1` must be different in the two executions. This means that before executing the load, another instruction will alter the value stored in `RS1`. In practice this is done by adding an `addi` (with the first operand being register zero which always holds zero) that stores one of two randomly generated numbers in the given register.

In this manner, a pair of two (sequences of) instructions can be generated to specifically test for an observation type specified in the contract template in Section 3.3. For these small programs, a random initial valuation for the registers as well as a random suffix is generated. The suffix allows to detect leakages that might only be observable after executing some more instructions. As those instructions are the same in both programs, they will not introduce many more possible observations, thus any leakage that occurs can be quite precisely mapped to the altered instruction.

The above process is repeated with every instruction type until the desired number of test cases is reached. This ensures that every considered instruction is actually contained in the set of test cases and that leakages that are triggered by a single difference in the architectural state (specifically leakages that do neither depend on specific operands nor on the interaction between two or more instructions) will be present in this set of test cases.

5.4 Integrating the Ibex and CVA6 Processors

So far, the open-source cores Ibex [19] and CVA6 [30] have been integrated into the project.

5.4.1 The Ibex Core

The Ibex core is a fairly simple implementation of the RV32IMC ISA featuring up to three pipeline stages and different multiplication modules to be selected.

The integration of the Ibex core into the testbench presented in Section 5.1 is straightforward, only minor adjustments are needed to compile the core. As there are no instruction caches, fetches can be observed via the memory interface. In an initial

implementation, the architectural state was extracted directly from the microarchitectural state. This meant identifying a retirement predicate that indicates whenever an instruction retires as well as storing the executed instruction throughout the write-back stage as it is needed for contract evaluation but was previously discarded after being decoded. After integrating the CVA6 core using the RISC-V Formal Interface (RVFI), this interface was also used to integrate the Ibex core. The tested configuration corresponds to the `maxperf` default configuration except for the branch target ALU having been disabled for this test. In this configuration, the Ibex has a three-stage pipeline, with an instruction-decode, an execute, and a writeback stage. Furthermore, it has a single-cycle multiplication engine.

5.4.2 The CVA6 Core

The CVA6 core, which was previously known as Ariane, is more complex than the Ibex core. It has a six-stage pipeline, support for virtual memory, and various caches.

Unfortunately, it also uses various SystemVerilog features that are not supported by sv2v and Icarus Verilog. Thus, some manual adjustments were necessary. Most noticeably, Verilog does not support type parameters as they enable access to named members, e.g. an object of type `instr` may have a member `op`, thus one can access this member of `instr_a` using `instr_a.op`. Verilog does not have support for custom types, thus sv2v needs to convert every type to a bit array and calculate the position of the respective member. However, with type parameters, those offsets are not known before the module is instantiated and thus a module with type parameters cannot be translated. A workaround is to parameterize the size of the datatype instead of the datatype itself. This does not allow to access members by name either but is sufficient to allow the compilation of the CVA6 core.

As the CVA6 core implements the RFVI, it is easy to obtain all the required information about the retired instructions. As the CVA6 features multiple commit ports, it can retire up to two instructions at the same time. Some more adjustments were needed to be able to compare the correct architectural states when this happens.

Furthermore, the CVA6 implements the Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) [1] which is a memory interface developed by Arm. The interface defines five channels that are used to exchange information about reads, writes, and the respective addresses. While this interface is very powerful and allows for high throughput in practice, its complexity is rather high. Luckily, the CVA6 includes a module that decodes AXI requests and encodes the responses providing a much simpler interface.

Apart from that, no more adjustments were needed to integrate the CVA6 core.

6 Evaluation

To evaluate the performance of the approach described in the previous chapters, the results from the two integrated cores, Ibex and CVA6 have been analyzed.

6.1 The Generated Contract

First of all, it makes sense to look at the generated contracts to get an overview of the results of the contract candidate generation.

In both contracts, the majority of entries comes from observations directly related to branches. This is not very surprising given that branch targets are randomly generated, thus, there are likely to be different instructions at different targets which may lead to timing differences. For example, a contract for the CVA6 core obtained from 20,000 test cases includes:

```
BGE: IMM
BGE: REG_RS1
BGE: REG_RS2
```

This means that for every `BGE` instruction, the immediate (encoding the branch target) and the content of both operands may leak. This seems very sensible given that the content of both registers eventually decides whether a branch will be taken or not. However, in the case of `BEQ` the relevant part of the contract looks as follows:

```
BEQ: IMM
BEQ: REG_RS2
BEQ: RS1
```

The `RS1` means that the number of the register contaiing the first operand will leak. This does not seem very reasonable and is most probably due to the fact that the solver had the choice to either include `RS1` or `REG_RS1` at a similar cost. Given that randomly generated values are very likely not equal, there are probably only a few test cases in which this branch was taken, thus there might not be sufficient evidence to make clear that `REG_RS1` should be included here.

Interestingly, in the case of a `DIV` on the CVA6 core, the content of both registers seems to have an influence on the execution time. For the multiplication, however, the contract candidate generation was not able to find such a dependency.

Regarding the Ibex core, the loads and stores are interesting. At first sight, it is not clear why the execution time of loads or stores varies, given that the Ibex does not have any cache enabled and the memory responds in constant time to requests. Upon closer inspection it turned out that the Ibex memory interface guarantees that all accesses are word-aligned, thus unaligned loads will perform two memory requests and thus take longer than aligned loads.

6.2 The Evolution of the Contract during Generation

While normally it is sufficient to generate the contract once after evaluating every test case, it is also possible to update the contract after each individual test case and collect statistics about how the contract evolves over time.

The following measurements have been collected on the Ibex core unless indicated otherwise. 20,000 test cases have been evaluated, and, after each addition, the contract was updated and tested on a set of 100,000 different, also randomly generated test cases. Within those test cases, the altered instructions were evenly distributed accross all considered instruction types (all the instructions included in the RV32IM ISA subset). Table 6.1 shows that in both sets about 7% of the test cases can be distinguished on the Ibex core by the adversary that only observes retirements.

	Size	**Adversary Distinguishable**			
		`Ibex`		`CVA6`	
Training Set	20,000	1421	7.1%	1055	5.2%
Evaluation Set	100,000	7035	7.0%	5573	5.5%

TABLE 6.1: Sets of test cases used for evaluation.

6.2.1 The Size of the Computed Contract Candidate

When starting the evaluation of the project, one important question was how many test cases will actually be required to generate a "good" contract. To answer this question, it might be helpful to look at how often the contract changes during evaluation and how stable the contract becomes over time.

It turns out that the contract changes rather frequently throughout the evaluation, in total it changes after every third addition. However, this does not mean that the contract changes completely, but in many cases, the solver just has multiple options to choose from, and, depending on the exact inputs, it might choose another option. This is also indicated by the size of the contract, as shown in Figure 6.1. The size in this context reflects the number of pairs included in the computed contract candidate. In the beginning, the contract grows rapidly, but the second half of the test cases only slightly increases the size of the contract. This indicates that towards the end, the

generated contract candidate is rather stable and more test cases will only slowly improve the overall contract. The frequent changes could possibly be eliminated by hinting the solver towards the solution it has chosen in the previous iteration, but as in production the computation of the contract only happens once after evaluating all test cases, this would not improve the quality or the speed of the algorithm.

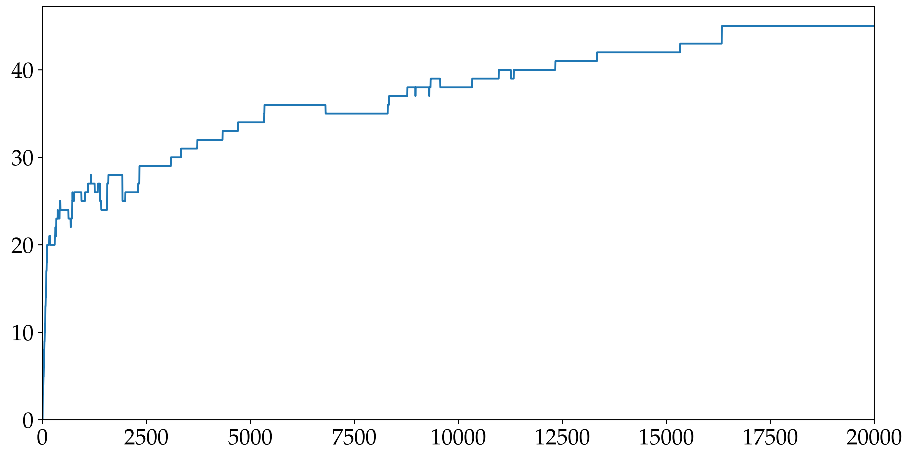


FIGURE 6.1: Size of the contract during the evaluation of the training set. Each pair (`type`, `observation`) is counted as 1.

6.2.2 The Quality of the Generated Contract Candidate

The previous measurements, however, do not give any information on the quality of the generated contract candidate. A stable contract not working on examples that were not included in the training set or a contract that classifies almost every example falsely as distinguishable does not help a lot in practice. Thus, the performance of the generated contract candidate on the evaluation set with 100,000 test cases was also measured.

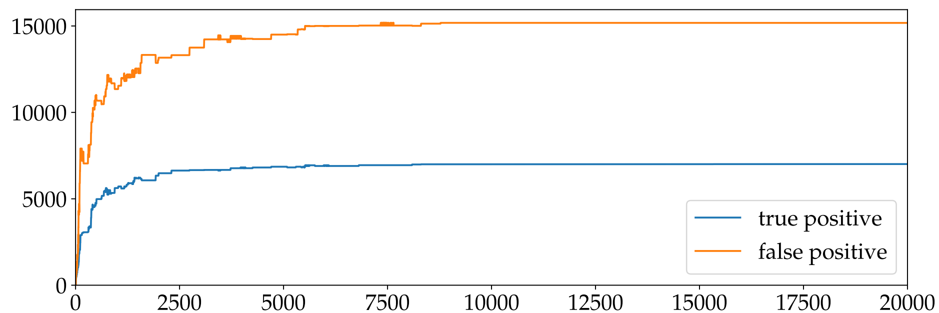
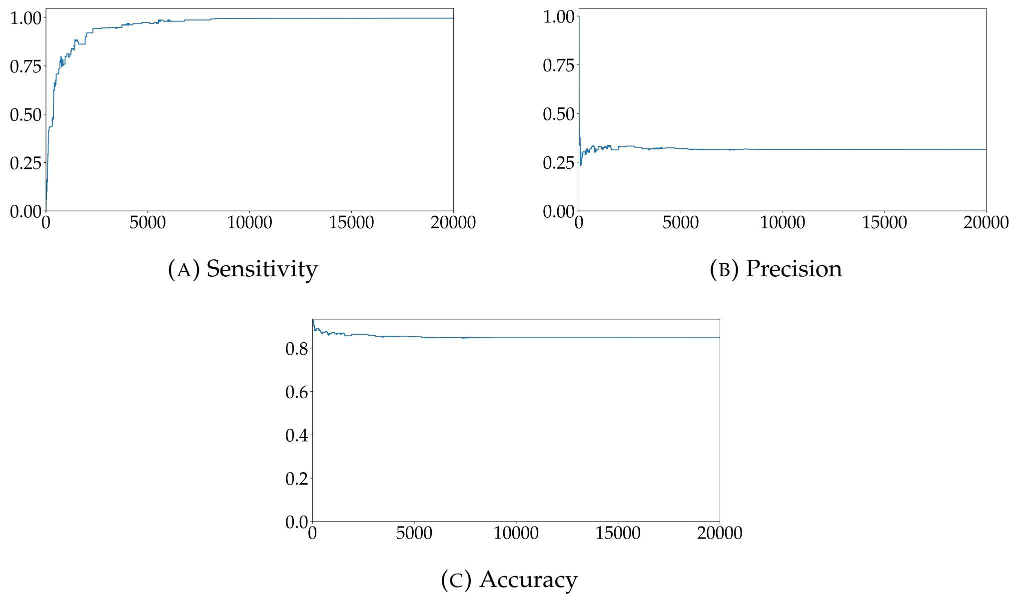


FIGURE 6.2: Absolute number of test cases in the evaluation set correctly and incorrectly classified as contract-distinguishable during the evaluation of the training set. In total, there were 7,035 adversary-distinguishable and 92,965 adversary-indistinguishable test cases.

At first glance, the absolute numbers shown in Figure 6.2 do not look too promising, but keeping in mind that only about 7% of the test cases were adversary distinguishable, the high number of false positives becomes more reasonable. It is also interesting to see that already after about 5,000 evaluated test cases the contract seems quite stable, although Figure 6.1 shows that the contract size increases by about one third. This illustrates that towards the end mostly cases affecting only a small number of executions are added to the contract.



(A) Sensitivity

(B) Precision

(C) Accuracy

FIGURE 6.3: Evolution of sensitivity, precision, and accuracy w.r.t to the evaluation set obtained during evaluation of the training set.

In order to obtain more comparable numbers, Figure 6.3 shows important statistical performance metrics. The sensitivity indicates the percentage of adversary-distinguishable test cases that were correctly classified as such. The precision represents the percentage of test cases which the adversary was able to distinguish out of the total number of test cases which the contract declared as distinguishable. The accuracy shows the percentage of test cases the contract correctly classified (both as distinguishable or indistinguishable).

Once again, after evaluating 5,000 test cases, the sensitivity is at about 97.5%. After evaluating the whole training set, the sensitivity increases to 99.6%. The precision, however, appears to be quite stable at about 32% which unfortunately does not seem to be considerable. The precision directly correlates to the numbers shown in Figure 6.2 where for every correctly classified sample, there are about two false positives. To some extent, this is influenced by the relatively low percentage of adversary-distinguishable samples in the test set. This also indicates the accuracy which measures the percentage of test cases that have been correctly classified (including both, adversary-distinguishable and adversary-indistinguishable cases). The

overall accuracy is at about 85%. This means there are about 15% incorrectly classified test cases. Given the high sensitivity, those 15% are almost all false positives.

This percentage seems high at first sight, but given that the current template is rather coarse-grained at some points, it does not seem unreasonable. For example, it is currently not possible to express in the contract whether a branch is taken or not. Given that it is very likely that branches are indistinguishable to an adversary if they are not taken and as in two randomly generated test cases the probability that a `BEQ` is not taken is very high, there are many possibilities for false positives. Furthermore, as explained in Section 6.1, the Ibex core treats aligned and unaligned memory accesses differently, thus the adversary can distinguish two executions in which one accesses an aligned address and the other one does not. However, in reality, it is more likely in a randomly generated test case that both addresses are either aligned or unaligned than that only one is aligned and the other one is unaligned. However, the current contract template only allows to classify two executions as distinguishable if the resulting addresses are different. Allowing more and especially more precise contract observations might help to improve the accuracy of the generated contract candidate.

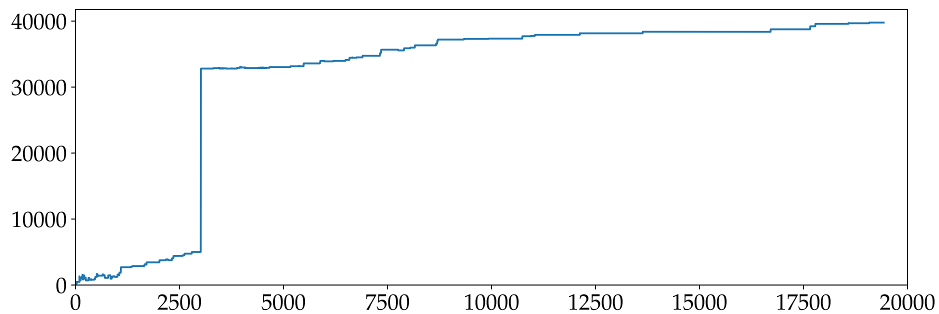


FIGURE 6.4: Absolute number of false positives, i.e. incorrectly classified as distinguishable, obtained during evaluation of the training set on the CVA6 core.

Looking at the number of false positives on the CVA6 core there is a very noticeable jump in the number of false positives as shown in Figure 6.4. With just one test case the number of false positives jumps from 5,006 to 32,807. Upon closer inspection, it turns out that this test case indeed is adversary distinguishable and that there is only one possible addition to the contract that would explain this leakage, the timing of the `SW` instruction depends on the number of the register that contains the value to be stored (`RS2`). In fact, when looking at the sequence of executed instructions, this addition becomes more reasonable:

```
r27 ← LB mem[r31 + 3792]
SW mem[r6 + 3169] ← r27
```

CODE 1: Program 1

```
r27 ← LB mem[r31 + 3792]
SW mem[r6 + 3169] ← r23
```

CODE 2: Program 2

There is a read-after-write dependency on register 27 in Program 1 but not in Program 2. This dependency causes a stall. However, as such a dependency is very rare, there are many test cases in which there is a store with different RS2 in which there is no adversary-observable difference. Thus, there are many false positives once this observation is added to the contract. This once more illustrates the limitations of the contract template as such a dependency cannot be expressed any better when using the contract template from Section 3.3.

6.3 Computation Time

Another important question is whether it is feasible to compute contract candidates with a sufficient number of test cases within a reasonable amount of time. This allows to efficiently use this workflow with different configurations in order to quickly get an idea of the core's overall behavior regarding timing side channels.

The results obtained on an Intel Core i7-8700 are shown in Table 6.2 comparing the performance of the Ibex and the CVA6 core. For the small Ibex core, the time required per test case is about 0.1 second and due to a high parallelization, the overall time to obtain a contract candidate from 20,000 test cases is only about 7.5 minutes. Looking at the CVA6 core, the required time is significantly higher which was to be expected given the higher complexity of this core. However, the overall computation time of 3.25 hours seems still reasonable.

	Ibex	CVA6
Compilation Time	3.4s	20.0s
Simulation Time[1]	83ms	2.8s
Extraction of Possible Observations[1]	3ms	21ms
Contract Candidate Computation[2]	3.2s	1.3s
Total Contract Candidate Generation Time[2]	7.5min	3.25h

[1] on average, per test case. [2] using the training set with 20,000 test cases, multi-threaded.

TABLE 6.2: Performance measurements of the contract candidate generation collected on an Intel Core i7-8700 CPU @ 3.20GHz with 12 threads and 16 GB of RAM.

7 Related Work

7.1 Side-Channel Attacks

In 1996, Paul Kocher described side-channel attacks/timing attacks on various cryptographic algorithms including RSA in [13]. Back then, the attack could be prevented by reducing timing differences using various methods, however, as of today, such methods are no longer viable as measurements have become much more precise and other attack vectors have been discovered.

Later on, in 2005, Colin Percival practically showed at BSDCan [16] how hyper-threading simplifies timing attacks on cache structures. One of his recommendations was to avoid sharing hardware among threads and to ensure that data from other threads does not have any influence on timing. While this in practice prevents many side-channel attacks, the performance of modern CPUs relies on sharing hardware, and, more importantly, other attacks on hardware that is inherently shared, e.g. RAM, will not be prevented. In 2006, further cache-based attacks were described in [23] alongside new concepts for cache design that would prevent the presented attacks with a small overhead in performance.

Since then, many different attacks using side channels have been presented. While the most popular, recent attacks, Meltdown [14] and Spectre [12], require transient execution, which is out of scope for this work, there are many other approaches that work without speculative execution.

Many popular attacks are based on Flush+Reload [28] or Flush+Flush [7] which extract memory accesses by repeatedly measuring access times which depend on whether the address is in the cache due to a recent access.

As most of these attacks are tailored for x86 CPUs, some adaption would be needed to make them work on RISV-V CPUs but the same attack vectors also apply to RISC-V as the general concept of optimizations such as caches are fundamentally the same accross different ISAs.

7.2 Hardware-Software Contracts

Hardware-software contracts have been presented in [8]. This paper introduced a lattice of different contracts starting from a contract that does not expose anything to contracts that allow to capture speculative memory accesses. As contracts only

rely on the architectural state, the contract evaluation has to simulate speculation by unrolling the mispredicted path. However, up to now, there is no formal method to actually prove contract satisfaction.

There are some more approaches that try to capture microarchitectural leakage, among these leakage containment models presented in [15]. Those try to track the microarchitectural information flow in order to decide what information will actually be visible at software-level. Unique Program Execution Checking (UPEC) [5] provides a property that aims to ensure that the architectural state of a processor does not depend on a secret part of the data memory. A violation of this property could indicate a vulnerability of this core and if this property can be proved, the secrets are protected against leakage through covert channels.

Another way to avoid leakage through side channels is by using constant-time or data-oblivious programming, in which neither the control flow nor the data flow may depend on secret data. This prevents many timing attacks as the sequence of executed instructions and the sequence of memory accesses do not depend on the secret. However, in some microarchitectures, the performance of some instructions depends on the operands as well [4] and thus the same control flow does not guarantee the same timing behavior. Data Oblivious ISA Extensions (OISA) [29] could provide guarantees which instructions can safely be used in data oblivious programming, however, this concept has not been realized in practice yet.

Apart from that, multiple projects try to obtain formal models of either the instruction set architecture such as Sail [2] or of a given microarchitecture described at RTL level [9]. Those models, given that important implementation details are contained in the model, can help to standardize formal methods such as the approach presented in this work and thus make it easier to apply a certain method on a variety of processors. The RISC-V Formal Interface [25] provides a standardized interface to obtain the architectural state from a running core. This interface simplified the integration of the CVA6 core in this work and allows to easily integrate other cores with support for this interface. With ISA-Formal [17], Arm has a verification technique that allows to formally capture the architectural behavior of a processor, however, as ARM processors are closed source, those processors are out of scope for approaches that do not rely on black-box testing.

8 Summary and Conclusion

Hardware-software contracts are still a relatively new area of research in microarchitectural security. While in theory these contracts could improve resilience against side-channel attacks, their practical use is very limited at the moment as there hardly exists any contract to work with. However, this work has shown that it is possible to infer relatively accurate contracts for existing microarchitectures even with a very limited set of execution traces. To efficiently work with those contracts, there is need for more restricted contract templates, like the one introduced in Chapter 3. Afterwards, Chapter 4 introduced an algorithm that allows to automatically generate a contract candidate according to this template. This algorithm generates the execution traces, analyzes them and infers the optimal contract candidate according to this analysis. This concept has been implemented and tested with open-source RISC-V processors written in Verilog. Only minimal adjustments in the processors sources were necessary to integrate them into the workflow. The feasibility of this approach has been demonstrated on the open-source RISC-V processors Ibex and CVA6.

Overall, the presented approach was able to generate reasonable contracts for the two examined processors, even with a rather small number of test cases. However, the evaluation showed that there is potential for improvement.

First of all, the contract template presented in Section 3.3 might be too coarse-grained in certain situations. This results in a relatively high number of executions falsely classified as distinguishable because it is just not possible to formulate a more precise contract with this specific template. It most certainly would be possible to improve the expressiveness of the contract template by adding more possible observations to the existing list of possible leakages. Possible additions could for example include reasoning about certain properties of the evaluated values, such as the equality, inequality, or if a value is zero to determine whether branches are taken or not. Furthermore, it might be helpful to allow to express whether a certain value is word-aligned. This could allow to reduce the false positives by capturing leakages in more detail. To further improve accuracy, one might consider to allow to compare specific bits of the operand to determine signedness or whether a value is above a certain threshold. However, to reliable detect such specific leakages much more execution traces will be required to reliably distinguish different types of leakage.

This directly relates to the second area that could be explored in further work. Other

test case generation methods might be useful to test specific but realistic and leakage-prone scenarios such as loops or load-store sequences. A combination with the existing test cases could improve the accuracy of the obtained contract candidate. Furthermore, this would allow to generate more and also more precise test cases, allowing to make use of a more fine-grained contract template as discussed above.

Bibliography

[1] Arm Limited. *AMBA® AXI Protocol Specification, Issue J*. https://developer.arm.com/documentation/ihi0022/latest. 2023.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 71:1–71:31. DOI: 10.1145/3290384.

[3] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[4] Lucas Biehl. "Characterising Input-Dependent Instruction Timing via Measurements". Master Thesis. Saarland University, 2021. URL: http://embedded.cs.uni-saarland.de/publications/theses/thesis_cs_msc_Biehl_Lucas_Alexander.pdf.

[5] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark W. Barrett, Subhasish Mitra, and Wolfgang Kunz. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. Ed. by Jürgen Teich and Franco Fummi. IEEE, 2019, pp. 994–999. DOI: 10.23919/DATE.2019.8715004.

[6] Jack E. Graver. "On the foundations of linear and integer linear programming I". In: *Math. Program.* 9.1 (1975), pp. 207–226. DOI: 10.1007/BF01681344.

[7] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*. Ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez. Vol. 9721. Lecture Notes in Computer Science. Springer, 2016, pp. 279–299. DOI: 10.1007/978-3-319-40667-1_14.

[8] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1868–1883. DOI: 10.1109/SP40001.2021.00036.

[9] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. "Synthesizing Formal Models of Hardware from RTL for Efficient

Verification of Memory Model Implementations”. In: *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021*. ACM, 2021, pp. 679–694. DOI: 10.1145/3466752.3480087.

[10] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language”. In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[11] “IEEE Standard for Verilog Hardware Description Language”. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[13] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[14] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 973–990. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[15] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. “Axiomatic hardware-software contracts for security”. In: *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*. Ed. by Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang. ACM, 2022, pp. 72–86. DOI: 10.1145/3470496.3527412.

[16] Colin Percival. *Cache missing for fun and profit*. Presented at BSDCan Ottawa, ON, Canada. 2005.

[17] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. “End-to-End Verification of Processors with ISA-Formal”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 42–58. DOI: 10.1007/978-3-319-41540-6_3.

[18] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. Ed. by Andrew Waterman and Krste Asanović. Dec. 2019.

[19] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *27th International Symposium on Power and Timing Modeling, Optimization and Simulation, PATMOS 2017, Thessaloniki, Greece, September 25-27, 2017*. IEEE, 2017, pp. 1–8. DOI: `10.1109/PATMOS.2017.8106976`.

[20] Zachary Snow. *sv2v: SystemVerilog to Verilog*. `https://github.com/zachjs/sv2v`. 2019.

[21] Wilson Snyder. *Verilator open-source SystemVerilog simulator and lint system*. `https://github.com/verilator/verilator`. 2003.

[22] Google Optimization team. *Google's Operations Research tools*. `https://github.com/google/or-tools`. 2010.

[23] Zhenghong Wang and Ruby B. Lee. "Covert and Side Channels Due to Processor Architecture". In: *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*. IEEE Computer Society, 2006, pp. 473–482. DOI: `10.1109/ACSAC.2006.20`.

[24] Stephen Williams. *The ICARUS Verilog Compilation System*. `https://github.com/steveicarus/iverilog`. 2000.

[25] Clifford Wolf. *RISC-V Formal Verification Framework*. `https://github.com/SymbioticEDA/riscv-formal`. 2017.

[26] Clifford Wolf. *SymbiYosys (sby) – Front-end for Yosys-based formal verification flows*. `https://github.com/YosysHQ/sby`. 2017.

[27] Clifford Wolf. *Yosys Open SYnthesis Suite*. `https://github.com/YosysHQ/yosys`. 2016.

[28] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Ed. by Kevin Fu and Jaeyeon Jung. USENIX Association, 2014, pp. 719–732. URL: `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom`.

[29] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL: `https://eprint.iacr.org/2018/808.pdf`.

[30] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Trans. Very Large Scale Integr. Syst.* 27.11 (2019), pp. 2629–2640. DOI: `10.1109/TVLSI.2019.2926114`.