Towards a Trustworthy Stack: Formal Verification of Low-Level Hardware and Software

NING DONG
Towards a Trustworthy Stack: 
Formal Verification of Low-Level Hardware and Software

NING DONG

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Philosophy on Monday the 5th February 2024, at 2:00 p.m. in F3, Lindstedtsvägen 26, Stockholm.

Doctoral Thesis in Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2024
To my parents and dear Wenqing
Abstract

Computer systems, consisting of hardware and software, have gained significant importance in the digitalised world. These computer systems rely on critical components to provide core functionalities and handle sensitive data. A fundamental requirement for these critical components is functional correctness, which ensures that the components work as their specifications prescribe. For instance, a pipelined processor will execute instructions concurrently in different stages such as fetch, decode and execute, and must produce results as specified in the instruction set architecture (ISA) manual. In addition to functional correctness, security properties such as confidentiality and integrity are important. In particular, confidentiality requires that sensitive data is only accessible to authorised users. To construct a correct and secure computer (i.e., a trustworthy stack), this thesis focuses on the functional correctness and confidentiality of peripherals and pipelined processors using the HOL4 interactive theorem prover.

For functional correctness, we use a refinement-based verification approach where the execution of a target system is constrained by a reference system. We have studied this for two different target systems, a synchronous serial peripheral interface (SPI) device along with its driver, and a 5-stage pipelined processor. Specifically, we formalise an SPI device and its driver, and present an abstract model as the reference system. The abstract model ensures correct communications in the SPI half- and full-duplex modes. The refinement between the abstract and SPI models is established using weak bisimulation. Secondly, we implement and verify a 5-stage in-order pipelined processor Silver-Pi for the RISC ISA Silver. The correctness of Silver-Pi is proved by exhibiting a refinement relation between the traces of the processor and the Silver ISA. Silver-Pi is implemented using the verified HOL4 Verilog library, which ensures the correctness of the processor down to its Verilog implementation.

For the SPI case study, weak bisimulation ensures that the SPI model has the same information flows as the abstract model, which prevents malicious driver operations e.g., an infinite loop based on a secret value.

In general, to prevent secret leaks caused by phenomena such as instruction pipelining and out-of-order (OoO) execution, the target and reference systems are augmented by observations that extract visible parts of the machine state to attackers. This allows a variety of information channels based on e.g. timing and cache access behaviour to be captured. As a security policy, we use the notion of conditional noninterference (CNI), which guarantees that a target system does not leak more information than what the reference system allows.

In order to analyse the timing channel of Silver-Pi, the observation function extracts the parts of the ISA state that may affect the execution time of a program. With this reference system, we prove that Silver-Pi satisfies CNI.

For OoO execution, we present the formalisation of a machine independent language, MIL, which uses a small set of primitive events to describe both in-order and OoO execution at a microarchitecture-like level. A notion
of CNI in MIL rules out trace-driven cache side channels by comparing OoO and in-order execution of a program. We present a semi-automated verification strategy for CNI using the executable semantics and demonstrate this strategy with several examples. The executable semantics computes results and generates observations during the execution of MIL programs.


Sammanfattning


För att analysera tidkanalen hos Silver-Pi, extraerar observationsfunktionen delar av ISA-tillståndet som kan påverka exekveringstiden för ett program. Med detta referenssystem bevisar vi att Silver-Pi uppfyller CNI.

För OoO-utförande presenterar vi formaliseringen av ett maskinoberoende språk, MIL, som använder en liten uppsättning primitiva händelser för
Acknowledgement

I remember the first day I was here at KTH, August 19th 2019. I was very nervous at that time because of a new environment and unfamiliar work for me. I was a little afraid since I was a rookie for formal verification and had no idea what would happen. The worst case I imagined was that I could not do anything useful and have to quit the PhD. Fortunately, this nightmare did not happen. I have learned a lot from others and published some papers.

First, I want to thank my supervisors Mads Dam and Roberto Guanciale. Mads has offered me the opportunity to explore the formal verification world. Since my background is not relevant to formal verification, I spent some time learning basic knowledge of formal methods and the usage of HOL4. I really appreciate the patience and encouragement Mads gives to me. Roberto is always willing to offer help and guidance when I have difficulties in my projects. During my doctoral studies, I have received a lot of valuable feedback and suggestions from Mads and Roberto. Without their help, I could not finish my research work.

I also want to thank my colleagues, my collaborators, and other people I met and talked to at conferences. To develop a verified processor, I have started to learn Verilog and the HOL4 Verilog library Andreas Lööw developed. Andreas works so efficiently and provides great convenience to support my work, for example, sharing the Vivado project and experiment settings of his processor. It is a great pleasure for me to work with him. In the MIL project, I have worked with Karl Palmskog and Xiaomo Yao. Although we focused on different tasks in the project, I have learned many things like HOL4 skills from Karl and Xiaomo. The STEP group is a warm family for me, I want to thank every member: Andreas Lindner, Anoud Alshnakat, Arve Gengelbach, Didrik Lundberg, Henrik Karlsson, Jonas Haglund, Karl Normman, Karl Palmskog, Pablo Buiras, and Xiaomo Yao. Especially I want to thank my officemate Anoud.

TCS provides a friendly work environment for PhD students. I want to thank Benoît Baudry, Cyrille Artho, Dilian Gurov, Douglas Wikström, Elena Troubitsyna, Karl Meinke, Mats Näslund, Martin Monperrus, Musard Balliu, and Philipp Haller. I want to thank Viggo Kann and Stefan Nilsson for yearly supervisory meetings to discuss my study progress. I thank my friends and colleagues: Aman Sharma, Amir M. Ahmadian, Christian Lidström, He Ye, Honglian Wang, Mikhail Shcherbakov, Fang Sen, Sijing Tu, Tianyi Zhou, Xin Zhao, Yogya Gamage, and Yuxin Liu.
Last but not least, I appreciate my wife Wenqing Yan and my parents Guoxiong Dong and Huimin Dong. Your support is priceless to me. Because of you, I did not give up when I lost my direction.

谢谢大家！
# Contents

Acknowledgement vii

Contents ix

I Thesis 1

Acronyms 3

1 Introduction 5
   1.1 Motivation ................................................. 5
   1.2 Research Questions ...................................... 9
   1.3 Contributions ............................................ 11
   1.4 Outline .................................................. 11

2 Background 13
   2.1 Computer System ......................................... 13
   2.2 Correctness and Security .................................. 20
   2.3 Formal Verification ....................................... 26

3 Related Work 41
   3.1 Peripheral and Driver Verification ....................... 41
   3.2 Processor Verification .................................... 42
   3.3 Low-Level Information Flow Security .................... 47

4 Summary of Publications 51
   4.1 Paper A: Refinement-Based Verification of Device-to-Device Information Flow ............................................. 51
   4.2 Paper B: Formal Verification of Correctness and Information Flow Security for an In-Order Pipelined Processor ............................................. 52
   4.3 Paper C: Information Flow Analysis of a Verified In-Order Pipelined Processor ............................................. 53
# CONTENTS


# 5 Conclusions and Future Work

5.1 Conclusions ........................................... 57
5.2 Future Work ........................................... 59

Bibliography ............................................. 63

## II Included Papers

A Refinement-Based Verification of Device-to-Device Information Flow

A.1 Introduction ........................................... 80
A.2 Background .......................................... 82
A.3 Architectural Model ................................. 83
A.4 SPI Hardware Model ................................. 84
A.5 SPI Driver Model .................................. 87
A.6 Abstract SPI Subsystem Specification ............... 89
A.7 Refinement ........................................... 89
A.8 System Properties ................................ 93
A.9 Information Flow Security ......................... 94
A.10 Application: Securing a Random Number Generator using SPI .... 97
A.11 Related Work ....................................... 98
A.12 Conclusion and Future Work ...................... 99

B Formal Verification of Correctness and Information Flow Security for an In-Order Pipelined Processor .................................................. 105
B.1 Introduction ........................................... 106
B.2 Background .......................................... 107
B.3 Pipeline Implementation ............................. 110
B.4 Correctness Proof ................................... 112
B.5 Information Flow ................................... 116
B.6 Discussion and Evaluation ......................... 120
B.7 Related Work ....................................... 122
B.8 Conclusion .......................................... 124

C Information Flow Analysis of a Verified In-Order Pipelined Processor .................................................. 129
C.1 Introduction ........................................... 130
C.2 Background .......................................... 130
C.3 Information Flow Security ......................... 137
D Foundations and Tools in HOL4 for Analysis of Microarchitectural Out-of-Order Execution 151
Part I

Thesis
Acronyms

The list contains the acronyms used in the first part of the thesis.

- AES  Advanced encryption standard
- AI   Artificial intelligence
- ALU  Arithmetic logic unit
- CCS  Calculus of communication systems
- CISC Complex instruction set computer
- CNI  Conditional noninterference
- CPU  Central processing unit
- DMA  Direct memory access
- DSL  Domain-specific language
- HDL  Hardware description language
- HOL  Higher-order logic
- ISA  Instruction set architecture
- ITP  Interactive theorem prover
- LCF  Logic for computable functions
- LTS  Labelled transition system
- MIL  Machine independent language
- ML   Meta language
- NI   Noninterference
- NIC  Network interface controller
- OoO  Out-of-order execution
- OS   Operating system
- RISC Reduced instruction set computer
- RNG  Random number generator
- RTL  Register transfer level
- SoC  System on chip
- SPI  Serial peripheral interface
- STS  State transition system
- TCB  Trusted computing base
- TRM  Technical reference manual
- UART Universal asynchronous receiver-transmitter
- USB  Universal serial bus
Chapter 1

Introduction

1.1 Motivation

Computer systems have become an indispensable component of society with a broad range of applications, for example from regular devices such as a smartphone to complex systems like a nuclear power plant control system. The basic demand for computer systems is that these systems can deliver services as users expect. For instance, users can transfer money to a friend using their bank’s application on a smartphone. Behind a successful money transfer, components in the smartphone perform their functionalities correctly. For example, the touchscreen allows users to enter necessary details such as the recipient’s information, and the bank application authenticates the identity of users typically using accounts plus passwords. In addition, the demands for the security of computer systems have grown dramatically. An essential property of security is confidentiality, which guarantees that only authorized users have access to certain data. For example, bank accounts and passwords stored in the smartphone are solely accessible to users themselves.

Figure 1.1 presents the overview of a typical computer system that is divided into two main parts: hardware and software. Hardware involves physical components in a computer that perform specific functionalities. For instance, the memory stores data, the CPU executes instructions, and peripherals enable interaction with the external environment. Software comprises programs running on top of hardware, including, for example, operating system (OS) kernels, device drivers operating peripherals, and various applications, etc. Interfaces between software and hardware provide details about physical components and technical guidelines for software to operate these components, e.g., technical reference manuals (TRM) for peripherals, and instruction set architectures (ISA). TRMs for peripherals describe functional features (e.g., data transmission and reception) of specific peripherals and suggested operations for drivers (e.g., configuration order between different registers of a peripheral [1]). ISAs (see details in Section 2.1.5) define the encodings and expected outcomes of instructions executed by processors, serving as the major interface
between software and hardware.

The processor is the fundamental hardware component in a computer system. Sequential processors handle one instruction at a time. Modern processors employ advanced techniques such as instruction pipelining, out-of-order (OoO) execution and speculation (described in Section 2.1.1) to optimize and parallelize instruction execution. Instruction pipelining divides the processing of instructions into different stages (e.g., fetch, decode, and execute). Instruction pipelining is a prerequisite of OoO execution and speculation. To better utilize pipeline resources, OoO execution permits the rearrangement of instructions’ execution order, deviating from the original order specified by programs. For instance, Berkeley out-of-order machine (BOOM) [2] and RSD [3] are OoO pipelined processors. Speculation allows processors to execute instructions ahead of branches or conditional operations. After determining the validity of these branches or conditions, the results can be directly committed or dismissed. For example, the SonicBOOM processor [4] supports speculative execution. Pipelined processors can handle multiple instructions concurrently, and can operate at a higher clock frequency compared to sequential processors since the internal stage circuit is simpler. ISAs hide the complexities of processors’ internal design (e.g., instruction pipelining) for software developers.

Processors need additional hardware to interact with the external environment. This is typically accomplished through peripherals. Peripherals and their drivers constitute a subsystem that provides communications (e.g., receiving users’ commands or transmitting sensitive data to another device) for other software. For example, we use the serial peripheral interface (SPI) for a random number generator (RNG) to connect a BeagleBone Black board and a camera to produce images of a lava lamp, which is a high-quality source of physical randomness [5,6].

Functional correctness is a basic requirement for critical components (e.g., peripherals and pipelined processors) in computer systems. A computer system should be a fully functional correct stack involving software and hardware. However, because of the size and complexity of computer systems, it is infeasible to ensure...
1.1. MOTIVATION

the correctness of every program and hardware. To simplify the problem, recent studies [7,8] present the end-to-end correctness of compilers and processors, where a correct compiler generates programs that are executed on a correct processor. In this thesis, our functional correctness work focuses on peripherals and pipelined processors, and we discuss the construction of a fully correct stack based on our results in Section 5.2.

Functional correct subsystems of peripherals must provide communication services according to their TRM. Similarly, pipelined processors are required to execute instructions according to the specification of their ISA. However, peripheral subsystems and processors present significant challenges to ensure their functional correctness. Device drivers are the major cause of functional failures in peripheral subsystems. For example, device drivers account for over 70% of the Linux v2.6 kernel source code (around 5 million lines of code) [9,10]. It is difficult to implement these large-scale drivers without any fault. Such faults may arise during execution and cause violations of functional correctness. A massive number of bugs in drivers has been discovered in the Linux kernel [11], and poorly-written driver code is responsible for more than 70% of OS crashes [10]. The major challenge of pipelined processors lies in their pipeline design and implementation. An implementation of pipelined processors contains thousands of (even more) lines of code written in hardware description languages (HDL). For example, even a simple research-purpose processor BOOM [2] consists of about 9,000 lines of code written in the Chisel HDL [12]. Furthermore, instructions may have internal dependencies. These dependencies can introduce hazards like data hazards (see Section 2.1.1) and affect the execution of instructions in a pipeline, preventing pipelined processors from producing correct results.

The functional correctness of peripherals and pipelined processors constrains a target system according to a reference system (in our case, TRM and ISA respectively). A reference system is correct-by-construct, and refinements can transfer the functional correctness and other properties from a reference system to a target system. Different relations such as simulation and bisimulation are used to establish refinement between these systems. Specifically, the notion of strong simulation requires that all operations of a target system are a subset of the reference system, while the notion of weak simulation only considers operations of a system that are observable to the external environment. Section 2.3.4 introduces formal definitions of strong and weak simulation and bisimulation.

A trustworthy computer system is not only functionally correct, but also secure. Confidentiality is a critical security requirement to protect software, sensitive data and other information processed in the systems. Confidential data can be leaked in various ways. For example, a secret key $s$ is directly copied into a public key $p = s$. Confidentiality can also be violated using side channel attacks, for example, based on timing and power consumption. To demonstrate side channel attacks, Figure 1.2 shows two programs that compare if a public key $p$ matches a secret key $s$. The compare_v1 function will stop as soon as a different bit is found. Suppose that the execution of each iteration in the loop takes the same amount of time,
def compare_v1(p, s):
    for x, y in zip(p, s):
        if x != y:
            return False
    return True

def compare_v2(p, s):
    res = 0
    for x, y in zip(p, s):
        res |= x ^ y
    return res == 0

Figure 1.2: Two compare programs in Python

this function will take a longer time if the two keys match more bytes than less. An attacker can infer the entire value of \( s \) based on variations of compare_v1’s execution time. Suppose the attacker can set input \( p \) and observe the execution time of compare_v1 and cannot observe \( s \) and the result of compare_v1. To extract the first byte, the attacker can try all 256 possibilities and observe which one uses a longer time than the others. Following this way, the attacker can learn the next byte until the entire value is recovered. The non-functional aspect (i.e. the execution time) of compare_v1 causes the secret leakage, and the execution time is called a side channel (Section 2.2.2) that allows attackers to extract confidential data. To avoid the timing side channel, the compare_v2 function updates the loop that always performs the same number of operations regardless of the contents of \( p \) and \( s \). So, compare_v2 will take a constant execution time and this measure is called constant time programming.

Side channels arise not just from software bugs but also from hardware vulnerabilities. For example, Spectre [13] and Meltdown [14] are well-known processor vulnerabilities using side channels like timing and cache access. These vulnerabilities are caused by speculation and OoO execution, which are processor pipeline features unavailable in the ISA. Information flow analysis at ISA level cannot detect secret leaks with respect to these vulnerabilities. Microarchitecture (Section 2.1.5) is an abstraction of processors below the ISA, which includes speculation and OoO execution and enables reasoning about information flows of programs under these features.

Variations of noninterference (NI) [15–17] are proposed to capture undesired information flows. Intuitively, NI regulates that the behaviour of a system, as observed from a specific channel, is independent of confidential data. NI contains a security policy that defines sensitive and public data in a system and describes allowed and forbidden information flows between data. For example, NI requires that the execution time of programs in Figure 1.2 is not affected by \( s \). Definitions of NI and variants (conditional noninterference CNI) are introduced in Section 2.3.6. In this thesis, we use variants of NI for peripherals and processors, which formulates that execution on a target system does not leak more information on specific channels than permitted on a reference system.

Formal verification (Section 2.3) has been broadly used to guarantee important properties such as functional correctness and information flow security of critical components. Notable examples of formal verification include the seL4 OS ker-
1.2. RESEARCH QUESTIONS

Formal methods are mathematical approaches and tools to model, specify, and verify systems to ensure that the systems satisfy their intended requirements. Formal methods include model checking [21], SMT/SAT solvers [22, 23], and interactive theorem proving [24, 25]. For the programs in Figure 1.2, we can construct a formal model to represent their operations and prove or disprove NI based on the formal model. For example, we define a function $\text{tm}$ to approximate the execution time of the program using the number of iterations in the loop, assuming each iteration takes the same amount of time. Based on the $\text{tm}$ function, we can verify whether the execution time of programs is affected by inputs or not.

Formal models and proofs can be developed using interactive theorem provers (ITP). ITP involves a formal language or logic in which mathematical models and machine-checked proofs can be expressed, and allows users to prove properties by combining human guidance with automated reasoning embedded in the prover. In this thesis, our verification for peripherals and processors is carried out in the HOL4 theorem prover [26].

1.2 Research Questions

This thesis aims to address gaps in the construction of a trustworthy (correct and secure) stack by verifying the functional correctness and information flow security of peripherals and processors using interactive theorem proving mainly HOL4.

The following two research questions motivate the work carried out for this thesis.

1. How to formally verify that the functional behaviour of a target system satisfies the specifications defined by a reference system?

This thesis presents refinement-based verification for both peripheral subsystems and pipelined processors.

We define a correct-by-construction abstract model of I/O devices and drivers to describe the expected results of I/O subsystems’ executions. The definition of the abstract model is based on the TRM of an SPI device [1]. The TRM includes details of the device and guidelines for the device driver. We formalise the SPI device with its driver and verify the refinement between the abstract and SPI models by establishing a weak bisimulation. The weak bisimulation ensures the functional behaviour (e.g., data transmission) of the SPI subsystem satisfies the abstract model.

We implement and formally verify an in-order pipelined processor Silver-Pi for the RISC ISA Silver [7]. For a verified circuit implementation, we construct the processor using a verified hardware development framework [27, 28] targeting the Verilog HDL. The correctness of the processor is established by exhibiting a trace relation (see Section 2.3.5) between the pipelined circuit and the Silver ISA.
2. How to ensure the information flow security of a target system given information flows in a reference system?

The refinement between the abstract and SPI model ensures information flow security for both single peripheral devices and when devices are connected in an end-to-end fashion. The weak bisimulation guarantees that two systems have the same information flow (up to channels that are not defined in the models like timing). The weak bisimilarity prevents malicious driver operations and allows to compose the SPI subsystem with nondeterministic components safely.

If the target system in Question 2 is a processor, the reference system (i.e., the ISA) does not contain enough information to reason about the processor’s security. For example, a processor that supports instruction pipelining and out-of-order execution can be exploited by an attacker to extract confidential data via side channels. Information flows at ISA level cannot prevent such information leakage in the processor. To ensure the information flow security of processors, additional reference systems are required depending on the underlying hardware and the power of attackers. In this thesis, we focus on two cases, in-order pipelined processors and microarchitectural out-of-order execution. We address the following two research questions.

3. How to guarantee the information flow security on the timing channel of a pipelined processor?

To reason about the timing channel of a processor, we extend an ISA model with an observation function as the reference system. The observation function extracts the parts of the ISA state that may affect the execution time of programs, including the program counter, data address, etc. We propose a notion of CNI, which formulates that a pipelined processor should not leak more information via the circuit’s timing channel than permitted by the observation function expressed at ISA level. The CNI for Silver-Pi is established by utilizing the trace relation between the circuit and ISA in the correctness proof. The information flow methodology is able to accommodate various processor designs, attacker models, and environments.

4. How to identify undesired information flows of programs caused by out-of-order execution?

We present the formalization of MIL (machine independent language), which abstractly describes microarchitectural in-order and out-of-order program execution and enables reasoning about program information flows. The microarchitectural in-order execution is the reference system for out-of-order execution. A notion of CNI states that the out-of-order execution of a program should not leak more information via trace-driven cache side channels (see Section 2.2.2) than the in-order execution of the program. MIL has a verified executable semantics that can compute results and observations of programs’ execution. We propose a semi-automated bisimulation-based strategy to establish CNI for programs partially using the executable semantics.
1.3 Contributions

This thesis includes the following publications:


- **Paper C** Ning Dong, Roberto Guanciale, Mads Dam, and Andreas Lööw, “Information Flow Analysis of a Verified In-Order Pipelined Processor,” *technical report*, 2023.


Paper A, B, and D have been peer-reviewed and published at the FMCAD conference. Paper C is a technical report of the nonmechanised CNI proofs that are abstractly presented in Paper B.

**Contributions:** The author of this thesis is the main author of Paper A, B, and C. The author has contributed to the implementation of applications, formal definitions and proofs, evaluation of results, and paper writing of the three papers. Paper D is a result of teamwork. In the formalization of Paper D, the author of this thesis has contributed to the implementation and correctness proof of the executable semantics, the security proof using the executable semantics, and other small tools.

1.4 Outline

This thesis is divided into two parts. The first part presents the introduction and background to understand the work of this thesis and a summary of the included publications. The second part consists of the four publications.

The first part contains five chapters which are structured in the following way. Chapter 2 describes the background related to the topic of this thesis, including hardware and software components, computer correctness and security, and formal verification. Chapter 3 presents related work such as formal verification of peripherals, drivers and processors, and verified tools for information flow analysis. Chapter 4 summarizes the major contents of each included paper. Chapter 5 concludes the first part of this thesis and discusses future work based on the results of this thesis.
Chapter 2

Background

This chapter introduces the background of this thesis. The background includes the description of critical components in computer systems (Section 2.1), functional correctness of critical components and security threats and countermeasures (Section 2.2), and related methods and techniques of formal verification e.g., interactive theorem provers, labelled transition systems, noninterference (Section 2.3). In the background, we use a pseudo assembly language as shown in Table 2.1 to represent programs.

2.1 Computer System

2.1.1 Processors

Processors are responsible for executing instructions stored in the memory and updating results to corresponding components (e.g., the memory and register file). As mentioned in Section 1.1, instruction pipelining has separate pipeline stages to handle multiple instructions concurrently. Figure 2.1 presents the overview of a 5-stage pipeline, including instruction fetch (IF) and decode (ID), execution (EX), memory access (MEM), write back (WB), and interfaces between two adjacent stages. The pipeline stages work as follows:

- **IF**: updates the program counter (PC) and fetches instructions from the memory.
- **ID**: decodes instructions and reads the register file for source operands data.
- **EX**: executes instructions using units like arithmetic logic unit (ALU).
- **MEM**: processes instructions that require interactions with external components like memory. The results of memory loads are delivered to the **WB** stage.
- **WB**: updates the results of instructions to the register file.
We have implemented a pipelined processor according to Figure 2.1 in Paper B.

Instruction pipelining presents general challenges that are independent of ISAs for the correct execution of programs. For instance, pipelines cause data hazards as shown in Table 2.1. The instruction i3 could read a wrong (old) value for the register R1 in the ID stage, because the previous instruction i2 is going to update R1’s value but i2’s result has not been committed to the register file. To prevent wrong inputs for instruction execution, there are two common solutions for data hazards: data forwarding and pipeline stalling. Data forwarding means that subsequent pipeline stages send the correct result of instructions to the ID stage. Pipeline stalling holds the instruction in the ID stage until the previous instructions that affect the registers are completed in the pipeline. Data forwarding is generally more effective than pipeline stalling, but a forward unit may still require a stall unit to work properly [29]. For example in Table 2.1, suppose i2 is a memory load to the register R1, the result of i2 is not available at the cycle t. Later, if the result of i2 is loaded from memory and forwarded to the ID stage, then a stall unit can release i3 for execution.

In-order pipelined processors execute and complete instructions in the order specified by programs. In-order pipelines may partially or totally stall to ensure correct executions when certain conditions (e.g., data hazards) happen. Advanced techniques such as out-of-order (OoO) and speculative execution are proposed to reduce pipeline stalls and improve the utilization of pipeline resources. OoO execution allows processors to rearrange the order of instructions by analysing dependencies between instructions and the availability of execution units. This means that OoO pipelines can execute instructions in an order that is different from the original order specified by programs. Speculative execution enables processors to execute instructions ahead of branches or conditional operations. After determining the ex-

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>i4</td>
<td>i3:R2:=R1-2</td>
<td>i2:R1:=R0+1</td>
<td>i1</td>
<td>i0</td>
</tr>
</tbody>
</table>

Table 2.1: Example of data hazards
ecution validity, speculative processors can commit the results directly if branches
or conditions hold, or dismiss the results if misprediction happens.

2.1.2 Peripherals

Many peripherals are serial devices to communicate with the external environment, for example, USB (universal serial bus), UART (universal asynchronous receiver-transmitter), SPI (serial peripheral interface) and NIC (network interface controller). In addition, there are peripherals as auxiliary hardware components in computer systems to perform specific functionalities. For instance, sound cards process audio data and artificial intelligence (AI) accelerators handle AI applications like artificial neural networks.

We choose the SPI protocol for close-range serial communication as the demonstrating example for peripheral verification. As mentioned in Section 1.1, we develop a random number generator that utilises the SPI protocol to connect a board and a camera for capturing a physical random resource. The SPI bus typically connects a controller and multiple peripherals through four kinds of lines including SCK (serial clock), SDI (serial data in), SDO (serial data out), and CS (chip select). The controller uses SCK to synchronise the timing information with peripherals and CS to select the desired communicating peripheral. The SPI protocol supports both half- and full-duplex data transmissions. The half-duplex mode means that the devices transmit or receive data at a time. The full-duplex mode allows transmitting and receiving data simultaneously on separate data lines SDI and SDO, which can improve the efficiency of communications. SPI devices provide various configurations such as the length of data (e.g., 8/16 bits per word) and the transfer mode (half-/full-duplex) for drivers.

2.1.3 Hardware Description Languages

Software programs are usually written in programming languages e.g., C, Java, and Python. Similarly, hardware devices such as processors can be implemented as digital circuits using hardware description languages (HDL) such as Verilog, VHDL, and Bluespec. HDLs provide abstractions of physical elements and allow developers to design and describe the structure and behaviour of complex digital systems. Hardware programs written in HDLs require further synthesis into netlists that represent the programs as a collection of interconnected electronic components like registers and flip-flops. Then, netlists can be used to produce the physical implementation through steps such as placement and routing. A typical abstraction used in HDLs is the register transfer level (RTL), which focuses on the data flow between registers and logical operations performed on data. However, the term “RTL” can refer to various meanings in hardware design. For example, RTL means resistor-transistor logic in the design of integrated circuits, using resistors and transistors to implement logic gates and functions in circuits. To avoid confusion, we use the
target HDL (Verilog) directly as the abstraction level of circuits in Paper B, rather than RTL.

Verilog is an extensively used HDL in the design and implementation of processors. Hardware developers can describe both combinational and sequential logic in Verilog. The combinational logic allows circuits to generate outputs based on the current values of inputs without the stored state (i.e., stateless logic), e.g., 

\[ \text{always}_{\text{comb}} \] blocks.\(^1\) In contrast, the sequential logic is a stateful logic that generates outputs based on the current inputs and the stored state using memory elements such as registers, e.g., \( \text{always}_{\text{ff}} \) blocks (\( \text{ff} \) stands for flip-flop, which is a type of digital memory circuit used in electronic devices). Verilog supports two types of assignments: blocking (=) and nonblocking (\( \leq \)) that are scheduled with/without blocking the execution of the following statements. For example, execute the following code where the value of \( x \) is 1 before executing, then \( y \) gets the value 0. If the first assignment is nonblocking, then \( y \) gets the value 1.

\[
\begin{align*}
\text{always}_{\text{ff}} & @ (\text{posedge} \ \text{clk}) \ \text{begin} \\
n & = 0; \\
y & = x; \\
\text{end}
\end{align*}
\]

\[ \text{always}_{\text{comb}} \text{ and } \text{always}_{\text{ff}} \text{ blocks are SystemVerilog features. For simplicity and to be consistent with the Verilog papers [27, 28], we describe them as a part of Verilog.} \]

2.1.4 Drivers

Software programs written in high-level programming languages, called source programs, cannot be executed by processors directly. To execute source programs, compilers translate them into low-level binary instructions specified by the ISA (see Section 2.1.5) according to the programming language and target hardware device. For example, the GCC compiler supports multiple languages like C and C++ for various architectures such as x86 and ARMv8 [30]. However, compilers themselves as large-scale software are error-prone and have massive bugs [31]. To avoid mistakes during compilation, the SPI driver model in Paper A is a manual translation of the driver’s binary.

Peripherals are usually configured by operating their registers located in specific regions or partitions of the memory. Drivers interact with peripherals through three major methods: register polling, interrupt, and direct memory access (DMA). Register polling means the CPU continually checks the status of specific registers of the device to determine if the following operations (e.g., data transmission) can be performed. Interrupts improve the efficiency of interactions between devices and the CPU. The CPU can work on other tasks rather than the driver’s operations, and the device issues interrupts to the CPU when certain conditions happen e.g., data is received from the external device. Then, the CPU suspends its current work to handle interrupts by executing interrupt handlers. The handlers perform certain operations e.g., reading data from the device and updating the device state.
2.1. COMPUTER SYSTEM

Some devices have a DMA controller that allows devices to access (read and write) the memory without involving the CPU. DMA controllers are configured by their drivers to coordinate the interaction between the device and memory. In Paper A, the verified SPI driver interacts with the SPI device through register polling.

2.1.5 Processor Abstraction

Different abstractions of processors describe the behaviour of processors without certain details e.g., the placement of gates. Figure 2.2 shows two common abstractions, ISA and microarchitecture.

Instructions Set Architectures

An ISA mainly captures functional features of processors, including supported instructions, available resources like registers, interfaces with the environment (e.g., data address and value for accessing the memory), etc. The major contents of ISAs are instruction encodings and semantics. The complexity of instructions can distinguish two major design methodologies of computer architectures, CISC and RISC (complex/reduced instruction set computer). CISC (e.g., x86 [32]) provides a large number of complex instructions that can perform high-level operations such as decimal arithmetic in a single instruction. CISC offers convenience for software engineers to develop large-scale programs such as compilers, since engineers can use complex instructions to perform high-level operations directly. However, CISC requires more complex hardware components to handle the wide range of instructions. On the contrary, RISC (e.g., RISC-V [33] and MIPS [34]) emphasizes the simplicity of instructions. The purpose of RISC is to provide a relatively small amount of simple instructions that can be executed quickly and efficiently and thus facilitate hardware implementation. The distinction between CISC and RISC has become less pronounced with the progress in processor technology, as numerous processors are designed using a blend of both CISC and RISC styles. For example, since 1995 Intel cores for the x86 ISA have implemented decoders that translate complex
instructions into low-level microinstructions that can be executed by hardware directly [35, 36], exhibiting a RISC-like feature. The MIPS ISA provides extensions for complex instructions such as MIPS-3D for improving 3D graphics operations, following CISC style. The simplicity of RISC ISAs makes them well-suited for high-performance processor design and implementation techniques such as instruction pipelining.

Lööw et al [7] propose a general-purpose RISC ISA called Silver and present a verified compiler and a non-pipelined processor for Silver. Paper B verifies the implementation of a pipeline in HOL4 (see Section 2.3.1). Since the Silver ISA is available in HOL4 and Paper B uses the same tools (the HOL4 Verilog library in Section 2.3.1) as Lööw et al [7], we choose the Silver ISA as the target for processor verification. The major advantages of Silver are its generality and simplicity. The supported instructions of Silver are similar to other RISC ISAs like RISC-V RV32I, including conditional and unconditional jumps, bytewise memory load and store, and arithmetic computations. The internal functionalities of these instructions are straightforward and easy to implement. The state of Silver ISA contains the program counter, memory, register file, I/O ports, etc. Notably, the Silver ISA has an acceleration instruction ACC that computes an addition of the lower and higher 16 bits value for a 32 bits input. The ACC instruction is a placeholder for more complex functionalities. The ACC instruction is supposed to be computed by a separate accelerator running in parallel with the processor in hardware. The existence of I/O ports and ACC instruction enables further extensions of Silver processors to support peripherals. For instance, we can extend the Silver ISA’s I/O ports to connect external devices, or implement a peripheral as an accelerator.

Microarchitectures

ISAs focus on functional features and do not cover hardware features such as OoO and speculative execution. These features are invisible to software developers, since the features are used to optimise hardware performance and do not affect the execution results of programs. However, vulnerabilities (e.g., Spectre and Meltdown, see Section 2.2.2) demonstrate that these features can cause information leakage during program execution. These information leakages cannot be detected at ISA level. Microarchitectures contain features like OoO execution and enables reasoning about the behaviour of programs under these features.

A microarchitecture is an intermediate between an ISA and a processor, providing the implementation principles for physical processors based on ISAs. Microarchitecture refers to the internal design of processors with hardware details that describe how instructions are processed, including for example instruction pipelining, advanced pipeline designs like OoO execution, execution unit, etc. For example, Marti et al. [37] propose the VB microarchitecture that supports OoO execution, as shown in Figure 2.3. Instructions are fetched into a queue and then decoded. The register alias table (RAT) maintains the mapping between logical registers used in instructions and corresponding physical registers in the register
file. Whenever a new physical register is mapped to a destination logical register, the frontend RAT is updated. An instruction is sent to the instruction or load/store queues for execution depending on the type of the instruction. Instructions are pushed into the validation buffer (VB) following the program order, and each entry in the buffer records the execution status of the instruction. The VB ensures that the results of instructions are committed in the program order while allowing the underlying OoO execution. Once an instruction leaves the VB, the changes that the instruction made are recorded by updating the retirement RAT (e.g., free some physical registers). The register status table (RST) collects information on instructions execution and determines if physical registers have restrictions (e.g., due to data hazards) to use. If the misprediction of the PC happens, the retirement RAT is copied into the frontend RAT, which drops the updates of wrongly fetched instructions to the machine state.

Different microarchitectures have various OoO mechanisms, leading to distinct scheduling of instructions for the same program. Instructions may have internal dependencies such as data hazards, and such dependencies affect the OoO execu-

![Diagram of VB microarchitecture](image)

Figure 2.3: VB microarchitecture

![Diagram of out-of-order execution](image)

Figure 2.4: Out-of-order execution of a program
tion. Figure 2.4 shows different out-of-order executions of the program in Table 2.1 where i2 and i3 have a data hazard, and the dashed line represents the in-order execution. Because of the data hazard, i3 cannot be executed before i2 and thus the red-coloured executions in Figure 2.4 are not allowed. To cover various microarchitectures, the formalisation of MIL in Paper D introduces nondeterminism to its OoO semantics rather than a specific OoO mechanism. Nondeterminism means multiple choices of instructions to be executed after a given instruction, and MIL checks dependencies between instructions to prevent invalid OoO executions. For example, the next instruction after i1 can be i2 or i4, but not i3.

2.2 Correctness and Security

This section introduces the general concept of functional correctness and security regarding peripherals and processors. To establish these properties, we present formal definitions of functional correctness and security in Section 2.3.

2.2.1 Functional Correctness

Functional correctness is a fundamental requirement to construct trustworthy computer systems. As described in Section 1.1, this thesis focuses on peripherals and processors. References describe the expected behaviour of peripherals and processors and are used as specifications.

Since many peripherals are operated through memory-mapped registers, technical reference manuals (TRM) of these peripherals describe details of registers and operation guidelines of the registers for drivers. The SPI model in Paper A is based on the TRM of the Texas Instruments McSPI device used in the AM335x family of SoCs [1]. This TRM introduces four functionalities of SPI, including initialization, full-duplex transfer, transmission and reception. We build an abstract model to describe the expected results of the four functionalities. For example, the model is ready to perform data transmission after initialization and data can be transmitted to an external device using transmission. The functional correctness of SPI means that the device and driver provide these functionalities as the abstract model defines. To achieve functional correctness, an SPI driver operates the device following the TRM. For example, the driver first writes 1 to the reset register of the SPI device for initialization. Then the driver continually checks the system status register of the device to know whether the reset is done by the device. After the reset is finished, the driver configures the device by writing several registers. Once the settings are finished, the initialization of the SPI device is finished and the device is ready for data transmission or can be reinitialized.

The functional behaviour of processors defined in ISA is generally more complex than peripherals. ISA contains different types of instructions, resources and interfaces of processors, etc. Intuitively, the functional correctness of a processor requires that the processor produces the same result for instructions as ISA
describes. Many ISAs like Silver are defined as sequential models that contain a state and present the execution of one instruction as a step to update the state. Since sequential processors execute instructions in a similar way to the ISA models, it is straightforward to ensure the functional correctness of sequential processors. However, the functional correctness of pipelined processors is more complicated as pipelines handle multiple instructions concurrently. To ensure the correct result of one instruction in a pipeline, it is necessary to keep track of the instruction in every stage and compare the computation of each stage to the ISA state that handles the same instruction. For example, the pipeline in Figure 2.1 processes a simple instruction \( R_1 := R_0 + 1 \). An ISA step executes the same instruction and becomes the reference to check the correctness of the pipeline. The correct result in the WB stage requires no modification to the result in the MEM stage and the correct computation in the EX stage, and the computation is based on the correct fetch and decode of the IF and ID stage. We introduce a methodology to ensure the correctness of a pipelined processor in Section 2.3.5.

### 2.2.2 Security

The requirements of trustworthy computer systems are not only functional correctness but also security. A major ambition of security is to protect the integrity and confidentiality [38] of software, sensitive data and other information handled in computer systems. We use a multilevel security (MLS) setting [39, 40] to interpret confidentiality and integrity. In the context of MLS, agents (e.g., a process or a program) run on top of a processor and can modify objects (e.g., a memory or a register file) in a computer system. These agents and objects are classified with security levels such as confidential, secret and top secret from lower to higher [40]. Confidentiality ensures that information in objects at a certain security level is not disclosed to agents with a lower security level. Integrity means that agents at a lower security level cannot modify objects at a higher level. This section focuses on the confidentiality of processors. Attackers can exploit indirect information (side channels) of processors’ execution to infer secret information, violating confidentiality. In the following, we first introduce side channel attacks and countermeasures, then describe processor vulnerabilities that enable side channel attacks.

#### Side Channel Attacks

The execution of programs on computer systems produces not only results according to the definition of source programs but also extra information (e.g., timing, power consumption and electromagnetic radiation) about the system. The extra information can be gathered from the system’s implementation and the environment. Since many abstractions of systems such as ISA primarily capture functional behaviour, the transfer of the extra information can create channels that are unintended at the abstract level. Attackers exploit these unintended channels to extract sensitive data stored in the system, i.e., side channel attacks.
Timing side channel: A common type of side channel attack is timing, where an attacker measures the duration a system takes to perform certain operations. The attacker learns sensitive data by analysing time variations. In Section 1.1, Figure 1.2 presents a simple program compare_v1 that checks whether a public key \( p \) matches a secret key \( s \). Suppose the execution of each iteration in the loop on hardware takes a constant time, the execution time of compare_v1 is longer if the two keys match more bytes than less. An attacker can measure the execution time of compare_v1 to extract the first byte of \( s \) by trying all 256 possibilities and finding the one used the longest time. Then the attacker learns the next byte using the same approach until the entire value of \( s \) is recovered. Similar to this simple example, many cryptography implementations are vulnerable to timing side channel attacks [41]. Hardware implementations may also cause timing side channels, and we have studied the timing security of a pipelined processor in Paper B and C.

Cache side channel: Almost all modern processors interact with caches to access instructions and data stored in the main memory, since caches have a smaller and faster memory that stores frequently used instructions and data to speed up the accesses. However, interactions between processors and caches introduce side channels using the patterns of caches like hit/miss rates. There are several common approaches to exploit cache side channels, e.g., PRIME+PROBE [42] and FLUSH+RELOAD [43].

PRIME+PROBE allows an unprivileged process (the attacker) to attack other processes (the victim) running in parallel on the same processor through cache. PRIME+PROBE typically operates as follows:

1. The attacker fills the cache with their own data in the main memory.
2. The victim executes and accesses the cache for memory operations, which may cause some of the attacker’s data to be evicted from the cache to accommodate the victim’s data.
3. The attacker accesses the memory used in Step 1 again to measure the time it takes. If the victim has accessed a cache line and caused an eviction in Step 2, then a cache miss happens and the access time of the cache line takes longer. Therefore, the attacker can learn whether or not the victim has accessed a certain partition of memory corresponding to the cache line.

Differing from PRIME+PROBE, FLUSH+RELOAD requires page sharing between the attacker and victim processes and targets the last level cache. The following shows the steps of FLUSH+RELOAD:

1. The attacker flushes specific cache lines using specific instructions like clflush in x86 processors. This step ensures that the targeted memory locations are not present in the cache.
2. The victim executes and accesses the main memory through the cache. If the victim accesses the memory locations flushed earlier, these locations will be loaded back into the cache.
3. The attacker accesses the memory locations flushed in Step 1 again. If the access of a location is faster (i.e. a cache hit), it indicates that the victim has accessed this location, otherwise not.

Attacks can gather information from different patterns of caches to learn secrets. Generally, there are three main categories of cache side channel attacks depending on the capabilities of the attacker:

- **time-driven** [44–46]: The attacker measures the total execution time of the victim. The attacker may infer a secret as the execution time depends on the number of secret-dependent memory accesses that result in cache misses and hits.

- **access-driven** [47, 48]: The attacker can observe specific resources the victim accesses. For example, PRIME+PROBE identifies individual cache lines that the victim has accessed.

- **trace-driven** [49, 50]: The attacker obtains the succession (trace) of certain activities during the victim execution. For instance, the attacker records the trace of cache misses and hits to learn the secret key in an implementation of Advanced Encryption Standard (AES) [49].

Information flow analysis in Paper D considers trace-driven cache side channels for programs represented in MIL. For more details, see Section 2.3.6.

**Countermeasure: Constant-Time Programming**

To prevent cache and timing side channels, the constant-time programming discipline is a set of policies to ensure that branch instructions and accessed memory addresses are independent of sensitive data. For example, the `compare_v2` function in Figure 1.2 checks all bits of inputs `s` and `p` and thus takes a constant execution time independent of the contents of inputs. Constant-time programming has been widely used in software development such as cryptography implementations [51–54]. For the compilation of constant-time programs, Almeida et al. [55] propose a framework Jasmin containing a programming language, a compiler and embedded tools for checking the constant-time property of programs. The Jasmin compiler preserves the constant-time property of programs during compilation. Similarly, Barthe et al. [56] present a modified version of the CompCert C compiler, which guarantees that the compiled code preserves the constant-time property of source programs.

**Processor Vulnerabilities**

Microarchitecture features such as speculative and OoO execution are extensively used in modern processors but may enable side channel attacks, as demonstrated by recent processor vulnerabilities like Spectre [13], Meltdown [14], and Orc [57]. These
vulnerabilities take advantage of performance optimizations that processors employ and promote security considerations in processor design and implementation.

Spectre is a class of processor vulnerabilities caused by speculative execution. Misprediction can arise during the speculative execution of instructions, leading to observable side effects that are influenced by confidential data. Spectre contains 4 main variants that exploit different components in speculative execution to extract secrets, including Spectre-PHT (pattern history table), Spectre-BTB (branch target buffer), Spectre-RSB (return stack buffer), and Spectre-STL (store to load). For example, PHT records patterns of conditional branch behaviours i.e., whether a branch was executed or not. When encountering the same branch in the future, PHT uses the stored history to predict if the branch will be executed or bypassed.

The program in Figure 2.5 [58] illustrates information leakage via Spectre-PHT, where $A1$ and $A2$ are two public arrays and $r0$ is a public register controlled by the attacker. The program can access $A2$ if and only if the value of $r0$ is in the bound of $A1$. However, the attacker can first use a permitted value of $r0$ so that PHT records a true for the branch (i2) and i3 is executed. Then, the attacker supplies a value of $r0$ that exceeds the size of $A1$. As a result of PHT, the processor performs an out-of-bounds memory access of sensitive data, leading to a secret leak. This thesis does not address the Spectre family of vulnerabilities, because the pipelined processor in Paper B does not support speculative execution and the formalisation of MIL in Paper D lacks the speculative semantics.

Spectre-like vulnerabilities are not solely a result of speculative execution, OoO execution may also cause such vulnerabilities. Guanciale et al. [58] present a program in Figure 2.6 called Spectre-OoO, which compromises secrets due to OoO execution. In Figure 2.6, $b1$ and $b2$ are public memory addresses, $r1$ and $r2$ are public registers, $z$ is a register storing the secret value. We assume an attacker that can observe the addresses of memory accesses via cache, including instruction and data load with an address ($il\ a$ and $dl\ a$), and data memory store with an address ($ds\ a$), and $dl\ a1::ds\ a2$ represents a trace of memory accesses. Executing the program on in-order processors always produces the same trace of cache accesses $dl\ b1::ds\ b2$. However, if the flag of $z$ is false, $i3$ does not have data dependency with $i1$, i.e., the value of $r1$ does not affect $r2$. Then, OoO processors can execute $i3$ before $i1$ and generate a cache trace $ds\ b2::dl\ b1$ if and only if $z$ is false, allowing the attacker to learn $z$’s value.

Meltdown exploits the side effects of OoO execution to compromise the memory isolation between privileged programs (e.g., an OS kernel) and other unprivileged programs.

\begin{verbatim}
i1: r1 = A1.size;
i2: if (r0 < r1)
i3: then y = A2[A1[r0]];
\end{verbatim}

Figure 2.5: A program example for Spectre-PHT
2.2. CORRECTNESS AND SECURITY

Figure 2.6: A program example for Spectre-OoO

\[
\begin{align*}
i1: & \ r1 = *b1; // load to r1 from mem[b1] \\
i2: & \ cmov \ z? \ r2 = r1; // copy r1 to r2 if z holds \\
i3: & \ *b2 = r2; // store r2 into mem[b2]
\end{align*}
\]

applications. This compromise enables attackers to gain access to protected memory locations that store sensitive data. Meltdown relies on race conditions in circuits between the fetch of a memory address and the corresponding privilege check for this address, and does not rely on any software vulnerabilities \[14, 59\]. This thesis focuses on the behaviour of programs under pipeline features and does not study race conditions in circuits, so we do not address Meltdown and omit its details here.

Basic in-order pipelined processors are not affected by Spectre- and Meltdown-like vulnerabilities since in-order pipelines lack speculative and OoO execution. However, in-order pipelined processors remain insecure against side channel attacks, as the Orc attack \[57\] in Figure 2.7 shows. The program has a secret address \( sa \) that stores the secret data \( \text{mem}[sa] \) and a public address \( pa \), \( v \) is the guessed value of the attacker, and other values are regular registers. A correct processor is required to raise an exception of \( i5 \) and forbid the execution of \( i6 \) because \( i5 \) attempts to read the secret data \( \text{mem}[sa] \) from the memory. However, before this control becomes valid, \( i6 \) has already been executed by the processor pipeline and started a memory request via the cache. The memory request can affect the cache state and thus the execution time of the program. If the cache is also pipelined i.e., allowing new memory requests while processing previous ones, the cache has a memory request trace \( ds \ pa+v::dl \ sa::dl \ mem[sa] \). If the values of \( ps+v \) and \( \text{mem}[sa] \) are identical, the cache will have a read-after-write hazard in its pipeline and spend a longer processing time. Therefore, the attacker infers the secret data from the guessed value using the timing side channel.

In software development, constant-time programming is usually guaranteed to the source or binary code level. The emergence of Spectre and other vulnerabilities shows violations of the constant-time property when programs are executed on processors. To ensure the security of programs, some studies \[58, 60\] extend

Figure 2.7: A program example for Orc

\[
\begin{align*}
i1: & \ r1 = sa; \\
i2: & \ r2 = pa; \\
i3: & \ r2 = r2 + v; \\
i4: & \ *r2 = r3; // store r3 to mem[r2] \\
i5: & \ r4 = *r1; // load to r4 from mem[r1] \\
i6: & \ r5 = *r4; // load to r5 from mem[r4]
\end{align*}
\]
CHAPTER 2. BACKGROUND

the concept of constant-time programming with respect to hardware features like speculative and OoO execution. Guanciale et al. [58] propose a general microarchitecture model MIL capturing OoO and speculative execution and define the MIL constant-time to prevent sensitive data from affecting programs’ execution time. Cauligi et al. [60] present an operational semantics for an abstract 3-stage machine supporting OoO and speculative execution and a formal definition of constant-time programs under their semantics to discover information leakage. Although these works have significant differences in their semantics and definitions, the main purpose of constant-time properties is consistent: the properties identify potential timing variations by analyzing the program’s behaviour under OoO and speculative execution.

2.3 Formal Verification

Formal verification is used to improve the trustworthiness of critical components in computer systems using mathematical analysis. Formal verification consists of various methods, for example, model checking, SAT/SMT solvers, and theorem proving. In addition, various models, theories and techniques are proposed and used in formal verification, e.g., labelled transition systems, calculus of communication systems, bisimulation, and noninterference.

Model checking verifies whether a finite-state model of a system satisfies a given property like functional correctness by exploring all possible states [21]. SAT (Boolean satisfiability) solvers [22] check the satisfiability of formulas constructed by Boolean variables, and SMT (satisfiability modulo theories) solvers [23] extend SAT to support more complex formulas such as real numbers, arrays, and strings. Theorem proving uses formal logic and mathematical reasoning to verify the properties (theorems) of a system with the help of theorem provers (also called proof assistants), and enables fine-grained proofs for theorems. In this thesis, we focus on theorem proving using interactive theorem provers (ITP).

2.3.1 Interactive Theorem Provers

ITPs are widely used in academia and industry for verifying software and hardware. Examples of ITP include ACL2 [61], Coq [62], HOL4 [26], and Isabelle [63]. These ITPs are built using the LCF (logic for computable functions) approach [64,65].

LCF

Robin Milner [64] first introduced a theorem proving framework LCF in 1972, based on the logic proposed by Dana Scott [66], called Stanford LCF. In order to offer high assurances about the correctness of proofs, the key idea of LCF is to build the theorem proving system with minimal trusted code (a small trusted kernel) and separate the kernel from other components of the system. The trusted kernel contains a set of inference and proof rules that provide the logical foundation for
the system and guarantee the correctness of the system. Gordon et al. [67] present a theorem prover LCF (Edinburgh LCF) which is a fully functional implementation following the LCF approach.

Proofs in the Stanford LCF are constructed in backward reasoning by declaring a desired goal and generating a proof tree from the goal. The theorem prover maintains the proof tree by recording established theorems and unsolved subgoals. Subgoals emerge at points where the proof branches, and a subgoal is solved when it reaches a leaf (corresponding to an axiom). The desired goal is proved once all subgoals within the proof tree are solved. The Stanford LCF has two major problems [68]:

1. The size of proofs is limited by available memory.

2. The set of proof commands is fixed and hard to extend.

The Edinburgh LCF adopts corresponding changes to solve the above problems and these features are continually utilized in modern ITPs like HOL4. For the first problem, the Edinburgh LCF records the results of proofs as theorems rather than the whole proofs and develops a structure to summarize and preserve only the necessary information of proofs for reconstruction. For the second problem, Milner and his colleagues have developed a functional programming language ML (meta language) [69] to implement the Edinburgh LCF. ML supports higher-order functions that can be used as arguments and returned as results and enables the definitions of proof commands as functions that can be easily composed and combined. In addition, the Edinburgh LCF supports proofs in a forward manner, which start from axioms and derive the desired theorem via a sequence of steps where each step is either an axiom or obtained by applying inference rules to a previous step.

**HOL4**

HOL4 is an interactive theorem prover for higher-order logic and a direct descendant of the Edinburgh LCF, inheriting key features including a small trusted kernel, proofs in both backward and forward manner, etc [26]. The essential logic implemented in HOL4 is Church’s simple type theory [70], which presents a formulation to construct well-typed terms. HOL4 uses the Standard ML (SML) programming language, which is a general-purpose functional programming language evolved from ML [71] allowing operations on terms. HOL4 uses terms to represent objects of interest and each term has a type which specifies the class of values the term can have. For example, a HOL4 term \( x \) has the type \( \text{N} \) as a natural number i.e., \( x : \text{N} \). As the fundamental elements in HOL4, terms are used to represent objects such as numbers and program variables, construct logical propositions using connectives like conjunction and implication, define functions, etc. Types are critical to ensure the consistent usage of terms and restrict operations only to those terms with compatible types.
HOL4 users prove properties of target systems where terms are employed to describe and define system behaviours. The proven properties with their assumptions are represented as theorems with the SML type `theorem`. HOL4 provides several constructs to derive theorems. *Rules* transform existing theorems into new ones. *Conversions* convert a term into a theorem that establishes the equivalence between the term and other term computed by the conversion. Rules and conversions generate theorems in a forward manner from known theorems to desired ones. HOL4 also offers backward reasoning using *tactics*. Tactics can reduce a proof goal to a list of simpler subgoals (e.g., the induction tactic `Induct`) and automate the proof process (e.g., prove a goal using first-order resolution `metis_tac`). In addition, HOL4 offers advanced automation in proofs including embedded decision procedures and rewrite rules, customised tactics by users, etc.

HOL4 provides an extensive collection of libraries covering various basic datatypes such as integers, words and lists. This allows users to express target systems using these libraries and construct new theorems based on existing ones in the libraries. Researchers continuously develop new libraries in HOL4 for specific purposes. For example, Lőöw et al. [27, 28] present the HOL4 Verilog library for verified circuit development. We use the Verilog library in Paper B to implement a pipelined processor. Our formalizations in Paper A, B and D are carried out in the HOL4 theorem prover.

**HOL4 Verilog Library**

The HOL4 Verilog library has a formal semantics for the subset of Verilog covering fundamental Verilog features mentioned in Section 2.1.3. The library takes a circuit represented as a HOL function as input and translates the function to a Verilog program. Figure 2.8 shows the workflow of circuit implementation using
2.3. FORMAL VERIFICATION

The library. Firstly, users create a state containing variables used by the circuit and define next-state functions to update the circuit state according to its specification. The correspondence theorem between the specification and next-state functions is supposed to be proved by users. Then, the library’s proof-producing translator generates a Verilog abstract syntax tree (AST) for the circuit where a correspondence theorem is automatically proved. Finally, given the AST, a Verilog text file is produced using an unverified pretty-print function in the library.

2.3.2 Labelled Transition Systems

ITPs provide a framework to model and verify a target system. In the following, we introduce various techniques to construct models and verify properties.

A labelled transition system (LTS) is a formal model to represent the behaviour of systems or processes, which is commonly used to define communication protocols and computer systems. For example, the SPI device and driver model in Paper A and out-of-order and in-order execution in Paper D are defined as LTS. LTS consists of the states of a system, the transitions between these states, and labels that appeared during transitions, as Definition 2.1 shows.

Definition 2.1. An LTS is a tuple \((S, \rightarrow, L, s_0)\) where \(S\) and \(L\) are a set of states and labels respectively, \(s_0\) is the initial state, and \(\rightarrow\) is a transition relation such that \(\rightarrow \subseteq S \times L \times S\). There is a transition from state \(s\) to \(s'\) with a label \(l\) written \(s \xrightarrow{l} s'\), if and only if \((s, l, s') \in \rightarrow\).

Labels in an LTS can represent actions or events that cause state transitions. For example, Paper A uses a label \(wr\) to represent the write operation of the SPI driver. The driver writes a byte \(v\) to a device register with the memory-mapped address \(ad (wr \ ad \ v)\). The SPI device model has a dual label \(\overline{wr}\), which means the device performs the corresponding action \(wr \ ad \ v\) to update the register. In this example, the data variables \(ad\) and \(v\) in labels are instantiated by potential values already when the SPI device and driver communicate on the write action and its dual action. This is called early semantics [72] because of the early instantiation of data variables. In contrast, later semantics means that the processes only synchronize on the action name and that the receiving LTS (the SPI device in this example) has to accept whatever value the transmitting LTS (the SPI driver) offers. The result of values (e.g., \(ad\) and \(v\)) is delayed until the device has received the value. For example, later semantics are used in Haskell for lazy evaluation of expressions. We use early semantics in Paper A because of the fact that the values of data (e.g., \(ad\) and \(v\)) are known when different components (e.g., the SPI device and driver) interact.

Labels can also be used to specify observations during transitions. For example, labels in MIL are the addresses of memory accesses during programs’ in-order and out-of-order execution, and the labels overapproximate what an attacker can observe through a cache.
CHAPTER 2. BACKGROUND

If for every state in $S$ there is at most one transition from that state, then the LTS is deterministic.

**Definition 2.2.** An LTS is deterministic if for every state $s$ in $S$ the following holds: $\forall s, s_1, s_2, l_1, l_2. s \xrightarrow{l_1} s_1 \land s \xrightarrow{l_2} s_2 \Rightarrow s_1 = s_2 \land l_1 = l_2$. Otherwise, the LTS is nondeterministic.

To interpret noninterference (Definition 2.9) in Section 2.3.6, we use another definition of determinism [73], which states that for each label in $L$ there is at most one transition with that label, i.e., $\forall s, s_1, s_2. s \xrightarrow{l} s_1 \land s \xrightarrow{l} s_2 \Rightarrow s_1 = s_2$. Following this determinism, a function $\text{step} : S \times L \rightarrow S$ represents an atomic transition $s \xrightarrow{l} s'$ in a deterministic LTS.

The SPI model in Paper A is nondeterministic, reflecting the general fact that a peripheral device and its driver can perform different operations for a given state depending on the communication progress with the external environment. The formalisation of MIL in Paper D has both deterministic and nondeterministic LTSes. The in-order semantics is deterministic since states can only execute programs according to the order of instructions. On the other hand, the out-of-order semantics is nondeterministic, since OoO execution does not follow the program’s order and allows multiple instructions to be executed in a given state.

**Definition 2.3.** An execution $\rho$ of an LTS is a finite or infinite sequence of transitions starting from an initial state $s_0$, $\rho = s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \cdots \xrightarrow{l_n} s_n \cdots$. A trace $\sigma$ is the sequence of transition labels in $\rho$, $\sigma = l_1, l_2, \cdots, l_n, \cdots$. In addition, $\text{Tr}(s)$ is a set of traces for a state $s \in S$ such that $\text{Tr}(s) = \{\sigma \mid \exists s' \in S. s \xrightarrow{\sigma} s'\}$.

Extending the $\text{step}$ function, a function $\text{step}^* : S \times L^* \rightarrow S$ represents the execution of multiple labels in a trace sequentially.

The state transition system (STS) is similar to LTS except without transition labels, also called unlabelled transition system. Formally,

**Definition 2.4.** An STS is a pair $(S, \rightarrow, s_0)$ where $S$ is a set of states and $s_0$ is the initial state, and $\rightarrow$ is a transition relation such that $\rightarrow \subseteq S \times S$. There is a transition from state $s$ to $s'$ written $s \rightarrow s'$, if and only if $(s, s') \in \rightarrow$.

The meaning of (non)deterministic in Definition 2.2 also applies to STS. In Paper B, the Silver ISA and the pipelined circuit are deterministic STSes, where $s \rightarrow s'$ represents the atomic execution of one instruction in the ISA and the execution of one cycle in the circuit. For Definition 2.3, the meaning of execution applies to STS where $\rho = s_0 \rightarrow s_1 \rightarrow s_2 \cdots \rightarrow s_n \cdots$, but the trace concept does not apply to STS due to the absence of labels.

---

2A trace sometimes refers to the execution instead of labels [74], i.e., $\sigma = s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \cdots \xrightarrow{l_n} s_n \cdots$. 

---
2.3.3 Calculus of Communication Systems

Robin Milner introduces the Calculus of Communication Systems (CCS) [75–77] to describe and reason about the interactions between components in communication systems. The SPI device and driver model follow the style of CCS to exhibit the interaction between other software and SPI driver, SPI driver and device, as well as SPI device and external device.

CCS is constructed upon processes and actions. Processes represent the basic agent that may communicate with other agents in a system, and a process’s behaviour is defined in terms of actions it can perform [78]. The syntax of CCS is defined as follows:

\[ P ::= 0 \mid a.P \mid P + Q \mid P|Q \mid P[f] \mid P \setminus A \]

- 0: inactive process. The process does not perform any action.
- \( a.P \): action prefix. \( a.P \) performs an action \( a \) and continues as the process \( P \). A special action \( \tau \) is the internal operation in a process \( P \) and is invisible to other processes.
- \( P + Q \): summation. \( P + Q \) can process either as \( P \) or as \( Q \).
- \( P|Q \): composition. \( P|Q \) combines two processes \( P \) and \( Q \) that perform operations concurrently and can synchronize on matching actions (for example, \( P \) sends data and \( Q \) receives).
- \( P[f] \): relabelling. \( P[f] \) allows to rename actions according to a relabelling function \( f \) in a process \( P \).
- \( P \setminus A \): restriction. \( P \setminus A \) excludes a set of actions \( A \) in a process \( P \).

The semantics of CCS is based on the syntax and inference rules that define the operational behaviour of CCS syntax. Inference rules are of the form \( \text{hypothesis} \vdash \text{conclusion} \). The hypothesis shows the behaviour of the component process and the conclusion defines the behaviour of the composite process.

We use LTS to interpret CCS semantics where processes are states and actions are transition labels, written \( P \xrightarrow{a} P' \) (\( P \) produces \( P' \) under an action \( a \)). For the above operators (action prefix, summation, composition, relabelling and restriction), we show their corresponding inference rules.

\[
\frac{\text{Act}}{a.P \xrightarrow{a} P}
\]

Rule Act has the same meaning as the action prefix i.e., a process \( a.P \) can perform an action \( a \) and continue as the process \( P \).
Rule $Sum_1$ and $Sum_2$ show that if a process $P$ (or $Q$) produces $P'$ (or $Q'$) under an action $a$, then $P + Q$ also produces $P'$ (or $Q'$) under $a$.

Rule $Com_1$ and $Com_2$ allow the parallel composition of two processes to perform actions of these component processes. Rule $Com_3$ shows the parallel composition yields a $\tau$ action in the case that two processes execute an action $a$ and its dual action $\overline{a}$ respectively. Rule $Com_3$ enables modelling the synchronisation between different components in communication systems. For example in Paper A, we use this rule with early semantics to represent write operations between the SPI device and driver ($\text{wr ad v}$ and $\overline{\text{wr ad v}}$ as Section 2.3.2 describes).

A relabelling function $f$ renames specific actions e.g., $a$ is replaced by $b$. Rule $Rel$ states that the replacement applies everywhere in a process $P$.

Rule $Res$ specifies that a set of actions $L$ is no longer available for a given process $P$. This rule plays a crucial role in constraining interactions of CCS processes. For example, an SPI subsystem consists of the SPI hardware device running in parallel with its device driver. The internal actions between the device and driver like $wr$ and $\overline{wr}$ are assumed invisible to the external world and thus are removed in the SPI subsystem using the $Res$ rule.

### 2.3.4 Equivalence of Behaviour

Given different LTSes, a straightforward question is whether they have the same behaviour. To solve this question, a direct problem is the meaning of “the same”. Various notions of equivalence are proposed for LTSes under different requirements for “the same” behaviour. For example, trace equivalence considers two LTSes to be equivalent if they can perform the same trace (a sequence of labels as Definition 2.3).
Simulation states the equivalence as that if one LTS can perform a transition then the other LTS can also perform such a transition.

As mentioned above, trace equivalence means that two LTSes are equivalent if they can perform the same trace starting from their initial states. Formally:

**Definition 2.5.** Two LTSes \((S, \rightarrow_1, L, s_0)\) and \((T, \rightarrow_2, L, t_0)\) are trace equivalent if and only if \(\text{Tr}(s_0) = \text{Tr}(t_0)\), written \(S \approx T\).

Bisimulation [77,79] captures that two LTS systems are indistinguishable from each other regarding their observable behaviours. The key idea of bisimulation is that two states are related if the transitions can be done in one state, can be done in the other state too [80]. The definition of simulation forms the foundation of bisimulation, as the following shows.

**Definition 2.6.** Given two LTSes \((S, \rightarrow_1, L, s_0)\) and \((T, \rightarrow_2, L, t_0)\), a binary relation \(R \subseteq S \times T\) is a simulation if every \((s, t) \in R\) and \(l \in L\) satisfy that for all \(s'\) if \(s \xrightarrow{l_1} s'\), there exists \(t'\) such that \(t \xrightarrow{l_2} t'\) and \((s', t') \in R\).

The relation \(R\) is a bisimulation if both \(R\) and its inverse \(R^{-1}\) are simulations, written \(S \simeq_R T\). Here \(R^{-1} \subseteq T \times S\).

Simulation and bisimulation in Definition 2.6 are usually called strong since they require that all labels of two LTSes are equally observable. However, this requirement is not always feasible in practice. Many operations of a system are completely internal and not observable to the external environment, usually represented by \(\tau\) transition. In contrast to strong bisimulation, weak bisimulation [81] allows an arbitrary number of \(\tau\) transitions before and after observable transitions \((s \xrightarrow{\tau^*l\tau^*} s')\).

**Definition 2.7.** Given two LTSes \((S, \rightarrow_1, L, s_0)\) and \((T, \rightarrow_2, L, t_0)\), a binary relation \(R \subseteq S \times T\) is a weak simulation if for every \((s, t) \in R\):

- if \(s \xrightarrow{l_1} s'\) then \(t \xrightarrow{\tau^*l\tau^*} t'\) for some \(t'\) such that \((s', t') \in R\).
- if \(s \xrightarrow{\tau} s'\) then \(t \xrightarrow{\tau^*} t'\) for some \(t'\) such that \((s', t') \in R\).

The relation \(R\) is a weak bisimulation if both \(R\) and its inverse \(R^{-1}\) are weak simulations, written \(S \sim_R T\).
Figure 2.9 illustrates weak (bi)simulation. In Paper A, we use weak bisimulation to establish the behavioural equivalence between the abstract I/O model and the concrete SPI model. The definition of weak bisimulation in Paper A is a variant of Definition 2.7, using $t \xrightarrow{\tau}^l t'$ instead of $t \xrightarrow{\tau^l} t'$. A key property of (strong and weak) bisimulations is compositionality, as Theorem 2.1 shows. This property allows two bisimilar systems to compose with an additional system and ensures that the composed systems are also bisimilar.

**Theorem 2.1.** If $A \sim_R B$, then $A | C \sim_{R'} B | C$, where $(a | c, b | c) \in R' \iff (a, b) \in R$.

Another key property of bisimulations is transitivity in Theorem 2.2. Transitivity states that if $B$ bisimulates $A$ and $C$ bisimulates $B$, then $C$ also bisimulates $A$.

**Theorem 2.2.** If $A \sim_{R_1} B$ and $B \sim_{R_2} C$, then $A \sim_{R_1 \circ R_2} C$ where $(a, c) \in R_1 \circ R_2 \iff \exists b. (a, b) \in R_1 \land (b, c) \in R_2$.

If two LTSes are (weak or strong) bisimilar, then the two systems have the same traces, satisfying the trace equivalence in Definition 2.5.

**Theorem 2.3.** If $A \sim_R B$, then $A \approx B$.

### 2.3.5 Refinement-based Verification

Formal verification of complex systems usually contains a high-level abstract model of the system, which defines the desired behaviour of the system and serves as a framework to verify the intended properties. The target systems are defined as more detailed implementations. The connection between abstract models and implementations relies on refinement, which ensures that the refined implementation preserves the properties of the abstract model. There are two major benefits of refinement-based verification:

- Proving properties is easier on abstract models than on concrete models in general.
- An abstract model can be reused to verify different implementations via corresponding refinement relations.

Refinements between different level models are normally established in terms of different types of simulations. Theorem 2.2 allows refinements through multiple levels of abstraction and divides the verification process into several steps. For example, Kami platform [82] has verified a 4-stage pipelined processor circuit with respect to its ISA model using refinements through intermediate models such as a 3-stage pipeline.

Paper A presents a refinement-based verification for peripherals, which contains an abstract model of I/O devices and their drivers and a concrete SPI model. We
verify the refinement between these models by establishing a weak bisimulation. In order to show such weak bisimulation, we introduce an intermediate model that is similar to the top-level abstract model but contains a specific register of the SPI device used for data transitions. The intermediate model ignores most internal operations of the SPI device and driver and retains internal operations that are related to data transitions. This stepwise approach facilitates the verification to build the desired bisimulation.

To prove the functional correctness of Silver-Pi, Paper B establishes a refinement between the Silver ISA and the pipelined circuit, as shown in Figure 2.10. Since the state of the pipelined circuit contains various instructions in different stages, the circuit state at a cycle maps to multiple ISA states. Following the pipeline proof for MIPS ISA [83, 84], we introduce a scheduling function and a trace relation between the Silver ISA and the pipelined circuit. Intuitively, a scheduling function (dashed lines in Figure 2.10) maps a processing instruction in a pipeline stage at a cycle to the ISA state that handles the same instruction. Based on the scheduling function, a trace relation regulates the equivalence between the signals of the pipeline circuit and the state of ISA. Therefore, the trace relation ensures that the pipelined circuit produces the same results of instructions as the ISA defines. The refinement is similar to the trace equivalence in Definition 2.5, meaning that the circuit can execute instructions and generate correct results as the ISA defines. Compared to Definition 2.5, the difference is that we do not prove the completeness of traces, which states that if a sequence of instructions is executed at ISA level, there exists a trace of the circuit to complete these instructions in a finite number of cycles.

2.3.6 Noninterference

The notion of noninterference (NI) is commonly used to formalize the confidentiality properties of formal models. NI regulates that certain sensitive data does not interfere with or affect the behaviour of a system in ways that attackers can observe.

NI was first introduced by Goguen and Meseguer [15, 16], and later extended
CHAPTER 2. BACKGROUND

by Rushby [17]. A target system is modelled as a deterministic LTS $\langle S, \rightarrow, L, s_0 \rangle$. NI of the system is defined using security domains and a security policy. Let $D$ be a set of security domains that classify labels as for example, sensitive/high $hi$ and public/low $lo$. A function $dom : L \rightarrow D$ assigns a domain for a label. To describe the externally visible behaviour of a system, we define a set of observations $O$, and a function $obs$ that specifies the observation associated with a label on a given state, $obs : S \times L \rightarrow O$. A security policy $\Pi$ is a set of assertions that regulates permitted information flows between different domains. A binary relation $\rightsquigarrow$ on domains $D$ represents assertions. For example, $\langle lo \rightsquigarrow hi \rangle$ allows $lo$ to $hi$ and $\langle hi \nleftrightarrow lo \rangle$ forbids $hi$ to $lo$. We define a function $\text{purge} : L^* \times D \rightarrow L^*$ that takes a trace and a domain and keeps only those labels in the trace that are permitted by the security policy to transfer information to the domain. The $\text{purge}$ function is recursively defined as Definition 2.8 shows.

**Definition 2.8.** $\text{purge}(\varepsilon, d) = \varepsilon$ where $\varepsilon$ is an empty trace.

\[
\text{purge}(l \cdot \sigma, d) = \begin{cases} 
\text{purge}(\sigma, d), & \text{if } \langle \text{dom}(l) \nleftrightarrow d \rangle \in \Pi \\
 l \cdot \text{purge}(\sigma, d), & \text{otherwise}
\end{cases}
\]

Definition 2.9 defines NI.

**Definition 2.9.** A system is noninterferent with respect to a policy $\Pi$ if and only if for any label $l$ and any trace $\sigma$, the following holds:

\[
\text{obs}(\text{step}^*(s_0, \sigma), l) = \text{obs}(\text{step}^*(s_0, \text{purge}(\sigma, \text{dom}(l))), l).
\]

NI requires that removing all actions in a trace that are not allowed to interfere with security domains does not affect the final observation of executing the trace. In this way, NI ensures the absence of undesired information flow between different domains.

The observations and security policy in NI depend on the power of attackers and the features of target systems. To accommodate different scenarios, variants of NI are proposed by specifying $obs$ and $\Pi$ in Definition 2.9. For example, termination-sensitive noninterference (TSNI) [85, 86] and progress-sensitive noninterference (PSNI) [85]. In the context of program execution, a program is considered secure when its publicly observable outputs, such as the produced results, are designed to minimise any dependence on the values of its secret inputs. Let $C$ be a set of commands and $E$ be a set of environment states. An environment state $e = (el, eh)$ contains public $el$ and sensitive $eh$ data.

**Definition 2.10.** Two environment states $e_1$ and $e_2$ are low equivalent if and only if their low data is equal, $e_1 \approx_{\text{low}} e_2 \iff e_1.el = e_2.el$.

A state $s$ in LTS is in the form of a pair $(p, e)$ where $p$ is a program consisting of a sequence of commands. A transition $(p, e) \xrightarrow{\cdot} (p', e')$ represents an execution step
of the program. A state can generate a trace $\sigma(p, e)$ if $\exists(p', e'). (p, e) \xrightarrow{\sigma} (p', e')$. We use $(p, e) \uparrow$ to represent a terminate state that has no transitions, and $\sigma \uparrow (p, e)$ means that $\exists(p', e'). (p, e) \xrightarrow{\sigma} (p', e') \land (p', e') \uparrow$ where $(p', e')$ is a terminate state. We assume an attacker that knows the initial values of $el$ and can observe the $el$ of the final environment.

**Definition 2.11.** The attacker’s knowledge from observing a trace $\sigma$ of a program $p$ with an environment state $e$ is the $el$ of the final environment. Formally, $\text{obs}_T(p, e, \sigma) = e'.el$ such that $(p, e) \xrightarrow{\sigma} (p', e')$.

The security policy $\Pi$ is $\{\langle lo \sim hi\rangle, \langle hi \not\sim lo\rangle\}$ where $lo$ and $hi$ applies to $el$ and $eh$ respectively. TSNI requires that for two low equivalent environment states, if the program terminates in one trace, then the program terminates in the other trace and the final environments are also low equivalent.

**Definition 2.12.** A program $p$ is termination-sensitive noninterferent, if for any two environment states $e_1$ and $e_2$ such that $e_1 \approx_{\text{low}} e_2$, and the program terminates in a trace $\sigma_1 \uparrow (p, e_1)$, there exists a trace $\sigma_2$ such that $\sigma_2 \uparrow (p, e_2)$ and $\text{obs}_T(p, e_1, \sigma_1) = \text{obs}_T(p, e_2, \sigma_2)$.

Compared to TSNI, PSNI assumes a stronger attacker that can observe the progress (intermediate steps) of program execution. The observation function $\text{obs}$ extracts public environments during a trace $\sigma$, rather than only the final environment in Definition 2.11.

**Definition 2.13.** For a trace $\sigma$ of a program $p$ with an environment state $e$, $\text{obs}_P(p, e, \sigma) = \{e'.el \mid \exists \sigma'. (p, e) \xrightarrow{\sigma'} (p', e') \land \sigma' \subset \sigma\}$

PSNI forces the progress of a program to be independent of secret inputs.

**Definition 2.14.** A program $p$ is progress-sensitive noninterferent, if for any two environment states $e_1$ and $e_2$ such that $e_1 \approx_{\text{low}} e_2$, and the program terminates in a trace $\sigma_1 \uparrow (p, e_1)$, there exists a trace $\sigma_2$ such that $\sigma_2 \uparrow (p, e_2)$ and $\text{obs}_P(p, e_1, \sigma_1) = \text{obs}_P(p, e_2, \sigma_2)$.

Informally, a program is secure from a certain side channel if the behaviour of the program on the channel cannot be distinguished by the attacker. To model side channels, labels are used to represent the observations of a channel (e.g., a cache). The observations are determined by the power of an attacker. For example, Paper D assumes an attacker that observes the addresses of memory accesses through a cache during program execution. If the execution of a program under different secret inputs has the same observations, then the attacker cannot infer the secrets through the channel that is modelled in labels. Therefore, NI ensures that the side effects of program execution are not affected by secret data.

In Paper A, we present a notion of PSNI to regulate allowed information flows in the SPI model, which is equivalent to Definition 2.14, or in terms of termination
only, to Definition 2.12. PSNI ensures that an attacker cannot learn secrets by observing channels included in the model during the execution of the SPI device and its driver.

**Conditional Noninterference**

The above NI and its variants compare what is leaked by a program with a security policy specifying the allowed and forbidden information flows in an LTS. In practice, some scenarios require checking the leakage of the same program under different models (LTSes). For example, identifying Spectre-OoO in Figure 2.6 demands the comparison of program leakage under out-of-order and in-order execution. To ensure the information flow security on a target system, one approach is adding declassification [87] for the target system. Declassification specifies allowed information leakage in the target system. Then, one can prove the NI of a program on the target system to prevent additional leakage than allowed by the declassification. However, declassification is laborious and error-prone since public information varies greatly for different programs. For example, b1 and b2 in the Spectre-OoO program are public in OoO execution but another program may have an entirely different setting. Another approach is conditional noninterference (CNI) [58, 88].

Based on a reference system for allowed information leakage, CNI prevents additional information leakage in a target system with respect to the reference system. CNI only captures information leaks introduced by the target system and ignores existing leaks in the reference system. This means that there is no need to declassify the public information of a program in the target system.

A system has a set of states \( S \) and a set of labels \( L \) where labels are observations during transitions. Two LTSes \((S, \rightarrow_r, L, s_0)\) and \((S, \rightarrow_t, L, s_0)\) represent the system under a reference model (e.g., in-order execution) and a target model (e.g., OoO execution) respectively. As Definition 2.3 shows, \( \rho \) represents an execution and \( \sigma \) is the corresponding trace of \( \rho \). The security policy specifies the public and sensitive information of a state \( s \). The policy is defined as a binary relation \( \sim \subseteq S \times S \). The relation \( \sim \) requires that two states have the same public information and therefore cannot be distinguished by the attacker prior to a transition. To ensure the security of \( \rightarrow_t \), a security condition requires that transitions starting from indistinguishable states on the target model \( \rightarrow_t \) do not leak more information than on the reference model \( \rightarrow_r \).

**Definition 2.15.** A system is conditional noninterferent if for all \( s_1, s_2 \in S \) such that \( s_1 \sim s_2 \), if for every \( \rho_1 = s_1 \rightarrow_r \cdots \) there exists \( \rho_2 = s_2 \rightarrow_r \cdots \) such that the traces of the executions are equal (\( \sigma_1 = \sigma_2 \)), then for every \( \rho_1 = s_1 \rightarrow_t \cdots \) there exists \( \rho_2 = s_2 \rightarrow_t \cdots \) such that \( \sigma_1 = \sigma_2 \).

CNI regulates that executing the same program on a target model does not leak more information than permitted on the reference model. In Paper D, we use the

\[3\] The CNI introduced here is different from standard ones [15], which refers to NI with dynamic security policies where assertions can change depending on the state of components of a system.
setting of observations as shown in Section 2.2.2, including \texttt{il a}, \texttt{dl a} and \texttt{ds a}. These observations overapproximate what an attacker can observe through a cache, and allow us to reason about the leakage through cache-based side channels transparently without an explicit cache model. Paper D considers in-order execution as the reference and out-of-order execution as the target. CNI in Paper D rules out trace-driven cache side channels of programs’ out-of-order execution compared to in-order execution. For example, the Spectre-OoO program in Figure 2.6 violates the CNI since the trace of in-order execution is always \texttt{dl b1::ds b2} but the trace of OoO execution depends on the value of \texttt{z} (i.e., \texttt{ds b2::dl b1} iff \texttt{z} is false).

Paper B uses CNI to prevent timing side channels of the pipelined processor Silver-Pi with respect to the Silver ISA. Paper B defines an observation function on the Silver ISA model that extracts the part of the ISA state that can affect the execution time of a program. The ISA model serves as the reference and the circuit model is the target. The CNI formulates that a processor should not leak more information via its timing channel than what is expected by the observation function expressed at ISA level. In Paper B, we demonstrate the adaptability of using the CNI to accommodate various processor designs, attacker models, and environments. Paper C presents the explicit nonmechanised proof of CNI for Silver-Pi.
Chapter 3

Related Work

This chapter presents related work, including verified I/O devices and drivers in Section 3.1, ISA formalisation, verified processors and stacks in Section 3.2, and information flow security of low-level programs in Section 3.3.

3.1 Peripheral and Driver Verification

Many studies [89–93] verify the functional correctness of device drivers for specific serial devices. Alkassar et al. [89] present a formal model of the serial interface controller UART (universal asynchronous receiver-transmitter) 16550A and a formal model of the DLX ISA in the Isabelle/HOL theorem prover. An assembler-level programming model is a combination of the two models where the UART model is a memory-mapped device of the ISA model and the two models can interact via interrupts and register polling. They implement a UART driver in assembly and verify the driver’s correctness. The correctness ensures that the driver completes its execution in a number of steps, and a certain number of words are transmitted from the processor to the external environment via the UART. Following this approach, Alkassar et al. [90] prove the functional correctness of an assembly driver for an ATAPI (advanced technology attachment packet interface) hard disk in Isabelle/HOL.

Penninckx et al. [93] verify the Linux USB (universal serial bus) BP (boot protocol) keyboard driver with properties including freedom of data races in the presence of concurrent callbacks, freedom of illegal memory accesses, and correct API (application programming interface) usage.

Similar to Paper A, Duan et al. [91] present an abstract device model supporting register polling in HOL4. The device model is plugged into a formal model of the ARMv4 ISA [94]. They instantiate the abstract model with a UART interface on the NXP LPC2129 chip and ensure the correctness of the transmit and receive functions from a driver for this UART device. Duan [92] extends the abstract model and the UART model to support interrupts under the ARMv7 ISA model [95] in
HOL4, and ensures the functional correctness of an interrupt-driven driver for the UART device. The major limitation of Duan’s work [91,92] is that the device state is merged into the ISA state, which requires careful handling of the interactions between the execution of the device and the CPU. This causes great complexity in modelling a specific device with respect to Duan’s abstract model. Paper A builds device and driver models independent of the ISA models.

Chen et al. [96] propose a framework in Coq for building verified interruptible OS kernels with device drivers. The framework includes a general device model and a formal model of interrupts, and the two models can be used to reason about interrupt-driven drivers. The general device model is a finite state transition system. They formalise several peripherals including a UART by instantiating the device state and transitions from the general model. Based on the framework, they extend the verified noninterruptible OS kernel CertiKOS [97] to support interrupts with verified device drivers. Their general device model has the following limitations. First, the model only considers device operations that the CPU can observe and ignores operations that are observable to the external environment. Second, the model does not cover full-duplex mode for data transmission, which means one cannot use it to verify an SPI device. In contrast, our abstract model in Paper A describes the device-to-device interface and full-duplex transmission.

The above verifications [89–93,96] focus on drivers that operate devices through interrupts and register polling. Some other studies [98–100] verify DMA-enabled I/O devices and drivers. Monniaux [98] verifies a device driver for a USB OHCI (open host controller interface) controller with an embedded DMA controller using the static C program analyzer Astrée. Haglund et al. [99] formalise a DMA-enabled NIC (network interface controller) in HOL4 and prove the isolation of the NIC model, which states that the device only accesses allowed memory regions and cannot access sensitive memory regions. Later, Haglund et al. [100] present a general formal framework in HOL4 for modelling DMA controllers and verifying their isolation under specific conditions. These conditions can be used as specifications that guarantee device drivers configure DMA controllers correctly. They verify a DMA controller of USB using the framework.

### 3.2 Processor Verification

This section presents existing works that have formally verified the functional correctness of processors. First, we look at ISA formalisation that constitutes the foundation for processor verification. Then, we review the rich history of verified processors especially pipelined. Finally, we introduce verified stacks including verified compilers and processors.

**ISA Formalisation**

Since ISAs define the functional behaviour of processors, ISA formalisation provides foundations for verifying processors. Various ISAs have been formalised in different
ITPs [94, 95, 101–103]. Anthony Fox [94] proposes a formal model of the ARMv4 ISA in HOL4, and Fox et al. [95] present a model of the ARMv7 ISA. Joloboff et al. [101] verify an instruction set simulator for the ARMv6 ISA in Coq. Goel et al. [102] define a model for the IA-32e mode of the x86 architecture in the ACL2 theorem prover. Bourgeat et al. [103] provide a formalisation of the RISC-V ISA written in Haskell, and the formalisation is then translated to Coq.

The above ISA models in various ITPs [94, 95, 101–103] rely on the features of underlying ITPs and differ in their presentation style, abstraction level and complexity. These differences make these models hard to be reused for different verifications. To solve this problem, various domain-specific languages (DSL) have been proposed and utilized [104–107]. Ulan Degenbaev [104] presents a DSL and a model of the x86 ISA using the DSL. This DSL lacks tools for translating specifications written in the DSL to ITPs. Anthony Fox [105,106] develops a DSL called L3, which has been used to formalise ISAs such as ARMv8, RISC-V and Silver. L3 can export ISA specifications to HOL4 and Isabelle/HOL. Similar to L3, Sail [107] is a DSL describing the sequential behaviour of ISAs such as ARMv8-A, RISC-V, and CHERI-MIPS. The RISC-V model in Sail is integrated with a weak memory model to reason about operations like load and store. Sail supports Coq, HOL4 and Isabelle/HOL. Using DSLs like L3 and Sail allows automatic translation of ISA models to different ITPs. This fact facilitates the work of ISA specification since developers do not need knowledge of concrete ITPs. The corresponding ISA models in ITPs can be used for different purposes, for example, reasoning about processors, compilers and operating systems [7,108,109]. In Paper B, we use the Silver ISA model, originally defined in L3 and translated into HOL4, to verify the functional correctness of a pipelined processor.

Processor Verification

The functional correctness of processors is the basic requirement for computer systems. To ensure the correctness, formal verification of processors has been studied extensively in both academia and industry.

Many of early efforts [110–114] focus on non-pipelined processors. Since non-pipelined processors are similar to ISA models that handle one instruction at a time, the correctness is typically established using variants of simulation. For instance, Hunt [110] prove the correctness of the non-pipelined FM8502 processor in the Nqthm theorem prover. The correctness states that the processor completes an ISA-level instruction within a finite number of clock cycles. Later, Hunt et al. [111] formalise a custom HDL DUAL-EVAL in Nqthm and verify a non-pipelined processor FM9001 written in DUAL-EVAL. Arora et al. [112] present the formal verification of a non-pipelined processor Viper demonstrating that a top-level ISA specification is satisfied by a low-level implementation represented as an abstraction of a conventional electronic block model.

The correctness of pipelined processors is more complicated than non-pipelined as Section 2.2.1 discusses. Many studies [115–119] verify simplified and abstract
models of pipelined processors. For example, Srivas et al. [115] verify a pipelined machine including 3 stages that are fetch, decode and execute, and write back respectively. The machine has a simple forward unit to address data hazards, transmitting the result from the write back stage to the decode and execute stage. The forward design is not applicable to more complex pipelines (e.g., 5-stage) because a previous instruction in the pipeline may not have a valid result that can be forwarded. Sawada et al. [116] present a framework for verifying pipelined microprocessors in the ACL2 theorem prover. The framework describes the behaviour of pipelined processors using table-based models. The table-based model abstractly records the execution trace of an instruction within a pipeline in a tabular form. They prove the correctness of the table-based model with respect to an ISA model. Since the table-based model only captures behaviour at the microarchitecture level, the verification does not guarantee the correctness of an implementation.

Enterprises have produced various formal tools to verify processors. ARM presents a framework ISA-Formal [120] to verify that processors correctly implement the specification of ISA. ISA-Formal uses bounded model checking to explore different sequences of instructions and allows engineers to detect bugs in the datapath, pipeline control and forward/stall logic of processors. To demonstrate the usage of ISA-Formal, they have applied it to 8 different ARM processors and successfully found bugs in all processors.

Intel has used symbolic simulation to formally verify their processors [121–124]. Since x86 is a CISC ISA but Intel’s microarchitecture is RISC-like, ISA-level instructions are translated into a smaller set of simpler instructions called microinstructions. Most ISA-level instructions correspond to a single microinstruction. For some more complicated instructions, microcode defines a sequence of microinstructions to achieve the functionality of these instructions [124]. Microcode is generally stored in an on-chip ROM [36]. Intel proposes an SMT/SAT-based tool MicroFormal [122] to check the functional backward compatibility of microcode. Since new functionalities of CPUs are often implemented in microcode, the compatibility of microcode ensures that new generations of microcode are backwards compatible with older generations. MicroFormal does not consider the correctness of processors at the circuit level. Kaivola et al. [123] present the verified core execution cluster of the Intel Core i7 processor using symbolic simulation. The execution cluster is responsible for the functional behaviour of all microinstructions.

Centaur proposes a framework [36] for verifying x86 processors. The framework includes a symbolic simulator GL [125, 126], ACL2, and SAT solvers. The framework translates RTL implementations written in SystemVerilog into a formal model in ACL2, and then generates function definitions corresponding to the design blocks in the formal model. GL reduces finite ACL2 theorems into propositional logic formulas using symbolic simulation for these function definitions. The theorems state the correctness of implementations based on an x86 ISA model. These formulas are either proved directly with binary decision diagrams (BDD), or simplified using and-inverter graph (AIG) algorithms and transmitted to SAT solvers to check their validity. The BDD and AIG algorithms are written in ACL2 and
proven correct. The verification of the framework relies on an unverified tool to translate implementations from SystemVerilog to ACL2.

Many of the above publications \[36, 110, 111, 113, 115, 116\] use ITPs to model the behaviour of a processor and verify its correctness with respect to the ISA. To accurately capture the behaviour of a processor at the circuit level, Lööw et al. \[7\] implement a non-pipelined processor for the Silver ISA (Section 2.1.5) using the HOL4 Verilog library (Section 2.3.1). The verified Silver-Pi in Paper B can be considered an extension of this work in many ways, targeting the same ISA and using the same Verilog library. The major difference is the pipeline design and implementation of Silver-Pi, while the non-pipelined Silver processor is similar to the sequential model of the ISA.

There are few verified pipelined processors \[20, 82, 127\] at the circuit level. We introduce the details of these verified processors and discuss the differences between their work and Paper B.

Beyer et al. \[20\] verify a pipelined processor VAMP for the DLX ISA in the PVS theorem proving system, and Tverdyshev et al. \[128\] port the VAMP processor to Isabelle/HOL. Similar to our work, the VAMP processor is translated to Verilog and then synthesised for FPGAs. However, the correctness of VAMP is not down to the Verilog implementation since they use unverified tools for the translation from models in ITPs to Verilog. In the PVS work, they use a tool called pvs2hdl \[129\] to generate the Verilog implementation for a gate-level PVS circuit definition. Tverdyshev et al. \[128\] develop a tool IHaVelt to generate a Verilog implementation from Isabelle/HOL. Differently, we verify the correctness of Silver-Pi down to its Verilog implementation following the workflow in Figure 2.8.

To ensure the correctness of a pipeline, our verification has its root in the proof for MIPS \[127, 130\] using a trace relation and scheduling function. Kovalev et al. \[127\] implement and verify a pipelined processor for the MIPS ISA, and Lutsyk et al. \[130\] upgrade the processor with operating system support. However, their proofs are not mechanized, while our proof is machine-checked in HOL4. More importantly, we have updated the proof methodology to address the challenges of non-MIPS ISAs. For example, their pipeline can determine the next PC in the ID stage and thus their proof ignores the misprediction of PC, because of the MIPS ISA’s delay slot. The delay slot is an instruction slot that allows the execution of instructions without the effects of a previous branch instruction like a jump. However, the delay slot is uncommon for other RISC ISAs such as RISC-V \[33\] and Armv8-A \[131\]. To address the common challenge of mispredicted PC, we refine the scheduling function to be partial, i.e., the scheduling function is undefined ∑ for wrongly fetched instructions.

Kami \[82\] is a hardware development platform in Coq to implement and verify circuits targeting the Bluespec HDL. Bluespec is a higher-level HDL compared to Verilog, which means the Bluespec code generated by Kami requires further translation into Verilog. To demonstrate the usage of Kami, they develop and verify an in-order pipelined processor targeting a subset of the RISC-V ISA. Erbsen et al. \[8\] extend this processor to support more RISC-V RV32I instructions like
bytewise store and load. Their pipeline design has obvious differences from Silver-Pi:

1. They implement a 4-stage pipeline merging the MEM and WB stage together, and Silver-Pi is a 5-stage pipeline as shown in Figure 2.1.

2. Their pipeline determines the next PC in the last stage, while our pipeline decides in the EX stage.

Furthermore, their verification is hard to be reused for other ISAs and pipelined processors. The correctness of the Kami processor is proved by refinement via intermediate modules that are tied to their specific ISA model. The ISA model uses an immutable instruction memory that remains unaffected by memory operations. As a result, the processor fetches the old instructions even when dealing with self-modifying programs.

**Verified Stacks**

Verified stacks ensure the functional correctness of computer systems with comprehensive proof across the software-hardware boundary. Most verified stacks [7,8,132,133] include a verified compiler and a verified processor, and present end-to-end verification for running verified software on verified processors.

An early attempt is the CLI stack [132] using the Nqthm prover. The CLI stack connects a verified compiler for a Pascal-like language to the non-pipelined FM8502 processor. The stack establishes the correctness of several software program examples (e.g., multiplying two integers) down to the processor model. Moore [134] builds the CLI stack for the FM9001 processor.

The Verisoft stack [133] presents a verified compiler in Isabelle/HOL for the C-like programming language C0 [135] targeting the verified VAMP processor. The verification covers different layers of the system using simulation, from the application layer over the system-level software like a microkernel and a compiler, down to the gate-level VAMP processor.

CakeML [136] is a mechanically verified ML system supporting a substantial subset of SML. CakeML includes a verified compiler that can produce a verified machine-code implementation of an input program for multiple ISAs, e.g., x86, Armv8 and Silver. Based on the verified CakeML compiler for the Silver ISA, Lööw et al. [7] present a verified stack on top of the non-pipelined Silver processor. Following this work, Silver-Pi enables end-to-end verification of running verified programs on a verified pipelined processor. For more details, see Section 5.2.

Erbsen et al. [8] introduce, according to the authors, the first verification of a realistic embedded system in Coq, including software applications, device drivers, a compiler, and a RISC-V processor (based on the above Kami processor). They ensure the functional correctness of the embedded system represented as a top-level theorem, which states that running an application on their processor only produces results (i.e., I/O traces) allowed by the application specification.
To my understanding, these verified stacks are still not trustworthy. The major problem is that these stacks focus only on functional correctness and ignore security properties like confidentiality completely. For a correct compiler, it is also crucial to prevent undesired secret leaks during compilation. For example, Besson et al. [137] propose a notion of information flow preservation (IFP) and use the notion to validate the CompCert C compiler for detecting secret leaks introduced by the compiler. The IFP ensures that a compiled target program is no more vulnerable to side-channel attacks than a source program. The above compilers in verified stacks [7,8,132,133] lack such information flow analysis. Correct processors should not introduce side channels due to the internal design. We discuss the information flow analysis with respect to processors in Section 3.3. In addition, the correctness of other necessary hardware components is omitted in these stacks [7, 8, 132, 133]. For example, the memory subsystem (e.g., a cache plus a main memory) is not verified [7, 8], or peripherals are dismissed [7].

3.3 Low-Level Information Flow Security

The discovery of Spectre [13] and Meltdown [14] is a strong motivation for security analysis of programs with respect to processor features like speculative and OoO execution. Paper B, C and D study the information flow security of processors on the circuit (represented in HDLs) and microarchitectural level. This section presents related work preventing information leakage caused by processor features.

To detect speculative leaks, various unformalised tools [138–140] have been proposed. For example, Yan et al. [138] present InvisiSpec that makes speculation invisible in the data cache hierarchy to defend against Spectre. InvisiSpec uses a speculative buffer to temporarily hold the read data of loads during speculative execution, rather than directly loading the data to the cache. After the loads become safe to be committed, the buffer will make the data visible to the cache. Wang et al. [139] propose a binary analysis framework oo7 that can detect and patch potentially vulnerable code snippets for Spectre in programs based on specific syntactic code patterns. However, these tools may have internal bugs and problems. For instance, as Guarnieri et al. [88] point out, oo7 misses some speculative leaks and incorrectly classifies programs’ security.

Formal approaches [57, 58, 88, 141–147] have been proposed to prevent information leakage caused by speculative and OoO execution. Since microarchitecture provides pipeline features including speculative and OoO execution, some works [58, 88, 142–144] are based on the microarchitectural level like our Paper D.

Guanciale et al. [58] present the machine independent language (MIL), which abstractly describes microarchitecture features including in-order, out-of-order and speculative execution and enables analysis of information flows of programs under these features. They present several program examples in MIL capturing Spectre vulnerabilities and discover Spectre-OoO (Figure 2.6) caused by OoO execution.
Paper D is the formalisation of MIL in the HOL4 theorem prover focusing on the in-order and out-of-order semantics, and the formalisation of speculative semantics is left out as future work.

Guarnieri et al. [88] propose a security notion, called speculative noninterference (SNI), against speculative execution attacks at the microarchitectural level. SNI requires that executing a program under a speculative semantics does not leak more information than executing the same program under a non-speculative semantics. They present a tool Spectector to check if programs satisfy SNI using symbolic execution.

Guarnieri et al. [144] present a framework for specifying hardware-software contracts for secure speculation. The framework includes an ISA language, a model of the microarchitecture including speculation, and an adversary model specifying the microarchitectural components that are observable by the attacker via side channels. A processor satisfies a contract if, whenever two program executions have the same observations at ISA level, the executions are indistinguishable by the adversary at the microarchitectural level. Similar to the adversary model, we use an observation function in Paper B to extract the parts of ISA that can affect the execution time of programs. Paper D also has an adversary model at the microarchitectural level where the reference is in-order execution and the target is out-of-order execution, instead of ISA and microarchitecture in [144].

Cheang et al. [142] introduce a notion of information flow security property called trace property-dependent observational determinism (TPOD). TPOD can capture secret leaks caused by the interaction of microarchitectural side channels with speculative execution. Similar to Paper D, Griffin et al. [143] formalise a framework for verifying TPOD in Isabelle/HOL using the operational semantics of Cheang et al. The major difference from Paper D is that their framework [143] did not consider OoO execution.

Godbole et al. [147] propose micro-update models capturing microarchitectural features and develop a semiautomated algorithm to generate micro-update models from register transfer level (RTL) processor designs. The micro-update models can be used for information flow analysis of programs, for example, establishing noninterference. They use a model checker to ensure the equivalence between a micro-update model and an RTL design. Our MIL semantics only preserves vulnerability-related microarchitectural features like OoO execution, while micro-update models include hardware components based on signals-of-interest to more accurately capture the behaviour of RTL designs. However, as mentioned in their paper, identifying signals-of-interest requires a significant amount of manual effort including consideration of RTL designs and ISAs. The abstract (compared to their work) MIL semantics is independent of ISAs and RTL designs, and the formalisation in Paper D has developed tools to transfer binary programs of different ISAs to MIL.

Microarchitecture does not capture all aspects of processors, e.g., the absence of accurate time information. Processors written in HDLs provide a timing channel that can be used to detect secret leakages through timing side channels caused
by processor designs. In Paper B, we propose a notion of conditional noninter-
ference, which formulates that executing a program on the pipelined processor
does not leak more information on its timing channel than permitted at ISA level.
Paper C presents the explicit nonmechanised CNI proof of Silver-Pi. Some stud-
ies [57, 141, 146] present formal approaches to capture vulnerabilities caused by the
implementations of processors.

Fadiheh et al. [57] propose an SAT-based model checking method called unique
program execution checking (UPEC), which detects processors’ vulnerabilities at
the register transfer level. They discover the Orc attack as shown in Figure 2.7,
which demonstrates that in-order pipelined processors can leak secrets due to their
design. In Paper B, we also show a correct but insecure in-order pipelined processor
because of its data hazards handling. Fadiheh et al. [141] extend UPEC for checking
OoO processors. Compared to our work, UPEC has a fixed security property
which makes it hard to adjust different ISA-level leakage models. In addition,
counterexamples generated by UPEC are not always caused by hardware design
(e.g., because of unreachable states), while the violations in our CNI proof are
directly related to the circuit implementation.

Wang et al. [146] introduce a verification tool LEAVE for checking processor im-
plementations against ISA-level leakage contracts via an SMT solver. To simplify
the security verification, LEAVE utilises a decoupling theorem that separates se-
curity and functional correctness requirements for contract satisfaction and ignores
the functional correctness in their verification. They only demonstrate the usage
of LEAVE with simple 2/3-stage RISC-V processors such as DarkRISCV [148] and
Ibex [149]. For such simple processors, functional correctness is not necessary for
security proof since the progress of instructions in pipelines is obvious. However, for
more complex pipelines like 5-stage, it is difficult to reason about security properties
without functional correctness, which formulates how instructions are processed in
a pipeline. Paper B presents a verification approach for processors including both
functional correctness and information flow security, and as our CNI proof shows,
the functional correctness proof can largely facilitate the security proof.

In our work (Paper B, C and D), conditional noninterference has been used to
formulate that executing a program on a target system does not leak more informa-
tion than executing the same program on a reference system. Our CNI represents
a variant of other similar NI definitions used in similar work e.g., SNI [88]. Cauligi
et al. [145] summarise related notions of NI to detect secret leakage caused by
speculative execution. In Paper B, because the Silver ISA and circuit design are
deterministic, the definition of CNI is a derivative of ignorance-preserving refine-
ment (IPR) [150]. Ignorance captures an observer’s inability to identify two related
traces. The IPR notion means that refinement steps must preserve the ignorance
in an abstract model.
Chapter 4

Summary of Publications

The goal of this thesis is to improve the trustworthiness of critical components of computer systems. This is achieved by formally verifying the functional correctness and information flow security of peripherals and processors using the HOL4 theorem prover. This thesis includes four publications as listed in Section 1.3. This chapter presents a summary of each publication with a statement of the author’s contribution.

Paper A verifies an SPI device and its driver with respect to an abstract model using a refinement-based approach. The refinement is established by a weak bisimulation, which can transfer functional and information flow properties of the abstract model to the SPI model. Paper B presents a verified in-order pipelined processor Silver-Pi for the Silver ISA. We define a notion of CNI to prevent secret leakage on the processor circuit’s timing channel. The verification strategy of CNI complements the verification of functional correctness. Paper C introduces the concrete nonmechanized proof of CNI for Silver-Pi. Paper D formalises the machine independent language (MIL) in HOL4 including microarchitectural in-order and OoO execution and develops verified tools embedded in HOL4 for executing MIL programs. In the formalisation of MIL, we propose a notion of CNI, which rules out trace-driven cache side channels of programs’ OoO execution compared to in-order execution.

4.1 Paper A: Refinement-Based Verification of Device-to-Device Information Flow

Content

This paper describes a refinement-based verification approach for I/O devices and their drivers. We build an abstract model of I/O devices and their drivers to describe the expected results of I/O subsystem execution, including initialization, full-duplex transfer, transmission and reception. The refinement-based verification concretises this correct-by-construction abstract model into an actual device and its driver. Specifically, we formalise an SPI device used in the AM335x family of SoCs [1] with a register polling driver. We prove the refinement between the abstract model and the SPI model using weak bisimulation. All models and weak bisimulation proofs have been formalized in the HOL4 theorem prover. Bisimilarity allows to transfer both functional correctness and information flow security (in terms of PSNI as Definition 2.14 shows) from the abstract model to the concrete one. The weak bisimilarity prevents malicious driver operations e.g., an infinite loop \( \text{while } (s) \{ \} \) before a data transmission where \( s \) is a secret, and allows to compose the SPI subsystem with nondeterministic components safely. For example, the driver is not allowed to perform \( \text{if } s \text{ then } v \text{ else } v' \) where \( v \) and \( v' \) are nondeterministically introduced from a communication medium between two SPI subsystems. The abstract model can be reused for verifying the functional behaviour of other SPI devices, and other peripherals after disabling the full-duplex transfer. The weak bisimulation guarantees that two systems have the same information flows up to channels that are not modelled here (e.g., the absence of a timing channel). This means that the information flow analysis of this paper does not deal with side channels like timing.

Based on the verified SPI subsystem, we develop a random number generator (RNG). We use a BeagleBone Black to connect with an Arducam Shield Mini 2MP Plus camera through the SPI protocol. The RNG captures images of the floating material in a lava lamp and uses the images as a source of physical randomness to produce numbers.

Contribution

I am the main contributor including defining formal models, writing proofs, and implementing and debugging the RNG application. Lots of inputs have been received from Roberto and Mads, for example, concepts like CCS in the formalisation. The writing of the paper is done together with Roberto and Mads.

4.2 Paper B: Formal Verification of Correctness and Information Flow Security for an In-Order Pipelined Processor

4.3. PAPER C: INFORMATION FLOW ANALYSIS OF A VERIFIED IN-ORDER PIPELINED PROCESSOR

Content

This paper presents a verified in-order pipelined processor Silver-Pi in the HOL4 theorem prover. Silver-Pi implements the RISC ISA Silver and features a 5-stage pipeline as shown in Figure 2.1. For formally verified hardware, Silver-Pi is constructed using the HOL4 Verilog library [27, 28]. The correctness of Silver-Pi is established using a refinement relation between the traces of the processor and the Silver ISA. As described in Section 2.3.1, the HOL4 Verilog library lifts the correctness of the processor to its Verilog implementation. The correctness proof is mechanized in HOL4.

We analyze the information flow properties of the processor by utilizing the refinement relation. The notion of conditional noninterference formulates that a processor should not leak more information via its timing channel than what is expected by a leakage model expressed at ISA level. We provide the nonmechanized proof of CNI for Silver-Pi and demonstrate the adaptability of the information flow methodology to accommodate various processor designs, attacker models, and environments. This paper presents the CNI proof in an abstract way due to limited space.

Silver-Pi has been successfully synthesized as a small computer system (in combination with other components such as a cache and interrupt handler) for the PYNQ-Z1 FPGA board, by using the Xilinx Vivado toolchain. We evaluate its performance with several software programs and the benchmark results show that Silver-Pi executes programs faster than the previous non-pipelined Silver processor [7].

Contribution

I am the main contributor to the implementation, formalisation, information flow analysis and evaluation of the processor. Roberto and Mads provide technical guidance for the proofs of the processor. Andreas Lööw has updated the HOL4 Verilog library to enable the development of the processor using the library. All authors have participated in the writing of this paper.

4.3 Paper C: Information Flow Analysis of a Verified In-Order Pipelined Processor


Content

This technical report is an extension of Paper B with a focus on CNI and proofs for Silver-Pi. The report introduces the correctness theorem of Silver-Pi with respect to the Silver ISA and explicitly shows related definitions e.g., an assumption mem_env
that describes the behaviour of a memory subsystem and an observation function \( \text{obs}_{ag} \) that extracts the part of the Silver ISA state that can affect the execution time of a program. According to these definitions and the correctness theorem, we prove the CNI of Silver-Pi via lemmas and subtheorems. We show concrete proofs for each lemma and subtheorem in this report. Combining these proofs, we demonstrate the nonmechanised CNI proof for Silver-Pi.

**Contribution**

I am the main contributor to the proof and the writing of this report. Co-authors have provided insights during the proof and refined this report.

### 4.4 Paper D: Foundations and Tools in HOL4 for Analysis of Microarchitectural Out-of-Order Execution

*Karl Palmskog, Xiaomo Yao, Ning Dong, Roberto Guanciale, and Mads Dam.*


**Content**

This paper presents the formalisation of MIL in the HOL4 theorem prover, encoding MIL’s in-order and out-of-order semantics. We formalise two key aspects of the metatheory of MIL:

- proof of memory consistency between in-order and out-of-order execution of microinstructions;

- a notion of conditional noninterference capturing trace-driven cache-based side channels.

To improve the automation of CNI proofs, we develop and verify an executable semantics in HOL4. The executable semantics computes results for the execution of MIL programs. Since the original MIL semantics [58] is defined using sets (e.g., sets of instructions), formalizing sets as HOL4 predicate sets is convenient for abstractly defining MIL and developing its metatheory. However, this encoding prevents many definitions from being computable. Instead, MIL executable semantics is list-based, and we prove the correctness of the executable semantics, which states that the list-based functions preserve the behaviour of set-based definitions.

Using the executable semantics, we propose a semi-automated bisimulation based strategy to verify CNI. We use the verification strategy for proving several MIL programs (e.g., Spectre-OoO in Figure 2.6) and show that out-of-order execution can introduce side channels by exhibiting a violation of CNI.
The speculative semantics [58] has not been formalised in this work and thus we do not address Spectre-like vulnerabilities in this paper. Karl Palmskog and Roberto Guanciale have been working on the formalisation of speculative semantics.

**Contribution**

This paper is a result of teamwork. I am the main contributor to the executable semantics and its correctness proof under the supervision and refinement of Karl Palmskog and Roberto Guanciale. I also have contributions for the CNI proof using the executable semantics to generate results of programs’ in-order execution. Karl Palmskog is the main contributor to the whole formalisation, including defining basic semantics and developing metatheory, refining and examining others’ work, etc. Xiaomo Yao is the main contributor to the CNI proof of several MIL programs. Roberto Guanciale and Mads Dam are two of the three authors of the initial MIL paper [58]. They provide supervision and technical suggestions for the formalisation. All authors have participated in the writing of this paper and the load is shared according to the workload in the formalisation.
Chapter 5

Conclusions and Future Work

This chapter presents the conclusions of this thesis (Section 5.1), discusses the limitations of the results and proposes future work (Section 5.2).

5.1 Conclusions

This thesis contributes to the construction of a trustworthy computer system by answering the following questions:

1. How to formally verify that the functional behaviour of a target system satisfies the specifications defined by a reference system?

2. How to ensure the information flow security of a target system given information flows in a reference system?

These two questions are difficult to solve if the target system is an entire computer involving software and hardware. Instead, we focus on two critical hardware components in computer systems, peripherals and processors.

To address Question 1, Paper A and B present refinement-based verification for peripherals and pipelined processors respectively. Paper A proposes an abstract model to describe the expected results of four functionalities of I/O devices and their drivers, including initialization, full-duplex transfer, transmission and reception. As an example, Paper A formalises an SPI device and its register polling driver. The refinement between the abstract and SPI model is established by a weak bisimulation. The weak bisimulation ensures the correctness of the four functionalities of the SPI model. Paper B proves the correctness of an in-order pipelined processor Silver-Pi with respect to the Silver ISA. Since the ISA model handles one instruction at a time and the pipeline handles multiple instructions concurrently, a scheduling function maps the state of a pipeline stage to an ISA state that processes the same instruction. Based on the scheduling function, we define a trace relation between the pipelined circuit and the Silver ISA. The trace relation ensures that fields in the circuit have the same results as the ISA prescribes.
In Paper A, we address Question 2 regarding the SPI device and its driver. The weak bisimulation guarantees that the abstract and SPI models have the same information flows with the ignorance of the device and driver’s internal operations. Based on the weak bisimulation, we show that the abstract and concrete models are PSNI. PSNI prevents malicious operations of the SPI driver, for example, an infinite loop depending on a secret before data transmission.

The discovery of Spectre, Meltdown and Orc vulnerabilities demonstrates that hardware features like instruction pipelining and out-of-order execution may enable side channel attacks. ISA mainly captures the functional behaviour of processors and ignores these features that processors employ to optimise performance. This means that Question 2 cannot ensure the security of processors even if a processor has the same information flow as the ISA. To guarantee the information flow security of pipelined processors, we focus on the following questions.

3. How to guarantee the information flow security on the timing channel of a pipelined processor circuit?

4. How to identify undesired information flows of programs caused by out-of-order execution?

We answer Question 3 in Paper B and C. An observation function extracts the parts of ISA states that can affect the execution time of programs. A processor written in the Verilog HDL describes the cycle-by-cycle behaviour of the circuit, providing a timing channel of the processor. We propose a notion of conditional noninterference, which formulates that executing programs on a processor should not leak more information through the timing channel than permitted by the observation function at ISA level. Paper B presents a verified in-order pipelined processor Silver-Pi. We abstractly describe the CNI proof of Silver-Pi, ensuring the absence of timing side channels on the circuit. In addition, we show a modified version of Silver-Pi that violates the CNI and can leak secrets through a timing side channel. Paper C explicitly presents the CNI proof of Silver-Pi.

Question 4 focuses on information leakage caused by out-of-order execution. Paper D presents the formalisation of MIL that abstractly describes microarchitectural in-order and out-of-order execution. An attacker observes the addresses of memory accesses. These observations overapproximate what an attacker can observe via cache and allow us to reason about information leakage through cache-based side channels without an explicit cache model. CNI in MIL requires that an OoO execution of a program does not leak more information through observations than the in-order execution of the program. MIL has an embedded executable semantics that compute results and generate observations of programs’ execution. We propose a semi-automated verification strategy to prove CNI for programs. The executable semantics generates the observations of programs’ in-order execution in the CNI proof. We apply this strategy to verify several programs like Spectre-OoO in MIL.
5.2 Future Work

The results of this thesis improve the trustworthiness of computer systems with a focus on peripherals and pipelined processors. Our ultimate goal is to build a trustworthy (correct and secure) stack including software and hardware. To achieve this goal, there are several directions to improve and extend the results of the thesis.

Some future work is straightforward based on the results of this thesis. For example, we suggest formalising the information flow proof of Paper C in HOL4, and extending the formalisation of MIL in Paper D with speculative semantics. We do not find major technical difficulties for the two extensions. In the following, we discuss the details of some directions that require more in-depth analysis.

Peripheral verification in Paper A presents an abstract model of I/O devices and their drivers. This abstract model describes the expected results of I/O subsystems without any details of devices or drivers. The abstract model can be used as a base of a framework for verifying multiple peripherals via bisimulation. To build such a framework, we can deeply embed the models in HOL4, which can provide proof automation for verifying bisimilarity. In addition, the information flow analysis in Paper A does not deal properly with side channels like timing, because the models do not include such channels. Considering the timing channel, it is difficult to capture the accurate timing information of the underlying hardware as the RTL design of most peripherals is not available. To address timing side channels, a practical solution is to define an observation model, which extracts the parts of a target system that can affect the execution time. Based on the observation model, a peripheral and its driver are secure if their executions are observational equivalence independent of secrets.

Processors play a central role in a trustworthy stack. Silver-Pi in Paper B is relatively simple compared to industrial processors. For example, Silver-Pi does not support data forwarding, OoO and speculative execution. We have implemented a version of Silver-Pi with a forward unit. To prove the correctness of a forward unit, the main work is the correctness of the ID stage. Data reading from the register file in the ID stage may be incorrect due to data hazards. If data hazards happen, we have to identify the previous instructions that affect the registers. If the previous instruction requires interactions with external components, then the stall unit prevents wrong data reading from the register file and the correctness of reading data is similar to the existing proof in Paper B. Otherwise, for regular instructions, we need to prove that the forward unit sends the correct result of the previous instruction to the ID stage. The correctness of OoO and speculative execution is much more complicated than an in-order pipeline. The correctness proof in Paper B relies on a scheduling function and a trace relation. It is hard to directly apply our approach to prove a pipeline with OoO execution. The major reason is the difficulty of defining the scheduling function as the definition requires manually mapping stages in such complex pipelines with ISA states. A possible solution is to define an OoO ISA model, and then it is feasible to map stages in an OoO pipeline with the OoO ISA model. Although MIL does not capture a
concrete OoO design, the formalisation of MIL provides insights to define formal ISA models of OoO execution. For example, MIL describes available resources like registers and semantics of OoO execution for common instructions like conditional jumps and memory operations.

A key point of a trustworthy stack is to establish end-to-end verification, which guarantees the correctness of software programs down to the processor’s implementation. The verification typically contains two main components, a verified compiler for generating the correct binary code of programs, and a verified processor that correctly executes the binary code as the ISA defines. CakeML [136] has a verified compiler that generates binary code for the Silver ISA from programs written in CakeML. Lööw et al. [7] prove the end-to-end verification for the non-pipelined Silver processor using the CakeML compiler. To establish the verification for Silver-Pi, there are two main problems, self-modifying programs and the progress of the pipelined circuit. The correctness of Silver-Pi relies on a software condition that no instruction is modified in the memory by the previous 4 instructions in the pipeline (no self-modifying). This condition is not necessary for non-pipelined processors that fetch a new instruction after the previous one is finished. On the other hand, the condition should be satisfied for any program running on Silver-Pi, otherwise, the processor fetches a wrong (old) instruction. The CakeML compiler does not produce a theorem, which ensures that the generated programs satisfy the self-modifying condition. Moreover, the correctness of compiled programs in CakeML states that the binary program will only produce behaviours (e.g., output a number) that are consistent with the source program in a finite number of steps. To use the correctness of the compiler for a verified stack, the end-to-end theorem requires a progress theorem of Silver-Pi. The progress theorem states that the circuit can finish any ISA-level instruction in a finite number of clock cycles. If the self-modifying condition is formally satisfied by the compiler and the progress theorem is proved, we can construct a verified stack for Silver-Pi. The stack guarantees that the execution of a program on Silver-Pi produces the same results as specified by the program written in CakeML within a finite number of cycles. If the verified stack is achieved, then the next step is to prove the security properties of the compiler for a trustworthy stack. The idea is to prove the CNI of binary programs compiled by CakeML. An observation function is defined at the source program level to extract public information during program execution. The CNI states that a compiled program does not leak more information than allowed by the observation function at the source program level. The CNI ensures that the compilation of CakeML does not introduce additional information leaks.

A computer system does not only consist of a compiler and a processor. In my opinion, hardware in a realistic stack contains at least a peripheral, a pipelined processor and a memory subsystem (a cache plus a main memory). Based on our verification for peripherals and processors, we can include a verified peripheral in the verified stack. The Silver ISA has an input and an output port, which can be used to connect a peripheral. However, this connection can only achieve simple input and output operations between two devices. For most serial devices, such
5.2. FUTURE WORK

I/O connections are not enough, because of, for example, the lack of a time port to synchronise. Since most serial devices are memory-mapped, a solution is to integrate a peripheral like SPI into the memory. However, the Silver ISA is a self-contained machine and the memory cannot be modified by external components. If a peripheral is merged into the memory, the peripheral affects the interaction between the memory and Silver-Pi (e.g., values in the memory are uncertain), and thus the correctness of Silver-Pi is violated. Since processors usually do not have an internal memory, we can remove the memory in the Silver ISA. Then, the Silver ISA and the processor interact with the memory in the same way and the correctness of Silver-Pi can be proved. Another solution to include a peripheral in the stack is to implement the peripheral as an accelerator running in parallel with the processor. We can extend the Silver ISA to specify more complex functionalities of the accelerator, e.g., processing audio data received from the environment.
Bibliography


[38] W. Stallings, Computer security principles and practice. 2015.


