Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ETH Dissertation 17500

Formal Approaches to Countering Side-Channel Attacks

Boris Alexander Köpf
Department of Computer Science

2007

DISS. ETH NR. 17500

Formal Approaches to Countering Side-Channel Attacks

Abhandlung
zur Erlangung des Titels
Doktor der Wissenschaften
der
Eidgenössischen Technischen Hochschule Zürich

vorgelegt von
Boris Alexander Köpf
Dipl. Math., Universität Konstanz
geboren am 05. Dezember 1976
deutscher Staatsbürger

angenommen auf Antrag von
Prof. Dr. David Basin, Referent
Prof. Dr. Andrei Sabelfeld, Korreferent

2007

DISS. ETH NO. 17500

FORMAL APPROACHES TO COUNTERING SIDE-CHANNEL ATTACKS

A dissertation
submitted to
ETH ZURICH
for the degree of
Doctor of Sciences

presented by
BORIS ALEXANDER KÖPF
Dipl. Math., University of Konstanz
born on 05 December 1976
citizen of Germany

accepted on the recommendation of
Prof. Dr. David Basin, examiner
Prof. Dr. Andrei Sabelfeld, co-examiner

2007

Abstract

Cryptographic algorithms are often modeled as idealized mappings from input to output. This is a problematic over-simplification. A villain determined to break cryptography will use all available information and will not restrict himself to the analysis of ciphertexts and public key material. So-called side-channel attacks demonstrate that characteristics such as the timing behavior of an algorithm's implementation can be effectively exploited for cryptanalysis.

In this thesis, we provide mathematically rigorous methods that allow for the detection, quantification, and elimination of side-channels. We focus on side-channels due to timing behavior and side-channels that arise from thread interleavings in multithreaded programs.

We present a novel method for detecting timing leaks in synchronous systems. The method is based on a parameterized and timing-sensitive notion of security that allows for the fine-grained modeling of information leakage. We present an efficient decision procedure for system security and show how it can be implemented in standard model-checking tools. We also present a model of adaptive side-channel attacks, which we combine with information-theoretic metrics to quantify the information revealed to an attacker. This allows us to express an attacker's remaining uncertainty about a secret as a function of the number of side-channel measurements made. We present algorithms and approximation techniques for computing this measure. Furthermore, we demonstrate how both of our methods can be used to analyze the resistance of hardware implementations of cryptographic functions to timing attacks.

We also present a method to detect and eliminate side-channels due to thread interleavings in multithreaded programs. Our approach uses unification on sub-programs to enforce that a program's alternative execution paths do not reveal information about the secrets involved in branching decisions. We demonstrate that integrating our approach into an existing transforming type system can improve the precision of the analysis and the quality of the resulting programs.

Kurzfassung

Kryptographische Algorithmen werden oft idealisiert als Funktionen von Eingaben auf Ausgaben modelliert. Dies ist eine problematische Vereinfachung, da sich ein Angreifer nicht auf die Analyse von öffentlichen Schlüsseln und Chiffraten beschränken muss, sondern auch andere verfügbare Informationen mit einbeziehen kann. In sogenannten Seitenkanalangriffen wurde gezeigt, dass es möglich ist, Charakteristika der Implementierung von kryptographischen Algorithmen, wie zum Beispiel deren Zeitverbrauch, für die Kryptoanalyse auszunutzen.

Die vorliegende Arbeit stellt mathematische Methoden bereit, um Seitenkanäle zu erkennen, zu quantifizieren und zu beseitigen. Der Schwerpunkt liegt hierbei auf Seitenkanälen durch Laufzeitcharakteristika und durch Scheduler-Verhalten in nebenläufigen Programmen.

Wir stellen einen neuen Ansatz zur Erkennung von laufzeitbasierten Seitenkanälen in synchronen Systemen vor. Die Grundlage dieses Ansatzes ist ein parametrisierter Sicherheitsbegriff, der es erlaubt, die preisgegebene Information genau zu beschreiben. Wir stellen ein effizientes Entscheidungsverfahren für die Sicherheit eines Systems vor und wir zeigen, wie es in handelsüblichen Modelcheckern implementiert werden kann. Des weiteren stellen wir ein Modell adaptiver Angriffe vor. Wir kombinieren dieses Modell mit Metriken aus der Informationstheorie. Dies erlaubt uns, die Information zu quantifizieren, die durch einen adaptiven Seitenkanalangriff gewonnen werden kann und die Unsicherheit des Angreifers über den Wert eines Geheimnisses als eine Funktion der durchgeführten Messungen auszudrücken. Wir stellen Algorithmen und Näherungsverfahren bereit, um diese Metriken zu berechnen und zeigen, wie Hardware-Implementierungen von kryptographischen Funktionen mit Hilfe unserer Methoden auf Schwachstellen durch Seitenkanalangriffe untersucht werden können.

Weiter stellen wir eine Methode vor, um schedulingbasierte Seitenkanäle in Programmen mit mehreren Threads zu erkennen und zu entfernen. Unser Ansatz wendet Unifikation auf Unterprogramme an und erreicht dadurch, dass Verzweigungen im Kontrollfluss keine Information über die Verzweigungsbedingung preisgeben. Wir integrieren diesen Ansatz in ein existierendes transformierendes Typsystem und zeigen, dass dadurch die Präzision der Analyse und die Qualität der ausgegebenen Programme erhöht werden kann.

Contents

List of Figures

Acknowledgments

I am especially grateful to Prof. David Basin for his guidance and support throughout my doctoral studies, and for giving me great freedom to investigate into what I deem interesting. I would like to thank Prof. Andrei Sabelfeld for his valuable feedback to my work and for serving on my thesis committee. I am particularly happy about this, as his work was the starting point for my research in information-flow security. I also want to thank Prof. Heiko Mantel for his collaboration in the beginning of my studies, and Prof. Patrick Schaumont for patiently answering all my questions related to hardware.

The information security group at ETH Zurich is an inspiring work environment and I would like to thank all group members for this great experience. Dr. Paul Hankes Drielsma and Michael Wahler are friends and great people to share an office with. Dr. Alexander Pretschner, Dr. Felix Klaedtke, and Dr. Achim Brucker helped me shape my mindset with their insights about life, science, and LaTeX. Manuel Hilty was an important companion on numerous mountain treks and in the last months while finishing our theses. Dr. Paul Sevinç generously shared his software-engineering know-how and countless chocolate bars. Michael Näf is a friend without whom my time in Zurich would not have been nearly as enjoyable. All of these people have given valuable feedback to texts leading to this dissertation, with the result that the end product has become much stronger for their suggestions.

Most importantly, I want to thank my parents for their love and constant support.

Chapter 1

Introduction

Cryptographic algorithms are often modeled as idealized mappings from input to output. This is a problematic over-simplification. A villain determined to break cryptography will use any available information and will not restrict himself to the analysis of ciphertexts and public key material. Information that has proven valuable for cryptanalysis includes the cryptosystem's timing characteristics [43, 15], its cache behavior [62], its power consumption [44], and its electromagnetic emanations [68, 33] during computation. Attacks that exploit the information revealed by an algorithm's physical execution are called *side-channel* attacks. Such attacks sidestep cryptographic security guarantees and are now so effective that they pose a real threat to systems that can be subjected to different kinds of measurements.

Side-channels are instances of covert channels, which are unintended ways of signaling information in computing environments. An information-flow analysis aims at detecting or guaranteeing the absence of covert channels. Ideally, the security guarantees obtained by such an analysis are rigorously backed up by mathematical proof. Many mathematical methods for the analysis of different kinds of covert channels have been developed. Maybe surprisingly, existing work does not provide satisfactory solutions for reasoning about side-channel attacks against cryptography – arguably the most daunting and certainly the best-documented threat within the scope of the field.

In this thesis, we investigate two different kinds of side-channels and develop mathematically rigorous methods for their analysis and elimination. In the first part of this thesis, we investigate side-channels that arise due to the timing behavior of cryptographic algorithms. We develop models and algorithms for their detection and for characterizing the information that is revealed through them. In the second part, we investigate scheduling side-channels, which are side-channels that arise due to thread interleavings in multithreaded programs. We develop a method for the security analysis of multithreaded programs that automatically removes certain scheduling side-channels by applying a transformation to the given program.

Throughout this thesis, we aim to provide a uniform presentation wherever possible. To this end, we develop a parameterized notion of secure information flow that we instantiate to capture both timing side-channels in synchronous hardware and scheduling side-channels in multithreaded programs.

In the remainder of this chapter, we will present the two kinds of side-channels that are relevant for our work. We will then briefly introduce, and discuss the scope of, formal methods for information-flow analysis. This allows us to identify concrete research goals. We conclude with a summary of our contributions towards achieving those goals.

1.1 Side-Channels

A *channel* is a mechanism for communicating information in a computing system. A *covert channel* is a channel whose primary use is not information transfer. Typically, the existence of such channels is unintended by design. Covert channels have received considerable attention in the context of multi-user operating systems, where processes can communicate via patterns in using, or obtaining locks on, shared resources such as disk space, memory, and processor time. Exploiting covert channels typically requires the cooperation of a sending and a receiving process.

In this thesis, we focus on covert channels that arise when a program unintentionally reveals information about the data it computes with. What constitutes a covert channel in this setting depends on a (possibly remote) observer of the program and his capabilities for distinguishing computations of the program with different data. To emphasize that the program does not intentionally reveal information and that the observer need not be a process on the same machine, we will use the terms *information leak* or *side-channel* instead of the term covert channel.

Below, we give examples of the side-channels of interest for this thesis: side-channels due to timing behavior and side-channels due to thread interleavings in multithreaded programs.

```
1. x := 1
2. for i := p − 1 to 0
3.     x := x ∗ x
4.     if k[i] = 1 then
5.         x := x ∗ c
6. return x
```

Figure 1.1: Modular exponentiation with a timing side-channel

1.1.1 Timing Side-Channels

A program has a timing side-channel if an observer can distinguish between computations with different secret input values by means of a clock.

We illustrate timing side-channels with an example from cryptography. For two reasons, timing leaks in cryptographic algorithms are arguably the most relevant instance of timing side-channels: first, they have been successfully exploited, as a considerable number of documented attacks show [43, 15, 19, 2]. Second, many of the attacks against timing leaks in cryptographic algorithms successfully recover secret key material, which is highly sensitive information.

Example 1.1. The RSA decryption step is a single modular exponentiation operation, in which a plaintext m is derived from a ciphertext c and a secret key k by computing $m = c^k \mod n$ for a given $n \in \mathbb{N}$. Figure 1.1 shows a a simple square-and-multiply algorithm for modular exponentiation. Here, the basis c is stored in a variable c, the exponent k is stored in an array k of p bits, and $*$ denotes multiplication modulo n. Consider the conditional branch in line 4. The multiplication operation in line 5 is carried out only if the ith exponent bit is set to 1. It is not difficult to see that, in this way, the algorithm's overall running time depends on the number of 1-bits in the exponent. An observer who can measure the running time of this algorithm will be able to deduce information about the exponent which, in the case of RSA decryption, is the decrypting agent's secret key. $\Diamond$

Example 1.1 illustrates a timing leak, but it is not obvious how such a leak can be exploited to systematically recover the key. An example of such a key recovery attack is given in Section 4.2. An important prerequisite for this and for many similar exploits is that the attacker can observe a large number of computations with different ciphertexts, which he can choose at his discretion.

```
h = 0    ▶ if h then (skip; l := 1) else l := 1   ||  skip; l := 0
           if h then (skip; l := 1) else l := 1   ||  ▶ skip; l := 0
         if h then (skip; l := 1) else ▶ l := 1   ||  skip; l := 0
           if h then (skip; l := 1) else l := 1   ||  skip; ▶ l := 0       l = 0

h = 1    ▶ if h then (skip; l := 1) else l := 1   ||  skip; l := 0
           if h then (skip; l := 1) else l := 1   ||  ▶ skip; l := 0
         if h then (▶ skip; l := 1) else l := 1   ||  skip; l := 0
           if h then (skip; l := 1) else l := 1   ||  skip; ▶ l := 0
         if h then (skip; ▶ l := 1) else l := 1   ||  skip; l := 0         l = 1
```

Figure 1.2: Scheduling side-channel

Several countermeasures have been proposed to eliminate timing side-channels, which we discuss in Section 7.1. In this thesis, we will focus solely on methods for detecting and quantifying timing side-channels.

1.1.2 Scheduling Side-Channels

A multithreaded program has a scheduling side-channel when a scheduler maps secret-dependent variations in the control flow of individual threads to variations in the observable output of the program.

Example 1.2. The program $C_1 = \mathsf{if\ h\ then\ (skip; l := 1)\ else\ l := 1}$ does not reveal the value of the variable h to an observer who may only see the value of the variable l after C_1 has terminated. However, if C_1 runs as a parallel thread with $C_2 = \mathsf{skip; l := 0}$ under a round-robin scheduler that assigns a time-slice to every instruction (including skip), the final value of l reflects the initial value of h. Figure 1.2 illustrates this phenomenon with two executions of $C_1 \parallel C_2$ in which h is initially set to 0 and 1, respectively, and where ▶ marks the instruction pointer's position. ◊

Scheduling channels are an important obstacle to provably secure computation in multithreaded programs and they have received considerable attention from researchers in language-based information flow. Their practical relevance, however, remains unclear: we are not aware of any documented exploit of a scheduling channel in software that has not been written for demonstration purposes.

Which countermeasures against scheduling side-channels are appropriate depends on the assumptions about the scheduler's behavior, and we will discuss different approaches in Section 7.2.3. In Example 1.2, where we assume that the scheduler assigns one time-slice to each instruction, the scheduling process can be decoupled from h's value by inserting a skip before the assignment in C_1's second branch.

1.2 Formal Methods and Models

Informally, a program has *secure information flow* if it keeps secrets confidential during computation, that is, if it has no side-channels. Three ingredients are required for a mathematically rigorous (synonymously: a *formal*) analysis of a program's information flow. The first is a mathematical representation (a *model*) of the program that performs the computation. The second is a mathematical representation of what we mean by secure information flow (a *flow property*), which must be expressed in terms of the program model. A violation of this flow property implies the existence of a channel through which secret information is revealed. The third is a mathematically sound method for determining whether a given program is secure, that is, a method for determining whether the model satisfies the flow property.

Analyzing the information flow in a program requires techniques that go beyond the analysis of safety and liveness properties. Rather than inspecting each of the program runs in isolation, the entirety of possible program behaviors must be analyzed [55, 91]. In particular, it is not possible to dynamically enforce all information flow properties using techniques such as execution monitoring [80] without severely restricting the program's behavior. Static analysis, on the other hand, allows one to derive guarantees that cover the entirety of program runs.

Example 1.3. Security type systems for programming languages [74] are examples of such a formal, static analysis technique. The model of a program is given in terms of a formal semantics for the programming language. The flow property is expressed in terms of this semantics. Security type systems are a syntax-driven method for statically enforcing a program's security: a successful type check implies that the program satisfies the flow property, a claim that is typically backed up by a soundness proof of the type system. However, most security type systems produce false negatives: a failed type check does not imply that a side-channel indeed exists. $\Diamond$

Formal approaches for analyzing side-channels have advantages and pitfalls, and we briefly discuss those that are relevant for this thesis. A clear advantage of using mathematical language is that it leaves no ambiguity about what is meant by, e.g., "secure information flow." Another advantage is that formal reasoning techniques can support a designer in determining whether a model actually satisfies the desired flow property. Ideally, this process can be automated. Security type systems, for example, provide mechanized support for checking that a program is secure, while keeping the overhead for the programmer low.

However, mathematical security guarantees need to be interpreted with care: they do not cover threats against features of the real system that are not reflected in the model. Choosing an adequate system model is thus a crucial prerequisite for a meaningful formal security analysis. Below, we will motivate our choices of models for this thesis.

1.2.1 Timing Models

A model for analyzing a program's timing side-channels must faithfully reflect the timing behavior of the program's implementation. A formal analysis of implementations on today's multi-purpose processors does not seem feasible, as providing precise timing models is currently out of reach. Giving upper bounds for the real-time behavior of multi-purpose processors is already a daunting task [67], and is still not sufficient for proving the absence of timing side-channels. In this thesis, we therefore approach the problem at a lower level of abstraction. Namely, we analyze timing side-channels at the level of synchronous (clocked) hardware, which is interesting because special-purpose hardware implementations of cryptographic algorithms are important in resource-critical application domains. As is standard, we model synchronous circuits as Mealy machines in which one transition corresponds to one clock tick. The hardware description language GEZEL [78, 77, 79] provides the link between the model and concrete hardware implementations. Namely, GEZEL allows for the specification of synchronous circuits in terms of automata, and it comes with a tool for translating the designs into a subset of the industrial-strength hardware description language VHDL. The translation is *cycle-true*, which means that it preserves the timing behavior within the granularity of clock ticks. Moreover, the output is *synthesizeable*, i.e. it can be mapped to a physical implementation. In this way, the security guarantees obtained by a formal analysis translate into guarantees for real-world hardware implementations.

1.2.2 Scheduling Models

For the analysis of scheduling side-channels, we follow the literature in the field and use a more abstract system model. We formalize multithreaded programs in terms of the operational semantics of a simple programming language with dynamic thread creation. As in Example 1.2, we assume that each time-slice allows for the execution of exactly one command in the language. An advantage of this assumption is that it leads to a simple model that allows one to study thread interleavings, which are the essence of scheduling side-channels. However, security guarantees that are obtained by a formal analysis using this model translate directly only to systems with schedulers that behave according to our assumptions. For modeling systems in which a scheduler's time-slices are based on the elapse of processor time, one is faced with problems like those sketched in Section 1.2.1. This is why we use the term *scheduling leak* instead of the term *internal timing leak*, which is often used in the literature.

1.3 Problem Statement

Below, we identify three obstacles to automatically detecting, quantifying, and eliminating information leaks.

Detecting Timing Side-Channels. There is a large body of work on rigorous methods for analyzing timing side-channels in high-level programming languages. However, there is only one formal approach that tackles timing leaks in hardware circuits [88, 87]. This approach is based on a security type system for VHDL and provides only an approximate solution to the problem of detecting timing leaks in hardware: it is tailored to VHDL and, like other type-based approaches, produces false negatives.

It has been an open problem to detect information leaks in synchronous hardware.

Quantifying Side-Channels. While the outcome of a standard information flow analysis is binary – either the program satisfies the security property or it does not – quantitative approaches aim to assess how much information is actually revealed. Existing approaches quantify the information that a passive observer can gain, but they do not adequately capture attackers who can interact with the system as in typical exploits. These interactions are often expensive or limited, and a meaningful quantitative measure must take their number into account.

It has been an open problem to quantify the information that an adaptive attacker can extract from a system's side-channels within a given number of measurements.

Eliminating Scheduling Side-Channels. Security type systems are an attractive language-based approach for detecting scheduling leaks. Typically, however, type checking is not a decision procedure for security, and a non-successful type check leaves the programmer with two open questions: whether the program is indeed insecure and, if so, how this flaw can be corrected. Transforming type systems address both problems. During the type-check, potential leaks are removed from the program and, hence, more programs are accepted as secure. However, today's transforming type systems lack precision: they correct only a subset of the programs that are intuitively correctable.

It has been an open problem to support the automated correction of insecure programs.

1.4 Contributions

This section gives an overview of our contributions to solving the problems sketched in Section 1.3. We have published our main results in [46], [47], and [48, 49], respectively.

Detecting (Timing) Side-Channels. Our contribution is a formal method for analyzing the information flow in hardware implementations. For this, we extend well-studied notions of confidentiality to a model of synchronous hardware. Our notion of security captures (but is not restricted to) timing side-channels. We give algorithms and complexity bounds for efficiently deciding system security, and we show how our decision procedures can be encoded in off-the-shelf model checkers. We demonstrate the feasibility of a hardware-level information-flow analysis by analyzing nontrivial examples.

Quantifying Side-Channels. We solve the problem of quantifying the information that an adaptive attacker can extract from a deterministic and stateless side-channel. Technically, our contribution is a model of adaptive attacks, which we combine with information-theoretic metrics to derive a meaningful measure of what an attacker can achieve within a given number of side-channel measurements. We show that our model can be easily automated and we apply it to analyze hardware implementations of cryptographic algorithms for their vulnerability to adaptive side-channel attacks.

Eliminating Scheduling Side-Channels. A program is secure if all program runs look identical to an observer. This intuitive notion of security can be expressed as the program relating to itself under a (non-reflexive) relation on program terms. We build on this characterization to develop a novel technique for repairing insecure programs. Namely, we apply unification to program terms in order to relate a program to itself and thus achieve security. We use this approach to improve an existing security type system.

Our contribution is a transforming type system that improves on previous transformations to enforce scheduler-independent security in the sense that it can repair programs for which all previously existing approaches fail. Furthermore, we improve the quality of corrections: programs resulting from a transformation by our type system are more efficient and smaller in size than when transformed by related approaches.

1.5 Outline

The remainder of this thesis is structured as follows. In Chapter 2, we introduce the system models and the notions of secure information flow that we use throughout this thesis. In Chapter 3, we present efficient algorithms for detecting information leaks in deterministic and nondeterministic finite-state systems. Moreover, we show how to implement the algorithm for the deterministic case in a temporal logic model-checker. In Chapter 4, we show how to express and determine the information that is leaked through a side-channel both quantitatively and qualitatively. In Chapter 5, we present a unification-based transforming type system for eliminating scheduling leaks from multithreaded programs. In Chapter 6, we evaluate the techniques developed in Chapters 3 and 4 by analyzing hardware implementations of cryptographic algorithms. We present related work and conclude in Chapters 7 and 8, respectively.

Chapter 2

Background

2.1 Introduction

In this chapter, we introduce the building blocks of a rigorous information flow analysis, namely, a formal system model and a formal notion of secure information flow. We carve out the common basis behind the two kinds of side-channels of interest for this thesis. To this end, we define a general system model and a parametric flow property that we later instantiate to capture both scheduling leaks in multithreaded programs and timing leaks in synchronous hardware.

Apart from the conceptual appeal of having a general basis for both kinds of side-channels, our approach has a concrete advantage over a separate treatment: the procedures we develop for deciding system security (see Chapter 3) are applicable for both detecting timing leaks in synchronous hardware and scheduling leaks in multithreaded programs with finite memory.

We begin with a brief overview of equivalence relations and partitions – two notions we will use extensively throughout this thesis.

2.2 Equivalence Relations and Partitions

An *equivalence relation* on a set S is a binary relation $R \subseteq S \times S$ that is reflexive, transitive, and symmetric. The (R-)*equivalence class* $[x]_R$ of $x \in S$ is the set of all elements of S that relate to x under R, i.e, $[x]_R = \{y \in S \mid x \; R \; y\}$. The *quotient set* S/R is the set of all R-equivalence classes, i.e., $\{[x]_R \mid x \in S\}$. A *partial equivalence relation* (*Per*) on S is a binary relation $Q \subseteq S \times S$ that is transitive and symmetric. The *domain* of a partial equivalence relation Q is the set $dom(Q) = \{x \in S \mid x\ Q\ x\}$ on which Q is reflexive, and hence an equivalence relation.

A *partition* of a set S is a set $\pi = \{B_1, \ldots, B_n\}$ of pairwise disjoint *blocks* with the requirement that $\bigcup_{i=1}^{n} B_i = S$. A *refinement* of a partition π is a partition π' such that every block of π' is contained in some block of π. This relationship is denoted by $\pi' \leq \pi$. For $A \subseteq S$ and a partition π of S, we define the *restriction* of π to A as $\{A \cap B \mid B \in \pi\}$ and denote it by $A \cap \pi$. Clearly, $A \cap \pi$ is a partition of A. For partitions π_1 and π_2, we define $\pi_1 \cap \pi_2$ as the partition $\{A \cap B \mid A \in \pi_1, B \in \pi_2\}$. Clearly, $\pi_1 \cap \pi_2 \sqsubseteq \pi_1$ and $\pi_1 \cap \pi_2 \sqsubseteq \pi_2$.

Equivalence relations and partitions are closely related. The quotient set S/R of an equivalence relation $R \subseteq S \times S$ is a partition of S. Conversely, a partition π of S gives rise to an equivalence relation R_π on S, where $x\ R_\pi\ y$ if and only if there is a $B \in \pi$ with $x \in B$ and $y \in B$.

2.3 Languages and System Models

Throughout this thesis, we use transition systems as a general system model. In this section, we first give a definition of such transition systems. Subsequently, we define specializations that are tailored to the two classes of systems of interest for this thesis: multi-threaded programs and synchronous hardware circuits, respectively.

Definition 2.1. A *transition system* is a 5-tuple $M = (S, \Sigma, \Gamma, \delta, s_0)$, where S is a set of states, Σ is an input alphabet, Γ is an output alphabet, $\delta \subseteq S \times \Sigma \times \Gamma \times S$ is a transition relation, and $s_0 \in S$ is the initial state. A transition system is *input enabled* if for all $s \in S$ and $a \in \Sigma$, there is a $b \in \Gamma$ and a $s' \in S$ with $(s, a, b, s') \in \delta$. A transition system is *finite*, if S, Σ, and Γ are finite sets. We write $\delta(s, a, b)$ to denote the set $\{s' \mid (s, a, b, s') \in \delta\}$.

In the remainder of this thesis, we will mainly consider input-enabled transition systems, and we will usually leave this implicit. The transition systems representing multi-threaded programs are possibly infinite, and we use a programming language for their finite representation. The transition systems representing synchronous hardware circuits will be finite, and we only informally sketch their syntactic representation.

2.3.1 Synchronous Hardware

In our case studies, we specify synchronous circuits in terms of the hardware description language GEZEL. GEZEL allows for circuits to be described as datapaths with a finite-state controller that can easily be interpreted as Mealy machines, i.e. deterministic transition systems. Below, we will illustrate the syntax of GEZEL using an example. Subsequently, we will define Mealy machines as our formal model of synchronous hardware and sketch the (informal) translation of GEZEL code to this model. For a full reference for the GEZEL language, refer to [77].

GEZEL **Syntax by Example**

The basic design unit in GEZEL is a *module*. A module consists of a datapath and a finite state controller and communicates by means of input and output signals. *Signals* (keyword: `sig`) represent wires that carry integer values of a fixed bit-width. A *datapath* (`dp`) consists of registers and signal flow graphs. A *register* (`reg`) stores an integer of a fixed bit-width and a *signal flow graph* (`sfg`) is a sequence of assignments `l:=r`, where `l` is a register or an output signal, and where `r` is an arithmetic expression over input signals and registers. A register may be assigned to at most once during each clock cycle and has the assigned value in the subsequent cycle. An output signal must be assigned to exactly once during each clock cycle. The *controller* (`fsm`) schedules the execution of signal flow graphs and is specified in terms of a finite automaton in which every transition corresponds to one clock cycle. Transitions may conditionally depend on boolean expressions over signals and registers and they are labeled with the signal flow graphs to be executed during the corresponding cycle. Cyclic dependencies between signal assignments in the signal flow graphs of one transition are forbidden.

Example 2.1. Figure 2.1 depicts a GEZEL implementation of a counter. The datapath `counter` (lines 1 to 8) has two 4-bit input signals `start` and `stop`, an output signal `done` and two 4-bit registers `current` and `upper` for storing the counter's current and maximal values, respectively. The datapath contains signal flow graphs for initializing the counter's bounds (`init`), increasing the counter (`oneup`) and for signaling termination (`term` and `noterm`). The controller `counter_ctl` (lines 9 to 16) represents an automaton with three states `s0`, `s1`, and `s2`. Lines 12 to 15 specify that, after an initialization step in line 12, the automaton remains in state `s1` until the value of `current` reaches the value of `upper`. Then the automaton signals termination and loops in state `s2`. $\Diamond$

```
dp counter(in  start, stop : ns(4);
           out done        : ns(1)) {
  reg current, upper : ns(4);
  sfg init    {current=start; upper=stop;}
  sfg oneup   {current=current+1;}
  sfg term    {done=1;}
  sfg noterm  {done=0;}
}

fsm counter_ctl(counter) {
  initial s0;
  state s1, s2;
  @s0 (init, noterm) -> s1;
  @s1 if (current==upper)  then (term)         -> s2;
                           else (oneup,noterm) -> s1;
  @s2 (term) -> s2;
}
```

Figure 2.1: A counter in GEZEL

Semantics

As is standard [29], we use Mealy machines as a semantic model for synchronous hardware circuits and we assume that one transition corresponds to one clock cycle of a global clock. We will be particularly interested in Mealy machines that compute using the input provided in their initial state and that ignore all inputs given in subsequent states.

Definition 2.2. A *Mealy machine* is a finite transition system $M = (S, \Sigma, \Gamma, \delta, s_0)$ where the transition relation $\delta \subseteq S \times \Sigma \times \Gamma \times S$ is a function on $S \times \Sigma$. We call a Mealy machine *triggered* if s_0 has no ingoing transitions and if for all $a, b \in \Sigma$ and for all $s \in S \setminus \{s_0\}$ $(s, a, c_1, s_1), (s, b, c_2, s_2) \in \delta$ implies $c_1 = c_2$ and $s_1 = s_2$.

Examples of triggered Mealy machines are the counter of Example 2.1 and all of the circuits we will present in Chapter 6.

Giving GEZEL a semantics in terms of Mealy machines is straightforward. Input and output signals are mapped to Σ and Γ, respectively, which will be of the form

$\{0,1\}^n$, for some $n \in \mathbb{N}$. The set of states S is the Cartesian product of the set of possible register values and the set of control states in the datapath's controller. The transition function δ is specified by the datapath's controller.

Example 2.2. Our formal representation of the counter of Example 2.1 is a Mealy machine $M = (S, \Sigma, \Gamma, \delta, s_0)$ with $S = \{s_0, s_1, s_2\} \times \{0,1\}^{4\cdot 2}$, $\Sigma = \{0,1\}^{4\cdot 2}$ and $\Gamma = \{0,1\}$. Here, S is the product of the set of control states with the set of possible values of the registers `current` and `upper`. Σ and Γ represent the possible values of the input and output signals, respectively. Deriving δ from lines 11-15 of the code in Figure 2.1 is straightforward. $\Diamond$

2.3.2 Multithreaded Programs

We specify programs with multiple threads in terms of the multi-threaded while language (MWL) from [75]. Below, we briefly define the syntax and the operational semantics of MWL. For an introductory text to the formal semantics of programming languages see, e.g., [95].

Syntax

The language includes assignments, conditionals, loops, and a command for dynamic thread creation. We represent programs as vectors of threads, where the set of single threads *Com* is given by the grammar

$$C ::= \mathsf{skip} \mid \mathit{Id} := \mathit{Exp} \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C\,V)\ .$$

Here, C, C_1, C_2 denote programs in *Com* and *Id* ranges over a set of variable identifiers *Var*. *Exp* and B denote arithmetic and boolean expressions that consist of variables, constants, or terms resulting from applying binary operators to expressions. They will not be further specified. We represent multi-threaded programs $V = \langle C_0, \ldots, C_n \rangle$ as vectors of threads (also called *thread pools*), that is, as elements of $\vec{Com} = \bigcup_{n \in \mathbb{N}} Com^n$. We will identify thread pools of length 1 with single threads. We denote the concatenation of two thread pools $V = \langle C_0, \ldots, C_n \rangle$ and $W = \langle C'_0, \ldots, C'_m \rangle$ by VW, i.e. $VW = \langle C_0, \ldots, C_n, C'_0, \ldots, C'_m \rangle$.

Semantics

Let *Var* be an arbitrary set of variable identifiers and let *Val* be an arbitrary set. A *memory* is a mapping ν from variables in *Var* to values in *Val*. We denote the set of all

$$\llparenthesis \mathsf{skip}, \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle\rangle, \nu \rrparenthesis \qquad \frac{\llparenthesis Exp, \nu \rrparenthesis \downarrow n}{\llparenthesis Id := Exp, \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle\rangle, [Id = n]\nu \rrparenthesis}$$

$$\frac{\llparenthesis C_1, \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle\rangle, \mu \rrparenthesis}{\llparenthesis C_1; C_2, \nu \rrparenthesis \rightarrowtriangle \llparenthesis C_2, \mu \rrparenthesis} \qquad \frac{\llparenthesis C_1, \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle C_1' \rangle V, \mu \rrparenthesis}{\llparenthesis C_1; C_2, \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle C_1'; C_2 \rangle V, \mu \rrparenthesis}$$

$$\frac{\llparenthesis B, \nu \rrparenthesis \downarrow \mathsf{True}}{\llparenthesis \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \nu \rrparenthesis \rightarrowtriangle \llparenthesis C_1, \nu \rrparenthesis} \qquad \frac{\llparenthesis B, \nu \rrparenthesis \downarrow \mathsf{False}}{\llparenthesis \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \nu \rrparenthesis \rightarrowtriangle \llparenthesis C_2, \nu \rrparenthesis}$$

$$\frac{\llparenthesis B, \nu \rrparenthesis \downarrow \mathsf{True}}{\llparenthesis \mathsf{while}\ B\ \mathsf{do}\ C, \nu \rrparenthesis \rightarrowtriangle \llparenthesis C; \mathsf{while}\ B\ \mathsf{do}\ C, \nu \rrparenthesis} \qquad \frac{\llparenthesis B, \nu \rrparenthesis \downarrow \mathsf{False}}{\llparenthesis \mathsf{while}\ B\ \mathsf{do}\ C, \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle\rangle, \nu \rrparenthesis}$$

$$\llparenthesis \mathsf{fork}(CV), \nu \rrparenthesis \rightarrowtriangle \llparenthesis \langle C \rangle V, \nu \rrparenthesis$$

Figure 2.2: Operational semantics of threads

memories $\nu : Var \to Val$ by *Mem*. For $\nu \in Mem$, $Id, Id' \in Var$ and $n \in Val$, we denote by $[Id = n]\nu$ the memory defined by $([Id = n]\nu)(Id) = n$, and $([Id = n]\nu)(Id') = \nu(Id')$ if $Id \neq Id'$.

We use the judgment $\llparenthesis Exp, \nu \rrparenthesis \downarrow n$ to denote that expression *Exp* evaluates to value n in memory ν and we assume that expression evaluation is total. We say that expressions *Exp* and *Exp'* are *equivalent* (denoted by $Exp \equiv Exp'$) if and only if they evaluate to identical values in each memory, i.e. if for all memories $\nu \in Mem$ we have $\llparenthesis Exp, \nu \rrparenthesis \downarrow n \Leftrightarrow \llparenthesis Exp', \nu \rrparenthesis \downarrow n$. A *configuration* is a pair $\llparenthesis V, \nu \rrparenthesis$, with threads $V \in \vec{Com}$ and memory $\nu \in Mem$.

The operational semantics for MWL is formalized in Figures 2.2 and 2.3 and comprises deterministic and nondeterministic transitions. *Deterministic transitions* (see Figure 2.2) are denoted by $\llparenthesis C, \nu \rrparenthesis \rightarrowtriangle \llparenthesis W, \mu \rrparenthesis$, expressing that a single thread C performs a computation step with memory ν, yielding a new memory μ and a thread pool W. W has length zero if C has terminated, length one if C has neither terminated nor spawned any new threads, and length greater than one if new threads were spawned.

Nondeterministic transitions (see Figure 2.3) model the execution of a multi-threaded program on a single processor and are denoted by $\llparenthesis V, \nu \rrparenthesis \rightarrowtriangle \llparenthesis W, \mu \rrparenthesis$, where V and W are thread pools, expressing that some thread C_i in V performs a step with memory ν resulting in memory μ and some thread pool W'. The global thread pool W results then by replacing C_i with W'.

Nondeterminism models the possible thread choices of a scheduler, but it does not mandate any particular scheduling strategy. As information flow properties are, in

$$\frac{\langle\!| C_i, \nu |\!\rangle \twoheadrightarrow \langle\!| W', \mu |\!\rangle}{\langle\!| \langle C_0 \dots C_{n-1} \rangle, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle C_0 \dots C_{i-1} \rangle W' \langle C_{i+1} \dots C_{n-1} \rangle, \mu |\!\rangle}$$

Figure 2.3: Operational semantics of thread pools

general, not preserved under refinement [52] (e.g., the choice of a particular scheduler), using a possibilistic system model for an information flow analysis needs to be justified. This justification is the subject of [75], where Sabelfeld and Sands give a probabilistic semantics for $\vec{Com}$ that is based on an explicit model of a scheduler, together with a probabilistic notion of security. The authors put forward *strong security*, a notion of secure information flow that can be expressed in terms of a possibilistic semantics for $\vec{Com}$, and they prove that strong security implies probabilistic security for a large class of schedulers, including round-robin and uniform schedulers. In this thesis, we will analyze programs with respect to strong security, hence it is sufficient to work with the more abstract, possibilistic semantics for $\vec{Com}$.

Multi-threaded programs as transition systems

It is intuitively clear that the semantics of MWL defines a transition system. We will make this intuition explicit by giving two alternative interpretations of the operational semantics as a transition system according to Definition 2.1.

In a transition $\langle\!| V, \nu |\!\rangle \twoheadrightarrow \langle\!| V', \mu |\!\rangle$, the thread pools V and V' take the role of the states of a transition system, the memory ν takes the role of the input to the system, and the memory μ takes the role of the output of the system. For this, observe that mappings of type $Var \to Val$ can be represented as elements of an alphabet $\prod_{v \in Var} Val = Val^{Var}$. Choosing thread pools as states and allowing arbitrary memories as inputs to every transition is required for expressing strong security, which we will use as a notion of secure information flow in multi-threaded programs. When a program is actually executed, the input may be given in terms of the initial memory only.

We say that $W \in \vec{Com}$ is *reachable* from $V \in \vec{Com}$ if there are memories $\nu_0, \nu_1, \dots, \nu_n$ and thread pools $V = V_0, V_1, \dots, V_n = W$ such that $\langle\!| V_i, \nu_i |\!\rangle \twoheadrightarrow \langle\!| V_{i+1}, \nu_{i+1} |\!\rangle$ is derivable for $0 \leq i \leq n-1$.

Definition 2.3. The transition system *induced* by $V \in \vec{Com}$ is defined as $M_V = (S_V, \Sigma_V, \Gamma_V, \delta_V, V)$, where $S_V \subseteq \vec{Com}$ is the set of programs that are reachable from V, and where $\Sigma_V = \Gamma_V = Val^{Var}$. For $W, W' \in S_V$ and $\nu, \nu' \in Mem$ we define $(W, \nu, \nu', W') \in \delta$ if and only if $\langle\!| W, \nu |\!\rangle \twoheadrightarrow \langle\!| W', \nu' |\!\rangle$ is derivable.

Next we will give an alternative definition of an induced transition system, where we augment the output of each transition with scheduling-relevant data, namely, the number of currently active threads and the index of the thread that was scheduled for execution in this transition.

Definition 2.4. We define the *augmented* transition system induced by $V \in \vec{Com}$ as $\hat{M}_V = (S_V, \Sigma_V, \Gamma'_V, \delta'_V, V)$, where S_V, Σ_V and Γ_V are as in Definition 2.3 and where $\Gamma'_V = \Gamma_V \times \mathbb{N} \times \mathbb{N}$. For $W = \langle C_0, \ldots, C_{n-1} \rangle$, $W' = \langle C_0, \ldots, C_{i-1} \rangle W'' \langle C_{i+1}, \ldots, C_{n-1} \rangle$ and $v, v' \in Mem$, we define $(W, v, (v', n, i), W') \in \delta$ if and only if $\langle\!| W, v |\!\rangle \twoheadrightarrow \langle\!| W', v' |\!\rangle$ is derivable.

Note that $\hat{M}_V$ outputs (v', n, i) during a transition only if the ith thread from a pool of n threads is scheduled for execution. As we assume that expression evaluation is total in MWL, the transition relation of $\hat{M}_V$ is defined for every input unless V has terminated. To formally make $\hat{M}_V$ input enabled, one can add $(\langle\rangle, v, (v, 0, 0)), \langle\rangle)$ for every $v \in Mem$ to δ.

It is not difficult to see that if the set of variable values *Val* is finite, then the transition system induced by a single thread without fork() (that is, without dynamic thread creation) is finite. Consequently, the transition system induced by a thread pool without dynamic thread creation and with finite memory is also finite.

2.4 Expressing Secure Information Flow

A system keeps secrets confidential during computation if it exhibits *secure information flow*. In this section, we will show how secure information flow can be formally expressed. As in [53], we introduce (and distinguish between) flow policies and flow properties.

Flow policies are high-level requirements regarding the flow of information between abstract security domains. Examples of security domains are classifications such as "confidential" and "non-confidential" and a typical requirement is that no confidential information may flow into the non-confidential domain. Flow policies are system independent; in particular, they are independent of any particular means of signaling information.

By contrast, *flow properties* are low-level requirements regarding the flow of information in a system and they are expressed in terms of a model of the system and a model of its observer. The observer model expresses an agent's capabilities for distinguishing variations in the system's behavior due to different input data, thereby capturing a potential side-channel. A flow policy can be mapped to a flow property by associating both the system's input and the system's observer with security domains.

Below, we will introduce flow policies and a parameterized flow property, and we will show how to implement a given flow policy by instantiating the parameters of the flow property.

2.4.1 Flow Policies

A *flow policy* is a pair $(\mathcal{D}, \rightsquigarrow)$, where $\mathcal{D}$ is a nonempty finite set of *security domains* and where $\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ is a reflexive and transitive binary relation, which we call the *flow relation*. For two security domains $A, B \in \mathcal{D}$, the relationship $A \rightsquigarrow B$ denotes that information may flow from A to B. Conversely, if $A \not\rightsquigarrow B$, then the flow of information from A to B is forbidden.

Example 2.3. Consider two security domains, one for confidential (*high*) and one for non-confidential (*low*) information, together with the requirement that no information must flow from the confidential into the non-confidential domain. This requirement is formalized by the policy $P_0 = (\{H, L\}, \rightsquigarrow)$, where H and L represent the domains for high and low data, respectively, and where $\rightsquigarrow = \{H \rightsquigarrow H, L \rightsquigarrow L, L \rightsquigarrow H\}$. We will often abbreviate P_0 as $H \not\rightsquigarrow L$. $\Diamond$

Throughout this thesis, we focus on the policy P_0. As the following example shows, we do not lose any generality with this restriction.

Example 2.4. Consider an arbitrary flow policy $P = (\mathcal{D}, \rightsquigarrow)$ with $|\mathcal{D}| \geq 2$ (Note that a flow policy with $|D| = 1$ does not pose any restrictions on the flow of information and is thus trivially fulfilled). Enforcing P means guaranteeing that there is no illegal information flow into any $D \in \mathcal{D}$. To express this requirement in terms of P_0, define $L_D = D$, and identify all $E \in \mathcal{D}$ for which $E \not\rightsquigarrow D$ is required with H_D. In this way, enforcing P can be reduced to enforcing $H_D \not\rightsquigarrow L_D$ for every $D \in \mathcal{D}$. We will use this reduction in Example 5.9, where we enforce a multi-level flow policy using a type system for P_0. $\Diamond$

2.4.2 A Parametric Flow Property

We model the information flow in a program with respect to an observer of the program. An observer is modeled in terms of two parameters that express what he *can observe* of the program's behavior, and what he *may know* about the program's input.

The program satisfies the flow property if, for all possible inputs, what the observer can see does not allow him to refine what he may know about the input. We will show how the parameters can be instantiated to express both secure information flow and the leakage of a well-defined part of the input.

Technically, we model the observer in terms of his capabilities for distinguishing different program behaviors.

Distinguishing atomic inputs and outputs. We capture that the observer cannot distinguish between two outputs $a, b \in \Gamma$ with an equivalence relation $R_O \subseteq \Gamma \times \Gamma$. We say that a and b are *observationally equivalent*, or simply R_O*-equivalent*, if and only if $a\ R_O\ b$. In other words, if the system outputs $x \in \Gamma$, the observer can only deduce the equivalence class $[x]_{R_O}$. Similarly, we use the equivalence relation $R_I \subseteq \Sigma \times \Sigma$ to model to what extent the observer may know the inputs of the system. If the system computes with input x, the observer may only know that some element of $[x]_{R_I}$ is processed.

In the following, Id_X denotes the identity on a set X and All_X denotes $X \times X$. For relations $R \subseteq \Gamma_1 \times \Gamma_1$ and $Q \subseteq \Gamma_2 \times \Gamma_2$, we overload notation and define $R \times Q \subseteq (\Gamma_1 \times \Gamma_2)^2$ as $(r_1, q_1)\ (R \times Q)\ (r_2, q_2)$ if and only if $r_1\ R\ r_2$ and $q_1\ Q\ q_2$.

Example 2.5. The relation $R_O = \mathrm{All}_\Gamma$ formalizes an observer who cannot distinguish between any two system outputs. In contrast, the relation $R_O = Id_\Gamma$ models an observer who can determine the (singleton) Id_Γ-equivalence class of the output, or equivalently, who can precisely determine the output. If $\Sigma = \Sigma_1 \times \Sigma_2$, the relation $R_I = Id_{\Sigma_1} \times \mathrm{All}_{\Sigma_2}$ models an observer who may only know the Σ_1-component of the input. We can also model more fine-grained capabilities. Consider, for example, $\Sigma = \{0, 1\}^n$ and the predicate $\Psi_\Sigma = \{(a, b) \in \Sigma \times \Sigma \mid \|a\| = \|b\|\}$, where $\|x\|$ denotes the Hamming weight of x, i.e. the number of bits set to 1. Ψ_Σ models that an observer may only know the Hamming weight (determine the Ψ_Σ-equivalence class) of the input. $\Diamond$

R_I/R_O**-security.** Two states of a system are *observationally equivalent* if every output from one state can be matched by an R_O-equivalent output from the other state whenever the corresponding inputs are R_I-equivalent, and if the resulting states are again observationally equivalent. We capture this observational equivalence of states by R_I/R_O-equivalence, a parameterized notion of strong bisimulation. R_I/R_O-equivalence allows one to distinguish program behaviors that differ with regard to the number of transitions that lead to some output, thereby capturing timing behavior.

Definition 2.5 (R_I/R_O-Equivalence). Let $M = (S, \Sigma, \Gamma, \delta, s_0)$ be an automaton with output, and let $R_I \subseteq \Sigma^2$ and $R_O \subseteq \Gamma^2$ be equivalence relations. We define $\simeq_{R_O}^{R_I}$ as the union of all symmetric and transitive relations $\mathcal{R}$ on S with the property that for all $s_1, s_2 \in S$:

$$s_1 \;\mathcal{R}\; s_2 \Rightarrow \forall a_1, a_2 \in \Sigma.(a_1 \; R_I \; a_2 \Rightarrow \begin{array}{l} \forall (s_1, a_1, o_1, s_1') \in \delta. \\ \exists (s_2, a_2, o_2, s_2') \in \delta. \\ \; s_1' \;\mathcal{R}\; s_2' \wedge o_1 \; R_O \; o_2). \end{array} \tag{2.1}$$

Two states $s_1, s_2 \in S$ are R_I/R_O*-equivalent* iff $s_1 \simeq_{R_O}^{R_I} s_2$.

It is easy to see that $\simeq_{R_O}^{R_I}$ is a partial equivalence relation on S and that $\simeq_{R_O}^{R_I}$ itself satisfies Property (2.1) of Definition 2.5.

Example 2.6. Consider the Mealy machine from Example 2.2. If the observer may know the input to `start` but not that to `stop` (that is, R_I relates all inputs with identical `start`-components) and if he can see whether the `done` flag is set (that is, $R_O = Id_\Gamma$), then M is not R_I/R_0 secure: for a fixed value of `start`, M produces distinguishable sequences of output when it receives different values of `stop` in s_0. Hence $s_0 \not\simeq_{R_O}^{R_I} s_0$. $\diamondsuit$

In general, if the initial state of a system is *not* observationally equivalent to itself, then running the system on R_I-equivalent input sequences may lead to observable differences in the system behavior. This constitutes a refinement of the observer's knowledge about the input (modeled by R_I), and thus is an information leak. We will make this refinement explicit in Section 4.3.4. If, on the other hand, the initial state is observationally equivalent to itself, then we say that the system is R_I/R_O*-secure*.

Definition 2.6 (R_I/R_O-Security). Let $M = (S, \Sigma, \Gamma, \delta, s_0)$ be a transition system and let $R_I \subseteq \Sigma^2$ and $R_O \subseteq \Gamma^2$ be equivalence relations. Then M is R_I/R_O*-secure* iff $s_0 \simeq_{R_O}^{R_I} s_0$.

The idea that security can be modeled as a system being observationally equivalent to itself is formalized in the Per model of secure information flow [76].

2.4.3 Mapping Flow Policies to Flow Properties

To turn a flow policy $P = (\mathcal{D}, \rightsquigarrow)$ into a flow property on a transition system $M = (S, \Sigma, \Gamma, \delta, s_0)$, we need to interpret $\mathcal{D}$ and $\rightsquigarrow$ in terms of M. We illustrate this with the policy $P_0 = H \not\rightsquigarrow L$. As Example 2.4 shows, arbitrary flow policies can be reduced to this case.

Mapping Security Domains to I/O. We assume that high and low inputs are provided simultaneously to the system and we model this by choosing $\Sigma = \Sigma_H \times \Sigma_L$, where Σ_H and Σ_L are the sets of possible high and low inputs, respectively. Similarly, we assume that the output consists of high and low components, that is, $\Gamma = \Gamma_H \times \Gamma_L$.

Example 2.7. In the case of synchronous hardware circuits, the set of input signals is split into signals that carry confidential data $\Sigma_H = \{0,1\}^n$ and input signals that carry non-confidential data $\Sigma_L = \{0,1\}^m$, that is, $\Sigma = \{0,1\}^{m+n}$. Output signals are split analogously. $\Diamond$

Example 2.8. In a programming-language setting, a security domain is assigned to every variable in *Var*. This corresponds to a partitioning of *Var* according to the security domains in $\mathcal{D}$. For $\mathcal{D} = \{H, L\}$ we have $Var = Var_H \uplus Var_L$, where Var_L and Var_H are the variables of low and high domains, respectively. Consequently, we have $\Sigma = Val^{Var_L} \times Val^{Var_H}$ for the transition system induced by a multithreaded program. $\Diamond$

As notation, we write $\nu =_L \mu$ to denote that memories $\nu, \mu \in Mem$ coincide on the variables of domain L, i.e., that $\nu\ (\mathrm{All}_H \times Id_L)\ \mu$ holds.

Mapping Flow Relations to R_I/R_O-security. The policy that no information must flow from the high into the low domain, i.e. $H \not\rightsquigarrow L$, corresponds to the requirement that an observer with low clearance must not be able to distinguish variations in the system behavior that result from different high inputs. This requirement is captured by $\mathrm{All}_{\Sigma_H} \times Id_{\Sigma_L} / \mathrm{All}_{\Gamma_H} \times Id_{\Gamma_L}$-security, which is often coined *non-interference* and which we will abbreviate as *NI-security*.[1] More generally, we say that $\mathrm{All}_{\Sigma_H} \times Id_{\Sigma_L} / R_O$-secure programs have *secure information flow*, as their execution does not leak any secret information to the observer modeled by R_O; that is, they have no side-channel with respect to this observer.

2.4.4 Examples

In this section, we will take a closer look at variants of non-interference on different system models and for different observers. In Section 4.3.4, we will investigate how R_I/R_O-security can also be used to express the leakage of a well-defined portion of the secret input. When Σ is understood, we write Id_L as an abbreviation for Id_{Σ_L}. We abbreviate analogously for All_{Σ_L}, Γ and the high domain.

[1]Non-interference was originally defined in a different context [37], but has become the standard term for naming the absence of illegal information flow in many settings.

Example 2.9. On Mealy machines, $\simeq^{\mathrm{All}_H \times Id_L}_{\mathrm{All}_H \times Id_L}$ represents a notion of observational equivalence closely related to Agat's Γ-*bisimulation* [4]. In Agat's approach, timing is captured by transition labels that represent the duration of the corresponding operations on the underlying machine. Γ-bisimulation relates programs in which the timing between observable events is equivalent. In our setting, one transition of the Mealy machine corresponds to one clock cycle in a synchronous circuit. Hence, a NI-secure circuit does not leak any secret information to a low observer who can determine the circuit's execution time up to single clock ticks. $\Diamond$

Example 2.10. For a multithreaded program V and the induced transition system M_V, NI-security represents a possibilistic notion of security similar to Volpano and Smith's *concurrent non-interference* [84], which has been used to model the security of multithreaded programs in the presence of a purely nondeterministic scheduler. Note that NI-security is more restrictive with respect to the number of transitions that lead to an output, as it is based on strong bisimulation equivalence, as opposed to the weak bisimulation-based concurrent non-interference. $\Diamond$

Strong security is a security property on multithreaded programs that corresponds to the policy P_0. A program that satisfies strong security does not leak high information to a low observer, in the presence of an arbitrary scheduler. Strong security assumes an observer who can see the values of the low variables during the entire program run. By interpreting certain low variables as communication channels, strong security can be used for capturing information leaks through patterns in a program's observable communication.

Definition 2.7 ([75]). The *strong low-bisimulation* $\cong_L$ is the union of all symmetric relations R on command vectors $V, V' \in \vec{Com}$ of equal size, i.e. $V = \langle C_1, \ldots, C_n \rangle$ and $V' = \langle C'_1, \ldots, C'_n \rangle$, such that

$$\begin{aligned}
&\forall \nu, \nu', \mu \in Mem. \forall i \in \{1 \ldots n\}. \forall W \in \vec{Com}.\\
&[(V \; R \; V' \wedge \nu =_L \nu' \wedge \langle\!|C_i, \nu|\!\rangle \twoheadrightarrow \langle\!|W, \mu|\!\rangle)\\
&\quad \Rightarrow \exists W' \in \vec{Com}. \exists \mu' \in S.(\langle\!|C'_i, \nu'|\!\rangle \twoheadrightarrow \langle\!|W', \mu'|\!\rangle\\
&\qquad\qquad\qquad\qquad \wedge W R W' \wedge \mu =_L \mu')] \, .
\end{aligned} \tag{2.2}$$

Similar to the definition of R_I/R_O-security, a program V is defined as *strongly secure* iff $V \cong_L V$.

The following example shows how strong security on multithreaded programs can be seen as an instance of R_I/R_O-security.

Example 2.11. Observe that strong low-bisimulation only relates thread pools (1) with the same number of threads and (2) in which threads of equal indices are low-bisimilar. By assuming that scheduling-relevant information is visible to the observer, one can also express (1) and (2) in terms of R_I/R_O-equivalence. For this, recall that the output alphabet of the augmented transition system $\hat{M}_V$ is of the form $\Gamma'_V = Val^{Var} \times \mathbb{N} \times \mathbb{N}$, where the second and third components of $(v, n, i) \in \Gamma'_V$ represent the number of currently active threads and the index of the thread scheduled for execution, respectively. Note that (1) corresponds to observational equivalence with respect to an observer who can determine the number of threads in a program, that is, the second component of the output of $\hat{M}_V$. (2) corresponds to observational equivalence with respect to an observer who can determine the index of the thread that is scheduled for execution, that is, the third component of the output of $\hat{M}_V$. Hence, V is strongly secure if and only if $\hat{M}_V$ is $(\mathrm{All}_H \times Id_L)/((\mathrm{All}_H \times Id_L) \times Id_{\mathbb{N}} \times Id_{\mathbb{N}})$-secure. Note that this property is termination-sensitive even if $\hat{M}_V$ is input-enabled, as termination of V is signaled in terms of 0 active threads. $\diamondsuit$

2.5 Summary

In this chapter, we introduced transition systems as a general system model, and we showed how they can be specialized to both models of synchronous hardware and models of multithreaded programs. We also introduced R_I/R_O-security, a parametric flow property, and we showed how it can be instantiated to both strong security (for multithreaded programs) and a timing-sensitive notion of non-interference (for synchronous hardware). This underlying common basis allows us to use the algorithms from Chapter 3 for detecting both timing leaks in hardware and scheduling leaks in multithreaded programs with finite induced transition systems.

Chapter 3

Detecting Side-Channels

3.1 Introduction

A system has a side-channel if it exhibits insecure information flow. Detecting side-channels hence corresponds to deciding whether a system has secure information flow, which we model in terms of R_I/R_O-security. In this chapter, we give procedures for deciding the R_I/R_O-security of transition systems with finite alphabets and a finite number of states. For deterministic systems (i.e., for Mealy machines), we reduce deciding R_I/R_O-security to a reachability problem on a special kind of product Mealy machine. In the nondeterministic case, we reduce it to a generalization of the Partition Refinement Problem. Moreover, we show how the decision procedure for deterministic systems can be easily implemented using the symbolic model checker SMV. When given an insecure system, SMV produces a counterexample, that is, a sequence of input symbols that lead to distinguishable output. We report on experimental results with our prototype in Chapter 6, where we use it to analyze hardware implementations of cryptographic algorithms for their resistance to timing attacks.

3.2 Deterministic Case

We first reduce the problem of deciding the R_I/R_O-equivalence of states to a reachability problem on a special kind of product machine. This may seem surprising as, in general, information flow properties are properties of sets of traces rather than properties of individual traces [55]. The intuition behind our construction is that every sequence of inputs to the product machine corresponds to a pair of input sequences of the original machine. As R_I/R_O-equivalence is transitive, it suffices to analyze each individual input sequence of the product machine in order to establish R_I/R_O-security for the original system.

Definition 3.1. Let $M_i = (S_i, \Sigma, \Gamma, \delta_i, s_{0,i})$, with $i \in \{1,2\}$, be Mealy machines and let R_I and R_O be equivalence relations on Σ and Γ, respectively. The *R_I/R_O-product* $M_1 \times_{R_O}^{R_I} M_2$ is the Mealy machine $(S_1 \times S_2, R_I, \{0,1\}, \delta', (s_{0,1}, s_{0,2}))$, where

$$\begin{aligned}\delta' = \{((s_1,s_2),(a,b),\chi,(t_1,t_2)) \mid\ & a\ R_I\ b \wedge (\chi = \text{if } c\ R_O\ d \text{ then } 1 \text{ else } 0) \wedge \\ & (s_1,a,c,t_1) \in \delta_1\ \wedge\ (s_2,b,d,t_2) \in \delta_2\}\,.\end{aligned}$$

A *falsifying state* is a state with an outgoing transition labeled with 0, that is, where $c\ R_O\ d$ is violated. We now show that deciding the observational equivalence of states is equivalent to determining whether a falsifying state can be reached in $M \times_{R_O}^{R_I} M$.

Theorem 3.1. *Let $M = (S, \Sigma, \Gamma, \delta, s_0)$ be a Mealy machine, $R_I \subseteq \Sigma \times \Sigma$ and $R_O \subseteq \Gamma \times \Gamma$ equivalence relations, and let $s_1, s_2 \in S$. Then*

$$s_1 \simeq_{R_O}^{R_I} s_2 \Leftrightarrow \textit{no falsifying state is reachable from } (s_1,s_2) \textit{ in } M \times_{R_O}^{R_I} M\,.$$

Proof. ($\Rightarrow$) We show that no input $w \in (R_I)^*$ can trigger a transition labeled with 0. We proceed by induction on the length of w. The assertion is clear for $w = \epsilon$. Suppose now that $w = (a,b)w'$. As $s_1 \simeq_{R_O}^{R_I} s_2$ and δ is a function, there are unique transitions (s_1,a,c,t_1) and $(s_2,b,d,t_2) \in \delta$, with $t_1 \simeq_{R_O}^{R_I} t_2$ and $(c,d) \in R_O$. Hence $M \times_{R_O}^{R_I} M$ outputs 1 on this transition and we apply the induction hypothesis to (t_1,t_2) and w'.

($\Leftarrow$) We show that $\mathcal{Q} = \{(t_1,t_2) \mid (t_1,t_2) \text{ can be reached from } (s_1,s_2)\}$ fulfills (2.1) of Definition 2.5. Pick $(t_1,t_2) \in \mathcal{Q}$ and $(a,b) \in R_I$. Since δ is a function, there are unique transitions (t_1,a,c,t_1') and $(t_2,b,d,t_2') \in \delta$. Clearly, (t_1',t_2') can also be reached from (s_1,s_2) in $M \times_{R_O}^{R_I} M$ and, as no transition labeled with 0 can be triggered by assumption, $(c,d) \in R_O$ holds. Hence $\mathcal{Q}$ is contained in the union of all relations with (2.1) of Definition 2.5. □

Theorem 3.1 justifies a simple decision procedure for R_I/R_O-equivalence that is based on searching the R_I/R_O-product. We use breadth-first search, as it will find a shortest path to a falsifying state.

Corollary 3.1. *Let $M = (S, \Sigma, \Gamma, \delta, s_0)$ be a Mealy machine, let $s_1, s_2 \in S$, and let $R_I \subseteq \Sigma$ and $R_O \subseteq \Gamma$ be equivalence relations. Then $s_1 \simeq_{R_O}^{R_I} s_2$ can be decided in time $\mathcal{O}(|S|^2|R_I|)$, given the product $M \times_{R_O}^{R_I} M$.*

Proof. Breadth-first search can be implemented in time $\mathcal{O}(|V| + |E|)$ on a graph $G = (V, E)$. $M \times_{R_O}^{R_I} M$ has $|S|^2$ states and $|S|^2|R_I|$ transitions. This yields an $\mathcal{O}(|S|^2|R_I|)$ upper bound for the time complexity of deciding R_I/R_O-equivalence. □

By determining whether $s_0 \simeq_{R_O}^{R_I} s_0$ holds for the initial state s_0, Corollary 3.1 allows one to efficiently decide the R_I/R_O-security of a Mealy machine M.

3.3 Nondeterministic Case

A straightforward extension of the above reduction does not appear possible in the nondeterministic case. In a similar nondeterministic product machine, a falsifying transition shows only that both underlying transitions produce distinguishable output, which does not imply a violation of nondeterministic R_I/R_O-security. Instead, we use a partition refinement-based algorithm for deciding process equivalence [42] as a starting point. We introduce the more general Partial Partition Refinement problem, and we cast the problem of deciding R_I/R_I-equivalence of states in arbitrary finite-state transition systems as an instance. We then generalize the algorithm from [42] and apply it to decide R_I/R_O-equivalence.

The Partition Refinement Problem is, given a partition π of a set S and a property P of partitions of S, to find the coarsest refinement π' of π such that π' satisfies P. This is equivalent to finding the greatest equivalence relation $R_{\pi'}$, with $R_{\pi'} \subseteq R_\pi$, such that $R_{\pi'}$ satisfies P. Since R_I/R_O-equivalence is a partial equivalence relation, we need to generalize the Partition Refinement Problem. The *Partial Partition Refinement Problem* is, given a partial equivalence relation R and a property P, to find the coarsest refinement R' of R, such that R' satisfies P. We next show that the problem of deciding R_I/R_O-equivalence can be cast as an instance of this problem. Then, generalizing the ideas in [42], we compute this coarsest refinement as the maximal fixed point of a monotone mapping Θ.

A *partial partition* of a set S is a pair $\langle\{A_1, \ldots, A_n\}, C\rangle$, where the A_i are pairwise disjoint blocks with $\bigcup_{i=1}^n A_i \cup C = S$ and $\bigcup_{i=1}^n A_i \cap C = \emptyset$. There is a one-to-one correspondence between Pers R of a set S and partial partitions $\langle\{A_1, \ldots, A_n\}, C\rangle$, where the A_i correspond to the equivalence classes of R, and $C = S \setminus dom(R)$. As notation, we denote this correspondence as $\langle\{A_1, \ldots, A_n\}, C\rangle \triangleq \langle R, C\rangle$. Let $\pi_1 = \langle\{A_1, \ldots, A_n\}, C_1\rangle$ and $\pi_2 = \langle\{B_1, \ldots, B_m\}, C_2\rangle$ be partial partitions of S. We define $\pi_1 \leq \pi_2$ to hold whenever $C_1 \supseteq C_2$ and if every block of π_1 is contained in some block of π_2. The relation $\leq$ is a partial order on the set of all partial partitions of a set S. In fact, it is also a lattice when we define the meet $\sqcap$ as $\langle\{A_1,\dots,A_n\},C_1\rangle \sqcap \langle\{B_1,\dots,B_m\},C_2\rangle = \langle\{A_i \cap B_j \mid i \in \{1,\dots,n\},\ j \in \{1,\dots,m\}\},\ C_1 \cup C_2\rangle$

In the remainder of this subsection, let $M = (S,\Sigma,\Gamma,\delta,s_0)$ be a finite transition system, and let $R_I \subseteq \Sigma \times \Sigma$ and $R_O \subseteq \Gamma \times \Gamma$ be equivalence relations.

Definition 3.2. An *R_I/R_O-partition of S* is a partial partition $\langle\{A_1,\dots,A_n\},C\rangle$ of S, with

$$\begin{aligned}&\forall i,j \in \{1,\dots,n\}.\ \forall s_1,s_2 \in A_i.\ \forall (a_1,a_2) \in R_I.\ \forall x \in \Gamma/_{R_O}.\\ &\quad \overline{\delta}(s_1,a_1,x) \cap A_j \neq \emptyset \ \Leftrightarrow\ \overline{\delta}(s_2,a_2,x) \cap A_j \neq \emptyset \ \wedge\\ &\quad \overline{\delta}(s_1,a_1,x) \cap C \ =\ \overline{\delta}(s_2,a_2,x) \cap C \ =\ \emptyset,\end{aligned} \tag{3.1}$$

where $\overline{\delta}(s,a,x)$ denotes the set $\bigcup_{c\in x}\delta(s,a,c)$. A R_I/R_O-partition π of S is *maximal* if $\pi \geq \pi'$ holds for every R_I/R_O-partition π' of S.

We adapt (3.1) of Definition 3.2 to a mapping on partial partitions whose fixed points are precisely the R_I/R_O-partitions of S. To this end, let $\pi = \langle\{A_1,\dots,A_n\},C_1\rangle \mathrel{\hat{=}} \langle R_1, C_1\rangle$ be a partial partition of S. We define $\Theta(\pi) := \langle R_2, S \setminus dom(R_2)\rangle$, where $s_1\ R_2\ s_2$ if and only if

$$\begin{aligned}s_1\ R_1\ s_2 \ \wedge\ &\forall j \in \{1,\dots,n\}.\ \forall (a_1,a_2) \in R_I.\ \forall x \in \Gamma/_{R_O}.\\ &\quad \overline{\delta}(s_1,a_1,x) \cap A_j \neq \emptyset \ \Leftrightarrow\ \overline{\delta}(s_2,a_2,x) \cap A_j \neq \emptyset \ \wedge\\ &\quad \overline{\delta}(s_1,a_1,x) \cap C_1 \ =\ \overline{\delta}(s_2,a_2,x) \cap C_1 \ =\ \emptyset.\end{aligned}$$

Lemma 3.1. *Let $\langle R,C\rangle$ be a partial partition of the set of states S. Then the following are equivalent:*

1. *$\langle R,C\rangle$ is a fixed point of Θ.*
2. *$\langle R,C\rangle$ is a R_I/R_O-partition of S.*
3. *R satisfies (2.1) of Definition 2.5.*

Proof. (1. $\Rightarrow$ 2.) The assertion follows by setting $R_1 = R_2 = R$ in the definition of Θ, and observing that two states s_1 and s_2 relate in R whenever they are contained in the same set A_i of the corresponding partial partition.

(2. $\Rightarrow$ 3.) Let $s_1\ R\ s_2$, $a_1\ R_I\ a_2$, and $(s_1,a_1,c_1,s_1') \in \delta$. As $\overline{\delta}(s_1,a_1,[c_1]) \cap C = \emptyset$, we have $s_1' \in \overline{\delta}(s_1,a_1,[c_1]) \cap A$ for some equivalence class A of R. By hypothesis, we also have $\overline{\delta}(s_2,a_2,[c_1]) \cap A \neq \emptyset$, and hence there is a transition $(s_2,a_2,c_2,s_2') \in \delta$ with $c_1\ R_O\ c_2$ and $s_1'\ R\ s_2'$.

(3. ⇒ 1.) Let $\langle R, C\rangle \mathrel{\hat{=}} \langle\{A_1, \ldots, A_n\}, C\rangle$ and $\Theta(\langle R, C\rangle) = \langle R', C'\rangle$. It suffices to show that $R' = R$. The implication $R' \subseteq R$ follows directly from the definition of Θ. To show that $R \subseteq R'$, choose $s_1\ R\ s_2$ and $a_1\ R_I\ a_2$. If $\overline{\delta}(s_1, a_1, x) \cap A_j \neq \emptyset$, then there is a $(s_1, a_1, c_1, s_1') \in \delta$ with $c_1 \in x$ and $s_1' \in A_j$. As R satisfies (2.1) of Definition 2.5, there is also a $(s_2, a_2, c_2, s_2') \in \delta$, with $c_2 \in x$ and $s_2' \in A_j$. Hence $\overline{\delta}(s_2, a_2, x) \cap A_j \neq \emptyset$, and $R \subseteq R'$ follows. □

From Lemma 3.1, it follows that the relation $\simeq^{R_I}_{R_O}$ is a maximal fixed point of the function Θ. In particular, $\simeq^{R_I}_{R_O}$ itself satisfies (2.1) of Definition 2.5 and is thus contained in every maximal fixed point of Θ. Conversely, as every fixed point of Θ satisfies Property (2.1), the maximal fixed point is contained in $\simeq^{R_I}_{R_O}$, the union of all such relations.

The following theorem gives rise to a construction for maximal R_I/R_O-partitions.

Theorem 3.2. *There is a unique maximal R_I/R_O-partition π^* of S, namely, $\pi^* = \Theta^n(\langle\{S\}, \emptyset\rangle)$, for some $n \in \mathbb{N}$.*

Proof. We show that Θ is monotone with respect to $\leq$. Since the set of partial partitions of S is a complete lattice, it follows from the Knaster-Tarski fixed point theorem that a unique maximal fixed point of Θ exists (see, e.g., [95]). By Lemma 3.1, this fixed point is also a maximal R_I/R_O-partition.

For the monotonicity of Θ, consider the partial partitions $\pi_1 = \langle\{A_1, \ldots, A_n\}, C_1\rangle \mathrel{\hat{=}} \langle Q_1, C_1\rangle$ and $\pi_2 = \langle\{B_1, \ldots, B_m\}, C_2\rangle \mathrel{\hat{=}} \langle Q_2, C_2\rangle$, where $\pi_1 \leq \pi_2$. Furthermore, let $\Theta(\pi_1) = \langle Q_1', C_1'\rangle$ and $\Theta(\pi_2) = \langle Q_2', C_2'\rangle$. We need to show that $s_1\ Q_1'\ s_2$ implies $s_1\ Q_2'\ s_2$. Assume $s_1\ Q_1'\ s_2$. By the definition of Θ, this implies $s_1\ Q_1\ s_2$, which implies $s_1\ Q_2\ s_2$. Furthermore, for all $(a_1, a_2) \in R_I$, and for all $x \in \Gamma/_{R_O}$, we have $\overline{\delta}(s_1, a_1, x) \cap C_1 = \overline{\delta}(s_2, a_2, x) \cap C_1 = \emptyset$. As $C_1 \supseteq C_2$, we also have $\overline{\delta}(s_1, a_1, x) \cap C_2 = \overline{\delta}(s_2, a_2, x) \cap C_2 = \emptyset$. Finally, let $(a_1, a_2) \in R_I$ and $x \in \Gamma/_{R_O}$, and suppose $s_1' \in \overline{\delta}(s_1, a_1, x) \cap B_i$. s_1' is also contained in some $A_j \subseteq B_i$, as otherwise this would contradict the assumption $\overline{\delta}(s_1, a_1, x) \cap C_1 = \emptyset$. Then, as $s_1\ Q_1'\ s_2$, we also have $\overline{\delta}(s_2, a_2, x) \cap A_j \neq \emptyset$. Hence we conclude $\overline{\delta}(s_2, a_2, x) \cap B_i \neq \emptyset$. The proof that $\overline{\delta}(s_2, a_2, x) \cap B_i \neq \emptyset$ implies $\overline{\delta}(s_1, a_1, x) \cap B_i \neq \emptyset$ follows along the same lines and concludes the proof of the monotonicity of Θ.

As S is finite, the lattice of partial partitions of S is also finite and hence complete. The Knaster-Tarski fixed-point theorem guarantees the existence of a unique maximal fixed point π^*. We have $\Theta(\pi) \leq \pi$ for every partial partition π of S, and hence iteratively applying Θ to $\pi_\top = \langle\{S\}, \emptyset\rangle$ leads to the fixed point $\pi^* = \Theta^n(\pi_\top)$ after a finite number of steps n. □

Theorem 3.2 provides the basis of an algorithm for deciding the R_I/R_O-equivalence of states in polynomial time.

Corollary 3.2. *For two states $s_1, s_2 \in S$, we can decide $s_1 \simeq^{R_I}_{R_O} s_2$ in time*

$$\mathcal{O}(|S|^4 \cdot |R_I| \cdot |\Gamma/_{R_O}|)\,,$$

under the assumption that $\overline{\delta}(s,a,x) = \bigcup_{c \in x} \delta(s,a,c)$ can be determined in $\mathcal{O}(1)$ for all $s \in S$, $a \in \Sigma$, and $x \in \Gamma/_{R_O}$.

Proof. It suffices to show that a single application of Θ can be computed in time $\mathcal{O}(|S|^3 \cdot |R_I| \cdot |\Gamma/_{R_O}|)$. Due to Theorem 3.2, a fixed point can be obtained by iteratively applying Θ. As $\Theta(\pi) \leq \pi$ for every partition π, this process terminates within at most $|S|$ applications.

We assume $S = \{s_1, \dots, s_n\}$ and that the equivalence class of each state is given by a representative s_i with minimal i, and by a distinguished symbol $* \notin S$ if the state is outside the domain of the relation. For example, in the case of $\pi_\top = \langle \{S\}, \emptyset \rangle$, the canonical representative for every state is s_1. Suppose now we are given a partial partition $\pi = \langle R, C \rangle$ and we want to compute $\Theta(\pi) = \langle R', C' \rangle$. To decide whether two states s_i and s_j relate in R', we perform the following procedure: for all $(a_1, a_2) \in R_I$, and for all $x \in \Gamma/_{R_O}$, we compare the corresponding sets of R-equivalence classes of the target states. If all of the corresponding sets coincide, s_i and s_j are in the same R'-equivalence class. By iterating i stepwise from 1 to n, we perform this check for every $j \in \{1, \dots, n\}$. Under this ordering, the canonical representative of the R'-equivalence class of each s_j is the s_i with minimal index such that equivalence of s_i and s_j can be established, and $*$ if there is no such s_i. In this way, each application of Θ can be computed in time $\mathcal{O}(|S|^3 \cdot |R_I| \cdot |\Gamma/_{R_O}|)$. $\square$

Similar to the deterministic case, Corollary 3.2 allows one to efficiently decide the R_I/R_O-security of a Mealy machine M by determining whether $s_0 \simeq^{R_I}_{R_O} s_0$ holds for the initial state s_0.

3.4 An Implementation

In this section, we present an implementation of our algorithm for deciding the R_I/R_O-security of deterministic systems. Instead of implementing the search procedure from Corollary 3.1 by hand, we use a temporal logic model-checker for this task. Temporal

```
1.  MODULE main
2.  VAR
3.   lo,hi1,hi2 : array (n-1)..0 of boolean;
4.   sys1 : system;
5.   sys2 : system;

6.  ASSIGN
7.   sys1.lo_in:=lo;
8.   sys1.hi_in:=hi1;
9.   sys2.lo_in:=lo;
10.  sys2.hi_in:=hi2;

11. SPEC !EF(!sys1.lo_out=sys2.lo_out)
```

Figure 3.1: Product construction in SMV

logic model-checking allows for the verification whether a transition system satisfies a specification given in temporal logic [23]. In the model-checker SMV, the verification is based on the efficient manipulation of binary decision diagrams (BDD) and allows systems with very large state spaces to be verified. By implementing our search procedure in SMV, we make use of these sophisticated techniques and data structures. The use of alternative temporal logic model-checkers such as SPIN [41], which relies on the lazy explicit enumeration of states, is straightforward. A detailed investigation of which model-checking technique is more adequate for the class of systems considered in this thesis, e.g. along the lines of [12], is beyond the scope of this work.

Figure 3.1 lists an SMV-fragment that encodes the product construction $\times^{\mathrm{All}_H\times Id_L}_{\mathrm{All}_H\times Id_L}$ of Definition 3.1. The system to be analyzed is an arbitrary module (a SMV-specification of a transition system) that we call `system` and that reads input Σ_L and Σ_H from variables `lo_in` and `hi_in`, respectively, and that writes output to the variables `lo_out` and `hi_out`. In our example $\Sigma_L = \Sigma_H = \Gamma_L = \Gamma_H = \{0,1\}^n$ for some $n \in \mathbb{N}$. For building the product, we instantiate `system` twice in lines 4 and 5. Both instances, `sys1` and `sys2`, are provided with the same low input `lo` (as specified by Id_{Σ_L}), and are provided with combinations of high inputs `hi1` and `hi1`, respectively (as specified by All_{Σ_H}). This is implemented in lines 7-10. In fact, all such input combinations are considered, as no assignments are made to the variables `lo`, `hi1`, and `hi2`.

Reachability of a falsifying state of the product corresponds to a violation of the

CTL-formula `!EF(!sys1.lo_out=sys2.lo_out)` in line 11. If we reach a state in which the `lo_out` variables of the two instances of `system` differ, then we have found a falsifying state of the product. In this case, SMV computes a counterexample, namely, two $\mathrm{All}_H \times Id_L$-equivalent input sequences that lead to distinguishable low output.

We will report on experimental results in Section 6.3, where we use this implementation to detect (and prove the absence of) timing leaks in GEZEL-implementations of algorithms for integer multiplication and finite field exponentiation, and where we discuss its performance.

3.5 Summary

We presented algorithms and complexity bounds for deciding R_I/R_O-security for both deterministic and nondeterministic finite-state transition systems. Our decision procedure for nondeterministic systems can be applied to, e.g., nondeterministic models of hardware and for deciding strong security of multithreaded programs with finite induced transition systems. We showed how our decision procedure for the security of deterministic systems can be encoded in the model-checker SMV. Our experimental results in Section 6.3 show that this prototype implementation is a powerful tool for accurately analyzing the information flow in synchronous hardware, which has been an open problem until now.

Chapter 4

Quantifying Side-Channels

4.1 Introduction

In this chapter, we develop a method for quantifying the amount of secret information that an attacker can extract from a system's side-channels.

As was shown in Chapter 3, the detection of side-channels in deterministic systems corresponds to deciding whether an adversary can distinguish system runs with different secret inputs. For this, it suffices to consider passive attackers, i.e., mere observers of the system. In a number of documented side-channel attacks, however, the attacker is more powerful, in the sense that he can interactively provide input to the system. In this way, he can partially control the system and direct its behavior towards maximizing the leakage of confidential information. A meaningful quantitative measure must take these interactions into account. In particular, it must allow for reasoning about their number, as the attacker's interactions with the system are often expensive or limited. In this chapter, we present such a measure and show how it can be computed for a given system.

For this, we first give a definition of attack strategies, which are explicit representations of the adaptive decisions made by an attacker during attacks. We combine attack strategies with information-theoretic entropy measures; this allows us to express the attacker's expected uncertainty about the secret after he has performed a side-channel attack following a given strategy. By quantifying over all attack strategies of a fixed length n, we express what attackers can, in principle, achieve in n attack steps. We use this to define a function Φ that gives a lower bound on the expected uncertainty about the secret as a function of the number of side-channel measurements. Since the bounds given by Φ are information-theoretic, they hold for any kind of analysis technique that a computationally unbounded attacker might apply to analyze the measurements. Note that such strong bounds are realistic. In template attacks [21], the entire information contained in each measurement is effectively exploited for the recovery of the secret.

We give algorithms and (exponential) complexity bounds for computing Φ. Furthermore, we propose two heuristic techniques that reduce this complexity and thereby allow us to estimate the vulnerability of systems of a size for which the direct computation of Φ is infeasible.

Our approach is parametric in the physical characteristics of the side-channel, which can be described by deterministic hardware models of the target system. In this way, the accuracy of our method depends only on the accuracy of the system model used. Furthermore, our approach accommodates different notions of entropy that correspond to different kinds of brute-force guessing.

Finally, we have implemented our approach in the functional programming language HASKELL [14]. We report on experimental results using the resulting prototype in Chapter 6.4, where we analyze hardware implementations of cryptographic algorithms for their resistance to timing attacks.

4.2 A Timing Attack

To motivate our formal model of attack strategies, we briefly sketch a timing attack against modular exponentiation. We choose the attack from [31] for its simplicity.

Consider again the simple square-and-multiply modular exponentiation algorithm from Example 1.1. Assume an implementation that computes $*$ with Montgomery's algorithm [61] for modular multiplication, which is a common choice in practice. An important property of Montgomery multiplication is that it is typically performed fast and in approximately constant time. For some input values, however, so-called Montgomery reductions have to be performed, which are very time-consuming operations.

Assume an attacker who has knowledge about this implementation and who can measure the algorithm's running times on the target system. Furthermore, assume that he can trick the target system into decrypting arbitrary ciphertexts c (as, e.g., in [15]).

Under these assumptions, the attacker can extract the secret key k as follows. Suppose he already knows key bits $k_{p-1}, \dots, k_i$. For determining the $i-1$st bit, he randomly chooses a set A of possible inputs (i.e., ciphertexts) to the algorithm. Computing the loop for key bits $p-1, \dots, i$ on his private machine, he partitions A into two disjoint sets A_1 and A_2: A_1 is the subset of inputs for which a Montgomery reduction has to be carried out when computing x ∗ c in line 5 of the $p - i$th pass of the loop. A_2 is the set of initial values of c for which no such reduction is necessary. The attacker now uses the target system for decrypting all inputs from A and measures the corresponding running times. If the average running time for inputs from A_1 is higher than that for inputs from A_2, the attacker concludes that the target system indeed computes x ∗ c in the $p - i$th pass of the loop – and hence the $p - i$th key bit must be 1.[1] In this way, he can successively extract the entire key from the system.

Note that this particular attack is not adaptive, as the attacker chooses his inputs at random. However, nothing in the attack scenario prevents the attacker from optimizing his queries with respect to previously revealed side-channel information. This must be taken into account when giving bounds on what attackers can, in principle, achieve in a side-channel attack.

4.3 Attack Strategies

In this chapter, we present our model of attack strategies. It is based on an explicit representation of the side-channel to be analyzed, which allows for a simpler presentation. We will point out the precise connections to the system models and security notions from Chapter 3.

4.3.1 Attackers and Side-Channel Measurements

Attack Scenario. Let Σ_H be a finite set of secrets, Σ_L be a finite set, and D be an arbitrary set. We consider cryptographic functions of type $F : \Sigma_H \times \Sigma_L \to D$, where we assume that F is invoked by two collaborating callers. One caller is an honest agent that provides a secret argument $h \in \Sigma_H$ and the other caller is a malicious agent (the *attacker*) that provides the argument $l \in \Sigma_L$. Examples of F are encryption and decryption functions and MACs. We assume that the attacker has no access to the values of h and $F(h, l)$, but that he can subject F's implementation I_F to side-channel measurements.[2] Typically, the secret h is a long-term secret such as a key, which remains constant during different calls to F. The malicious agent performs an attack in order to gather information to deduce h or narrow down its possible values. Such an *attack*

[1]Note that this rationale relies on the assumption that all other timing variations, such as Montgomery reductions in other passes of the loop, manifest as random noise.

[2]For many cryptographic functions F (for example, the one-time pad), the knowledge of $F(h, l)$ and l determines h, which would yield trivial information-theoretic bounds.

consists of a sequence of *attack steps*, each with two parts: a *query* phase, in which the attacker decides on an input l and sends it to the system, and a *response* phase, in which he observes I_F while it computes $F(h, l)$. The attack is *adaptive* if the attacker can use the observations made during the first n steps to choose the query for the n+1st step. An attack ends if either the honest agent changes the secret (assuming the independence of the old and new secrets) or if the attacker stops querying the system.

Explicit Models of Side-Channels. We assume that the attacker can make one side-channel measurement per invocation of the function F and that no errors occur in the measurement. We assume that the attacker has full knowledge about the implementation I_F. Furthermore, we assume that the attacker's side-channel measurements are independent of I_F's internal state and that they depend only on the values of the input to I_F. Given our assumptions, an *(explicit) side-channel* is a function $f_{I_F} : \Sigma_H \times \Sigma_L \to O$, where O is the set of possible observations, and we assume that f_{I_F} is known to the attacker. We will usually leave I_F implicit and abbreviate f_{I_F} as f.

Example 4.1. Suppose that F is implemented in synchronous (clocked) hardware and that the attacker is able to determine I_F's running times up to single clock cycles. Then the timing side-channel of I_F can be modeled as a function $f : \Sigma_H \times \Sigma_L \to \mathbb{N}$ that represents the number of clock ticks consumed by an invocation of F. $\Diamond$

If the function f accurately models the side-channel, then any randomness in a physical attacker's measurements is due to noise and the assumption of error-free measurements is a safe worst-case characterization of the attacker's capabilities.

Although one can take this direct approach to modeling side-channels, they can also be given in terms of the Mealy machine models from Chapter 2.3.

Example 4.2. If infinite system runs are considered, a triggered Mealy machine $M = (S, \Sigma, \Gamma, \delta, s_0)$ defines a function $f_M : \Sigma \to \Gamma^\omega$, where $\Gamma^\omega = \mathbb{N} \to \Gamma$ and where $f_M(a)$ denotes the infinite sequence of outputs produced by M when it is provided with input a in its initial state. We lift an equivalence relation $R_O \subseteq \Gamma \times \Gamma$ to an equivalence relation on Γ^ω by defining $u \; R_O \; v$ for $u, v \in \Gamma^\omega$ if and only if $u(i) \; R_O \; v(i)$ for all $i \in \mathbb{N}$. Suppose now $\Sigma = \Sigma_H \times \Sigma_L$. The function $f_M^{R_O} : \Sigma_H \times \Sigma_L \to \Gamma^\omega / R_O$ given by $f_M^{R_O}(h, l) = [f_M(h, l)]_{R_O}$ defines the explicit side-channel of M with respect to an attacker with observational capabilities given by R_O. $\Diamond$

Example 4.2 connects the notion of side-channels from Chapter 2 with the definition of explicit side-channels given in this chapter. This connection will allow us to give an interpretation of R_I/R_O-security in terms of the quantitative model we will present in Section 4.4.

As the next example shows, f can also be derived from the implementation I_F.

Example 4.3. Suppose a hardware implementation I_F of F is given. As in template attacks [21], average values of I_F's time consumption for fixed input values h and l can be used to define $f(h, l)$. $\Diamond$

As in template attacks, the attacker can use noise models of the target implementation to extract the maximal information from his measurements, that is, the value of f.

Finally, it is straightforward to model that the attacker has (partial) knowledge of the values of F.

Example 4.4. Consider a function $F : \Sigma_H \times \Sigma_L \to D$, with $D = \{0, 1\}^n$ for some $n \in \mathbb{N}$, an arbitrary side-channel $f : \Sigma_H \times \Sigma_L \to O$ of I_F, and a function $g : D \to E$ mapping to an arbitrary set E. By interpreting the function $(g \circ F, f) : \Sigma_H \times \Sigma_L \to E \times O$ defined as $(g \circ F, f)(h, l) = (g(F(h, l)), f(h, l))$ as the side-channel, one can model that the attacker has access to the information given by f and $g \circ F$. Possible instantiations of g include the identity, projections to components of D, and the Hamming weight of values in D. $\Diamond$

4.3.2 Formalizing Attack Strategies

An adaptive attacker chooses his queries with the knowledge of previously revealed side-channel information. We use trees to define attack strategies, which capture these adaptive choices. Subsequently, we also formalize non-adaptive attacks, that is, attacks in which the attacker gathers all side-channel information before performing any analysis. To begin with, we motivate an abstract view of attack steps, which is the key to the simplicity of our model.

Attacker's Choices and Knowledge. During the query phase, the attacker decides which input $l \in \Sigma_L$ to query the system with. In the response phase, he learns the value $f(h, l)$. In general, he cannot deduce h from $f(h, l)$. What he can deduce, though (assuming full knowledge about the implementation I_F and unbounded computational power), is the set of secrets that are coherent with the observation $f(h, l)$. Namely, assuming a fixed f, we say that a secret h is *coherent with* $o \in O$ *under* $l \in \Sigma_L$ whenever

$f(h,l) = o$ holds. Two secrets h and r are *indistinguishable under* l iff $f(r,l) = f(h,l)$. Note that for every $l \in \Sigma_L$, *indistinguishability under* l is an equivalence relation on Σ_H. For every $o \in O$, the set of secrets that are coherent with o under l forms an equivalence class of indistinguishability under l. The set of secrets that are coherent with the attacker's observation under the attacker's input is the set of secrets that could possibly have led to this observation; we use this set to represent the attacker's knowledge about the secret after an attack step. In this way, a function $f : \Sigma_H \times \Sigma_L \to O$ gives rise to a set of partitions $\mathcal{P}_f = \{\pi_l \mid l \in \Sigma_L\}$, where π_l is the partition induced by indistinguishability under l.

In terms of the set of partitions $\mathcal{P}_f$, the two phases of an attack step can be described as follows:

1. In the query phase, the attacker chooses a partition $\pi \in \mathcal{P}_f$.
2. In the response phase, the system reveals the block $B \subseteq \pi$ that contains h.

Conversely, given a set of partitions $\mathcal{P}$, one can easily define a (non-unique) function f, with $\mathcal{P}_f = \mathcal{P}$. In this sense, the partition-based and the functional viewpoints are equivalent. Formalizing f in terms of $\mathcal{P}_f$ only abstracts from the concrete values that f takes, which are irrelevant for assessing the information that is revealed by f. For clarity of presentation, we will subsequently focus on the partition-based viewpoint and generalize from single attack steps to entire attacks.

Attack Strategies as Trees. To model adaptive attacks, we proceed as follows. We assume a fixed set of partitions $\mathcal{P}$ of Σ_H and we use a tree whose vertices are labeled with subsets of Σ_H for capturing the attacker's decisions with respect to his possible observations. In this tree, an attack step is represented by a node together with its children. The label A of the parent node is the set of secrets that are coherent with the attacker's observation at this point; hence it represents the basis for the attacker's decision. The labels of the children form a partition of that set. We require that this partition be of the form $A \cap P$ for some $\pi \in \mathcal{P}$. This corresponds to the attacker's choice of a query. By observing the system's response, the attacker learns which successor's block actually contains the secret. This node is the starting point for subsequent attack steps. With this formalization of an attack strategy, an actual attack corresponds to a path from the root in this tree.

Example 4.5. Let $\Sigma_H = \{1,2,3,4\}$ and consider the set of partitions $\mathcal{P} = \{\{\{1\},\{2,3,4\}\}, \{\{1,2\},\{3,4\}\}, \{\{1,2,3\},\{4\}\}\}$ of Σ_H. Suppose the attacker picks $\{\{1,2\},\{3,4\}\}$ as his

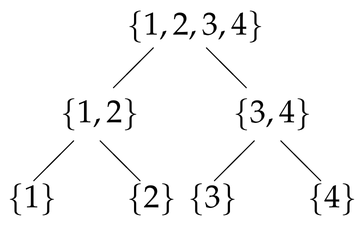

Figure 4.1: An attack strategy

first query. If the system responds $\{1,2\}$, the attacker chooses $\{\{1\},\{2,3,4\}\}$ as his next query. Otherwise, he chooses $\{\{1,2,3\},\{4\}\}$. In this way, he can determine any secret within two steps. The corresponding attack strategy is depicted in Figure 4.1. $\Diamond$

Formally, let $T = (V, E)$ be a tree with nodes V and edges $E \subseteq V \times V$. For every node $v \in V$, we denote the set of its successors as $succ(v) = \{w \mid (v, w) \in E\}$. The *height* of a tree T is the length of a longest path in T.

Definition 4.1. Let $\mathcal{P}$ be a set of partitions of Σ_H. An *attack strategy against* $\mathcal{P}$ is a triple (T, r, λ), where $T = (V, E)$ is a tree, $r \in V$ is the root, and $\lambda : V \to 2^{\Sigma_H}$ is a node labeling with the following properties:

1. $\lambda(r) = \Sigma_H$, and

2. for every $v \in V$, there is a $\pi \in \mathcal{P}$ with $\lambda(v) \cap \pi = \{\lambda(w) \mid w \in succ(v)\}$.

An attack strategy is of *length* k if T has height k. An *attack* is a path $(r, \ldots, t)$ from the root r to a leaf t of T.

Requirement 1 of Definition 4.1 expresses that, a priori, every secret in Σ_H is possibly chosen by the honest agent. Requirement 2 expresses that the labels of the children of each node form a partition of their parent's label and that this partition is obtained by intersecting the label with a $\pi \in \mathcal{P}$. A simple consequence of requirements 1 and 2 is that the labels of the leaves of an attack strategy partition the label of the root node. This leads to the following definition.

Definition 4.2. The partition *induced* by the attack strategy $\mathfrak{a} = (T, r, \lambda)$ is the set $\{\lambda(v) \mid v \text{ is a leaf of } T\}$, which we denote by $\pi_{\mathfrak{a}}$. We define the set of secrets that are *coherent with an attack* $a = (r, \ldots, t)$ as $\lambda(t)$.

Observe that this definition of coherence corresponds to our prior definition considering attacks (r, t) of length 1: the secrets that are coherent with an observation o under l form the block $\lambda(t)$ that the system reveals when queried with π_l.

To clearly distinguish between adaptive and non-adaptive attacks, we briefly describe how the latter can be cast in our model.

4.3.3 Non-adaptive Attack Strategies

An attack strategy is called *non-adaptive* if the attacker does not have access to the system's responses until the end of the attack. Thus, when choosing an input, he cannot take into account the outcomes of his previous queries. In our model, this corresponds to the attacker choosing the same partition in all nodes at the same level of the attack strategy.

Formally, the *level* of a node $v \in V$ in an attack strategy $\mathfrak{a} = (T, r, \lambda)$, with $T = (V, E)$, is the length of the path from the root r to v. A tree is *full* if all leaves have the same level.

Definition 4.3. An attack strategy $\mathfrak{a} = (T, r, \lambda)$ is *non-adaptive* iff T is full and for every level i there is a $\pi_i \in \mathcal{P}$ such that $\lambda(v) \cap \pi_i = \{\lambda(w) \mid w \in succ(v)\}$, for every v of level i.

Note that we require the tree to be full to exclude observation-dependent termination of attacks. The structure of non-adaptive attacks is simpler than that of adaptive attacks and we can give explicit representations of the partitions induced.

Proposition 4.1. *Let $\mathfrak{a}$ be a non-adaptive attack strategy of length k against $\mathcal{P}$. Then we have*

$$\pi_{\mathfrak{a}} = \bigcap_{i=0}^{k-1} \pi_i \, ,$$

where $\pi_i \in \mathcal{P}$ is the partition chosen at level $i \in \{0, \ldots, k-1\}$ of $\mathfrak{a}$.

Proof. We prove the assertion by induction on the length k of $\mathfrak{a} = (T, r, \lambda)$. If $k = 0$, we have $\pi_{\mathfrak{a}} = \lambda(r) = \Sigma_H = \bigcap \emptyset$. If $k > 0$, consider the full subtree T' of height $k-1$ of T. We have $\pi_{\mathfrak{a}} = \{\lambda(w) \mid w \text{ is a leaf of } T\} = \bigcup_v \{\lambda(w) \mid w \in succ(v)\}$, where v ranges over the leaves of T'. According to Definition 4.3 and the induction hypothesis, we conclude $\pi_{\mathfrak{a}} = \bigcup_v \lambda(v) \cap \pi_{k-1} = \bigcap_{i=0}^{k-2} \pi_i \cap \pi_{k-1} = \bigcap_{i=0}^{k-1} \pi_i$. $\square$

Observe that, since $\cap$ is commutative, the order of the queries is irrelevant. This is coherent with the intuitive notion of a non-adaptive attack, as the side-channel information is only analyzed when the attack has finished.

4.3.4 Attack Strategies and R_I/R_O-Security

We next establish a formal connection between R_I/R_O-security and attack strategies. For this, consider a triggered Mealy machine $M = (S, \Sigma_H \times \Sigma_L, \Gamma, \delta, s_0)$, an equivalence relation $R_O \subseteq \Gamma \times \Gamma$, and let $f = f_M^{R_O}$ be M's side-channel with respect to R_O. Furthermore, let $\mathcal{P}_f$ be the set of partitions of Σ_H that is induced by f.

Proposition 4.2. *If M is $R \times Id_L/R_O$-secure for an equivalence relation $R \subseteq \Sigma_H \times \Sigma_H$, then $\pi_R \sqsubseteq \pi_{\mathfrak{a}}$ for all attack strategies $\mathfrak{a}$ against $\mathcal{P}_f$.*

Proof. Choose an arbitrary $l \in \Sigma_L$ and arbitrary $h_1, h_2 \in \Sigma_H$ with $h_1 \; R \; h_2$. M is $R \times Id_L/R_O$-secure, hence $f(h_1, l) = f(h_2, l)$. That is, h_1 and h_2 are indistinguishable under l. In terms of partitions, this implies $\pi_R \sqsubseteq \pi_l$. As l was chosen freely, $\pi_R \sqsubseteq \bigcap_{\pi \in \mathcal{P}_f} \pi$ follows. $\bigcap_{\pi \in \mathcal{P}_f} \pi \sqsubseteq \pi_{\mathfrak{a}}$ holds for every attack strategy $\mathfrak{a}$ against $\mathcal{P}_f$, hence the assertion follows from the transitivity of $\sqsubseteq$. $\square$

Proposition 4.2 implies that an attacker cannot deduce more than the R-equivalence class of the secret input of a triggered and $R \times Id_L/R_O$-secure Mealy machine, independently of the attack strategy he follows.

Example 4.6. Let M be a triggered Mealy machine with $\Sigma_H = \{0,1\}^n$ and let $\Psi = \{(a,b) \in \Sigma_H \times \Sigma_H \mid \|a\| = \|b\|\}$, where $\|x\|$ denotes the Hamming weight of x. If M is $\Psi \times Id_L/\mathrm{All}_L \times Id_L$-secure, then any adaptive attack does not reveal more than the Ψ-equivalence class of the high input to M, which corresponds to the high input's Hamming weight. If M is $\mathrm{All}_H \times Id_L/\mathrm{All}_L \times Id_L$-secure (i.e., NI-secure), then any adaptive attack does not reveal more than the All_H-equivalence class of the input to M, which corresponds to no information gain about the secret input. $\lozenge$

In untriggered and $R \times Id_L/R_O$-secure Mealy machines, the R-equivalence class of every input symbol may be revealed to the observer.

Example 4.7. Consider a Mealy machine M with a single state s_0, alphabets $\Sigma_H = \Sigma_L = \Gamma = \{0,1\}$, and transitions $\{(s_0, (h,l), h, s_0) \mid h, l \in \{0,1\}\}$. Note that M maps every high input h to the identical low output. However, M is $\Psi \times Id_L/Id_\Gamma$-secure, where Ψ is defined as in Example 4.6. This is due to the fact that the Ψ-equivalence classes of Σ_H are singleton and illustrates that, in an untriggered system, $R \times Id_L/R_O$-security may still allow for the flow of arbitrary information to the observer. $\lozenge$

Proposition 4.2 shows how the techniques introduced in Chapter 3 can be used to provide bounds on what an adaptive attacker can learn about the secret input. However, these bounds hold against attack strategies of arbitrary length and they cannot be used to reason about the attacker's effort (in terms of measurements) to obtain this information.

In the next section, we will extend our model of attack strategies presented with measures for their quantitative evaluation. Afterwards, we use this quantitative model to give bounds on what attackers can possibly achieve within a given number of attack steps.

4.4 Quantitative Evaluation of Attack Strategies

In Section 4.3, we used the induced partition $\pi_{\mathfrak{a}}$ to represent what an attacker learns about the secret by following an attack strategy $\mathfrak{a}$. Intuitively, the attacker obtains more information (or equivalently, reduces the uncertainty) about the secret as $\pi_{\mathfrak{a}}$ is refined. Information-theoretic entropy measures can be used to express this remaining uncertainty. Focusing on the remaining entropy instead of the attacker's information gain provides a concrete, meaningful measure that quantifies the attacker's effort for the recovery of the secret by brute-force guessing under the worst-case assumption that he can actually determine the set of secrets that are coherent with his observations during the attack. The viewpoints are informally related by the equation *initial uncertainty = information gain + remaining uncertainty*, which we will make explicit below.

4.4.1 Measures of Uncertainty

We now introduce three entropy measures, which correspond to different notions of resistance against brute-force guessing. Presenting these different measures serves two purposes. First, it accommodates the fact that different types of guesses and different notions of success for brute-force guessing correspond to partially incomparable notions of entropy [54, 18, 66]. Second, it demonstrates how the possibilistic model presented in Section 4.3 can serve as a basis for a variety of probabilistic extensions.

In the following, assume a probability measure p is given on Σ_H and is known to the attacker. For a random variable $X : \Sigma_H \to \mathcal{X}$ with range $\mathcal{X}$, we define $p_X : \mathcal{X} \to \mathbb{R}$ as $p_X(x) = \sum_{h \in X^{-1}(x)} p(h)$, which in the literature is often denoted by $p(X = x)$. For a partition π of Σ_H, there are two variables of particular interest. The first is the random variable U that models the random choice of a secret in Σ_H according to p (i.e., $U = id_{\Sigma_H}$, where id is the identity function). The second is the random variable V_P that represents the choice of the enclosing block (i.e., $V_\pi : \Sigma_H \to \pi$, where $h \in V_\pi(h)$). For an attack strategy $\mathfrak{a}$, we abbreviate $V_{\pi_{\mathfrak{a}}}$ by $V_{\mathfrak{a}}$.

Shannon Entropy. The *(Shannon) entropy* [81] of a random variable $X : \Sigma_H \to \mathcal{X}$ is defined as

$$H(X) = -\sum_{x \in \mathcal{X}} p_X(x) \log_2 p_X(x) \; .$$

The entropy is a lower bound for the average number of bits required to represent the results of independent repetitions of the experiment associated with X. Thus, in terms of guessing, the entropy $H(X)$ is a lower bound for the average number of binary questions that need to be asked to determine X's value [18].

Given another random variable $Y : \Sigma_H \to \mathcal{Y}$, one denotes by $H(X|Y = y)$ the entropy of X given $Y = y$, that is, with respect to the distribution $p_{X|Y=y}$. The *conditional entropy* $H(X|Y)$ of X given Y is defined as the expected value of $H(X|Y = y)$ over all $y \in \mathcal{Y}$, namely,

$$H(X|Y) = \sum_{y \in \mathcal{Y}} p_Y(y) H(X|Y = y) \; .$$

Entropy and conditional entropy are related by the equation $H(XY) = H(Y) + H(X|Y)$, where XY is the random variable defined by $XY(x) = (X(x), Y(x))$.

Consider now an attack strategy $\mathfrak{a}$ and the corresponding variables U and $V_{\mathfrak{a}}$. $H(U)$ is the attacker's initial uncertainty about the secret and $H(U|V_{\mathfrak{a}} = B)$ is the attacker's remaining uncertainty about the secret after learning the secret's enclosing block $B \in \pi_{\mathfrak{a}}$. $H(U|V_{\mathfrak{a}})$ is the attacker's expected remaining uncertainty about the secret after performing an attack with strategy $\mathfrak{a}$. As the value of $V_{\mathfrak{a}}$ is determined by that of U, we have $H(UV_{\mathfrak{a}}) = H(U)$. The equation $H(U) = H(V_{\mathfrak{a}}) + H(U|V_{\mathfrak{a}})$ is the formal counterpart of the informal equation given at the start of this section.

Guessing Entropy. The guessing entropy of a random variable X is the average number of questions of the kind "does $X = x$ hold" that must be asked to guess X's value correctly [54].

As we assume p to be public, the optimal procedure is to try each of the possible values in order of their decreasing probabilities. W.l.o.g., let $\mathcal{X}$ be indexed such that $p_X(x_i) \geq p_X(x_j)$, whenever $i \leq j$. Then the *guessing entropy* $G(X)$ of X is defined as $G(X) = \sum_{1 \leq i \leq |\mathcal{X}|} i \, p_X(x_i)$. Analogously to the conditional Shannon entropy, the *conditional guessing entropy* $G(X|Y)$ is defined as

$$G(X|Y) = \sum_{y \in \mathcal{Y}} p_Y(y) G(X|Y = y) \; .$$

$G(X|Y)$ represents the expected number of guesses needed to determine X when the value of Y is already known.

Hence, $G(U|V_{\mathfrak{a}})$ is a lower bound on the expected number of off-line guesses that an attacker must still perform to recover the secret after having carried out a side-channel attack with strategy $\mathfrak{a}$.

Marginal Guesswork. For a fixed $\alpha \in [0,1]$, the marginal guesswork of a random variable X quantifies the number of questions of the kind "does $X = x$ hold" that must be asked until X's value is correctly determined with a chance of success given by α [66].

Again, w.l.o.g. let $\mathcal{X}$ be indexed such that $p_X(x_i) \geq p_X(x_j)$, whenever $i \leq j$. Then the (α)-*marginal guesswork* of X is defined as

$$W_\alpha(X) = \min\{j \mid \sum_{1\leq i\leq j} p_X(x_i) \geq \alpha\} .$$

We define the *conditional marginal guesswork* $W_\alpha(X|Y)$ analogously to the conditional entropy. As before, $W_\alpha(U|V_{\mathfrak{a}})$ is a lower bound on the expected number of guesses that an attacker needs to perform in order to determine the secret with a success probability of more than α after having carried out a side-channel attack with strategy $\mathfrak{a}$.

Uniform Distributions. If p is uniformly distributed, simple explicit formulae for the entropy measures presented so far can be derived.

Proposition 4.3. *Let $\mathfrak{a}$ be an attack strategy with $\pi_{\mathfrak{a}} = \{B_1, \dots, B_r\}$, $|B_i| = n_i$, and $|\Sigma_H| = n$. If p is uniformly distributed, then*

1. $H(U|V_{\mathfrak{a}}) = \frac{1}{n}\sum_{i=1}^{r} n_i \log n_i$,
2. $G(U|V_{\mathfrak{a}}) = \frac{1}{2n}\sum_{i=1}^{r} n_i^2 + \frac{1}{2}$, *and*
3. $W_\alpha(U|V_{\mathfrak{a}}) = \frac{1}{n}\sum_{i=1}^{r} n_i \lceil \alpha n_i \rceil$.

Proof.
1. The entropy of a uniformly distributed random variable with finite range $\mathcal{X}$ is given by $\log_2 |\mathcal{X}|$ (see, e.g. [6]). Consequently, $H(U|V_{\mathfrak{a}} = B_i) = \log_2 n_i$ and $H(U|V_{\mathfrak{a}}) = \sum_{i=1}^{r} \frac{n_i}{n} H(U|V_{\mathfrak{a}} = B_i) = \frac{1}{n}\sum_{i=1}^{r} n_i \log n_i$.
2. We have $G(U|V_{\mathfrak{a}}) = \sum_{i=1}^{r} \frac{n_i}{n} G(U|V_{\mathfrak{a}} = B_i) = \sum_{i=1}^{r} \frac{n_i}{n} \sum_{j=1}^{n_i} j \frac{1}{n_i} = \frac{1}{n}\sum_{i=1}^{r} \frac{(n_i+1)n_i}{2} = \frac{1}{2n}\sum_{i=1}^{r} n_i^2 + \frac{1}{2}$.

3. The assertion follows from $W_\alpha(U|V_\mathfrak{a}) = \sum_{i=1}^{r} \frac{n_i}{n} W_\alpha(U|V_\mathfrak{a} = B_i)$ and the observation that $W_\alpha(U|V_\mathfrak{a} = B_i) = \lceil \alpha n_i \rceil$. □

While there are clear connections between the entropy measures in the uniform case, there is no general relationship between them for arbitrary probability distributions. Massey [54] shows that lower bounds can be given for $G(X)$ in terms of $H(X)$, but that there are no general upper bounds for $G(X)$ in terms of $H(X)$. Pliam [66] shows that there can be no general inequality between marginal guesswork and Shannon entropy.

Worst-Case Entropy Measures. All entropy measures presented so far are average-case measures. We use the example of guessing entropy to illustrate this and to show how they can be adapted to accommodate stronger, worst-case guarantees.

The conditional guessing entropy $G(U|V_\mathfrak{a})$ weights each value $G(U|V_\mathfrak{a} = B)$ by the probability that a randomly chosen secret from Σ_H is contained in $B \in \pi_\mathfrak{a}$. As $G(U|V_\mathfrak{a} = B)$ measures the difficulty of guessing a secret if its enclosing block B is known, $G(U|V_\mathfrak{a})$ quantifies whether secrets are, on the average, hard to guess after an attack with strategy $\mathfrak{a}$.

Our model also accommodates entropy measures for a worst-case analysis, in the sense that they quantify the guessing effort for the secrets in Σ_H that are easiest to guess. To capture this, we define the *minimal guessing entropy* $\hat{G}(U|V_\mathfrak{a})$ of U given $V_\mathfrak{a}$ as $\hat{G}(U|V_\mathfrak{a}) = \min\{G(U|V_\mathfrak{a} = B) \mid B \in \pi_\mathfrak{a}\}$. The value $\hat{G}(U|V_\mathfrak{a})$ is a lower bound on the expected guessing effort for the weakest secrets in Σ_H.

The following example illustrates the difference between worst-case and average-case entropy measures.

Example 4.8. Consider a set of uniformly distributed secrets $\Sigma_H = \{1, \ldots, 10\}$ and the partitions $\pi_1 = \{\{1\}, \{2, \ldots, 10\}\}$ and $\pi_2 = \{\{1\}, \ldots, \{10\}\}$. We have $\hat{G}(U|V_\pi) = 1$, which reflects that there exists a secret that is trivial to guess with knowledge of its enclosing block in π_1. The conditional entropy yields $G(U|V_{\pi_1}) = 4.6$, which reflects that, on the average, 4.6 guesses are still necessary to recover the secret. Note that $\hat{G}(U|V_{\pi_1}) = \hat{G}(U|V_{\pi_2})$, and that $G(U|V_{\pi_2}) = 1 < G(U|V_{\pi_1})$. That is, only the average-case measure can distinguish between the partitions π_1 and π_2. ◊

Ultimately, it will depend on the application whether worst-case or average-case measures are appropriate. As Example 4.8 illustrates, average-case measures can better distinguish between partitions, and they might be preferable when comparing implementations for their vulnerability to side-channel attacks. Worst-case measures are preferable when developing countermeasures against side-channel attacks. To this end, let m be a lower bound for the number of brute-force guesses that are considered to be infeasible for an attacker. The number $n^* = \max\{n \mid \Phi_{\hat{G}}(n) \geq m\}$ can be used as a bound for the number of queries that the system can safely answer before the key should be changed. In the remainder of this chapter, we will focus solely on average-case measures, for the reason that they are better suited for distinguishing between partitions. All of our technical results, however, carry over to the worst-case versions with only minor modifications.

Given entropy measures for evaluating attack strategies, we can now define the optimality of attacks and give bounds for what an attacker can, in principle, achieve by performing a side-channel attack.

4.4.2 Measuring the Resistance to Optimal Attacks

There is a trade-off between the number of attack steps and the attacker's uncertainty about the secret. More side-channel measurements imply less uncertainty, which entails fewer guesses. Below, we give a formal account of this for the entropy measures introduced. We then define a function $\Phi_{\mathcal{E}}$ that is parameterized by an entropy measure $\mathcal{E} \in \{H, G, W_\alpha\}$ and whose value is the expected remaining uncertainty about the secret after n steps of an optimal attack strategy. As we will show, $\Phi_{\mathcal{E}}$ can be used for assessing an implementation's vulnerability to side-channel attacks.

For assessing the vulnerability of an implementation to active side-channel attacks, we make the worst-case assumption that the attacker proceeds optimally. A strategy is optimal if an attacker who follows it can expect to have less uncertainty about the secret than with any other strategy of the same length.

Definition 4.4. Let $\mathfrak{a} = (T, r, \lambda)$ be an attack strategy of length n against a set of partitions $\mathcal{P}$ of Σ_H. We call $\mathfrak{a}$ *optimal with respect to* $\mathcal{E} \in \{H, G, W_\alpha\}$ iff $\mathcal{E}(U|V_{\mathfrak{a}}) \leq \mathcal{E}(U|V_{\mathfrak{b}})$ holds for all attack strategies $\mathfrak{b}$ against $\mathcal{P}$ of length n.

Next, we define the expected remaining uncertainty as a function of the number of attack steps taken by an optimal attacker. In this way, we relate two important aspects of a system's vulnerability; namely, how much information an attacker can obtain and how many queries he needs for this.

Definition 4.5. Let $\mathcal{P}$ be a set of partitions of Σ_H and let $\mathcal{E} \in \{H, G, W_\alpha\}$. We define the *resistance* $\Phi_\mathcal{E}$ to an attack against $\mathcal{P}$ by

$$\Phi_\mathcal{E}(n) = \mathcal{E}(U|V_\mathfrak{a}) ,$$

where $\mathfrak{a}$ is an optimal attack of length n with respect to $\mathcal{E}$.

We now formally justify the intuition that more attack steps lead to less uncertainty about the secret. In particular, we prove that $\Phi_\mathcal{E}$ decreases monotonously. As notation, we say that an attack strategy $\mathfrak{a} = (T, r, \lambda)$ is the *prefix* of an attack strategy $\mathfrak{b} = (T', r', \lambda)$ if T is a subtree of T', $r = r'$, and if λ and λ' coincide on T. We denote this by $\mathfrak{a} \leq \mathfrak{b}$.

Proposition 4.4. *Let $\mathcal{E} \in \{H, G, W_\alpha\}$ be an entropy measure and let $\mathfrak{a}$ and $\mathfrak{b}$ be attack strategies.*

1. *$\mathfrak{a} \leq \mathfrak{b}$ implies $\mathcal{E}(U|V_\mathfrak{a}) \geq \mathcal{E}(U|V_\mathfrak{b})$.*

2. *For all $n \in \mathbb{N}$, we have $\Phi_\mathcal{E}(n) \geq \Phi_\mathcal{E}(n+1)$.*

Proof. We prove 4.4.1 for the example of the guessing entropy G. Consider a partition π of Σ_H. It is easy to see that $G(U|V_\pi) = \sum_{B\in\pi} \sum_{i=1}^{|B|} i\, p_U(x_i^B)$, where the elements x^B of block B are indexed in order of their decreasing probabilities. Observe that the probabilities in the sum do not depend on π, but that the indices of the elements decrease as π is refined. As $\mathfrak{a} \leq \mathfrak{b}$ implies $\pi_\mathfrak{a} \sqsupseteq \pi_\mathfrak{b}$, 4.4.1 follows. Assertion 4.4.2 is a simple consequence of 4.4.1. □

4.5 Automated Vulnerability Analysis

Below, we first show that $\Phi_\mathcal{E}$ is computable for $\mathcal{E} \in \{H, G, W_\alpha\}$ and we give algorithms and complexity bounds. The algorithms require exponential time and cannot be used for direct computation. We then present a greedy heuristic for approximating $\Phi_\mathcal{E}$ to address this problem.

Throughout this section, let $\mathcal{P}$ be a set of partitions of Σ_H and let $r \geq 2$ be the maximum number of blocks of a partition in $\mathcal{P}$, i.e., $r = \max\{|\pi| \mid \pi \in \mathcal{P}\}$. We assume that partitions are represented using standard disjoint-set data structures with operations UNION and FIND (see, e.g., [26]). Furthermore, we assume that O and Σ_H are ordered sets for which two elements can be compared in $\mathcal{O}(1)$. It is not difficult to see that, given a function $f : \Sigma_H \times \Sigma_L \to O$, one can build disjoint-set data structures for $\mathcal{P}_f$ in time $\mathcal{O}(|\Sigma_L|\,|\Sigma_H| \log |\Sigma_H|)$, under the assumption that f can be computed in time $\mathcal{O}(1)$.

4.5.1 Computing $\Phi_{\mathcal{E}}$

We begin by establishing an upper bound on the number of attack strategies of a given length; we will use this later, when we compute $\Phi_{\mathcal{E}}$ by enumerating strategies.

Lemma 4.1. *The number of attack strategies of length n against $\mathcal{P}$ is bounded from above by $|\Sigma_L|^{\frac{r^n-1}{r-1}}$. Furthermore, every attack strategy of length n can be encoded by an r^n-tuple over $\{1, \dots, |\Sigma_L|\}$.*

Proof. A straightforward inductive argument shows that the partition induced by an attack strategy of length n has at most r^n blocks. We prove the claimed bound by induction on n. For $n = 0$, the bound is clearly valid. Assume now that there are at most $|\Sigma_L|^{\frac{r^n-1}{r-1}}$ attack strategies of length n. Each such attack strategy can be extended to an attack strategy of length $n+1$ by assigning one of the $|\Sigma_L|$ partitions to every block of the induced partition. There are at most r^n blocks, so there are at most $|\Sigma_L|^{r^n}$ possible extensions. In total, there are at most $|\Sigma_L|^{\frac{r^n-1}{r-1}} \cdot |\Sigma_L|^{r^n} = |\Sigma_L|^{\frac{r^{n+1}-1}{r-1}}$ attack strategies of length $n+1$, which concludes our inductive proof. Now observe that the choices of partitions at level j can be encoded by a r^j-tuple $(i_{j,1}, \dots, i_{j,r^j})$ over $\{1, \dots, |\Sigma_L|\}$. As $\sum_{j=0}^{n-1} r^j = \frac{r^n-1}{r-1} \leq r^n$, the entire strategy can be encoded by a r^n-tuple. □

Computing $\Phi_{\mathcal{E}}(n)$ requires identifying an optimal attack of length n. We may compute $\Phi_{\mathcal{E}}(n)$ directly by brute force: enumerate all attack strategies and compute $\mathcal{E}$ for each induced partition. This algorithm yields an upper bound for the complexity of computing $\Phi_{\mathcal{E}}$.

Theorem 4.1. *The value $\Phi_{\mathcal{E}}(n)$ can be computed in time*

$$\mathcal{O}(n\,|\Sigma_L|^{r^n}|\Sigma_H|\,\log|\Sigma_H|)$$

under the assumption that $\mathcal{E}$ can be computed in time $\mathcal{O}(|\Sigma_H|)$.

Proof. Let $(i_0; \dots; i_{n-1,1}, \dots, i_{n-1,r^{n-1}})$, with $1 \leq i_j \leq |\Sigma_L|$, represent an attack strategy $\mathfrak{a}$ of length n, where the choices of partitions at each level are encoded as in the proof of Lemma 4.1 and where the individual levels are separated by ";". Iterate over all $h \in \Sigma_H$. For each h, call FIND(h) on the representation of partition i_0 to obtain the index j of h's enclosing block in π_{i_0}. Use $\textsc{Find}(h)$ to obtain h's block in $\pi_{i_{1,j}}$. Repeat this procedure until h's block in the partition at depth n is determined. Save these n block indices in a list and store it in an array I at index h. Performing this procedure for all $h \in \Sigma_H$ has a time complexity of $\mathcal{O}(n\,|\Sigma_H| \log |\Sigma_H|)$. Two secrets are in the same block of the partition induced by $\mathfrak{a}$ if and only if their corresponding index lists coincide. To obtain the equivalence classes, sort Σ_H according to the lexicographic order given by the lists in I in $\mathcal{O}(n\,|\Sigma_H| \log |\Sigma_H|)$, which dominates the running time for evaluating $\mathcal{E}$ on the resulting partition. Performing this procedure for all attack strategies yields an overall running time of $\mathcal{O}(n\,|\Sigma_L|^{r^n} |\Sigma_H|\, \log |\Sigma_H|)$. □

4.5.2 Approximating $\Phi_{\mathcal{E}}$

Brute-force computation of $\Phi_{\mathcal{E}}$ requires time doubly exponential in the number of attack steps and is hence infeasible even for small parameter sizes. To address this problem, we present a more efficient greedy heuristic and describe properties that help us approximate $\Phi_{\mathcal{E}}$.

A Greedy Heuristic. Consider an attacker who has performed a number of attack steps against a set of partitions $\mathcal{P}$ and has narrowed down the set of possible secrets to a subset $A \subseteq \Sigma_H$. A greedy choice for the subsequent query is a partition $\pi \in \mathcal{P}$ that minimizes the remaining entropy of $A \cap \pi$. To formalize this, consider the random variable $U_A = id_A$ that models the random choice of a secret according to the conditional probability distribution $p(\cdot|A)$, and the random variable $V_{\pi\cap A} : A \to \pi \cap A$ that models the choice of the enclosing block in $\pi \cap A$.

Definition 4.6. An attack strategy $\mathfrak{a} = (T, r, \lambda)$ against $\mathcal{P}$, with $T = (V, E)$, is *greedy with respect to* $\mathcal{E} \in \{H, G, W_\alpha\}$ iff for every $v \in V$ and all $\pi_1, \pi_2 \in \mathcal{P}$, $\{\lambda(w) \mid w \in succ(v)\} = \lambda(v) \cap \pi_1$ implies $\mathcal{E}(U_A|V_{\pi_1\cap A}) \leq \mathcal{E}(U_A|V_{\pi_2\cap A})$.

We next define an approximation $\hat{\Phi}_{\mathcal{E}}$ of $\Phi_{\mathcal{E}}$ based on the partition induced by a greedy strategy. Note that greedy strategies are not unique and that the induced partitions of two greedy strategies of the same length need not even have the same entropy. Hence to define an approximation $\hat{\Phi}_{\mathcal{E}}$, we assume a fixed greedy strategy $\mathfrak{a}$ of sufficient length k whose underlying tree is full. For all $n \leq k$, we denote the full prefix of $\mathfrak{a}$ with length n by $\mathfrak{a}(n)$. We define $\hat{\Phi}^{\mathfrak{a}}_{\mathcal{E}}$ as $\hat{\Phi}^{\mathfrak{a}}_{\mathcal{E}}(n) = \mathcal{E}(U|V_{\mathfrak{a}(n)})$, for all $n \leq k$. We only use $\mathfrak{a}$ as an artifact to consistently resolve the nondeterminism of greedy strategies of different lengths. From now on, we assume that a greedy strategy $\mathfrak{a}$ of sufficient length is fixed and write $\hat{\Phi}_{\mathcal{E}}$ instead of $\hat{\Phi}^{\mathfrak{a}}_{\mathcal{E}}$.

Theorem 4.2. *The value $\hat{\Phi}_{\mathcal{E}}(n)$ can be computed in time*

$$\mathcal{O}(n\, r\, |\Sigma_L|\, |\Sigma_H|^2)\ ,$$

under the assumption that $\mathcal{E}$ can be computed in time $\mathcal{O}(|\Sigma_H|)$.

Proof. For computing intersections of partitions, we assume a list representation of the blocks of every partition, in which every list is ordered with respect to the order on Σ_H. This can be extracted from the given disjoint-set data structures in time $\mathcal{O}(|\Sigma_L|\, |\Sigma_H|^2)$. For a fixed subset of Σ_H that is represented as an ordered list, a greedy refinement can then be computed by intersecting it with each of the (at most r) blocks of each of the $|\Sigma_L|$ partitions. As the set representations are ordered, this can be done in time $\mathcal{O}(r\, |\Sigma_L|\, |\Sigma_H|)$. As the number of blocks in every partition of Σ_H is bounded by $|\Sigma_H|$, computing n greedy steps can be done in time $\mathcal{O}(n\, r\, |\Sigma_L|\, |\Sigma_H|^2)$. □

We next state several inequalities between the values of $\Phi_{\mathcal{E}}$ and $\hat{\Phi}_{\mathcal{E}}$, which we will use later, when interpreting our experimental results.

Relating $\Phi_{\mathcal{E}}$ and $\hat{\Phi}_{\mathcal{E}}$. The definition of a greedy strategy begs the question of whether greedy strategies are also optimal. The following example illustrates that this is not the case in general.

Example 4.9. Consider the set of partitions $\mathcal{P} = \{\{\{1\},\{2\},\{3,4,5\}\},\{\{1\},\{2,3,4\},\{5\}\},\{\{1,2,3\},\{4,5\}\}\}$, a uniform distribution, and the guessing entropy as a measure. A greedy strategy refines Σ_H to $\{\{1\},\{2\},\{3,4,5\}\}$ in a first step, and to $\{\{1\},\{2\},\{3,4\},\{5\}\}$ in a second step. Optimally, however, one would first pick $\{\{1,2,3\},\{4,5\}\}$ and refine it to $\{\{1\},\{2\},\{3\},\{4\},\{5\}\}$ in a second step. ◊

Although Example 4.9 implies that $\hat{\Phi}_{\mathcal{E}}$ and $\Phi_{\mathcal{E}}$ do not coincide in general, we can establish the following relationships.

Proposition 4.5. *For $\mathcal{E} \in \{H, G, W_\alpha\}$, we have*

1. $\hat{\Phi}_{\mathcal{E}}(1) = \Phi_{\mathcal{E}}(1)$,
2. *for all $n \in \mathbb{N}$, $\hat{\Phi}_{\mathcal{E}}(n) \geq \Phi_{\mathcal{E}}(n)$, and*
3. *if $\hat{\Phi}_{\mathcal{E}}(n) = \hat{\Phi}_{\mathcal{E}}(n+1)$, then we have $\Phi_{\mathcal{E}}(n') = \hat{\Phi}_{\mathcal{E}}(n') = \hat{\Phi}_{\mathcal{E}}(n)$, for all $n' \geq n$.*

```
greedy :: [Part k] -> Int -> [k] -> Part k
greedy f n secs = app n (greedystep f) [secs]

greedystep :: [Part k] -> Part k -> Part k
greedystep f pt = concat (map refine pt)
  where refine b = minimumBy order (restrict b f)
```

Figure 4.2: Computing $\hat{\Phi}_{\mathcal{E}}$ in HASKELL

Proof. Assertions 1 and 2 follow directly from Definitions 4.4 and 4.6. For Assertion 3, let $\mathfrak{a}$ be the greedy strategy underlying the definition of $\hat{\Phi}_{\mathcal{E}}$. $\hat{\Phi}_{\mathcal{E}}(n) = \hat{\Phi}_{\mathcal{E}}(n+1)$ implies that $\pi_{\mathfrak{a}(n)}$ cannot be refined by intersection with a partition from $\mathcal{P}$, hence $\pi_{\mathfrak{a}(n)} = \bigcap_{\pi\in\mathcal{P}} \pi$, which refines every partition that can be induced by intersection of elements from $\mathcal{P}$. □

We will make use of Proposition 4.5 in our experiments. 4.5.2 shows that an implementation that is shown to be vulnerable when analyzed with $\hat{\Phi}_{\mathcal{E}}$ must also be vulnerable with respect to $\Phi_{\mathcal{E}}$. 4.5.3 implies that if $\hat{\Phi}_{\mathcal{E}}$ levels off, then so does $\Phi_{\mathcal{E}}$, and their values coincide. Hence we do not need to compute $\Phi_{\mathcal{E}}$ for arguments beyond this point.

4.5.3 An Implementation

For our experiments we have implemented $\hat{\Phi}_{\mathcal{E}}$ in HASKELL [14]. We have chosen simplicity over efficiency, forgoing sophisticated data structures and optimizations. Instead, we represent sets as lists and partitions as lists of lists and recursively compute greedy refinements of partitions. The core routines are given in Figure 4.2.

The function `greedy` takes as arguments a list `secs` of secrets, a list of partitions `f` of the list `secs`, and an integer `n`. It refines the trivial partition `[secs]` by `n`-fold application of a greedy refinement step through `app`. The refinement step is implemented in `greedystep`, where each partition `pt` is refined by greedily refining each individual block. This is done in `refine`, which maps each block to its partition with minimal rank among those obtained by restricting the elements of `f` to `b` with `restrict`. The rank of a partition is given by the function `order`, which can be instantiated to $\mathcal{E} \in \{H, G, W_\alpha\}$. Applying `order` to the result of `greedy` yields $\hat{\Phi}_{\mathcal{E}}$. The simplicity of this implementation shows that the automation of our techniques is indeed straightforward.

4.6 Summary

We have presented a quantitative model for reasoning about adaptive side-channel attacks. It allows us to express an attacker's remaining uncertainty about a secret as a function of the number of his side-channel measurements. This function provides a relevant metric for assessing a system's vulnerability to side-channel attacks.

Our model builds on explicit representations of side-channels. These representations can be given by hardware simulation environments and they can also derived from Mealy machine models of hardware. This allows us to give an interpretation of R_I/R_O-security in terms of our quantitative model.

We report on experimental results with our prototype in Chapter 6, where we show that our results can be practically used for giving information-theoretic bounds for the side-channel leakage of cryptographic algorithms in the presence of adaptive attackers.

Chapter 5

Eliminating Side-Channels

5.1 Introduction

5.1.1 Security Type Systems and Transformations

Security type systems are an attractive approach for automating information flow analyses of programs in high-level programming languages: security type-checking is static, modular and allows one to syntactically determine if a program is secure without the need to perform a direct analysis of the transition system induced by a program with large or infinite memory.

If type-checking succeeds, then the program is secure. If type-checking fails, the program might be insecure and should not be run. The correction of the program can be a tedious and error-prone process and is usually left to the programmer. Transforming type systems offer automated support for this task: during type-checking, the program is modified with the objective of removing potential information leaks. Similarly to non-transforming type systems, transforming type systems produce false negatives. However, by removing certain information leaks, they can be successfully applied to a larger class of programs.

A program transformation to remove information leaks must modify a program in such a way that the output program's observable behavior does not depend on any secret data. Information leaks that arise through branching decisions that depend on secret data are particularly difficult to avoid. For securing a conditional with a secret guard, for example, one must ensure that both branches are observationally equivalent; then an observer cannot deduce which branch was taken and the confidentiality of the guard is preserved. In this chapter, we develop a novel transformation to secure such conditionals.

The first transformational approach for securing conditionals with secret guards is the *cross-copying* technique from [4]. The technique was originally proposed as a method to transform out timing leaks in sequential program, but it has been shown that it is also suitable for eliminating scheduling leaks in multithreaded programs [75]. The intuition behind this is that, after the transformation, both branches of a conditional with a secret guard need the same amount of time slices to be executed. In this way, the scheduler's behavior is decoupled from the secret guard.

The cross-copying technique works as follows. For securing a conditional

$$\mathsf{if\ h\ then}\ C_1\ \mathsf{else}\ C_2\ ,$$

where h is a variable of domain H, one tries to construct programs S_1 and S_2 with the properties that (1) S_i is observationally equivalent to C_i, for $i \in \{1,2\}$, and (2) S_1 and S_2 do not contain assignments. If S_1 and S_2 can be constructed with (1) and (2), the conditional is transformed into

$$\mathsf{if\ h\ then}\ C_1; S_2\ \mathsf{else}\ S_1; C_2\ .$$

As C_i is observationally equivalent to S_i, the observer cannot distinguish between the execution of $C_1; S_2$ and $S_1; C_2$ and, hence, the result of the transformation is a secure program. As S_1 and S_2 do not change the memory, the program's semantics is only slightly modified by the transformation.

Example 5.1. The cross-copying technique transforms the program

$$\mathsf{if\ h\ then\ skip}; \mathsf{h}_1 := 1\ \mathsf{else}\ \mathsf{h}_2 := 2$$

into the program

$$\mathsf{if\ h\ then\ skip}; \mathsf{h}_1 := 1; \mathsf{skip\ else\ skip}; \mathsf{skip}; \mathsf{h}_2 := 2\ .$$

For this, note that for $C_1 = \mathsf{skip}; \mathsf{h}_1 := 1$ and $C_2 = \mathsf{h}_2 := 2$, and for an observer who cannot see the values of the variables $\mathsf{h}_1, \mathsf{h}_2$, the programs $S_1 = \mathsf{skip}; \mathsf{skip}$ and $S_2 = \mathsf{skip}$ satisfy properties (1) and (2). $\Diamond$

The cross-copying technique is intuitive and simple, but it solves the problem of transforming a program secure only partially. Its applicability is limited by the fact that S_1 and S_2 with (1) and (2) can only be constructed if C_1 and C_2 do not contain assignments to low variables. As a consequence, the cross-copying technique is not applicable for securing the thread $\mathsf{if\ h\ then\ skip; l := 0\ else\ l := 0}$ from Example 1.2, where l is a variable of domain L. Furthermore, the transformation may lead to an unacceptable blow-up of the program's running time. Finally, as we will explain later, it is not directly applicable for enforcing multi-level security policies.

5.1.2 Our Approach

In this chapter, we develop a novel approach for achieving the observational equivalence of the branches of a conditional, which addresses the aforementioned shortcomings of the cross-copying technique. We use this approach to improve an existing transforming type system for programs in $\vec{Com}$.

Main Idea. We define a partial equivalence relation $\doteq$ on program terms to syntactically approximate the notion of observational equivalence $\approxeq$ to be achieved in the sense that $C_1 \doteq C_2$ implies $C_1 \approxeq C_2$. For securing a conditional $\mathsf{if\ h\ then}\ C_1\ \mathsf{else}\ C_2$, where h is a variable of domain H, we proceed in three steps:

1. we introduce meta-variables as subterms into C_1 and C_2, yielding the *lifted* programs $\overline{C_1}$ and $\overline{C_2}$,
2. we find a substitution σ of meta-variables with program terms such that $\sigma\overline{C_1} \doteq \sigma\overline{C_2}$ holds, and
3. we remove remaining meta-variables from $\sigma\overline{C_i}$, for $i \in \{1,2\}$, and return the corrected program $\mathsf{if\ h\ then}\ \sigma\overline{C_1}\ \mathsf{else}\ \sigma\overline{C_2}$.

That is, we reduce the problem of making the branches of a program observationally equivalent to an (equational) *unification* problem. This seems a natural approach to solving the constraints given by the requirement $\overline{C_1} \approxeq \overline{C_2}$.

Example 5.2. Consider again the program

$$\mathsf{if\ h\ then\ skip; l := 0\ else\ l := 0}\ ,$$

for which the cross-copying technique fails. By inserting meta-variables α_i at suitable positions, we obtain the lifted branches $\mathsf{skip}; \alpha_1; \mathsf{l} := 0; \alpha_2$ and $\alpha_3; \mathsf{l} := 0; \alpha_4$. If the substitution $\sigma = \{\alpha_1 \backslash \epsilon, \alpha_2 \backslash \epsilon, \alpha_3 \backslash \mathsf{skip}, \alpha_4 \backslash \epsilon\}$ (here ϵ denotes the neutral element of the sequential composition operator) is applied to the lifted branches, both yield $\mathsf{skip}; \mathsf{l} := 0$.

A unification-based transformation would hence return

$$\mathsf{if\ h\ then\ skip; l := 0\ else\ skip; l := 0}\ ,$$

which is a secure program. $\Diamond$

While this approach is, in principle, applicable to a variety of notions of observational equivalence $\approxeq$, we will investigate in detail a concrete instance for eliminating scheduling leaks in MWL programs. Our notion of observational equivalence will be the strong low-bisimulation $\approxeq_L$ introduced in Section 2.4.

Outline. In the remainder of this chapter, we show how the strong low-bisimulation $\approxeq_L$ (see Chapter 2.4) can be approximated by a relation $\doteq_L \subseteq \vec{Com} \times \vec{Com}$ and we present a calculus for inserting meta-variables into programs in $\vec{Com}$. We show that the problem of finding substitutions for this instance can be reduced to a well-known unification problem (AC_1 unification with free constructors [40]), allowing existing unification algorithms to be employed for computing unifiers. We sketch that these algorithms do not yield an optimal solution, and we also propose a unification algorithm that is tailored to the specific structure of the unification problems that arise during the transformation. Finally, we integrate our approach into an existing transforming type system and prove its completeness in the sense that it successfully eliminates every scheduling side-channel that can be removed by insertion of dummy operations.

Advantages. In comparison to the state-of-the art transforming type system for achieving strong security [75], the main advantages of our transforming type system are the following: our transforming type system can correct a class of programs that cannot be corrected by the original type system; our transformation returns programs that are faster and often substantially smaller in size; our transformation can be applied in the context of security policies with more than two levels. Besides these technical advantages, unification yields a natural perspective on the problem of making two programs observationally equivalent.

5.2 Observational Equivalence by Unification

5.2.1 Objectives of the Transformation

The main objective of our program transformation is to achieve the observational equivalence $C_1 \approxeq_L C_2$ of given programs C_1 and C_2. To realize this, the programs' behavior might need to be changed in some way. Without any restrictions on the possible semantic changes, achieving observational equivalence is trivial: both C_1 and C_2 can simply be transformed to skip. We will use a weak bisimulation $\simeq$ to constrain the possible modifications to C_1 and C_2, and we require that the original program C_i and the transformed program C'_i satisfy $C_i \simeq C'_i$, for $i \in \{1,2\}$.

Hence, the objectives of our program transformation can be expressed by the two equivalence relations, $\approxeq_L$ and $\simeq$, where $\approxeq_L$ models the observational equivalence to be achieved and where $\simeq$ models the behavior to be preserved. It can be guaranteed that these objectives are met by unification. For achieving observational equivalence by unification under $\doteq_L$, it must be ensured that $C_1 \doteq_L C_2$ implies $C_1 \approxeq_L C_2$. For achieving the preservation of behavior, the liftings $\overline{C_i}$ of C_i and the range of the admissible substitutions σ must be chosen in such a way that $C_i \simeq \sigma\overline{C_i}$ holds, for $i \in \{1,2\}$.

Below, we introduce suitable syntactic approximations for observational equivalence, and substitutions and liftings for achieving weak bisimulation equivalence.

5.2.2 Approximating Observational Equivalence

The relation $\doteq_L \subseteq \vec{Com} \times \vec{Com}$ defined by the rules in Figure 5.1 provides a syntactic approximation of the strong low bisimulation relation. We will illustrate this with a few exemplary rules and give some necessary definitions, before formally stating and proving the assertion in Theorem 5.1.

An arithmetic expression Exp has the security domain L (denoted by $Exp : L$) if all variables in Exp have domain L. Otherwise, it has security domain H (denoted by $Exp : H$). We classify boolean expressions analogously. Hence, values of expressions with domain H may depend on secrets while values of expressions with domain L can only depend on public data. In the following, we use variable names h and l to denote variables of domain H and L, respectively.

The relation $\doteq_L$ provides a syntactic approximation of the strong low bisimulation relation: the rules [SHA_1] and [SHA_2] express that an assignment to a high variable is invisible to the observer, i.e., that it cannot be distinguished from a skip. However, l := h is not $\doteq_L$-related to itself because the precondition of [LA], the only rule in Figure 5.1 applicable to assignments to low variables, rules out that high variables occur on the right-hand side of the assignment. This meets our intuition, as the execution of an assignment l := h clearly violates the policy $H \not\rightsquigarrow L$. If the branches of a high conditional relate (rules [$SHCond_1$] and [$SHCond_2$]), then this conditional is related to the sequential composition of a skip and a program that is related to the branches. The

$$\frac{}{\mathsf{skip} \bumpeq_L \mathsf{skip}}\,[\mathit{Skp}] \qquad \frac{\mathit{Id} : L \quad \mathit{Exp} : L \quad \mathit{Exp}' : L \quad \mathit{Exp} \equiv \mathit{Exp}'}{\mathit{Id} := \mathit{Exp} \bumpeq_L \mathit{Id} := \mathit{Exp}'}\,[\mathit{LA}]$$

$$\frac{\mathit{Id} : H \quad \mathit{Id}' : H}{\mathit{Id} := \mathit{Exp} \bumpeq_L \mathit{Id}' := \mathit{Exp}'}\,[\mathit{HA}] \qquad \frac{\mathit{Id} : H}{\mathsf{skip} \bumpeq_L \mathit{Id} := \mathit{Exp}}\,[\mathit{SHA}_1] \qquad \frac{\mathit{Id} : H}{\mathit{Id} := \mathit{Exp} \bumpeq_L \mathsf{skip}}\,[\mathit{SHA}_2]$$

$$\frac{C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_2'}{C_1; C_2 \bumpeq_L C_1'; C_2'}\,[\mathit{Seq}] \qquad \frac{C_1 \bumpeq_L C_1', \ldots, C_n \bumpeq_L C_n'}{\langle C_1, \ldots, C_n \rangle \bumpeq_L \langle C_1', \ldots, C_n' \rangle}\,[\mathit{Par}]$$

$$\frac{C \bumpeq_L C' \quad V \bumpeq_L V'}{\mathsf{fork}(CV) \bumpeq_L \mathsf{fork}(C'V')}\,[\mathit{Frk}] \qquad \frac{B, B' : L \quad B \equiv B' \quad C \bumpeq_L C'}{\mathsf{while}\ B\ \mathsf{do}\ C \bumpeq_L \mathsf{while}\ B'\ \mathsf{do}\ C'}\,[\mathit{Whl}]$$

$$\frac{B, B' : L \quad B \equiv B' \quad C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_2'}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\,[\mathit{LCond}]$$

$$\frac{B, B' : H \quad C_1 \bumpeq_L C_1' \quad C_1 \bumpeq_L C_2' \quad C_1 \bumpeq_L C_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\,[\mathit{HCond}]$$

$$\frac{B' : H \quad C_1 \bumpeq_L C_1' \quad C_1 \bumpeq_L C_2'}{\mathsf{skip}; C_1 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\,[\mathit{SHCond}_1] \qquad \frac{B : H \quad C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_1'}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathsf{skip}; C_1'}\,[\mathit{SHCond}_2]$$

$$\frac{\mathit{Id} : H \quad B' : H \quad C_1 \bumpeq_L C_1' \quad C_1 \bumpeq_L C_2'}{\mathit{Id} := \mathit{Exp}; C_1 \bumpeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}\,[\mathit{HAHCond}_1]$$

$$\frac{\mathit{Id}' : H \quad B : H \quad C_1 \bumpeq_L C_1' \quad C_2 \bumpeq_L C_1'}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \bumpeq_L \mathit{Id}' := \mathit{Exp}'; C_1'}\,[\mathit{HAHCond}_2]$$

Figure 5.1: A syntactic approximation of observational equivalence

rule models that the execution of the branches cannot be distinguished and that the evaluation of the guard looks like a skip to a low observer. Loops with high guards are considered insecure. Finally, only loops with low guards relate in $\bumpeq_L$ (see rule [*Whl*]). As the following theorem shows, $\bumpeq_L$ approximates $\approxeq_L$ as required.

Theorem 5.1. *If $V \bumpeq_L V'$ is derivable for $V, V' \in \vec{Com}$, then $V \approxeq_L V'$ holds.*

The proof relies on two auxiliary lemmas and proceeds by induction on the number of rule applications in the derivation of $V \bumpeq_L V'$. It is given in full detail in Appendix A.1.

It is not difficult to see that $\bumpeq_L$ is a partial equivalence relation, i.e. it is transitive and symmetric, but not reflexive. Nevertheless, $\bumpeq_L$ is an equivalence relation, even a congruence relation, if one restricts programs to the language *Slice*, which we define, following [75], as the largest sub-language of *Com* without assignments of high expressions to low variables, assignments to high variables, and loops or conditionals with high guards. Formally, *Slice* is given by the grammar

$$S ::= \mathsf{skip} \mid Id_L := Exp_L \mid S_1; S_2 \mid \mathsf{if}\ B_L\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{while}\ B_L\ \mathsf{do}\ S \mid \mathsf{fork}(S\vec{V})\ ,$$

where S, S_1, S_2 denote programs in *Slice*, and where Id_L, Exp_L and B_L range over variable identifiers and (boolean) expressions of domain L. The sub-language *Slice* provides the context in which we will apply unification.

5.2.3 Liftings, Substitutions, and Preservation of Behavior

Liftings We insert meta-variables from a set $\mathcal{V} = \{\alpha_1, \alpha_2, \dots\}$ into a program $C \in Com$ by sequential composition with the subterms of C. The set $Com_{\mathcal{V}}$ of programs with meta-variables is defined by

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid C; X \mid X; C \\ & \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C\vec{V})\ , \end{aligned}$$

where the placeholder X ranges over $\mathcal{V}$. The set of all command vectors with meta-variables is $\vec{Com}_{\mathcal{V}} = \bigcup_{n \in \mathbb{N}} Com^n_{\mathcal{V}}$. Note that the ground programs in $\vec{Com}_{\mathcal{V}}$, i.e., those without meta-variables, are exactly the programs in $\vec{Com}$. The operational semantics for ground programs remains unchanged, whereas programs in $\vec{Com}_{\mathcal{V}}$ are not meant to be executed. Along the same lines, we define the sets $Slice_{\mathcal{V}}$ and $\vec{Slice}_{\mathcal{V}}$, respectively, by inserting meta-variables into commands in *Slice*, and vectors thereof.

Substitutions Meta-variables may be substituted with programs, meta-variables, or the special symbol ϵ, which acts as the neutral element of the sequential composition operator (";"), i.e. $\epsilon; C = C$ and $C; \epsilon = C$. Note that skip is not a neutral element of (";") with respect to $\doteq_L$, as skip requires a computation step. When talking about programs in $Com_{\mathcal{V}}$ under a given substitution, we implicitly assume that these equations have been applied (from left to right) to eliminate the symbol ϵ from the program. Moreover, we view sequential composition as an associative operator and implicitly identify programs that differ only in the use of parentheses for sequential composition. That is, $C_1; (C_2; C_3)$ and $(C_1; C_2); C_3$ denote the same program.

Definition 5.1. A mapping $\sigma : \mathcal{V} \to (\{\epsilon\} \cup \mathcal{V} \cup Com_{\mathcal{V}})$ is a *substitution* if its *domain* $dom(\sigma) = \{\alpha \in \mathcal{V} \mid \sigma(\alpha) \neq \alpha\}$ is finite. The *range* of σ is defined as the set $ran(\sigma) = \sigma(dom(\sigma))$. We denote substitutions σ with $dom(\sigma) = \{\alpha_1, \dots, \alpha_n\}$ and $\sigma(\alpha_i) = C_i$, for $i \in \{1, \dots, n\}$, as sets of assignments $\sigma = \{\alpha_1 \backslash C_1, \alpha_2 \backslash C_2, \dots, \alpha_n \backslash C_n\}$. A substitution mapping each meta-variable in a program V to $\{\epsilon\} \cup Com$ is a *ground substitution of V*. A substitution π mapping all meta-variables in V to ϵ is a *projection of V*. Given a program V in $\vec{Com}$, we call every program V' in $\vec{Com}_{\mathcal{V}}$ with $\pi V' = V$ a *lifting of V*.

As previously explained, the lifting of a program forms the basis for our transformation by unification.

Example 5.3. The program $\mathsf{if}\ \mathsf{h}\ \mathsf{then}\ (\alpha_1; \mathsf{skip}; \alpha_2; \mathsf{l} := 1)\ \mathsf{else}\ (\alpha_3; \mathsf{l} := 1)$ is a lifting of the program $\mathsf{if}\ \mathsf{h}\ \mathsf{then}\ (\mathsf{skip}; \mathsf{l} := 1)\ \mathsf{else}\ \mathsf{l} := 1$. ◊

Preservation of Behavior We introduce an equivalence relation $\simeq$ to constrain the modifications caused by the transformation. Intuitively, this relationship requires a transformed program to be a slowed-down version of the original program.

Definition 5.2. The *weak possibilistic bisimulation* $\simeq$ is the union of all symmetric relations R on command vectors, such that whenever $V\ R\ V'$, then for all memories ν, μ and all thread pools W, there is a thread pool W', such that

$$\begin{aligned} &\langle\!|V, \nu|\!\rangle \twoheadrightarrow \langle\!|W, \mu|\!\rangle \implies (\langle\!|V', \nu|\!\rangle \twoheadrightarrow^* \langle\!|W', \mu|\!\rangle \wedge WRW') \\ &\wedge V = \langle\rangle \implies \langle\!|V', \nu|\!\rangle \twoheadrightarrow^* \langle\!|\langle\rangle, \nu|\!\rangle . \end{aligned}$$

Here, $\twoheadrightarrow^*$ denotes the reflexive and transitive closure of the relation $\twoheadrightarrow$.

The requirement that the transformed program must be $\simeq$-related to the original program is stronger than the requirement in [75]. There, it is only required that the original program can simulate the transformed program. One advantage of our more restrictive choice is that a transformation cannot introduce nontermination.

In the remainder of this chapter, we will focus on substitutions with a restricted range, namely those that map meta-variables to (possibly empty) sequential compositions of meta-variables and skips.

Definition 5.3. A substitution with range $\{\epsilon\} \cup Stut_{\mathcal{V}}$ is called *preserving*, where $Stut_{\mathcal{V}}$ is defined by

$$C ::= X \mid \mathsf{skip} \mid C_1; C_2 ,$$

where the C_i range over $Stut_{\mathcal{V}}$ and X ranges over meta-variables in $\mathcal{V}$.

The term *preserving* substitution is justified by the fact that such substitutions preserve a given program's semantics as specified in Definition 5.2.

Theorem 5.2 (Preservation of Behavior).

1. *For all preserving substitutions σ, ρ that are ground for $V \in \vec{Com}_{\mathcal{V}}$, we have $\sigma(V) \simeq \rho(V)$.*
2. *For each lifting V' of a ground program $V \in \vec{Com}$ and each preserving substitution σ with $\sigma(V')$ ground, we have $\sigma(V') \simeq V$.*

The proof of Theorem 5.2 is given in Appendix A.2.

5.2.4 Unification of Programs.

The problem of finding a substitution that relates the branches of conditionals with high guards by the relation $\doteq_L$ can be viewed as the problem of finding a unifier for the branches under $\doteq_L$. To this end, we lift the relation $\doteq_L \subseteq \vec{Com} \times \vec{Com}$ to a relation on $\vec{Com}_{\mathcal{V}}$ that we also denote by $\doteq_L$.

Definition 5.4. For $V_1, V_2 \in \vec{Com}_{\mathcal{V}}$, we define $V_1 \doteq_L V_2$ iff $\sigma V_1 \doteq_L \sigma V_2$ holds for each preserving substitution σ that is ground for V_1 and V_2.

Definition 5.5. A $\doteq_L$*-unification problem* Δ is a finite set of statements of the form $V_i \doteq^?_L V'_i$, i.e.

$$\Delta = \{V_0 \doteq^?_L V'_0, \dots, V_n \doteq^?_L V'_n\}$$

with $V_i, V'_i \in \vec{Com}_{\mathcal{V}}$ for all $i \in \{0, \dots, n\}$. A substitution σ is a *preserving unifier for* Δ if and only if σ is preserving and $\sigma V_i \doteq_L \sigma V'_i$ holds for each $i \in \{0, \dots, n\}$. A $\doteq_L$-unification problem is *solvable* if the set of preserving unifiers $\mathcal{U}(\Delta)$ for Δ is not empty.

5.3 Automating the Transformation

In this section, we show how to automate the correction of programs. To this end, we first extend the transforming type system from [75] to incorporate unification. Subsequently, we show how the insertion of meta-variables and the unification process can be automated.

Throughout the section, we will assume that the equivalence of expressions is decidable. This assumption is required, as we have left expressions unspecified and as, during the unification process, we need to determine whether $Exp_1 \equiv Exp_2$ holds. For expression theories for which equivalence is not decidable, the precondition $Exp_1 \equiv Exp_2$ can be replaced with syntactic equality $Exp_1 = Exp_2$ in the rules of Figure 5.1 and in the unification calculus. In this way, precision in the approximation $\doteq_L$ of $\approxeq_L$ is lost and decidability is gained. All of our results carry over to this modified scenario.

5.3.1 A Transforming Type System

The transforming type system in Figure 5.2 has been derived from the one in [75]. We use the judgment

$$V \hookrightarrow V' : S$$

to denote the transformation of a program $V \in \vec{Com}_{\mathcal{V}}$ into a program $V' \in \vec{Com}_{\mathcal{V}}$. The intention is that V' has secure information flow and reflects the semantics of V as specified by Definition 5.2. The *slice* S is a program in the sub-language $Slice_{\mathcal{V}}$ that is observationally equivalent to V'.

The novelty over [75] is that our type system operates on $\vec{Com}_{\mathcal{V}}$ (rather than on $\vec{Com}$) and that the rule for high conditionals has been altered. In the original type system, a high conditional is transformed by sequentially composing each branch with the slice of the other branch. Instead of cross-copying slices, our rule instantiates the meta-variables that occur in the branches using preserving unifiers. The advantages of this modification are discussed in Section 5.4. Note that the rule [*THCond*] does not mandate the choice of a specific preserving unifier of the branches. Nevertheless, we can prove that the type system meets our previously described intuition about the judgment $V \hookrightarrow V' : S$. For this, we need a lemma that describes the relationship between V' and S.

Lemma 5.1. *If $V \hookrightarrow V' : S$ can be derived then $V' \doteq_L S$ holds.*

The proof of Lemma 5.1 is given in Appendix A.3. The following theorem is an immediate consequence of Theorems 5.2 and Lemma 5.1. It shows that lifting a program, applying the transforming type system, and afterwards projecting out remaining meta-variables meets the objectives discussed in Section 5.2.1.

Theorem 5.3 (Soundness of the Type System).
If $V^* \hookrightarrow V' : S$ is derivable for some lifting $V^* \in \vec{Com}_{\mathcal{V}}$ of a program $V \in \vec{Com}$ then

1. *$\pi V'$ is strongly secure, and*

2. *$\pi V' \simeq V$.*

$$\frac{}{\mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}}\ [TSkp] \qquad \frac{}{X \hookrightarrow X : X}\ [TVar]$$

$$\frac{Id : H}{Id := Exp \hookrightarrow Id := Exp : \mathsf{skip}}\ [THA] \qquad \frac{Id : L \qquad Exp : L}{Id := Exp \hookrightarrow Id := Exp : Id := Exp}\ [TLA]$$

$$\frac{C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : S_1; S_2}\ [TSeq] \qquad \frac{B : L \qquad C \hookrightarrow C' : S}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ S}\ [TWhl]$$

$$\frac{C_1 \hookrightarrow C_1' : S_1 \qquad V_2 \hookrightarrow V_2' : S_2}{\mathsf{fork}(C_1 V_2) \hookrightarrow \mathsf{fork}(C_1' V_2') : \mathsf{fork}(S_1 S_2)}\ [TFrk]$$

$$\frac{C_1 \hookrightarrow C_1' : S_1 \quad \ldots \quad C_n \hookrightarrow C_n' : S_n}{\langle C_1, \ldots, C_n \rangle \hookrightarrow \langle C_1', \ldots, C_n' \rangle : \langle S_1, \ldots, S_n \rangle}\ [TPar]$$

$$\frac{B : L \qquad C_1 \hookrightarrow C_1' : S_1 \qquad C_2 \hookrightarrow C_2' : S_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2}\ [TLCond]$$

$$\frac{B : H \quad C_1 \hookrightarrow C_1' : S_1 \quad C_2 \hookrightarrow C_2' : S_2 \quad \sigma \in \mathcal{U}(\{S_1 \overset{?}{\doteq}_L S_2\})}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2' : \mathsf{skip}; \sigma S_1}\ [THCond]$$

Figure 5.2: A transforming security type system for programs with meta-variables

Proof. From Lemma 5.1 and the symmetry and transitivity of $\doteq_L$, we obtain $V' \doteq_L V'$ which, by Definition 5.4, entails $\pi V' \doteq_L \pi V'$. Assertion 1 then follows directly from Theorem 5.1 and the definition of strong security.

By induction on the height of the derivation of $V^* \hookrightarrow V' : S$, one obtains $V' = \rho V^*$ for some preserving substitution ρ. Assertion 2 follows from applying Theorem 5.2.2 to $(\pi\rho)(V^*)$. □

For a fully automated analysis, we will now define more concretely where meta-variables are inserted and how unifiers are determined.

5.3.2 Automatic Insertion of Meta-Variables.

When lifting a program, one is faced with a trade-off: inserting meta-variables means creating possibilities for correcting the program, but it also increases the complexity of the unification problem. Within this spectrum, our objective is to minimize the number of inserted meta-variables without losing the possibility of correcting the program. To this end, consider the sub-language $Pad_{\mathcal{V}}$, the extension of $Stut_{\mathcal{V}}$ with assignments to high variables, which is defined by the following grammar:

$$C ::= X \mid \mathsf{skip} \mid Id_H := Exp \mid C_1; C_2$$

Here X is a placeholder for meta-variables in $\mathcal{V}$, Id_H is a placeholder for program variables in *Var* with domain H, and C_1, C_2 are placeholders for commands in$Pad_{\mathcal{V}}$.

Observe that two programs C_1 and C_2 in $Pad_{\mathcal{V}}$ are related via $\doteq_L$ whenever they contain the same number of skips and assignments to high variables and the same number of occurrences of each meta-variable α. The positioning of assignments and meta-variables within the programs, however, is irrelevant. To formalize this, we denote the number of occurrences of skips and assignments in a program $C \in Pad_{\mathcal{V}}$ by $const(C)$ (skips and assignments will be the constants in our unification problem) and the number of occurrences of each meta-variable α by $|C|_\alpha$.

Lemma 5.2. *For two commands C_1 and C_2 in $Pad_{\mathcal{V}}$, we have $C_1 \doteq_L C_2$ if and only if $const(C_1) = const(C_2)$ and $\forall \alpha \in \mathcal{V}\colon |C_1|_\alpha = |C_2|_\alpha$.*

The proof of Lemma 5.2 is given in Appendix A.4.

Moreover, observe that inserting one meta-variable next to another does not create new possibilities for correcting a program. This, together with Lemma 5.2, implies that inserting one meta-variable into every subprogram within *Pad* is sufficient for allowing every possible correction. We use this insight to define the language $Mgl_{\mathcal{V}}$ of *most general liftings*. It is the sub-language of $Com_{\mathcal{V}}$ that contains all liftings of programs in *Com* in which the rightmost subterm of every sub-program in $Pad_{\mathcal{V}}$ is a (unique) meta-variable. A technical side effect of the definition of $Mgl_{\mathcal{V}}$ is the fact that it simplifies inductive proofs.

Definition 5.6. The language $Mgl_{\mathcal{V}}$ is the subset of commands given by the following grammar and in which every meta-variable occurs at most once.

$$T ::= P \mid P; Id_L := Exp; T \mid P; \mathsf{if}\ B\ \mathsf{then}\ T_1\ \mathsf{else}\ T_2; T \mid$$
$$P; \mathsf{while}\ B\ \mathsf{do}\ T_1; T \mid P; \mathsf{fork}(T_1 V); T$$

Here Id_L is a placeholder for program variables in *Var* with domain L, T, T_1, T_2 are placeholders for commands in $Mgl_{\mathcal{V}}$, V is a placeholder for a command vector in $\vec{Mgl}_{\mathcal{V}}$, and P is a placeholder for a command of the form X or of the form $C; X$ with $C \in Pad_{\mathcal{V}}$.

The liftings in $\vec{Mgl}_{\mathcal{V}}$ are most general in the sense that if two programs can be made observationally equivalent for some lifting, they can be made equivalent for any lifting chosen from $\vec{Mgl}_{\mathcal{V}}$.

$$\frac{X \text{ fresh}}{\mathsf{skip} \rightharpoonup \mathsf{skip}; X} \qquad \frac{Id : H \quad X \text{ fresh}}{Id := Exp \rightharpoonup Id := Exp; X} \qquad \frac{Id : L \quad X, Y \text{ fresh}}{Id := Exp \rightharpoonup X; Id := Exp; Y}$$

$$\frac{C_1 \rightharpoonup C_1'; X \quad C_2 \rightharpoonup C_2'}{C_1; C_2 \rightharpoonup C_1'; C_2'} \qquad \frac{C_1 \rightharpoonup C_1', \ldots, C_n \rightharpoonup C_n'}{\langle C_1, \ldots, C_n \rangle \rightharpoonup \langle C_1', \ldots, C_n' \rangle}$$

$$\frac{C_1 \rightharpoonup C_1' \quad V_2 \rightharpoonup V_2' \quad X, Y \text{ fresh}}{\mathsf{fork}(C_1 V_2) \rightharpoonup X; (\mathsf{fork}(C_1' V_2')); Y} \qquad \frac{C \rightharpoonup C' \quad X, Y \text{ fresh}}{\mathsf{while}\ B\ \mathsf{do}\ C \rightharpoonup X; (\mathsf{while}\ B\ \mathsf{do}\ C'); Y}$$

$$\frac{C_1 \rightharpoonup C_1' \quad C_2 \rightharpoonup C_2' \quad X, Y \text{ fresh}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \rightharpoonup X; (\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'); Y}$$

Figure 5.3: A calculus for computing most general liftings

Lemma 5.3. *Let $V_i \in \overrightarrow{Com}$, with liftings $V_i' \in \overrightarrow{Com}_{\mathcal{V}}$ and $V_i^* \in \overrightarrow{Mgl}_{\mathcal{V}}$ for $i = 1, 2$. Suppose V_1^* (V_2^*) shares no meta-variables with V_1', V_2', and V_2^* (V_1', V_2', and V_1^*). Then we have*

$$\mathcal{U}(\{V_1' \overset{?}{\simeq}_L V_2'\}) \neq \emptyset \textit{ implies } \mathcal{U}(\{V_1^* \overset{?}{\simeq}_L V_2^*\}) \neq \emptyset\,.$$

The proof of Lemma 5.3 is given in Appendix A.5. The calculus $\rightharpoonup$: $\overrightarrow{Com} \rightarrow \overrightarrow{Com}_{\mathcal{V}}$ turns the choice of a lifting in $\overrightarrow{Mgl}_{\mathcal{V}}$ into an algorithm. The mapping is defined inductively: a fresh meta-variable is sequentially composed with the right-hand side of each subprogram. Another fresh meta-variable is sequentially composed with the left-hand side of each assignment to a low variable, fork, while loop, or conditional. A lifting of a sequentially composed program is computed by sequentially composing the liftings of the subprograms while removing the terminal variable of the left program. The full calculus is given in Figure 5.3.

Example 5.4. Consider the program $C = \mathsf{h}_1 := 1; \mathsf{h}_2 := 1; \mathsf{l} := 0; \mathsf{h}_3 := 2$. Applying the lifting calculus to C results in the lifted program $\overline{C} = \mathsf{h}_1 := 1; \mathsf{h}_2 := 1; \alpha_1; \mathsf{l} := 0; \mathsf{h}_3 := 2; \alpha_2$. $\diamondsuit$

The program $\overline{C}$ from Example 5.4 contains substantially fewer meta-variables than the program obtained by naive insertion of meta-variables between every two subprograms. However, it still allows for every possible correction. This is a consequence of the following lemma, which shows that liftings computed by $\rightharpoonup$ are most general in the sense of Definition 5.6 and Lemma 5.3.

Lemma 5.4. *Let $V \in \vec{Com}$ and $\overline{V} \in \vec{Com}_{\mathcal{V}}$. If $V \rightharpoonup \overline{V}$ can be derived, then*

1. *$\overline{V}$ is a lifting of V and*
2. *$\overline{V} \in \vec{Mgl}_{\mathcal{V}}$.*

The proof of Lemma 5.4 is given in Appendix A.6. Putting Lemmas 5.3 and 5.4 together, we can prove the *completeness* of the calculus $\rightharpoonup$: if two programs can be made observationally equivalent for some choice of liftings, then they can also be made equivalent if we restrict ourselves to the liftings computed by $\rightharpoonup$.

Theorem 5.4. *Let $V_i \in \vec{Com}$ with liftings $V_i', \overline{V_i} \in \vec{Com}_{\mathcal{V}}$ for $i = 1, 2$. Suppose $\overline{V_1}$ ($\overline{V_2}$) shares no meta-variables with V_1', V_2', and $\overline{V_2}$ (V_1', V_2', and $\overline{V_1}$). If $V_1 \rightharpoonup \overline{V_1}$ and $V_2 \rightharpoonup \overline{V_2}$ can be derived, then*

1. *$\mathcal{U}(\{V_1' \overset{?}{\simeq}_L V_2'\}) \neq \emptyset$ implies $\mathcal{U}(\{\overline{V_1} \overset{?}{\simeq}_L \overline{V_2}\}) \neq \emptyset$, and*
2. *$\mathcal{U}(\{V_1' \overset{?}{\simeq}_L V_1'\}) \neq \emptyset$ implies $\mathcal{U}(\{\overline{V_1} \overset{?}{\simeq}_L \overline{V_1}\}) \neq \emptyset$.*

Part 2 of Theorem 5.4 shows that if a program can be repaired for some lifting (recall that if a program is $\simeq_L$-equivalent to itself, then it is also secure), then it can also be repaired if we restrict ourselves to the lifting computed by $\rightharpoonup$. Its proof is given in Appendix A.7.

5.3.3 Automating Unification

Integrating Standard Unification Algorithms. Standard algorithms for the unification modulo an associative and commutative operator with neutral element and constants (see, e.g., [8] for background information on AC_1 unification) build on solving linear equations over the number of variables and constants that occur in the unification problem. These equations correspond to the one given in in Lemma 5.2. This allows for the employment of existing algorithms for AC_1-unification problems with constants and free function symbols (such as, e.g., the one in [40]) to the unification problems that arise when applying the rule for conditionals and then to filter the output such that only preserving substitutions remain. For the reader familiar with AC_1 unification: in the language $Stut_{\mathcal{V}}$, ϵ is viewed as the neutral element, skip as the constant, and ; as the operator. Other language constructs, i.e., assignments, conditionals, loops, forks, and ; (outside the language $Stut_{\mathcal{V}}$) must be treated as free constructors.

On Most General Unifiers. When applying our transforming type system to programs with nested high conditionals, the rule [*THCond*] is applied iteratively. When choosing a unifier of the branches, care must be taken not to rule out possible corrections in subsequent applications of the rule [*THCond*]. A natural way to avoid this pitfall is to use a *most general unifier*, i.e. a unifier that can be specialized to every other unifier. Unfortunately, AC_1-unification problems with constants do, in general, not allow for most general unifiers [8]. This result carries over to $\doteq_L$.

Example 5.5. The substitutions $\eta_1 = \{\alpha_1\backslash\epsilon, \alpha_2\backslash\mathsf{skip}\}$ and $\eta_2 = \{\alpha_1\backslash\mathsf{skip}, \alpha_2\backslash\epsilon\}$ both solve the unification problem $\Delta = \{\alpha_1;\alpha_2 \doteq^?_L \mathsf{skip}\}$. In fact, every possible solution of Δ must be ground for $\alpha_1;\alpha_2$, so there can be no unifier σ with substitutions ρ_1 and ρ_2 such that $\eta_1 = \rho_1 \circ \sigma$ and $\eta_2 = \rho_2 \circ \sigma$ hold. In other words, no most general unifier exists for Δ. ◊

As in AC_1-unification problems with constants, the role of most general unifiers for $\doteq_L$ can be replaced by the more general notion of a *complete set of unifiers*. For a given unification problem Δ, a complete set of unifiers is a set $A \subseteq \mathcal{U}(\Delta)$, such that for every unifier $\sigma \in \mathcal{U}(\Delta)$, there is a substitution ρ and a $\eta \in A$ such that $\sigma = \rho\eta$. Minimal complete sets of unifiers can be computed and used in a transforming type system. To avoid backtracking in the search, such a type system could return an entire set of transformed commands. Typing a high conditional then amounts to computing a complete set of unifiers for each combination of a command from the set returned for the then-branch with a command returned for the else-branch. Each unifier is then applied to the respective pair, resulting in a set of possible transformations for the high conditional. The transforming process succeeds, if the set of possible transformations of a program is not empty. Otherwise, the program is rejected as uncorrectable. Fortunately, we can avoid this explosion in complexity by using a more problem-tailored unification algorithm.

A Problem-Tailored Solution. We next present a unification algorithm that makes use of additional information on the positions where we inserted the meta-variables and the limited range of preserving substitutions. Recall that we operate on programs in $Slice_{\mathcal{V}}$, i.e., on programs without assignments to high variables, without assignments of high expressions to low variables, and without loops or conditionals with high guards.

The unification algorithm in Figures 5.4 and 5.5 is given in the form of a calculus for judgments of the form $C_1 \doteq^?_L C_2 :: \eta$, meaning that η is a preserving unifier of

$$\frac{C_1 \stackrel{?}{\simeq}_L C_1' :: \eta_1 \quad C_2 \stackrel{?}{\simeq}_L C_2' :: \eta_2 \quad C_1, C_1' \in NSeq_{\mathcal{V}}}{C_1; C_2 \stackrel{?}{\simeq}_L C_1'; C_2' :: \eta_1 \cup \eta_2}\ [USeq_3]$$

$$\frac{C_1 \stackrel{?}{\simeq}_L C_1' :: \eta_1 \quad C_2 \stackrel{?}{\simeq}_L C_2' :: \eta_2 \quad C_1, C_1' \in Stut_{\mathcal{V}} \cup \{\epsilon\} \quad C_2, C_2' \in NStut_{\mathcal{V}}}{C_1; C_2 \stackrel{?}{\simeq}_L C_1'; C_2' :: \eta_1 \cup \eta_2}\ [USeq_4]$$

$$\frac{C_1 \stackrel{?}{\simeq}_L C_2 :: \eta \quad B_1, B_2 : L \quad B_1 \equiv B_2}{\mathsf{while}\ B_1\ \mathsf{do}\ C_1 \stackrel{?}{\simeq}_L \mathsf{while}\ B_2\ \mathsf{do}\ C_2 :: \eta}\ [UWhl] \qquad \frac{Id : L \ \ Exp_1 \equiv Exp_2}{Id := Exp_1 \stackrel{?}{\simeq}_L Id := Exp_2 :: \emptyset}\ [UAsg]$$

$$\frac{C \stackrel{?}{\simeq}_L C' :: \eta_1 \quad V \stackrel{?}{\simeq}_L V' :: \eta_2}{\mathsf{fork}(CV) \stackrel{?}{\simeq}_L \mathsf{fork}(C'V') :: \eta_1 \cup \eta_2}\ [UFrk] \qquad \frac{C_1 \stackrel{?}{\simeq}_L C_1' :: \eta_1, \ldots, C_n \stackrel{?}{\simeq}_L C_n' :: \eta_n}{\langle C_1, \ldots, C_n \rangle \stackrel{?}{\simeq}_L \langle C_1', \ldots, C_n' \rangle :: \bigcup_{i=1}^{n} \eta_i}\ [UPar]$$

$$\frac{C_1 \stackrel{?}{\simeq}_L C_1' :: \eta_1 \quad C_2 \stackrel{?}{\simeq}_L C_2' :: \eta_2 \quad B_1, B_2 : L \quad B_1 \equiv B_2}{\mathsf{if}\ B_1\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \stackrel{?}{\simeq}_L \mathsf{if}\ B_2\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' :: \eta_1 \cup \eta_2}\ [UCond]$$

Figure 5.4: Unification calculus part I

$$\frac{C \in Stut_{\mathcal{V}} \cup \{\epsilon\}}{X \stackrel{?}{\simeq}_L C :: \{X \backslash C\}}\ [UVar_1] \qquad \frac{C \in Stut_{\mathcal{V}} \cup \{\epsilon\}}{C \stackrel{?}{\simeq}_L X :: \{X \backslash C\}}\ [UVar_2]$$

$$\frac{C_1 \stackrel{?}{\simeq}_L C_2 :: \eta \quad C_1, C_2 \in Stut_{\mathcal{V}}}{X; C_1 \stackrel{?}{\simeq}_L C_2 :: \eta \cup \{X \backslash \epsilon\}}\ [USeq_1] \qquad \frac{C_1 \stackrel{?}{\simeq}_L C_2 :: \eta \quad C_1, C_2 \in Stut_{\mathcal{V}}}{C_1 \stackrel{?}{\simeq}_L X; C_2 :: \eta \cup \{X \backslash \epsilon\}}\ [USeq_1']$$

$$\frac{C_1 \stackrel{?}{\simeq}_L C_2 :: \eta \quad C_1, C_2 \in Stut_{\mathcal{V}}}{\mathsf{skip}; C_1 \stackrel{?}{\simeq}_L \mathsf{skip}; C_2 :: \eta}\ [USeq_2]$$

Figure 5.5: Unification calculus part II

the commands C_1 and C_2. The operative intuition behind the rules in Figure 5.4 is to scan two program terms from left to right and distinguish two cases: if both leftmost subcommands are free constructors, (low assignments, loops, conditionals and forks), they are compared, and, if they agree, unification is recursively applied to pairs of corresponding subprograms and the residual programs (see rules [*UAsg*], [*UWhl*], [*UCond*], and [*UFrk*]). If one leftmost subcommand is skip, both programs are decomposed into their maximal initial subprograms in $Stut_{\mathcal{V}}$ and the remaining program (see rule [$USeq_4$]). Formally, we define the language $NSeq_{\mathcal{V}}$ of commands in $Slice_{\mathcal{V}} \setminus \{\mathsf{skip}\}$ without sequential composition as a top-level operator. The language $NStut_{\mathcal{V}}$ is the set of commands in $Slice_{\mathcal{V}}$ given by the grammar $C ::= C_1; C_2$, where $C_1 \in NSeq_{\mathcal{V}}$ and $C_2 \in Slice_{\mathcal{V}}$. The rule [$USeq_3$] can then be applied to the remainders, and separates the initial free constructors from the programs that are sequentially composed to their right-hand side. Recursive decomposition eventually leads to unification problems on $Stut_{\mathcal{V}}$.

The unification algorithms for programs in $Stut_{\mathcal{V}}$ is given in Figure 5.5. The operative intuition behind it is to scan two programs in $Stut_{\mathcal{V}}$ from left to right. The rule $[USeq_2]$ removes initial skips. The rules $[USeq_1]$, $[USeq_1']$, $[UVar_1]$, and $[UVar_2]$ govern how basic unifiers are constructed: meta-variables that occur at the end of a program in $Stut_{\mathcal{V}}$ are mapped to the program to be unified with. All other meta-variables are mapped to ϵ. Note that the unifiers obtained from recursive application of the algorithm to sub-programs are combined by set union. This is admissible if the meta-variables in all subprograms are disjoint, which is the case for the unification problems that arise during transformation.

Lemma 5.5. *Let $V_1, V_2 \in \vec{Slice}_{\mathcal{V}}$ and assume that no meta-variable occurs more than once in (V_1, V_2). If $V_1 \overset{?}{\simeq}_L V_2 :: \eta$, then*

1. *$\eta \in \mathcal{U}(\{V_1 \overset{?}{\simeq}_L V_2\})$, and*

2. *η is idempotent, and*

3. *$dom(\eta) \cup var(ran(\eta)) \subseteq var(V_1) \cup var(V_2)$, and*

4. *$V_1, V_2 \in \vec{Mgl}_{\mathcal{V}}$ implies $\eta V_1, \eta V_2 \in \vec{Mgl}_{\mathcal{V}}$,*

where $var(\cdot)$ returns the set of meta-variables occurring in a command or a set of commands in $\vec{Com}_{\mathcal{V}}$.

The proof of Lemma 5.5 is given in Appendix A.8.

As Property 2 of Lemma 5.5 shows, the unifiers η computed by our calculus are idempotent, i.e. $\eta \circ \eta = \eta$ holds. Property 3 is the reason why unifiers may be combined using set union. Property 4 is a substitute for the existence of most general unifiers as, in combination with Lemma 5.3, it implies that we do not lose the possibility for subsequent corrections by applying the unifiers computed with our calculus. This fact allows us to prove the completeness of our transforming type system in the following section.

5.3.4 Completeness

Below, we show the completeness of our approach, in the sense that every program that can be repaired by inserting skip commands at arbitrary positions in the program can also be repaired using our method. In other words, by first applying the lifting calculus from Figure 5.3 and then applying the transforming type system shown in Figure 5.2, where unification is instantiated with the algorithm in Figures 5.4 and 5.5, we do not lose any possible corrections. In particular, our approach is complete with respect to applying the transforming type system to arbitrary liftings and instantiating it with an unification algorithm of choice.

Theorem 5.5 (Completeness). *Let $V \in \vec{Com}$, $\overline{V}, W \in \vec{Com}_{\mathcal{V}}$, W be a lifting of V, and $V \rightharpoonup \overline{V}$.*

1. *If there is a preserving substitution σ with $\sigma W \doteq_L \sigma W$, then $\overline{V} \hookrightarrow' V' : S$ for some $V', S \in \vec{Com}_{\mathcal{V}}$.*
2. *If $W \hookrightarrow W' : S$ for some $W', S \in \vec{Com}_{\mathcal{V}}$ then $\overline{V} \hookrightarrow' V' : S'$ for some $V', S' \in \vec{Com}_{\mathcal{V}}$.*

Here, the judgment $V \hookrightarrow' V' : S$ denotes a successful transformation of V to V' by the transforming type system, where the precondition $\sigma \in \mathcal{U}(\{S_1 \overset{?}{\doteq}_L S_2\})$ is replaced by $S_1 \overset{?}{\doteq}_L S_2 :: \sigma$ in rule [*THCond*]. The proof of Theorem 5.5 is given in Appendix A.9.

5.4 Comparison

Section 5.3 made our novel approach to transformational typing concrete in the context of a multithreaded programming language. Below, we show that this instance compares favorably with the cross-copying technique from [75], outlined in Section 5.1.1. A comparison to related approaches for sequential languages [4, 10, 39] and less restrictive notions of security for multithreaded programs [70] will be made in Chapter 7.

Improved Precision and Quality of Transformations. The type system introduced in Section 5.3 is capable of analyzing programs where assignments to low variables appear in the branches of conditionals with high guards, which is not possible with the type system in [75].

Example 5.6. If one lifts $C = \mathsf{if}\ \mathsf{h}_1\ \mathsf{then}\ (\mathsf{h}_2 := \mathit{Exp}_1; \mathsf{l} := \mathit{Exp}_2)\ \mathsf{else}\ (\mathsf{l} := \mathit{Exp}_2)$, where $\mathit{Exp}_2 : L$, using our lifting calculus, applies our transforming type system, and finally removes all remaining meta-variables by applying a projection, then this results in

$$\mathsf{if}\ \mathsf{h}_1\ \mathsf{then}\ (\mathsf{h}_2 := \mathit{Exp}_1; \mathsf{l} := \mathit{Exp}_2)\ \mathsf{else}\ (\mathsf{skip}; \mathsf{l} := \mathit{Exp}_2)\ ,$$

a program that is strongly secure and also weakly bisimilar to C. Note that the program C cannot be repaired by applying the type system from [75], as assignments to low variables occur in the branches. $\Diamond$

Another advantage of our unification-based approach in comparison to the cross-copying technique is that the resulting programs are faster and smaller in size.

Example 5.7. The program $\mathsf{if\ h\ then}\ (\mathsf{h_1} := Exp_1)\ \mathsf{else}\ (\mathsf{h_2} := Exp_2)$ is returned unmodified by our type system, while the type system from [75] transforms it into the bigger program $\mathsf{if\ h\ then}\ (\mathsf{h_1} := Exp_1; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; \mathsf{h_2} := Exp_2)$. If this type system is applied a second time, an even bigger program is obtained, namely

$$\mathsf{if\ h\ then}\ (\mathsf{h_1} := Exp_1; \mathsf{skip}; \mathsf{skip}; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; \mathsf{skip}; \mathsf{skip}; \mathsf{h_2} := Exp_2)\ .$$

In contrast, our type system realizes a transformation that is idempotent, i.e. the program resulting from the transformation remains unmodified under a second application of the transformation. This property turns out to be helpful in the context of multi-level security policies (see Section 2.4.1). $\Diamond$

The chosen instantiation of our approach preserves the program behavior in the sense of a weak bisimulation. Naturally, more programs can be corrected if one is willing to relax this relationship between input and output of the transformation. For this reason, there are also some programs that cannot be corrected with our type system, although they can be corrected with the type system in [75] (which assumes a weaker relationship between the input and the output). As the following example shows, the downside of such a more permissive transformation is that it might lead to undesired program behavior.

Example 5.8. The program $\mathsf{if\ h\ then}\ (\mathsf{while\ l\ do}\ (\mathsf{h_1} := Exp))\ \mathsf{else}\ (\mathsf{h_2} := 1)$ is rejected by our type system. The type system in [75] transforms it into the strongly secure program

$$\mathsf{if\ h\ then}\ (\mathsf{while\ l\ do}\ (\mathsf{h_1} := Exp); \mathsf{skip})\ \mathsf{else}\ (\mathsf{while\ l\ do}\ (\mathsf{skip}); \mathsf{h_2} := 1)\ .$$

Note that this program is not weakly bisimilar to the original program, as the cross-copying of the while loop introduces possible non-termination. $\Diamond$

If one wishes to permit such transformations, one could, for instance, choose a simulation instead of the weak bisimulation in a variant of our approach. This would allow for an extended range of substitutions beyond $Stut_{\mathcal{V}}$. For instance, for correcting the program in Example 5.8, one would need to instantiate a meta-variable with a while loop. In such a setting, using our approach could even further broaden the scope of corrections while retaining the advantage of transformed programs that are comparably small and fast. It is not clear to us yet, however, how the concurrent version of cross-copying from [73] could be simulated in our approach.

Multi-Level Security Policies. Non-transforming security type systems for the two-level security policy can even be used to analyze programs under a policy with more domains. To this end, multiple type checks are performed, where each pass ensures that no illegitimate information flow can occur into a designated domain. For instance, consider a three-domain policy with domains $\mathcal{D} = \{top, left, right\}$, where information may only flow from *left* and from *right* to *top*. To analyze a program under this policy, all variables with label *top* and *left* are considered as if labeled H in a first type check (ensuring that there is no illegitimate information flow to *right*) and, in a second type check, all variables with label *top* and *right* are considered as if labeled H. There is no need for a type check from the perspective of *top*, as all information may flow to *top*. When adopting this approach for transforming type systems, one must take into account that the guarantees established by the type check for one domain might not be preserved under the modifications caused by the transformation for another domain. Therefore, the process must be iterated until a fixpoint is reached for all security domains.

Example 5.9. For the three-level policy from above (assuming $\mathsf{t}_1, \mathsf{t}_2 : top$, $\mathsf{r}_1, \mathsf{r}_2 : right$ and $\mathsf{l}_1, \mathsf{l}_2 : left$), the program $\mathsf{if}\ \mathsf{t}_1\ \mathsf{then}\ (\mathsf{t}_1 := \mathsf{t}_2; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{l}_1 := \mathsf{l}_2)\ \mathsf{else}\ (\mathsf{r}_1 := \mathsf{r}_2; \mathsf{l}_1 := \mathsf{l}_2)$ is lifted to $\mathsf{if}\ \mathsf{t}_1\ \mathsf{then}\ (\mathsf{t}_2 := \mathsf{t}_2; \mathsf{r}_1 := \mathsf{r}_2; \alpha_1; \mathsf{l}_1 := \mathsf{l}_2; \alpha_2)\ \mathsf{else}\ (\mathsf{r}_1 := \mathsf{r}_2; \alpha_3; \mathsf{l}_1 := \mathsf{l}_2; \alpha_4)$ and transformed into

$$\mathsf{if}\ \mathsf{t}_1\ \mathsf{then}\ (\mathsf{t}_1 := \mathsf{t}_2; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{l}_1 := \mathsf{l}_2)\ \mathsf{else}\ (\mathsf{r}_1 := \mathsf{r}_2; \mathsf{skip}; \mathsf{l}_1 := \mathsf{l}_2)$$

when analyzing security w.r.t. an observer with domain *left*. Lifting for *right* then results in

$$\mathsf{if}\ \mathsf{t}_1\ \mathsf{then}\ (\mathsf{t}_1 := \mathsf{t}_2; \alpha_1; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{l}_1 := \mathsf{l}_2; \alpha_2)\ \mathsf{else}\ (\alpha_3; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{skip}; \mathsf{l}_1 := \mathsf{l}_2; \alpha_4)\,.$$

Unification and projection gives

$$\mathsf{if}\ \mathsf{t}_1\ \mathsf{then}\ (\mathsf{t}_1 := \mathsf{t}_2; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{l}_1 := \mathsf{l}_2; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{skip}; \mathsf{l}_1 := \mathsf{l}_2)\,.$$

Observe that this program is no longer secure from the viewpoint of a *left*–observer. Applying the transformation again for domain *left* results in the secure program

$$\mathsf{if}\ \mathsf{t}\ \mathsf{then}\ (\mathsf{t}_1 := \mathsf{t}_2; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{skip}; \mathsf{l}_1 := \mathsf{l}_2; \mathsf{skip})\ \mathsf{else}\ (\mathsf{skip}; \mathsf{r}_1 := \mathsf{r}_2; \mathsf{skip}; \mathsf{l}_1 := \mathsf{l}_2; \mathsf{skip})\,,$$

which is a fixpoint of both transformations. $\Diamond$

Note that the idempotence of the transformation is necessary (but not sufficient) for the existence of a fixpoint and, hence, for the termination of such an iterative approach. As is illustrated in Example 5.7, the transformation realized by our type system is idempotent, whereas the transformation from [75] is not.

Another possibility for tackling multi-level security policies in our setting is to unify the branches of a conditional with guard of security level D' under the theory $\bigcap_{D \not\geq D'} \doteq_D$. This would result in a multi-level, transforming security type system that supports a single-pass transformation. However, an investigation of this possibility remains to be done.

5.5 Summary

We proposed a novel approach to transformational typing, where the key idea is to use unification in order to establish the observational equivalence of alternative execution paths due to different secret inputs to a program. This yielded a new perspective on the problem of eliminating scheduling leaks.

We proved that the resulting type system is sound in the sense that every typeable program is also strongly secure. Furthermore, we proved that the transformation is complete in the sense that it can repair any program that is repairable by insertion of dummy computation steps. The main advantages of our approach are that a larger class of insecure programs can be corrected, the resulting programs are faster and smaller in size, and it offers the possibility of analyzing programs under security policies with more than two security domains.

Chapter 6

Applications

6.1 Introduction

In this chapter, we apply the techniques developed in Chapters 3 and 4 for analyzing the resistance of hardware implementations of cryptographic algorithms to timing attacks: we determine the information that is leaked through the execution time, both qualitatively (in terms of what part of the secret is leaked) and quantitatively (in terms of the resistance Φ).

We analyze three examples: two circuits for bit-serial multiplication of nonnegative integers, and a circuit for exponentiation in the field $GF(2^k)$. Exponentiation over $GF(2^k)$ is relevant, for example, in the generalized ElGamal encryption scheme, where decryption consists of one exponentiation and one multiplication step [56]. All of our example circuits are specified in the hardware description language GEZEL, which we introduced in Section 2.3.

To begin with, we will give a short description of the functionality of the circuits to be analyzed. Subsequently, we will analyze their timing side-channels qualitatively and quantitatively in Sections 6.3 and 6.4, respectively.

6.2 The Circuits

Bit-serial Multiplication. For multiplying two natural numbers m and n bitwise, consider the representation $n = \Sigma_{i=0}^{k-1} n_i 2^i$, where n_i denotes the ith bit of n. The product $m \cdot \Sigma_{i=0}^{k-1} n_i 2^i$ can be expanded to

$$(\dots((n_{k-1} \cdot m) \cdot 2 + n_{k-2} \cdot m) \cdot 2 + \dots) \cdot 2 + n_0 \cdot m\,,$$

which can easily be turned into an algorithm: starting with $p = 0$, one iterates over all the bits of n, beginning with the most significant bit. If $n_i = 1$, one updates p by adding m and then doubling p's value. Alternatively, if $n_i = 0$, one updates p by just doubling its value. At the end of the loop, $p = m \cdot n$.

We implemented two versions of this algorithm. In the first version, the doubling and adding operations each take one clock cycle. Hence, the running time reflects the number of 1-bits in n. In the second version, we introduce a dummy step whenever no addition takes place. The running time of the version padded in this manner is independent of the operands.

Both GEZEL-implementations receive inputs n and m through signals `n_in` and `m_in`, respectively. If the computation has completed, the output is given in an output signal `result` and a flag `done` that is set to signal termination. Our implementations read their input values during the first clock cycle and ignore subsequent inputs. That is, the corresponding Mealy machines are triggered (see Section 4.3.4). The full GEZEL-code of the unpadded algorithm is given in Appendix B.1.

Exponentiation in a Finite Field. We analyzed a hardware implementation of the finite field exponentiation algorithm from [29]. Basically, it consists of the following three building blocks:

1. To compute the exponentiation of a field element x with exponent $a = \Sigma_{i=0}^{n-1} a_i 2^i$, one iterates over all bits of the exponent
$$x^a = (\dots(((x^{a_{n-1}})^2 \cdot x^{a_{n-2}})^2 \cdot x^{a_{n-3}})^2 \cdot \dots)^2 \cdot x^{a_0}\,. \tag{6.1}$$
In finite fields, every element x is represented by the coefficients of a polynomial, and thus each square and each multiplication operation in Equation 6.1 is again implemented by a loop.

2. Multiplication of polynomials q and $x = \Sigma_{j=0}^{r-1} x_j T^j$ is computed using the expansion $(\dots((x_{r-1} \cdot q) \cdot T + x_{r-2} \cdot q) \cdot T + \dots) + x_0 \cdot q$ in a loop similar to the one for bit-serial multiplication.

3. At the bit level, multiplication by T of a polynomial represented by coefficients $s = (s_{r-1}, \dots, s_0)$ can be implemented as follows. If $s_{r-1} = 0$, left-shift s by one. If $s_{r-1} = 1$, left-shift s by one and XOR the result with the coefficients of the field polynomial.

The full GEZEL-code of the exponentiation algorithm is given in Appendix B.2. It receives inputs x and a through signals `x_in` and `a_in`, respectively. If the computation has completed, the output is given in an output signal `result` and a flag `done` that is set to signal termination. The implementation is triggered, that is, it reads its inputs only during the first clock cycle.

6.3 Qualitative Analysis

We analyze our examples with respect to two security properties, namely NI-security and HW-security. As previously explained, a NI-secure system does not leak any information from the high into the low domain. HW-security is a weaker property: a HW-secure system does not leak more than the Hamming weight of its secret input. Both NI-security and HW-security are instantiations of the general notion of R_I/R_O-security introduced in Section 2.4. For deciding whether a given system satisfies NI-security or HW-security, we can hence use the decision procedure developed in Section 3.2.

We implemented this decision procedure in SMV (see Section 3.4); therefore we need to translate the GEZEL-implementation to the input language of SMV. For our experiments, we did this translation by hand. However, the semantic gap between both languages is so small that an automated translation is straightforward and has already been implemented in the context of a student project [50].

6.3.1 Security Properties

NI-security. NI-security is an abbreviation of $\mathrm{All}_{\Sigma_H} \times Id_{\Sigma_L} / \mathrm{All}_{\Gamma_H} \times Id_{\Gamma_L}$-security (see Section 2.4). In Section 3.4, we showed how SMV can be used for deciding NI-security. We use this implementation in our experiments. As in Section 3.4, we assume that the translations to SMV of our circuits receive high and low input in variables `hi_in` and `lo_in`, respectively, and provide high and low output in variables `hi_out` and `lo_out`, respectively.

In our translation from GEZEL to SMV, the secret input signals of the exponentiation and multiplication algorithms (`a_in` and `n_in`, respectively) are mapped to `hi_in`. The public input signals (`x_in` and `m_in`, respectively) are mapped to `lo_in`. The output signals `done` are mapped to the SMV-variable `lo_out`. As we are only interested in timing side-channels, we ignore the output `result` by mapping it to the variable

```
sys2.hi_in:=
case
  hi1[0]+...+hi1[SIZE-1]=hi2[0]+...+hi2[SIZE-1] : hi2;
  1 : hi1;
esac;
```

Figure 6.1: Deciding HW-security in SMV

hi_out.

HW-security. For $\Sigma_H = \{0,1\}^n$, we define HW-security as an abbreviation for $\Psi \times Id_{\Sigma_L}/\mathrm{All}_{\Gamma_L} \times Id_{\Gamma_L}$-security, where $\Psi = \{(a,b) \in \Sigma_H \times \Sigma_H \mid \|a\| = \|b\|\}$ and where $\|x\|$ denotes the Hamming weight of x. That is, a HW-secure system produces indistinguishable output whenever it is provided with two input sequences that are indistinguishable with respect to the Hamming weight of corresponding high inputs. As Proposition 4.2 shows, we have $\pi_\Psi \sqsubseteq \pi_\mathfrak{a}$ for all attack strategies $\mathfrak{a}$ against a HW-secure system. Hence no attacker can deduce more than the Ψ-equivalence class of the secret input or, equivalently, the secret input's Hamming weight.

As for NI-security, we can implement the decision procedure for HW-security in SMV. The construction is similar to that for NI-security, described in Section 3.4; the only difference is that we modify the input to hi_in of sys2 in line 10 of Figure 3.1, as described in Figure 6.1. The variables hi1 and hi2 both take all possible values in their range. Only when their Hamming weight coincides is sys2 fed with hi2. Otherwise its input is hi1. In this way, we ensure that the inputs to both instances of circuit always have the same Hamming weight and that all such combinations are considered.

For deciding the HW-security with respect to an observer who may only measure the timing behavior, we map the secret input signals of the exponentiation and multiplication algorithms (a_in and n_in, respectively) to hi_in. The public input signals (x_in and m_in, respectively) are mapped to lo_in. The output signals done are mapped to the SMV-variable lo_out. As we are only interested in timing side-channels, we ignore the output result by mapping it to the variable hi_out.

6.3.2 Results

The table in Figure 6.2 presents the results of our analysis. The first column corresponds to the serial multiplication algorithm where dummy steps are inserted to avoid

	Multiplication (padded)	Multiplication (unpadded)	Exponentiation
NI-security	✓	×	×
HW-security	✓	✓	×

Figure 6.2: Results of the analysis

timing leaks. The second column corresponds to the multiplication algorithm without dummy steps, and the third column contains the results for the finite-field exponentiation algorithm. The rows correspond to NI-security and HW-security, respectively. An entry ✓ denotes that the model is secure with respect to the corresponding notion of security, whereas × denotes that this is not the case.

Integer Multiplication (padded). The first column reflects what was intended by inserting dummy computation steps into the design: the circuit's execution time is independent of the input to the signal `n_in` and, hence, it is NI-secure.

Integer Multiplication (unpadded). The second column shows that the circuit implementing multiplication without dummy computation is not NI-secure, that is, its running time depends on the input to the signal `n_in`. However, it is HW-secure, that is, if the implementation is only run on inputs with equal Hamming weight, then no differences in the circuit's running times can be observed. As the circuit is triggered, no more than the Hamming weight of the input can be leaked.

Finite Field Exponentiation. The third column shows that the running time of the exponentiation algorithm depends on the input to the input signal `a_in`, which is the exponent. When considering only loop 1 (see Section 6.2), one might expect the same result as for bit-serial multiplication. However, the circuit is not HW-secure, which shows that this is not the case: even when provided with input of the same Hamming weight, the system shows differences in its running times. In the counterexample computed by SMV, the first difference between the sequences of states reached in both instances of `circuit` occurs after 20 steps. Distinguishable output is not produced until 36 steps, which corresponds to the minimal number of clock cycles required for any exponentiation with this circuit. A failed check for HW-security implies that information other than the Hamming weight can be extracted in an attack. In Section 6.4, we will subject this circuit to a quantitative analysis, which will allow us to analyze the information that is leaked in more detail.

Performance. We performed our experiments on a 2.4 GHz machine with 3 gigabytes of RAM. In the case of serial multiplication, we were able to analyze designs up to 10 bits per operand within one minute. In the case of exponentiation, we were able to analyze designs with up to 3 bits per operand within 2 minutes.[1] For larger bit-widths, the running times increased notably. Note that these numbers were obtained by using SMV "out of the box", that is, without applying one of the many existing optimization techniques. We expect a significant performance gain by tailoring the search procedure to our specific problem instance, for example by adopting abstraction techniques for handling bit-vectors.

6.4 Quantitative Analysis

In this section, we analyze the presented circuits with the quantitative techniques developed in Chapter 4. Our goal is to determine the resistance Φ of our implementations with respect to timing attacks. Throughout this section, we use the guessing entropy G as a measure of uncertainty and we abbreviate Φ_G by Φ and $\hat{\Phi}_G$ by $\hat{\Phi}$, respectively. We assume a uniform probability distribution of the secrets and we compute the remaining uncertainty with the formula given in Proposition 4.3.2.

6.4.1 Approximation Techniques

Computing Φ using the algorithm from Theorem 4.1 is expensive. The time required is doubly exponential in the number of attack steps, and the sizes of the input parameter sets are exponential in the number of bits used to represent the parameters. Hence, we cannot feasibly compute Φ for large parameter sizes. We use two approximation techniques to address this problem.

1. We approximate Φ by $\hat{\Phi}$. We will see that $\hat{\Phi}$ matches Φ on our example data, although this does not hold in general (see Example 4.9).

2. We parameterize each algorithm by the bit-width w of its operands. Our working assumption is that regularity in the values of Φ for $w \in \{2, \dots, w_{\max}\}$ reflects the

[1]This corresponds to a state-space size of approximately 2^{52} for the product automaton.

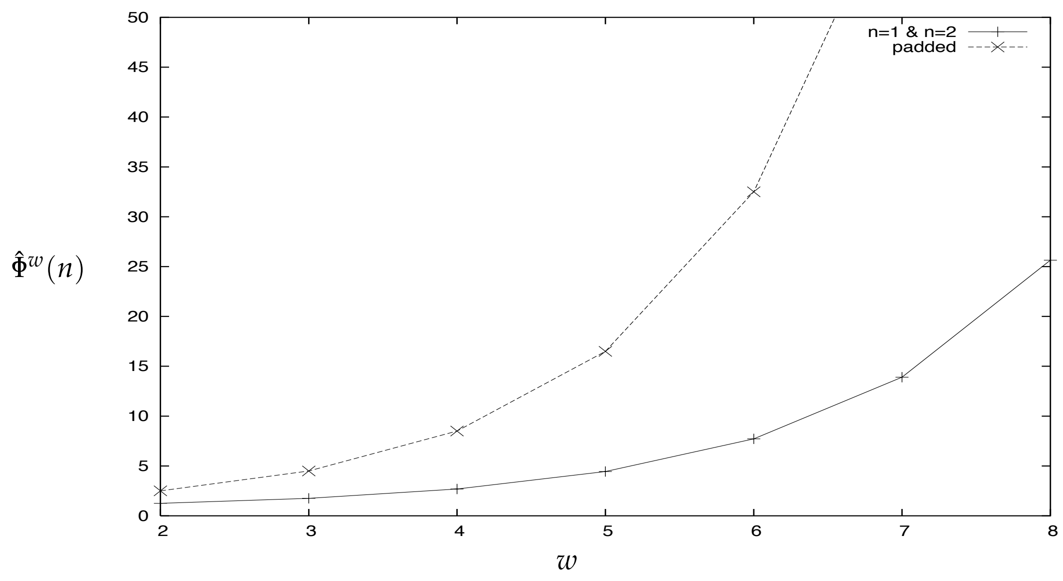

Figure 6.3: Integer multiplication

structural similarity of the parameterized algorithms. This allows us to extrapolate to values of w beyond w_{max}. To make this explicit, we will write Φ^w to denote that Φ is computed on w-bit operands.

For each algorithm and each bit-width $w \in \{2,\dots,8\}$, we use the GEZEL simulator to build up value tables for the timing side-channel $f : \{0,1\}^w \times \{0,1\}^w \to \mathbb{N}$. From this value table, we extract the partitions that we use as input to the HASKELL-implementation of $\hat{\Phi}$.

Next we present our experimental results and discuss their implications.

6.4.2 Results

Integer Multiplication (unpadded). The results of the analysis of the unpadded integer multiplication algorithm are depicted by the solid line in Figure 6.3. For their interpretation, observe that $\hat{\Phi}^w(1) = \hat{\Phi}^w(2)$ holds. Hence, according to Proposition 4.5, the graph actually depicts Φ^w. There are two conclusions to be drawn from Figure 6.3. First, the circuit's timing behavior depends on the number of 1-bits in the secret. This leads to the hypothesis that the Hamming weight of the secret is revealed or, equivalently, that two secrets are indistinguishable iff they have the same Hamming weight. The equivalence class of w-bit arguments with Hamming weight k has precisely $\binom{w}{k}$ elements. Hence, according to Proposition 4.3, the conditional guessing entropy for

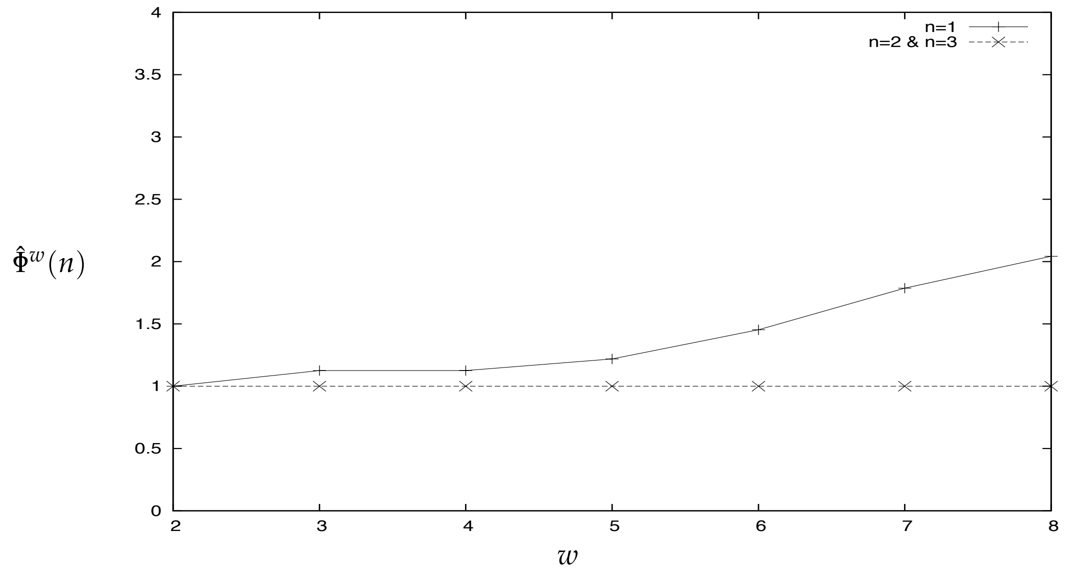

Figure 6.4: Finite-field exponentiation

the corresponding partition is given by $\frac{1}{2^{w+1}}\sum_{k=0}^{w}\binom{w}{k}^2+\frac{1}{2}$. The values computed using this expression match the solid curve in Figure 6.3, which confirms our hypothesis and the result from the qualitative analysis in Section 6.3.2.

Second, Figure 6.3 shows that a single side-channel measurement is enough to extract the maximal information revealed by the circuit's timing behavior. This follows, as $\Phi^w(1)$ and $\Phi^w(2)$ coincide, and is due to the fact that the circuit's running time is independent of the public parameter. It is beyond the scope of information-flow analyses, such as the ones developed in Chapter 3, to reason about the number of measurements needed to obtain information.

Integer Multiplication (padded). The result of the analysis of the padded integer multiplication algorithm is given by the dashed line in Figure 6.3. The curve matches the guessing entropy for a secret without side-channel information, given by $0.5(2^w+1)$. This implies that the circuit's timing behavior does not leak any secret information, which confirms the result from our qualitative analysis in Section 6.3.2.

Finite Field Exponentiation. The results of the analysis of the finite field exponentiation algorithm are given in Figure 6.4. For their interpretation, observe that $\Phi^w(1)=\hat{\Phi}^w(1)$ and $\hat{\Phi}^w \geq \Phi^w$ follow from Proposition 4.5. We conclude that one timing measurement reveals a quantity of information larger than that contained in the Hamming weight, but that it does not completely determine the secret. A second measurement, however, can reveal all remaining information about the secret. Hence, the exponentiation algorithm is vulnerable to timing attacks.

Performance and Scaling-Up. With precomputed value tables for the time consumption of the finite field exponentiation algorithm, the computation of $\hat{\Phi}^8(2)$ took 40 minutes on a 2.4 GHz machine with 3 gigabytes of RAM. However, the algorithms presented in Section 4.5 rely on the complete enumeration of the set of secrets and therefore do not scale. Fortunately, our data shows regularity and we can successfully extrapolate to larger bit-widths. Under our working assumption that the regularity in the data reflects the structural similarity of the parameterized algorithms, we conclude that the interpretations given for each algorithm hold, regardless of the implementation's bit-width.

6.5 Conclusions

In this chapter, we analyzed nontrivial hardware implementations for their vulnerability to timing attacks. We thereby illustrated the scope and the power of the analysis techniques developed in Chapters 3 and 4.

The analysis methods we presented in Section 3 allowed us to precisely characterize the maximal information that an attacker can extracted from a circuit by observing its timing behavior. Both the formal characterization and the formal verification of the leakage of the Hamming weight through timing behavior are beyond the scope of existing methods for information flow analysis. Our method also allowed us to detect that the finite field exponentiaton algorithm leaks information other than the Hamming weight of its exponent. However, it is beyond the scope of the techniques described in Chapter 3 to fully characterize this information or to reason about the attacker's effort (in terms of side-channel measurements) to obtain it.

The techniques we presented in Chapter 4 allowed for a more detailed analysis. For the integer multiplication algorithms, they allowed us to conclude that the maximally leaked information is already contained in one measurement. For the exponentiation algorithm, they allowed us to conclude that all the exponent information can be extracted, and that this can be achieved in only two measurements. To our knowledge, this is the first approach that allows for the expression (and computation) of the side-channel leakage of a system as a function of the number of measurements made by an attacker.

Chapter 7

Related Work

7.1 Attacks and Countermeasures

Kocher's timing attack [43] is the first side-channel attack against a cryptographic algorithm. Since its publication, the timing behavior of a number of algorithms and implementations has been exploited for side-channel cryptanalysis [31, 19, 15, 2]. Most notably, Boneh and Brumley [15] extracted 1024 bit private keys within 2 hours by attacking a standard OpenSSL RSA implementation on a remote server. Cache attacks (see, e.g., [62, 3]) are timing attacks that exploit the timing differences between cache misses and cache hits in multi-purpose processors. Attacks against the branch target buffer of instruction pipelines [1] exploit similar mechanisms and can hence be seen as instances of cache attacks. Caching side-channels cannot be directly captured in the quantitative model presented in Chapter 4, as our model of explicit side-channels is stateless.

Characteristics such as power consumption [44, 21, 69] and electromagnetic radiation [33, 68] have also been exploited for cryptanalysis. In [47], we showed that the model presented in Chapter 4 can also be used to give information-theoretic bounds for power attacks.

A number of countermeasures against side-channel attacks have been proposed. Here, we focus on countermeasures for defeating timing attacks, and we illustrate them with the algorithm from Example 1.1. One approach is to add random delays to the running time in order to make precise timing measurements impossible [43]. However, the author already observed that this technique can be rendered useless by increasing the number of timing measurements and averaging their values. Another countermeasure are blinding techniques [43, 15], and we focus on exponent blinding for modular exponentiation for illustration purposes. In exponentiation modulo n, the exponent k can be blinded by adding $r\phi(n)$, where r is a random natural number and ϕ is Euler's function. Due to the Fermat-Euler theorem, $c^{k+r\phi(n)} = c^k c^{r\phi(n)} = c^k \mod n$. Applying this technique, c^k can be computed with the algorithm in Example 1.1, with a running time that depends on $k + r\phi(n)$ rather than directly on k. The blinding of the message instead of the exponent is also possible, but requires an unblinding step.

Although blinding techniques are often the preferred solution in practice, they are not a general remedy for timing leaks. First, as can be seen in our example, blinding relies on the algebraic properties of the computed function. It works for securing RSA, but is more difficult to apply to algorithms for computing functions with a less obvious algebraic structure, such as AES or examples outside the realm of cryptography. Second, potential new side-channels are introduced during the blinding (and unblinding) steps. In this light, blinding simply relocates the problem of side-channels to a different part of the implementation. Although we are not aware of a documented exploit, timing differences in the blinding steps can, in principle, be exploited by side-channel attacks. Another countermeasure is to ensure that the implementation exhibits a constant execution time [43]. As we have seen, this countermeasure yields the provable absence of timing leaks and its application can, to some extent, be automated. Its drawbacks are that it is difficult to achieve constant running times on many platforms and that the performance of a constant-time implementation may be unacceptable for some applications. A less restrictive way of dealing with timing leaks is to ensure that only an acceptable amount of secret information is revealed through them.

7.2 Related Theory

7.2.1 Notions of Security

A number of timing-sensitive notions of secure information flow have been proposed. Agat [4] augments the program semantics with labels that express the elapse of time consumed by the corresponding command. He uses bisimulation equivalence to capture timing side-channels. Barthe et al. [10] use a similar semantics, but express timing-sensitive security without bisimulations. Hedin and Sands [39] augment the semantics with timing information and an execution history, that (in principle) allows for cache behavior to be incorporated. All approaches only sketch how the connection between transition labels and the elapse of real time on target processors can be made, which is a nontrivial task. Tolstrup [87] augments the semantics of the hardware description language VHDL with timing information. This has the advantage of a very concrete system model, but requires him to deal with many peculiarities of the VHDL execution model, such as processes and delta-time.

First approaches that consider partial information flow can be found in [25]. The use of arbitrary equivalence relations for capturing partial information flow was proposed in a timing-insensitive context [9, 35]. The definition of R_I/R_O-security marries the timing-awareness of the bisimulation-based approaches with the accuracy of the parameterized approaches. In [36], a parameterized and timing-sensitive definition of secure information flow is given. However, it does not allow for input sequences of arbitrary length and it is unclear whether it can be checked efficiently.

A number of information flow properties to capture the information flow in multithreaded programs have been proposed [84, 92, 16, 83]. Initial approaches express security under possibilistic schedulers [84] and security under a fixed probabilistic scheduling policy [92]. Strong security [75] offers security guarantees that hold in the presence of a large class of schedulers and it can be expressed in terms of R_I/R_O-security, as we have seen. Furthermore, it provides security guarantees with respect to observers who can see the low variables during the entire program run. More recent approaches to counter scheduling leaks build on less restrictive notions of security. The information flow properties of [70, 71] rely on the assumption that the values of the low variables can only be observed after the program has terminated and they hold for a restricted class of schedulers. A consequence of stronger assumptions about the scheduler and weaker assumptions about the attacker is that a larger class of programs are considered as secure.

There has been substantial work in information-flow security on quantifying information leaks, but the results are only partially applicable to the problem of analyzing how much information a side-channel attacker can extract from a system. Early approaches focus on quantifying the capacity of covert channels between processes in multi-user systems [59, 96, 38]. The models predate the first published side-channel attack against cryptography [43] and are so general that it is unclear whether and how they could be instantiated to quantify the side-channel leakage of cryptographic algorithms. Di Pierro et al. [65] show how to quantify the number of statistical tests an observer needs to perform to distinguish two processes in a probabilistic concurrent language. Lowe [51] quantifies information flow in a possibilistic process algebra by counting the number of distinguishable behaviors. The information measures proposed by Clark et al. [22] are closest to those we introduced in Chapter 4. The authors relate observational equivalence to random variables and use Shannon entropy to quantify the information flow. However, their measure captures the information gain of a passive observer instead of an active attacker: the public input to the system is chosen with respect to a probability distribution and is not under the attacker's control.

7.2.2 Decidability Results

Product constructions, such as the one we use for deciding $R_{/}R_{O}$-security in the deterministic case, have been used for expressing information flow in timing-insensitive settings [28, 9]. Decision procedures also exist for timing-sensitive notions of security [32]. The machine model used is considerably different and no complexity results are given, making a detailed comparison difficult.

Our decision procedure for nondeterministic R_I/R_O-security is a generalization of the decision procedure for bisimulation equivalence given in [42]. The authors use partition refinement for deciding process equivalence, an approach that is further optimized in [63]. Independently of our work, Dam [27] proposed a decision procedure for strong security, which relies on the refinement of equivalence relations. It allows one to decide strong security for a while-language without dynamic thread creation, if the equivalence $Exp_1 \equiv Exp_2$ of arbitrary expressions Exp_1, Exp_2 is decidable. The time complexity of Dam's decision procedure is a polynomial of degree 4 in the size of the state space, which matches the complexity bounds of our approach. Dam's approach is more general in that our restriction to finite memories is a sufficient but not a necessary condition for the decidability of expressions. Our approach is more general in that we can instantiate R_I and R_O to security properties beyond non-interference.

7.2.3 Security Type Systems

Early approaches for analyzing the information flow at the level of programming languages can be found in [30, 25]. A rigorous connection of syntax-based analyses with semantic-based notions of security was made in [93]. This connection comes in the form of a soundness result for a security type system, which guarantees that a typeable program exhibits secure information flow. Such soundness proofs have since become standard in theoretical works in the field. For an overview of language-based approaches to information-flow security, refer to [74].

A number of security type systems to counter timing leaks on a programming-language level [4, 10, 39, 88] exist. We discussed the underlying notions of security in Section 7.2.1. Here, we focus on type systems for multithreaded programs and on mechanisms that involve program modifications. The focus of such transformations has been on the problem of making the branches of conditionals observationally equivalent. An early proposal, though not yet a transformation, can be found in the work by Volpano and Smith [92]. They investigate a simple multithreaded programming language where they require entire conditionals to be executed atomically, ruling out differences in the duration of alternative execution paths. Moreover, they forbid assignments to variables that are observable for the attacker in order to avoid other differences in the observable behavior. However, the atomic execution of entire conditionals constrains parallelism and it is not obvious how the requirement of atomic execution can be practically enforced [70]. The cross-copying technique, a less restrictive solution, was proposed by Agat [4] and modified to fit into a multithreaded scenario by Sabelfeld and Sands [75]. Both transforming type systems sequentially compose each branch of a conditional with a program that simulates the timing behavior of the other branch. This cross-copying transformation ensures identical timing behavior of the branches in the transformed program. As in [92], assignments to variables are forbidden if their values can be observed by the attacker. Another problem is that the transformation can introduce non-termination into a program. This problem is addressed, and partially solved, in [73] by composing the cross-copied programs concurrently (rather than sequentially).

There is a natural trade-off between the permissiveness of a type system and the strength of the security guarantees it provides. Recent transformational approaches achieve weaker security guarantees while broadening the scope of program transformations. Russo et al. [70] propose a transformation that spawns new threads to close timing leaks, and they prove its soundness for a round-robin scheduler and attackers who can see the values of the low variables only upon termination of the program. The transformation type system is more permissive than the transformation presented in Chapter 5, in that it allows one to secure loops with high guards. The approach from [71] allows for the securing of loops with high guards, and it is also applicable to multi-level security policies. It relies on a modified runtime environment rather than on a program transformation. The authors show that these modifications can be applied to existing thread libraries, which is a promising step towards practical applicability.

Finally, type systems and model-checking techniques can be combined. Unno et al.

[90] use a failed type check to identify subprograms that might cause illegal information flow. Model-checking techniques are then applied to these subprograms in order to identify a pair of execution paths that witness that non-interference is indeed violated. The restriction to subprograms reduces the complexity of the model-checking problems. In this way, the efficiency of a type-based analysis can be combined with the precision of a model-checking approach.

7.2.4 Models of Attacks

Models and theoretical bounds on what side-channel attackers can achieve are only now emerging. Chari et al. [20] are the first to present methods for proving hardware implementations secure. They propose a generic countermeasure for power attacks and prove that it resists a given number of side-channel measurements.

In our model of adaptive attacks, we represent the attacker's knowledge about the secret as a set of possible values. Similar representations of knowledge have been used in different contexts. König et al. [45] consider abstract storage devices. Although they do not explicitly address side-channel attacks, their formalization of a read operation corresponds to our formalization of single attack step in Chapter 4. It is not obvious whether and how the theoretical results for abstract storage devices can help with the analysis of side-channel attacks. Askarov and Sabelfeld [7] use the set of possible initial states of a program to express a passive observer's knowledge about a secret. This set diminishes as the program run evolves and the number of observations increases.

Clarkson et al. [24] develop a model that allows for reasoning about an attacker's changing belief about a secret when he observes a run of a program. By considering multiple runs where the attacker's (post-)belief after one run is his (pre-)belief before the next run, one can, in principle, capture adaptive attacks. Beliefs about secrets may also be wrong. This has no correspondence in our model, where we assume that the attacker can make noiseless measurements and has full knowledge about the implementation. It would be interesting to see whether beliefs could be used in order to weaken our assumption of error-free measurements.

Micali et al. [58] propose physically observable cryptography, a mathematical model that aims at providing provably secure cryptography on hardware that is only partially shielded. Their model has recently been specialized by Standaert et al. [86, 85], who show how assumptions on the computational capabilities of an attacker can be combined with leakage functions that measure the information that is revealed by the system's side-channels. How to instantiate these leakage functions is an open problem.

As the approach focuses on power attacks and does not take into account adaptive attackers, using our model to instantiate these functions is not straightforward and we leave a detailed investigation of this possibility to future work.

Chapter 8

Conclusions

8.1 Summary of Contributions

We have presented R_I/R_O-security, a parametric and timing-sensitive information flow property that can be instantiated to a number of relevant security properties, including the strong security of multithreaded programs and properties that allow for the expression of partial information release for Mealy machines. We have given algorithms and complexity bounds for the problem of deciding R_I/R_O-security on deterministic and nondeterministic finite-state systems. The algorithms for the deterministic case can easily be implemented in the model-checker SMV, and we have applied them to analyze the timing side-channels of nontrivial synchronous circuits with inputs of small bit-widths. For this application domain, we thus solve the open problem of detecting (timing) side-channels.

The mere detection of a side-channel does not give detailed information about how vulnerable to attacks an implementation actually is: it is possible that only a negligible amount of secret information is leaked through the channel, or that an attacker's effort for obtaining sufficient side-channel information is too high to mount an attack in practice. We have presented a quantitative model of adaptive side-channel attacks that allows for a more detailed assessment of an implementation's vulnerability to side-channel attacks. To this end, we have formalized attack strategies and combined them with information-theoretic entropy measures. This allowed us to define the attacker's remaining uncertainty about the secret as a function of the number of side-channel measurements made. We have shown how this function can be computed, and we have provided techniques that allow it to be approximated for larger bit-widths. We have implemented our technique and we have used it to derive meaningful assertions about the vulnerability of hardware implementations to timing attacks. For a deterministic and stateless model of side-channels, we thus solve the open problem of quantifying the information that an adaptive attacker can extract in a given number of measurements.

Finally, we have proposed a unification-based approach to eliminating scheduling leaks in multithreaded programs. It repairs insecure programs for which all existing approaches fail, and the results compare favorably with those of other transformations: the transformed programs are often faster and smaller in size. Furthermore, we showed that our transformation is applicable for enforcing multi-level security policies. We thus improve on the state of the art for transforming multithreaded programs secure.

8.2 Outlook

An area of future work for the detection of side-channels in synchronous hardware along the lines of Chapter 3 would be to investigate algorithms and abstractions that help us manage both larger systems and those with infinite state spaces. Techniques to consider are model-checking approaches that more efficiently handle bit-vectors, and data-independence-based abstractions. One could also investigate approaches to automatically eliminating information leaks, either on the level of a hardware description language or on the level of transition systems, e.g. as in [82, 64].

There are a variety of possible extensions to the quantitative model we presented in Chapter 4. First, one could weaken the assumption of error-free measurements. It would be interesting to see whether it is possible to retain the model's simplicity and applicability. Second, how to scale our techniques to larger bit-widths is an open issue. For this, it is necessary to investigate more efficient algorithms and approximation techniques for computing Φ. The most promising approach we have looked into so far is the use of techniques for entropy estimation [11, 13]. Initial experimental results of a student project [34] are encouraging: we were able to confirm that the presented integer multiplication algorithm reveals one operand's Hamming weight – for implementations with 100 bits per operand and with an error of less than 1%. However, the existing confidence intervals for this estimation are too large for practical use, and, as future work, we hope to improve them. Third, our model could be integrated with computational attacker models, e.g. along the lines of [86], for deriving even more realistic bounds for attackers that are computationally limited.

It is an open question as to whether and how the unification-based transformation we presented in Chapter 5 can be applied to other notions of observational equivalence and other notions of program equivalence. Finally, it would be desirable to integrate our fully automatic transformation into an interactive framework for supporting the programmer in correcting insecure programs.

8.3 Final Remarks

Preserving the confidentiality of secrets in computing environments is a fundamental problem of information security. A large body of work in information-flow security is concerned with models for its formalization and methods for its mitigation. Although the field has attracted many researchers and has produced a great number of models and methods, its results have not been widely adopted in practice. This has been observed critically by a number of researchers in the field [60, 72, 97] and they have identified different reasons for this. These reasons include that, first, non-interference is a too strong property for practical use. For most applications, the existence of potential channels is acceptable, as long as their "bandwidth" is sufficiently low. Second, it is difficult to give faithful abstract models of many systems. If the covert channel is not modeled precisely, it is unclear what is gained by a formal security analysis. Third, for many kinds of covert channels, there are no documented real-world exploits. This makes it difficult to justify laborious verification efforts for covert channel analysis.

Our work on eliminating scheduling side-channels was mainly motivated by an interest in exploring the use of unification as a means for achieving observational equivalence, which seems a natural choice in the Per model of security. Our primary goal was not immediate applicability and, indeed, our work shares the aforementioned limitations with many approaches in information-flow security.

In our work on the analysis of timing side-channels, we have addressed the gap between model and reality: we focused on synchronous hardware, where faithful timing models are available. As we are up against real exploits, we were also able to base our analysis on a concrete and realistic model of the attacker. This allowed us to derive meaningful quantitative notions of information flow beyond non-interference. Finally, we demonstrated by example that an analysis using our methods can be performed at the push of a button. Hence, for a well-defined application domain and a concrete threat model, our methods avoid the aforementioned obstacles for practical impact and we believe that they can become a valuable part of future side-channel analysis toolkits.

Appendix A

Proofs of the Technical Results

A.1 Proof of Theorem 5.1

Before proving Theorem 5.1, we introduce a lemma and prove it using the bisimulation-up-to technique.

Definition A.1. A binary relation R on commands is a *strong low bisimulation up to* $\cong_L$ if R is symmetric and

$$\begin{aligned}
&\forall C, C', C_1, \dots, C_n \in \mathit{Com}. \forall \nu, \nu', \mu \in \mathit{Mem}. \\
&\quad (C \; R \; C' \wedge \nu =_L \nu' \wedge \langle\!| C, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle C_1, \dots, C_n \rangle, \mu |\!\rangle) \\
&\quad \Rightarrow \exists C'_1, \dots, C'_n \in \mathit{Com}. \exists \mu' \in \mathit{Mem}. (\langle\!| C', \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle C'_1, \dots, C'_n \rangle, \mu' |\!\rangle \\
&\qquad\quad \wedge \forall i \in \{1, \dots, n\}. C_i \; (R \cup \cong_L)^+ \; C'_i \wedge \mu =_L \mu')
\end{aligned} \tag{A.1}$$

Lemma A.1. *If R is a strong low bisimulation up to $\cong_L$ then $R \subseteq \cong_L$ holds.*

Proof. Let $Q = (R \cup \cong_L)^{\uparrow}$, where $^{\uparrow}$ denotes the pointwise lifting of a relation on commands to command vectors (here, $\cong_L$ is viewed as a relation on commands). It is sufficient to show $Q \subseteq \cong_L$. To this end, we show that Q satisfies condition 2.2 in Definition 2.7, and is therefore contained in $\cong_L$, the union of all such relations. Let $V = \langle C_1, \dots, C_n \rangle$ and $V' = \langle C'_1, \dots, C'_n \rangle$ with $(V, V') \in Q$ and let $\nu, \nu' \in \mathit{Mem}$ with $\nu =_L \nu'$. Suppose $\langle\!| V, \nu |\!\rangle \twoheadrightarrow \langle\!| W, \mu |\!\rangle$. By definition of the operational semantics, there is an $i \in \{1, \dots, n\}$ with $\langle\!| C_i, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle C_{i,1}, \dots, C_{i,m} \rangle, \mu |\!\rangle$ and $W = \langle C_1, \dots, C_{i-1}, C_{i,1}, \dots, C_{i,m}, C_{i+1}, \dots, C_n \rangle$. We distinguish between the cases $(C_i, C'_i) \in \cong_L$ and $(C_i, C'_i) \in R$.
$(C_i, C'_i) \in \cong_L$: By Definition 2.7, there are $C'_{i,1}, \dots, C'_{i,m}$ with $\langle\!| C'_i, \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle C'_{i,1}, \dots, C'_{i,m} \rangle, \mu' |\!\rangle$ where $\langle C_{i,1}, \dots, C_{i,m} \rangle \cong_L \langle C'_{i,1}, \dots, C'_{i,m} \rangle$ and $\mu =_L \mu'$. From Definition 2.7 it follows that $C_{i,j} \cong_L C'_{i,j}$ holds for all $j \in \{1, \dots, m\}$.

$(C_i, C_i') \in R$: By Definition A.1, there are $C_{i,1}', \ldots, C_{i,m}'$ with $\langle\!| C_i', \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle C_{i,1}', \ldots, C_{i,m}' \rangle, \mu' |\!\rangle$ where $\mu =_L \mu'$ and $C_{i,j}(R \cup \approxeq_L)C_{i,j}'$ for all $j \in \{1, \ldots, m\}$.

Hence, with $W' = \langle C_1', \ldots, C_{i-1}', C_{i,1}', \ldots, C_{i,m}', C_{i+1}', \ldots, C_n' \rangle$ we have $\langle\!| V', \nu' |\!\rangle \twoheadrightarrow \langle\!| W', \mu' |\!\rangle$ with $(W, W') \in Q$ and $\mu =_L \mu'$. As $\approxeq_L$ is defined to be the union of all symmetric relations with the condition 2.2, $Q \subseteq \approxeq_L$ follows. □

Lemma A.2.

1. *If $Id : H$ then skip $\approxeq_L$ $Id := Exp$.*
2. *If $Exp, Exp' : L$ and $Exp \equiv Exp'$ then $Id := Exp \approxeq_L Id := Exp'$.*
3. *If $C_1 \approxeq_L C_1'$ and $C_2 \approxeq_L C_2'$ then $C_1; C_2 \approxeq_L C_1'; C_2'$.*
4. *If $C_1 \approxeq_L C_1'$ and $V_2 \approxeq_L V_2'$ then fork$(C_1 V_2) \approxeq_L$ fork$(C_1' V_2')$.*
5. *If $B, B' : L$, $B \equiv B'$, and $C_1 \approxeq_L C_1'$ then while B do $C_1 \approxeq_L$ while B' do C_1'.*
6. *If $B, B' : L$, $B \equiv B'$, $C_1 \approxeq_L C_1'$, and $C_2 \approxeq_L C_2'$ then if B then C_1 else $C_2 \approxeq_L$ if B' then C_1' else C_2'.*
7. *If $C_1 \approxeq_L C_1'$ and $C_1 \approxeq_L C_2'$ then skip; $C_1 \approxeq_L$ if B' then C_1' else C_2'.*

Proof. We illustrate the proof idea with three example cases. In each case, we prove the strong low bisimilarity of the two commands with the bisimulation up-to technique. That is, we define a binary relation R on commands that relates the two commands and prove that R is a strong low bisimulation up to $\approxeq_L$. From Lemma A.1, we then obtain that the two given commands are strongly low bisimilar.

1. Define R as the symmetric closure of the relation $\{(\text{skip}, Id := Exp) \mid Id : H\}$. Let $(\text{skip}, Id := Exp) \in R$ and $\nu, \nu' \in Mem$ be arbitrary with $\nu =_L \nu'$. From the operational semantics, we obtain $\langle\!| \text{skip}, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, \nu |\!\rangle$. Moreover, there is a $\mu' \in Mem$ such that $\langle\!| Id := Exp, \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, \mu' |\!\rangle$. From $\nu =_L \nu'$ and $Id : H$, we obtain $\nu =_L \mu'$.

   Let $(Id := Exp, \text{skip}) \in R$ and $\nu, \nu', \mu \in Mem$ be arbitrary with $\nu =_L \nu'$ and $\langle\!| Id := Exp, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, \mu |\!\rangle$. We have $\langle\!| \text{skip}, \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, \nu' |\!\rangle$. From $\nu =_L \nu'$ and $Id : H$, we obtain $\mu =_L \nu'$.

   Hence, R is a strong low bisimulation up to $\approxeq_L$.

2. Define R as the symmetric relation $\{(C_1;C_2, C_1';C_2') \mid C_1 \cong_L C_1', C_2 \cong_L C_2'\}$.

   Let $(C_1;C_2, C_1';C_2') \in R$ and $\nu, \nu', \mu \in \mathit{Mem}$ be arbitrary with $\nu =_L \nu'$ and $\langle\!| C_1;C_2, \nu |\!\rangle \twoheadrightarrow \langle\!| C^*, \mu |\!\rangle$ for some $C^* \in \vec{\mathit{Com}}$. We make a case distinction on C^* according to the operational semantics:

   (a) $C^* = C_2$: from the operational semantics, we obtain $\langle\!| C_1, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, \mu |\!\rangle$. Since $C_1 \cong_L C_1'$ and $\nu =_L \nu'$, there is a $\mu' \in \mathit{Mem}$ with $\langle\!| C_1', \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, \mu' |\!\rangle$ and $\mu =_L \mu'$ according to Definition 2.7. From the operational semantics, we obtain $\langle\!| C_1';C_2', \nu' |\!\rangle \twoheadrightarrow \langle\!| C_2', \mu' |\!\rangle$ with $C_2 \cong_L C_2'$ (by definition of R) and $\mu =_L \mu'$.

   (b) $C^* = \langle C;C_2\rangle V$ for some $C \in \mathit{Com}$ and $V \in \vec{\mathit{Com}}$ (possibly $V = \langle\rangle$). From the operational semantics, we obtain $\langle\!| C_1, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle C\rangle V, \mu |\!\rangle$. Since $C_1 \cong_L C_1'$ and $\nu =_L \nu'$, there are $C' \in \mathit{Com}$, $V' \in \vec{\mathit{Com}}$, and $\mu' \in \mathit{Mem}$ with $\langle\!| C_1', \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle C'\rangle V', \mu' |\!\rangle$, $C \cong_L C'$, $V \cong_L V'$, and $\mu =_L \mu'$. From the operational semantics, we obtain $\langle\!| C_1';C_2', \nu' |\!\rangle \twoheadrightarrow \langle\!| \langle C';C_2'\rangle V', \mu' |\!\rangle$ with $(C;C_2, C';C_2') \in R$ (follows from $C \cong_L C'$, $C_2 \cong_L C_2'$, and the definition of R), $V \cong_L V'$, and $\mu =_L \mu'$.

   Hence, R is a strong low bisimulation up to $\cong_L$.

3. Define R as the symmetric relation

$$\{(\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \mid B, B' : L, B \equiv B', C_1 \cong_L C_1', C_2 \cong_L C_2'\}$$

   Let $(\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2') \in R$ and $\nu, \nu' \in \mathit{Mem}$ be arbitrary with $\nu =_L \nu'$. We make a case distinction on the value of B in ν:

   (a) $\langle\!| B, \nu |\!\rangle \downarrow \mathsf{False}$: From $\nu =_L \nu'$, $B, B' : \mathit{low}$, and $B \equiv B'$, we obtain $\langle\!| B', \nu' |\!\rangle \downarrow \mathsf{False}$. From the operational semantics, we obtain $\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \nu |\!\rangle \twoheadrightarrow \langle\!| C_2, \nu |\!\rangle$ and $\langle\!| \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2', \nu' |\!\rangle \twoheadrightarrow \langle\!| C_2', \nu' |\!\rangle$ with $\nu =_L \nu'$. By definition of R, we have $C_2 \cong_L C_2'$.

   (b) $\langle\!| B, \nu |\!\rangle \downarrow \mathsf{True}$: From $\nu =_L \nu'$, $B, B' : \mathit{low}$, and $B \equiv B'$, we obtain $\langle\!| B', \nu' |\!\rangle \downarrow \mathsf{True}$. From the operational semantics, we obtain $\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \nu |\!\rangle \twoheadrightarrow \langle\!| C_1, \nu |\!\rangle$ and $\langle\!| \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2', \nu' |\!\rangle \twoheadrightarrow \langle\!| C_1', \nu' |\!\rangle$ with $\nu =_L \nu'$. By definition of R, we have $C_1 \cong_L C_1'$.

   Hence, R is a strong low bisimulation up to $\cong_L$. □

Theorem 5.1. If $V \doteq_L V'$ is derivable for $V, V' \in \vec{Com}$, then $V \approxeq_L V'$ holds.

Proof. The proof proceeds by induction on the number of rule applications in the derivation $\mathcal{D}$ of $V \doteq_L V'$.

Base case: $\mathcal{D}$ consists of only a single rule application. We make a case distinction on this rule.

[*Skp*] The judgment derived is $\mathsf{skip} \doteq_L \mathsf{skip}$. Lemma A.2(1) implies $\mathsf{skip} \approxeq_L Id := Id$ where Id is an arbitrary identifier with $Id : H$. From symmetry and transitivity of $\approxeq_L$, we obtain $\mathsf{skip} \approxeq_L \mathsf{skip}$.

[*SHA1*] The judgment derived is $\mathsf{skip} \doteq_L Id := Exp$ with $Id : H$. From Lemma A.2(1) follows $\mathsf{skip} \approxeq_L Id := Exp$.

[*SHA2*] The judgment derived is $Id := Exp \doteq_L \mathsf{skip}$ with $Id : H$. From Lemma A.2(1) and the symmetry of $\approxeq_L$, we obtain $Id := Exp \approxeq_L \mathsf{skip}$.

[*HA*] The judgment derived is $Id := Exp \doteq_L Id' := Exp'$ with $Id, Id' : H$. From Lemma A.2(1), we obtain $\mathsf{skip} \approxeq_L Id := Exp$ and $\mathsf{skip} \approxeq_L Id' := Exp'$. Symmetry and transitivity of $\approxeq_L$ imply $Id := Exp \approxeq_L Id' := Exp'$.

[*LA*] The judgment derived is $Id := Exp \doteq_L Id := Exp'$ with $Id : L$, $Exp, Exp' : L$, and $Exp \equiv Exp'$. Lemma A.2(2) implies $Id := Exp \approxeq_L Id := Exp'$.

Induction assumption: If $\mathcal{D}'$ is a derivation of $W \doteq_L W'$ with fewer rule applications than in $\mathcal{D}$ then $W \approxeq_L W'$ holds.

Step case: We make a case distinction on the rule applied at the root of $\mathcal{D}$.

[*Seq*] The judgment derived is $C_1; C_2 \doteq_L C_1'; C_2'$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_2'$, respectively. From the induction assumption, we obtain $C_1 \approxeq_L C_1'$ and $C_2 \approxeq_L C_2'$. Lemma A.2(3) implies $C_1; C_2 \approxeq_L C_1'; C_2'$.

[*Par*] The judgment derived is $\langle C_1, \ldots, C_n \rangle \doteq_L \langle C_1', \ldots, C_n' \rangle$ and there are derivations $\mathcal{D}_i$ of $C_i \doteq_L C_i'$ for $i = 1, \ldots, n$. From the induction assumption, we obtain $C_i \approxeq_L C_i'$ for $i = 1, \ldots, n$. From Definition 2.7, we obtain $\langle C_1, \ldots, C_n \rangle \approxeq_L \langle C_1', \ldots, C_n' \rangle$.

[*Frk*] The judgment derived is $\mathsf{fork}(C_1 V_1) \doteq_L \mathsf{fork}(C_1' V_1')$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $V_1 \doteq_L V_1'$, respectively. From the induction assumption, we obtain $C_1 \approxeq_L C_1'$ and $V_1 \approxeq_L V_1'$. Lemma A.2(4) implies $\mathsf{fork}(C_1 V_1) \approxeq_L \mathsf{fork}(C_1' V_1')$.

[*Whl*] The judgment derived is $\mathsf{while}\ B\ \mathsf{do}\ C_1 \doteq_L \mathsf{while}\ B'\ \mathsf{do}\ C_1'$ with $B, B' : L$, $B \equiv B'$, and there is a derivation $\mathcal{D}_1$ of $C_1 \doteq_L C_1'$. Lemma A.2(5) implies then that $\mathsf{while}\ B\ \mathsf{do}\ C_1 \approxeq_L \mathsf{while}\ B'\ \mathsf{do}\ C_1'$.

[*LCond*] The judgment derived is $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ with $B, B' : L$, $B \equiv B'$, and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_2'$, respectively. Lemma A.2(6) implies $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approxeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$.

[*SHCond1*] The judgment derived is $\mathsf{skip}; C_1 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ with $B' : H$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_1 \doteq_L C_2'$, respectively. From the induction assumption, we obtain $C_1 \approxeq_L C_1'$ and $C_1 \approxeq_L C_2'$. Lemma A.2(7) implies $\mathsf{skip}; C_1 \approxeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$.

[*SHCond2*] The judgment derived is $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L \mathsf{skip}; C_1'$ with $B : H$ and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_1'$, respectively. From the induction assumption, we obtain $C_1 \approxeq_L C_1'$ and $C_2 \approxeq_L C_1'$. Symmetry of $\approxeq_L$ and Lemma A.2(7) imply $\mathsf{skip}; C_1' \approxeq_L \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$. From the symmetry of $\approxeq_L$, we obtain $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approxeq_L \mathsf{skip}; C_1'$.

[*HAHCond1*] The judgment derived is $Id := Exp; C_1 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ with $Id : H$, $B' : H$, and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_1 \doteq_L C_2'$, respectively. From $\mathsf{skip}; C_1 \approxeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ (see Case [*SHCond1*]), $Id := Exp; C_1 \approxeq_L \mathsf{skip}; C_1$ (see Case [*SHA2*]), and transitivity of $\approxeq_L$, we obtain $Id := Exp; C_1 \approxeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$.

[*HAHCond2*] The judgment derived is $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L Id' := Exp'; C_1'$ with $Id' : H$, $B : H$, and there are derivations $\mathcal{D}_1$ and $\mathcal{D}_2$ of $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_1'$, respectively. From $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approxeq_L \mathsf{skip}; C_1'$ (see Case [*SHCond2*]), $\mathsf{skip}; C_1' \approxeq_L Id' := Exp'; C_1'$ (see Case [*SHA1*]), and transitivity of $\approxeq_L$, we obtain $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approxeq_L Id' := Exp'; C_1'$.

[*HCond*] The judgment derived is $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \doteq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ with $B, B' : H$ and there are derivations $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{D}_3$ of $C_1 \doteq_L C_1'$, $C_1 \doteq_L C_2'$, and $C_1 \doteq_L C_2$, respectively. From the induction assumption, we obtain $C_1 \approxeq_L C_1'$, $C_1 \approxeq_L C_2'$, and $C_1 \approxeq_L C_2$. Symmetry and transitivity of $\approxeq_L$ implies $C_1 \approxeq_L C_1$. From $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approxeq_L \mathsf{skip}; C_1$ (see Case [*SHCond2*]), $\mathsf{skip}; C_1 \approxeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ (see Case [*SHCond1*]), and transitivity of $\approxeq_L$, we then obtain $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \approxeq_L \mathsf{if}\ B'\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$. □

A.2 Proof of Theorem 5.2

For the proof of Theorem 5.2, we first strengthen our notion of bisimulation. We then prove a lemma that shows that this relation is a congruence by using the up-to technique. With the help of this lemma, the proof Theorem 5.2 is a straightforward induction over a program's term structure.

Definition A.2. The *pointwise weak possibilistic bisimulation* $\dot{\simeq}$ is the union of all symmetric relations R on command vectors $V, V' \in \vec{Com}$ of equal size, i.e. $V = \langle C_1, \ldots, C_n \rangle$ and $V' = \langle C_1', \ldots, C_n' \rangle$, such that whenever $V\ R\ V'$ then for all states ν, μ and all $i \in \{1 \ldots n\}$

and all thread pools W, there is a thread pool W' with

$$\begin{array}{l} \langle\!| C_i, \nu |\!\rangle \twoheadrightarrow \langle\!| W, \mu |\!\rangle \Rightarrow (\langle\!| C_i', \nu |\!\rangle \twoheadrightarrow^* \langle\!| W', \mu |\!\rangle \wedge W \; R \; W') \\ \text{and } V = \langle\rangle \implies \langle\!| V', \nu |\!\rangle \rightarrow^* \langle\!| \langle\rangle, \nu |\!\rangle. \end{array} \tag{A.2}$$

Observe that Property A.2 also holds for the entire relation $\doteq$. Furthermore, for two thread pools $V = \langle C_1, \ldots, C_n\rangle$ and $V' = \langle C_1', \ldots, C_n'\rangle$ we have $V \doteq V'$ if and only if for all $i \in \{1, \ldots, n\}$ we have $C_i \doteq C_i'$.

Lemma A.3. $V \doteq V' \Rightarrow V \simeq V'$

Proof. Follows directly from Definitions 5.2 and A.2 and the operational semantics for thread pools. □

Definition A.3. A binary relation R on *Com* is a *pointwise weak possibilistic bisimulation up to* $\doteq$ if R is symmetric and

$$\begin{array}{l} \forall C, C', C_1, \ldots, C_n \in Com : \forall \nu, \mu \in Mem : \\ \quad (C \; R \; C' \wedge \langle\!| C, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle C_1, \ldots, C_n\rangle, \mu |\!\rangle) \\ \quad \Rightarrow \exists C_1', \ldots, C_n' \in Com : (\langle\!| C', \nu |\!\rangle \twoheadrightarrow^* \langle\!| \langle C_1', \ldots, C_n'\rangle, \mu |\!\rangle \\ \qquad \wedge \forall i \in \{1, \ldots, n\} : C_i (R \cup \doteq) C_i'). \end{array}$$

Lemma A.4. *If R is a pointwise weak possibilistic bisimulation up to $\doteq$, then we have $R \subseteq \doteq$.*

Proof. Let $Q = (R \cup \doteq)^{\uparrow}$, where $^{\uparrow}$ denotes the pointwise lifting of a relation on commands to command vectors (here, $\doteq$ is viewed as a relation on commands). It is sufficient to show $Q \subseteq \doteq$. To this end, we show that Q satisfies condition A.2 in Definition A.2, and is therefore contained in $\doteq$, the union of all such relations. Let $V = \langle C_1, \ldots, C_n\rangle$ and $V' = \langle C_1', \ldots, C_n'\rangle$ and $(V, V') \in Q$. If $n = 0$, then we have $V = V' = \langle\rangle$ and the second part of condition A.2 is fulfilled. Suppose $n > 0$ and $\langle\!| V, \nu |\!\rangle \twoheadrightarrow \langle\!| W, \mu |\!\rangle$. By definition of the operational semantics, we know that there is $i \in \{1, \ldots, n\}$ with $\langle\!| C_i, \nu |\!\rangle \twoheadrightarrow \langle\!| \langle C_{i,1}, \ldots, C_{i,m}\rangle, \mu |\!\rangle$ and $W = \langle C_1, \ldots, C_{i-1}, C_{i,1}, \ldots, C_{i,m}, C_{i+1}, \ldots, C_n\rangle$. We distinguish between the cases $(C_i, C_i') \in \doteq$ and $(C_i, C_i') \in R$.
$(C_i, C_i') \in \doteq$: By Definition A.2, there are $C_{i,1}', \ldots, C_{i,m}'$ with $\langle\!| C_i', \nu |\!\rangle \twoheadrightarrow^* \langle\!| \langle C_{i,1}', \ldots, C_{i,m}'\rangle, \mu |\!\rangle$ and $\langle C_{i,1}, \ldots, C_{i,m}\rangle \doteq \langle C_{i,1}', \ldots, C_{i,m}'\rangle$. From Definition A.2 it follows that $C_{i,j} \doteq C_{i,j}'$ holds, for all $j \in \{1, \ldots, m\}$.

$(C_i, C'_i) \in R$: By Definition A.3, there are $C'_{i,1}, \ldots, C'_{i,m}$ with $\langle\!|C'_i, \nu|\!\rangle \twoheadrightarrow^* \langle\!|\langle C'_{i,1}, \ldots, C'_{i,m}\rangle, \mu|\!\rangle$ and $C_{i,j}(R \cup \dot{\simeq})C'_{i,j}$ holds, for all $j \in \{1, \ldots, m\}$.

With $W' = \langle C'_1, \ldots, C'_{i-1}, C'_{i,1}, \ldots, C'_{i,m}, C'_{i+1}, \ldots, C'_n\rangle$ we have $\langle\!|V', \nu|\!\rangle \twoheadrightarrow^* \langle\!|W', \mu|\!\rangle$ and $(W, W') \in Q$. As $\dot{\simeq}$ is defined to be the union of all symmetric relations with the condition A.2, we see $Q \subseteq \dot{\simeq}$. □

Lemma A.5.

1. *If $C_1 \dot{\simeq} C'_1$ and $C_2 \dot{\simeq} C'_2$ then $C_1; C_2 \dot{\simeq} C'_1; C'_2$.*
2. *If $C_1 \dot{\simeq} C'_1$ and $V_2 \dot{\simeq} V'_2$ then $\mathsf{fork}(C_1 V_2) \dot{\simeq} \mathsf{fork}(C'_1 V'_2)$.*
3. *If $C_1 \dot{\simeq} C'_1$ then $\mathsf{while}\ B\ \mathsf{do}\ C_1 \dot{\simeq} \mathsf{while}\ B\ \mathsf{do}\ C'_1$.*
4. *If $C_1 \dot{\simeq} C'_1$ and $C_2 \dot{\simeq} C'_2$ then $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \dot{\simeq} \mathsf{if}\ B\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2$.*
5. *If $C_1 \dot{\simeq} C'_1, \ldots, C_n \dot{\simeq} C'_n$ then $\langle C_1, \ldots, C_n\rangle \dot{\simeq} \langle C'_1, \ldots, C'_n\rangle$.*

Proof. We proceed as in the proof of Lemma A.2, only by using pointwise weak possibilistic bisimulations up to $\dot{\simeq}$ instead of strong low-bisimulations. □

Theorem 5.2.
1. For all preserving substitutions σ, ρ that are ground for $V \in \vec{Com}_{\mathcal{V}}$, we have $\sigma(V) \simeq \rho(V)$.
2. For each lifting V' of a ground program $V \in \vec{Com}$ and each preserving substitution σ with $\sigma(V')$ ground, we have $\sigma(V') \simeq V$.

Proof. We prove the first assertion for pointwise weak possibilistic bisimulations (rather than weak possibilistic bisimulations) by induction on the term structure of vectors of length 1, i.e. $V \in Com_{\mathcal{V}}$. By Lemma A.5(5), the assertion is then lifted to arbitrary vectors in $\vec{Com}_{\mathcal{V}}$, and with Lemma A.3 the assertion follows. Let σ, ρ be substitutions that are preserving and ground for V.

1. Suppose V is skip or an assignment. Then $\sigma(V) = \rho(V) = V$, and the assertion follows by reflexivity of $\dot{\simeq}$.
2. Suppose V is of the form $\alpha; C'$ or $C'; \alpha$. By induction hypothesis, we have $\sigma(C') \dot{\simeq} \rho(C')$. σ and ρ are preserving and ground for α, so we see $\sigma(\alpha) \dot{\simeq} \rho(\alpha)$. From Lemma A.5(1), $\sigma V \dot{\simeq} \rho V$ follows.

3. Suppose $V = C_1;C_2$ with $C_1, C_2 \in \mathit{Com}_{\mathcal{V}}$. By induction hypothesis, we have $\sigma C_1 \simeq \rho C_1$ and $\sigma C_2 \simeq \rho C_2$. From Lemma A.5(1), $\sigma V \simeq \rho V$ follows

4. Suppose $V = \mathsf{while}\ B\ \mathsf{do}\ C'$ with $C' \in \mathit{Com}_{\mathcal{V}}$. By induction hypothesis, we have $\sigma C' \simeq \rho C'$. From Lemma A.5(3), $\sigma V \simeq \rho V$ follows.

5. Suppose $V = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ with $C_1, C_2 \in \mathit{Com}_{\mathcal{V}}$. By induction hypothesis, we have $\sigma C_1 \simeq \rho C_1$ and $\sigma C_2 \simeq \rho C_2$. From Lemma A.5(4), $\sigma V \simeq \rho V$ follows.

6. Suppose $V = \mathsf{fork}(C_0\langle C_1, \ldots, C_n\rangle)$ with $C_i \in \mathit{Com}_{\mathcal{V}}$ for $i \in \{0, \ldots, n\}$. By induction hypothesis, we have $\sigma C_i \simeq \rho C_i$ for $i \in \{0, \ldots, n\}$. From Lemma A.5(5), we first obtain $\sigma\langle C_1, \ldots, C_n\rangle \simeq \rho\langle C_1, \ldots, C_n\rangle$ and then, by Lemma A.5(2), $\sigma V \simeq \rho V$.

The second assertion follows from part 1 of the theorem by instantiating ρ with a projection π. □

A.3 Proof of Lemma 5.1

For the proof of Lemma 5.1, we first state and prove two lemmas that simplify reasoning with $\doteq_L$ on $\mathit{Com}_{\mathcal{V}}$.

Lemma A.6. *If $V_1 \doteq_L V_2$ holds for two programs $V_1, V_2 \in \vec{\mathit{Com}}_{\mathcal{V}}$ then $\sigma V_1 \doteq_L \sigma V_2$ holds for each substitution σ that is preserving (but not necessarily ground).*

Proof. Given an arbitrary substitution η that is preserving and ground for σV_1 and σV_2, we obtain $\eta(\sigma V_1) \doteq_L \eta(\sigma V_2)$ from $V_1 \doteq_L V_2$, Definition 5.4, and the fact that $\eta \circ \sigma$ is preserving and ground for V_1 and V_2. Since η was chosen arbitrarily, $\sigma V_1 \doteq_L \sigma V_2$ follows. □

Lemma A.7. *Let $V, V', V_0, V_0', \ldots, V_n, V_n' \in \vec{\mathit{Com}}_{\mathcal{V}}$ be programs that may contain meta-variables. If $V_i \doteq_L V_i'$ holds for each $i \in \{0, \ldots, n\}$ according to Definition 5.4 and $V \doteq_L V'$ can be syntactically derived from the assumptions $V_0 \doteq_L V_0'$, …, $V_n \doteq_L V_n'$ with the rules in Figure 5.1 then $V \doteq_L V'$ holds according to Definition 5.4.*

Proof. The proof proceeds by induction on the size of $\mathcal{D}$, the derivation of $V \doteq_L V'$ from $V_0 \doteq_L V_0', \ldots, V_n \doteq_L V_n'$.

Base case: If $\mathcal{D}$ consists of zero rule applications, then $V \doteq_L V'$ equals one of the assumptions.

Induction assumption: The proposition holds for every derivation with less than n rule applications.

Step case: Assume $\mathcal{D}$ consists of n rule applications. We make a case distinction on the last rule applied in $\mathcal{D}$. Here, we consider only the case where [*Seq*] is the last rule applied. The reasoning is independent of the structure of the rule [*Seq*], and so the cases for the other rules can be shown along the same lines.

Let $C_1, C_1', C_2, C_2' \in Com_{\mathcal{V}}$ be arbitrary with $C_1 \doteq_L C_1'$ and $C_2 \doteq_L C_2'$. Let σ be an arbitrary substitution that is preserving and ground for C_1, C_1', C_2, C_2'. From $C_1 \doteq_L C_1'$, $C_2 \doteq_L C_2'$, and Definition 5.4 we obtain $\sigma C_1 \doteq_L \sigma C_1'$ and $\sigma C_2 \doteq_L \sigma C_2'$. An application of [*Seq*] (for ground programs) yields $(\sigma C_1; \sigma C_2) \doteq_L (\sigma C_1'; \sigma C_2')$. Since $\sigma(C_1; C_2) = (\sigma C_1; \sigma C_2) \doteq_L (\sigma C_1'; \sigma C_2') = \sigma(C_1'; C_2')$, and σ was chosen freely, we obtain $C_1; C_2 \doteq_L C_1'; C_2'$ from Definition 5.4. □

Lemma 5.1. If $V \hookrightarrow V' : S$ can be derived then $V' \doteq_L S$ holds.

Proof. We prove the proposition by induction on the minimal height of a given derivation $\mathcal{D}$ of $V \hookrightarrow V' : S$.

Base case: $\mathcal{D}$ consists of a single rule application. We perform a case distinction on this rule:

[*TVar*] We have $V = V' = S = \alpha$ for some meta-variable $\alpha \in \mathcal{V}$. Let σ be an arbitrary substitution that is preserving and ground for α. As $\sigma\alpha$ is a command in $Stut_{\mathcal{V}}$ that is free of meta-variables (i.e. a sequential composition of skip statements), we obtain $\sigma\alpha \doteq_L \sigma\alpha$ from [*Skp*] and [*Seq*] in Figure 5.1. Hence, $\alpha \doteq_L \alpha$ holds.

[*TSkp*] We have $V = V' = S = \mathsf{skip}$. From [*Skp*] in Figure 5.1, we obtain $\mathsf{skip} \doteq_L \mathsf{skip}$.

[*THA*] We have $V = V' = Id := Exp$ and $S = \mathsf{skip}$ with $Id : H$. From [SHA_2] in Figure 5.1, we obtain $Id := Exp \doteq_L \mathsf{skip}$.

[*TLA*] We have $V = V' = S = Id := Exp$ with $Id : L$ and $Exp : L$. From [*LA*] in Figure 5.1, we obtain $Id := Exp \doteq_L Id := Exp$.

Induction assumption: For any derivation $\mathcal{D}'$ of a judgment $W \hookrightarrow W' : S'$ with height less than the height of $\mathcal{D}$, $W' \doteq_L S'$ holds.

Step case: We make a case distinction on the rule applied at the root of $\mathcal{D}$.

[*TSeq*] We have $V = C_1; C_2$, $V' = C_1'; C_2'$, and $S = S_1; S_2$ with $C_1 \hookrightarrow C_1' : S_1$ and $C_2 \hookrightarrow C_2' : S_2$. By induction assumption, $C_1' \doteq_L S_1$ and $C_2' \doteq_L S_2$ hold. An application of [*Seq*] in Figure 5.1 (observe Lemma A.7) yields $C_1'; C_2' \doteq_L S_1; S_2$.

[*TPar*] We have $V = \langle C_1, \ldots, C_n \rangle$, $V' = \langle C_1', \ldots, C_n' \rangle$, and $S = \langle S_1, \ldots, S_n \rangle$ with $C_i \hookrightarrow C_i' : S_i$ for all $i \in \{1, \ldots, n\}$. By induction assumption, $C_i' \doteq_L S_i$ holds for all $i \in \{1, \ldots, n\}$. Application of [*Par*] in Figure 5.1 yields $\langle C_1', \ldots, C_n' \rangle \doteq_L \langle S_1, \ldots, S_n \rangle$.

[*TFrk*] We have $V = \mathsf{fork}(C_1 V_2)$, $V' = \mathsf{fork}(C_1' V_2')$, and $S = \mathsf{fork}(S_1 S_2)$ with $C_1 \hookrightarrow C_1' : S_1$ and $V_2 \hookrightarrow V_2' : S_2$. By induction assumption, $C_1' \simeq_L S_1$ and $V_2' \simeq_L S_2$ hold. An application of [*Frk*] in Figure 5.1 yields $\mathsf{fork}(C_1' V_2') \simeq_L \mathsf{fork}(S_1 S_2)$.

[*TWhl*] We have $V = \mathsf{while}\ B\ \mathsf{do}\ C_1$, $V' = \mathsf{while}\ B\ \mathsf{do}\ C_1'$, and $S = \mathsf{while}\ B\ \mathsf{do}\ S_1$ with $B : L$ and $C_1 \hookrightarrow C_1' : S_1$. By induction assumption, $C_1' \simeq_L S_1$ holds. An application of [*Whl*] in Figure 5.1 yields $\mathsf{while}\ B\ \mathsf{do}\ C_1' \simeq_L \mathsf{while}\ B\ \mathsf{do}\ S_1$.

[*TLCond*] We have $V = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, together with $V' = \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'$ and $S = \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$ with $B : L$, $C_1 \hookrightarrow C_1' : S_1$, and $C_2 \hookrightarrow C_2' : S_2$. By induction assumption, $C_1' \simeq_L S_1$ and $C_2' \simeq_L S_2$ hold. An application of [*LCond*] in Figure 5.1 yields $\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' \simeq_L \mathsf{if}\ B'\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$.

[*THCond*] We have $V = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, together with $V' = \mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2'$ and $S = \mathsf{skip}; \sigma S_1$ with $B : H$, $C_1 \hookrightarrow C_1' : S_1$, $C_2 \hookrightarrow C_2' : S_2$, and $\sigma \in \mathcal{U}(\{S_1 \simeq_L^? S_2\})$. By induction assumption, $C_1' \simeq_L S_1$ and $C_2' \simeq_L S_2$ hold. As σ is preserving, we obtain $\sigma C_1' \simeq_L \sigma S_1$ and $\sigma C_2' \simeq_L \sigma S_2$ from Lemma A.6. Then we conclude $\sigma C_2' \simeq_L \sigma S_1$ from $\sigma C_2' \simeq_L \sigma S_2$, $\sigma S_1 \simeq_L \sigma S_2$ (follows from $\sigma \in \mathcal{U}(\{S_1 \simeq_L^? S_2\})$), and from the fact that $\simeq_L$ is symmetric and transitive. An application of [*SHCond*$_2$] in Figure 5.1 yields $\mathsf{if}\ B\ \mathsf{then}\ \sigma C_1'\ \mathsf{else}\ \sigma C_2' \simeq_L \mathsf{skip}; \sigma S_1$. □

A.4 Proof of Lemma 5.2

We formally define $const(C) = |C|_{\mathsf{skip}} + \sum_{Exp, Id:H} |C|_{Id\ :=\ Exp}$, where $|C|_D$ denotes the number of occurrences of D as a subterm of C.

Lemma 5.2. For two commands C_1 and C_2 in $Pad_{\mathcal{V}}$, we have $C_1 \simeq_L C_2$ if and only if $const(C_1) = const(C_2)$ and $\forall \alpha \in \mathcal{V}$: $|C_1|_\alpha = |C_2|_\alpha$.

Proof. ($\Rightarrow$) Suppose $const(C_1) \neq const(C_2)$. Let σ be the substitution mapping all variables in C_1 and C_2 to ϵ. The judgment $\sigma C_1 \simeq_L \sigma C_2$ is not derivable with the rules in Figure 5.1, contradicting the assumption $C_1 \simeq_L C_2$, as σ is preserving. Assume now $const(C_1) = const(C_2)$ and $|C_1|_\alpha \neq |C_2|_\alpha$ for some meta-variable $\alpha \in \mathcal{V}$. Let σ be the substitution mapping α to skip and all other variables in C_1 and C_2 to ϵ. We have $const(\sigma C_1) \neq const(\sigma C_2)$ and thus a contradiction to the assumption $C_1 \simeq_L C_2$.

($\Leftarrow$) Let σ be an arbitrary preserving substitution that is ground for C_1 and C_2. From the assumption $const(C_1) = const(C_2)$ and $|C_1|_\alpha = |C_2|_\alpha$ follows that $const(\sigma(C_1)) = const(\sigma(C_2))$ holds. It is not difficult to see that $\sigma(C_1) \simeq_L \sigma(C_2)$ is derivable with the rules from Figure 5.1 and hence, by Definition 5.4, $C_1 \simeq_L C_2$ holds. □

A.5 Proof of Lemma 5.3

Lemma 5.3. Let $V_i \in \vec{Com}$, with liftings $V_i' \in \vec{Com}_{\mathcal{V}}$ and $V_i^* \in \vec{Mgl}_{\mathcal{V}}$ for $i = 1, 2$. Suppose V_1^* (V_2^*) shares no meta-variables with V_1', V_2', and V_2^* (V_1', V_2', and V_1^*). Then we have

$$\mathcal{U}(\{V_1' \overset{?}{\simeq}_L V_2'\}) \neq \emptyset \text{ implies } \mathcal{U}(\{V_1^* \overset{?}{\simeq}_L V_2^*\}) \neq \emptyset\,.$$

More precisely, we can find a $\rho \in \mathcal{U}(\{V_1^* \overset{?}{\simeq}_L V_2^*\})$ with $dom(\rho) \subseteq var(V_1^*) \cup var(V_2^*)$ and $var(ran(\rho)) \subseteq var(V_1') \cup var(V_2')$.

Proof. Suppose σ is a preserving substitution with $\sigma V_1' \simeq_L \sigma V_2'$. We will inductively construct preserving substitutions ρ_1 with $\rho_1 V_1^* \simeq_L \sigma V_1'$, and ρ_2 with $\rho_2 V_2^* \simeq_L \sigma V_2'$ with the property $dom(\rho_i) \subseteq var(V_i^*)$ and $var(ran(\rho_i)) \subseteq var(V_i')$ for $i = 1, 2$. The meta-variables in V_1^* and V_2^* are disjoint, so $\rho = \rho_1 \cup \rho_2$ is well–defined and a unifier of $V_1^* \overset{?}{\simeq}_L V_2^*$ because of $\rho V_1^* \simeq_L \sigma V_1' \simeq_L \sigma V_2' \simeq_L \rho V_2^*$. We prove the assertion by induction on the term structure of $V_1^* \in \vec{Mgl}_{\mathcal{V}}$, starting with $V_1^* = C_1^* \in Mgl_{\mathcal{V}}$ (and hence $V_1 = C_1 \in Com$ and $V_1' = C_1' \in Com_{\mathcal{V}}$).

Suppose $C_1^* \in Pad_{\mathcal{V}} \cap Mgl_{\mathcal{V}}$. Then by definition of $Mgl_{\mathcal{V}}$, C_1^* contains at least one meta-variable α. C_1' is also a lifting of C_1, so it must be in $Pad_{\mathcal{V}}$. Let $\alpha_1, \ldots, \alpha_n$ be the meta-variables in C_1'. Define $\rho(\alpha) := \sigma(\alpha_1); \ldots; \sigma(\alpha_n)$, and set $\rho(Y) = \epsilon$ for all $Y \neq \alpha$ occuring in C_1^*. C_1^* and C_1' are both liftings of C_1, so they contain the same number of skips and assignments to high variables. By definition of ρ, we see that $\sigma C_1'$ and ρC_1^* contain the same meta-variables and the same number of constants. Applying Lemma 5.2, we can conclude that $\rho C_1^* \simeq_L \sigma C_1'$. Furthermore, $dom(\rho) \subseteq var(C_1^*)$ and $var(ran(\rho)) \subseteq var(C_1')$ are satisfied.

Suppose $C_1^* = P; \text{if } B \text{ then } C_{1,1}^* \text{ else } C_{1,2}^*; C^*$. The command C_1' is also a lifting of C_1, so it can be written as $P'; \text{if } B \text{ then } C_{1,1}' \text{ else } C_{1,2}'; C'$, with (possibly empty) commands P', C'.

If B is a *low* conditional, we inductively construct substitutions $\rho_1, \rho_2, \rho_3, \rho_4$ such that $\rho_1 P \simeq_L \sigma P'$, $\rho_2 C_{1,1}^* \simeq_L \sigma C_{1,1}'$, $\rho_3 C_{1,2}^* \simeq_L \sigma C_{1,2}'$ and $\rho_4 C^* \simeq_L \sigma C'$. The domains of the ρ_i are disjoint by the hypothesis that $dom(\rho_i)$ is a subset of the meta-variables of the corresponding subcommand and the assumption that every meta-variable occurs only once in C_1^*, so $\rho = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4$ is well-defined. Using Lemma A.7, we can conclude $\rho C_1^* = \rho_1 P; \text{if } B \text{ then } \rho_2 C_{1,1}^* \text{ else } \rho_3 C_{1,2}^*; \rho_4 C^* \simeq_L \sigma P'; \text{if } B \text{ then } \sigma C_{1,1}' \text{ else } \sigma C_{1,2}'; \sigma C' \simeq_L \sigma C_1'$. Furthermore, $dom(\rho) \subseteq var(C_1^*)$ and $var(ran(\rho)) \subseteq var(C_1')$ are satisfied.

If B is a *high* conditional, the precondition $\sigma C_1' \simeq_L \sigma C_2'$ together with the definition of $\simeq_L$ on high conditionals shows that $\sigma C_{1,1}' \simeq_L \sigma C_{1,2}'$ holds. After applying the induction hypothesis, we obtain $\rho_{2,1}$ and $\rho_{2,2}$ with $\rho_{2,1} C_{1,1}^* \simeq_L \sigma C_{1,1}' \simeq_L \sigma C_{1,2}' \simeq_L \rho_{2,2} C_{1,2}^*$

and ρ_1, ρ_3 with $\rho_1 P \simeq_L \sigma P'$ and $\rho_3 C^* \simeq_L \sigma C'$. With $\rho = \rho_1 \cup \rho_{2,1} \cup \rho_{2,2} \cup \rho_3$, we see that ρC_1^* is equal to $\rho_1 P; \mathsf{if}\ B\ \mathsf{then}\ \rho_{2,1} C_{1,1}^*\ \mathsf{else}\ \rho_{2,2} C_{1,2}^*; \rho_3 C^*$ and, therefore, $\rho C_1^* \simeq_L \rho_1 P; \mathsf{skip}; \rho_{2,1} C_{1,1}^*; \rho_3 C^* \simeq_L \sigma P'; \mathsf{skip}; \sigma C_{1,1}'; \sigma C' \simeq_L \sigma P'; \mathsf{if}\ B\ \mathsf{then}\ \sigma C_{1,1}'\ \mathsf{else}\ \sigma C_{1,2}'; \sigma C' \simeq_L \sigma C_1'$. Furthermore, $dom(\rho) \subseteq var(C_1^*)$ and $var(ran(\rho)) \subseteq var(C_1')$ are satisfied.

The remaining induction steps for $Mgl_{\mathcal{V}}$ and the lifting to $\vec{Mgl}_{\mathcal{V}}$ can be treated in the same way as the low conditional case. □

A.6 Proof of Lemma 5.4

Lemma 5.4. Let $V \in \vec{Com}$ and $\overline{V} \in \vec{Com}_{\mathcal{V}}$. If $V \rightharpoonup \overline{V}$ can be derived, then

1. $\overline{V}$ is a lifting of V and
2. $\overline{V} \in \vec{Mgl}_{\mathcal{V}}$.

Proof. The proof of Assertion 1 is a straightforward inductive argument over the structure of the derivation of $V \rightharpoonup \overline{V}$ and an inspection of each rule in Figure 5.3.

For the proof of Assertion 2, we proceed by induction on the term structure of $V \in \vec{Com}$: First, suppose $V = C \in Com$.

If $C = \mathsf{skip}$, we have $C \rightharpoonup \mathsf{skip}; X$, which is in $Mgl_{\mathcal{V}}$. The same holds for $C = Id := Exp$ with $Id : H$.

If $C = Id := Exp$ with $Id : L$, we have $C \rightharpoonup X; Id := Exp; Y$, which is in $Mgl_{\mathcal{V}}$.

If $C = \mathsf{while}\ B\ \mathsf{do}\ C'$, we get $C \rightharpoonup \overline{C} = X; \mathsf{while}\ B\ \mathsf{do}\ \overline{C'}; Y$ with $C' \rightharpoonup \overline{C'}$. By induction hypothesis, $\overline{C'} \in Mgl_{\mathcal{V}}$, and by definition of $Mgl_{\mathcal{V}}$ we have $\overline{C} \in Mgl_{\mathcal{V}}$.

If $C = \mathsf{fork}(C'V)$, we have $C \rightharpoonup \overline{C} = X; \mathsf{fork}(\overline{C'V}); Y$ with $C' \rightharpoonup \overline{C'}$ and $V \rightharpoonup \overline{V}$. By induction hypothesis, $\overline{C'} \in Mgl_{\mathcal{V}}$, and $\overline{V} \in \vec{Mgl}_{\mathcal{V}}$ and by definition of $Mgl_{\mathcal{V}}$ we have $\overline{C} \in Mgl_{\mathcal{V}}$.

If $C = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, we have $C \rightharpoonup \overline{C}$ with $\overline{C} = X; \mathsf{if}\ B\ \mathsf{then}\ \overline{C_1}\ \mathsf{else}\ \overline{C_2}; Y$ with $C_1 \rightharpoonup \overline{C_1}$ and $C_2 \rightharpoonup \overline{C_2}$. By induction hypothesis, $\overline{C_1}, \overline{C_2} \in Mgl_{\mathcal{V}}$, and by definition of $Mgl_{\mathcal{V}}$ we have $\overline{C} \in Mgl_{\mathcal{V}}$.

If $C = C_1; C_2$ let $C_1 \rightharpoonup \overline{C_1}$, and let $C_2 \rightharpoonup \overline{C_2}$. By induction hypothesis, $\overline{C_1}, \overline{C_2} \in Mgl_{\mathcal{V}}$. As $\overline{C_1} \in Mgl_{\mathcal{V}}$, we can write it as (implicit induction on $Mgl_{\mathcal{V}}$) $\overline{C_1} = \overline{C_1'}; P; X$ for a $P \in Pad_{\mathcal{V}}$ and a meta-variable $X \in \mathcal{V}$. Observe that $P; \overline{C_2} \in Mgl_{\mathcal{V}}$, and thus $\overline{C_1'}; P; \overline{C_2} \in Mgl_{\mathcal{V}}$. This is what we wanted, as we have $C_1; C_2 \rightharpoonup \overline{C_1'}; P; \overline{C_2}$.

If $V = \langle C_1, \ldots, C_n \rangle \in \vec{Com}$, we have $V \rightharpoonup \overline{V}$ with $\overline{V} = \langle \overline{C_1}, \ldots, \overline{C_n} \rangle$ with $C_i \rightharpoonup \overline{C_i}$ for $i = 1, \ldots, n$. By induction hypothesis, $\overline{C_1}, \ldots, \overline{C_n} \in Mgl_{\mathcal{V}}$, and thus $\overline{V} \in \vec{Mgl}_{\mathcal{V}}$.

The assertion that each meta-variable occurs at most once follows from the requirement that each meta-variable inserted while lifting must be *fresh*. □

A.7 Proof of Theorem 5.4

Theorem 5.4. Let $V_i \in \vec{Com}$ with liftings $V_i', \overline{V_i} \in \vec{Com}_{\mathcal{V}}$ for $i = 1, 2$. Suppose $\overline{V_1}$ ($\overline{V_2}$) shares no meta-variables with V_1', V_2', and $\overline{V_2}$ (V_1', V_2', and $\overline{V_1}$). If $V_1 \rightharpoonup \overline{V_1}$ and $V_2 \rightharpoonup \overline{V_2}$ can be derived, then

1. $\mathcal{U}(\{V_1' \overset{?}{\simeq}_L V_2'\}) \neq \emptyset$ implies $\mathcal{U}(\{\overline{V_1} \overset{?}{\simeq}_L \overline{V_2}\}) \neq \emptyset$, and
2. $\mathcal{U}(\{V_1' \overset{?}{\simeq}_L V_1'\}) \neq \emptyset$ implies $\mathcal{U}(\{\overline{V_1} \overset{?}{\simeq}_L \overline{V_1}\}) \neq \emptyset$.

Proof. 1. From Lemma 5.4, it follows that $\overline{V_1}, \overline{V_2} \in \vec{Mgl}_{\mathcal{V}}$. The claim then follows immediately by applying Lemma 5.3.

2. From Lemma 5.4, it follows that $\overline{V} \in \vec{Mgl}_{\mathcal{V}}$, hence it suffices to show the assertion for an arbitrary $V^* \in \vec{Mgl}_{\mathcal{V}}$. We will inductively construct a substitution ρ, with $\rho V^* \overset{?}{\simeq}_L \rho V^*$ and $dom(\rho) \subseteq var(V^*)$, starting with $V^* = C^* \in Mgl_{\mathcal{V}}$ and $V' = C' \in Com_{\mathcal{V}}$

   Suppose $C^* \in Pad_{\mathcal{V}}$. The identity is then in $\mathcal{U}(\{C^* \overset{?}{\simeq}_L C^*\})$.

   Suppose $C^* = P; Id := Exp; C_1^*$ with $Id : L$. C' is also a lifting of C, so $C' = P'; Id := Exp; C_1'$ with possibly empty (i.e., equal to ϵ) P', C'. From $\sigma C' \simeq_L \sigma C'$, and the definition of $\simeq_L$ we know that $Exp : L$ and $\mathcal{U}(\{C_1' \overset{?}{\simeq}_L C_1'\}) \neq \emptyset$. We apply induction hypothesis to obtain $\rho \in \mathcal{U}(\{C_1^* \overset{?}{\simeq}_L C_1^*)$. From Lemma A.7, we obtain $\rho C^* \simeq_L \rho C^*$.

   Suppose $C^* = P; \mathsf{if}\ B\ \mathsf{then}\ C_1^*\ \mathsf{else}\ C_2^*; C_3^*$ with $B : L$. C' is also a lifting of C, so $C' = P'; \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'; C_3'$. From $\sigma C' \simeq_L \sigma C'$ and the definition of $\simeq_L$ we know that $\mathcal{U}(\{C_i' \overset{?}{\simeq}_L C_i'\}) \neq \emptyset$ for $i = 1, 2, 3$. We apply induction hypothesis to obtain $\rho_i \in \mathcal{U}(\{C_i^* \overset{?}{\simeq}_L C_i^*\})$ for $i = 1, 2, 3$. $\rho = \rho_1 \cup \rho_2 \cup \rho_3$ is well–defined as the domains are pairwise disjoint, and $\rho \in \mathcal{U}(\{C^* \overset{?}{\simeq}_L C^*\})$.

   Suppose $C^* = P; \mathsf{if}\ B\ \mathsf{then}\ C_1^*\ \mathsf{else}\ C_2^*; C_3^*$ with $B : H$. We then know that $C' = P'; \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'; C_3'$ because both C' and C^* are liftings of C. From $\sigma C' \simeq_L \sigma C'$ and the definition of $\simeq_L$ we get $\sigma C_1' \simeq_L \sigma C_2'$. By Lemma 5.3, we obtain ρ_1 with $\rho_1 C_1^* \simeq_L \rho_1 C_2^*$ (note that every meta-variable occurs at most once in (C_1^*, C_2^*)). Applying induction hypothesis to P and C_3^*, we obtain ρ_0 with $\rho_0 P \simeq_L \rho_0 P$ and

ρ_2 with $\rho_2 C_3^* \doteq_L \rho_2 C_3^*$. With $\rho = \rho_0 \cup \rho_1 \cup \rho_2$, we have $\rho C^* \doteq_L \rho C^*$, which is what we wanted.

The remaining induction steps for $Mgl_{\mathcal{V}}$ and the lifting to $\vec{Mgl}_{\mathcal{V}}$ can be treated in the same way as the low conditional. □

A.8 Proof of Lemma 5.5

Lemma 5.5. Let $V_1, V_2 \in \vec{Slice}_{\mathcal{V}}$ and assume that no meta-variable occurs more than once in (V_1, V_2). If $V_1 \doteq^?_L V_2 :: \eta$, then

1. $\eta \in \mathcal{U}(\{V_1 \doteq^?_L V_2\})$, and
2. η is idempotent, and
3. $dom(\eta) \cup var(ran(\eta)) \subseteq var(V_1) \cup var(V_2)$, and
4. $V_1, V_2 \in \vec{Mgl}_{\mathcal{V}}$ implies $\eta V_1, \eta V_2 \in \vec{Mgl}_{\mathcal{V}}$,

where $var(\cdot)$ returns the set of meta-variables occurring in a command or a set of commands in $\vec{Com}_{\mathcal{V}}$.

Proof. For proving Assertions 1, 2, and 3, we proceed by structural induction on the derivation tree $\mathcal{D}$ of the judgment $V_1 \doteq^?_L V_2 :: \eta$. For this, note that assertion 2 is equivalent to $dom(\eta) \cap var(ran(\eta)) = \emptyset$.

It is not difficult to see that the application of the rules in Figure 5.5 preserves Assertions 2 and 3. Hence, we focus on proving Assertion 1.

If $\mathcal{D}$ consists of an application of the rule [*UVar*$_1$], we have $V_1 = \alpha$ and $V_2 = C$. The assertion follows, as α does not occur in $var(C)$ by assumption. [*UVar*$_2$] follows similarly.

If the root of $\mathcal{D}$ is an application of rule [*USeq*$_1$], we have $V_1 = \alpha; C_1$ and $V_2 = C_2$, and $C_1 \doteq^?_L C_2 :: \eta$. By hypothesis, $\eta C_1 \doteq_L \eta C_2$ holds. $(\eta \cup \{\alpha \backslash \epsilon\})\alpha; C_1 = \eta C_1 \doteq_L \eta C_2 \doteq_L (\eta \cup \{\alpha \backslash \epsilon\}) C_2$, as α does not occur in C_1, C_2. [*USeq*$'_1$] follows similarly.

If the root of $\mathcal{D}$ is an application of rule [*USeq*$_2$], we have $V_1 = \mathsf{skip}; C_1$ and $V_2 = \mathsf{skip}; C_2$, and $C_1 \doteq^?_L C_2 :: \eta$. By hypothesis, $\eta C_1 \doteq_L \eta C_2$ holds. Then, by Lemma A.7 we also have $\eta(\mathsf{skip}; C_1) \doteq_L \eta(\mathsf{skip}; C_2)$.

We prove Assertions 1, 2, and 3 for the rules in Figure 5.4 for the example of the rule [*UCond*]. The remaining cases can be proved along the same lines.

If the root of $\mathcal{D}$ is the application of $[UCond]$, we have $V_1 = \mathsf{if}\ B_1\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ and $V_2 = \mathsf{if}\ B_2\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2$, together with $B_1 \equiv B_2$ and $C_1 \overset{?}{\simeq}_L C'_1 :: \eta_1$, $C_2 \overset{?}{\simeq}_L C'_2 :: \eta_2$. By hypothesis, we have $\eta_i \in \mathcal{U}(\{C_i \overset{?}{\simeq}_L C'_i\})$, and $dom(\eta_i) \cup var(ran(\eta_i)) \subseteq var(C_i) \cup var(C'_i)$ for $i = 1, 2$. As $var(C_1) \cup var(C'_1)$ and $var(C_2) \cup var(C'_2)$ are disjoint by hypothesis, $\eta = \eta_1 \cup \eta_2$ is well-defined, $dom(\eta) \cap var(ran(\eta)) = \emptyset$, and $dom(\eta) \cup var(ran(\eta)) \subseteq var(V_1) \cup var(V_2)$ hold. From Lemma A.7, we see that η is indeed a unifier of V_1 and V_2.

For proving Assertion 4, we proceed by induction on the term structure of V_1. We treat the case of commands $V_1 = S_1$ and $V_2 = S_2 \in Slice_{\mathcal{V}}$ first.

Suppose $S_1 \in Stut_{\mathcal{V}}$. Only the rules $[UVar_1]$,$[UVar_2]$,$[USeq_1]$, $[USeq'_1]$ and $[USeq_2]$ apply, so $S_2 \in Stut_{\mathcal{V}}$. As $S_1, S_2 \in Mgl_{\mathcal{V}}$, we see that both commands contain a terminal meta-variable, i.e., a meta-variable as the rightmost subterm. The two base cases for the derivation, $[UVar_1]$ and $[UVar_2]$, map the meta-variable at the end of one command to the respective other command. Hence, both ηS_1 and ηS_2 have terminal meta-variables. As (S_1, S_2) contains every meta-variable only once, the same holds for ηS_1 and ηS_2, and thus they are elements of $Mgl_{\mathcal{V}}$.

Suppose $S_1 = P_1; \mathsf{if}\ B_1\ \mathsf{then}\ S_{1,1}\ \mathsf{else}\ S_{1,2}; S_{1,3}$ with $B : L$. From $S_1 \overset{?}{\simeq}_L S_2 :: \eta$ and the rules $[UCond]$ $[USeq_3]$ and $[USeq_4]$ in Figures 5.4 and 5.5, we obtain that $S_2 = P_2; \mathsf{if}\ B_2\ \mathsf{then}\ S_{2,1}\ \mathsf{else}\ S_{2,2}; S_{2,3}$, with $B_1 \equiv B_2$ and $P_1 \overset{?}{\simeq}_L P_2 :: \eta_0$, $S_{1,1} \overset{?}{\simeq}_L S_{2,1} :: \eta_1$, $S_{1,2} \overset{?}{\simeq}_L S_{2,2} :: \eta_2$, and $S_{1,3} \overset{?}{\simeq}_L S_{2,3} :: \eta_3$ are derivable. By induction hypothesis, we have $\eta_0 P_1$, $\eta_1 S_{1,1}$, $\eta_2 S_{1,2}$, $\eta_3 S_{1,3} \in Mgl_{\mathcal{V}}$. With $\eta = \eta_0 \cup \eta_1 \cup \eta_2 \cup \eta_3$ and the fact that the domains and variable ranges of the η_i are mutually disjoint for $i \in \{0, 1, 2, 3\}$, we conclude $\eta S_1 = \eta_0 P_1; \mathsf{if}\ B_1\ \mathsf{then}\ \eta_1 S_{1,1}\ \mathsf{else}\ \eta_2 S_{1,2}; \eta_3 S_{1,3}$, which is in $Mgl_{\mathcal{V}}$ as it contains every meta-variable at most once.

All other constructors and the lifting to command vectors can be treated in the same way as the low conditional. $\square$

A.9 Proof of Theorem 5.5

The proof of Theorem 5.5 essentially proceeds by induction on the term structure of W. The main technical difficulty in the proof lies in showing that if the two branches C_1, C_2 of a conditional with high guard, with $C_i \hookrightarrow' C'_i : S_i$ for $i = 1, 2$, unify, then we can also unify the corresponding slices, i.e. $S_1 \overset{?}{\simeq}_L S_2 :: \eta$. To prove this, we proceed in two steps:

1. We show that S_1 and S_2 are structurally equivalent "modulo" commands in $Pad_{\mathcal{V}}$,

$$\frac{P, P' \in Stut_{\mathcal{V}} \cup \{\epsilon\}}{P \doteq P'} \qquad \frac{P, P' \in Stut_{\mathcal{V}} \cup \{\epsilon\} \quad C \doteq C' \quad Id : L \quad Exp_1 \equiv Exp_2}{P; Id := Exp_1; C \doteq P'; Id := Exp_2; C'}$$

$$\frac{P, P' \in Stut_{\mathcal{V}} \cup \{\epsilon\} \quad C_1 \doteq C'_1 \quad C_2 \doteq C'_2 \quad V \doteq V'}{P; \mathsf{fork}(C_1 V); C_2 \doteq P'; \mathsf{fork}(C'_1 V'); C'_2} \qquad \frac{C_1 \doteq C'_1, \ldots, C_n \doteq C'_n}{\langle C_1, \ldots, C_n \rangle \doteq \langle C'_1, \ldots, C'_n \rangle}$$

$$\frac{P, P' \in Stut_{\mathcal{V}} \cup \{\epsilon\} \quad C_i \doteq C'_i \quad i = 1, 2, 3 \quad B_1 \equiv B_2}{P; \mathsf{if}\ B_1\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3 \doteq P'; \mathsf{if}\ B_2\ \mathsf{then}\ C'_1\ \mathsf{else}\ C'_2; C'_3}$$

$$\frac{P, P' \in Stut_{\mathcal{V}} \cup \{\epsilon\} \quad C_1 \doteq C'_1 \quad C_2 \doteq C'_2 \quad B_1 \equiv B_2}{P; \mathsf{while}\ B_1\ \mathsf{do}\ C_1; C_2 \doteq P'; \mathsf{while}\ B_2\ \mathsf{do}\ C'_1; C'_2}$$

Figure A.1: Resembling commands

and that they are in $Mgl_{\mathcal{V}}$ (Definition A.4, Lemma A.8).

2. We show that we can unify every two structurally equivalent slices, given that they are elements of $Mgl_{\mathcal{V}}$ (Lemma A.10.1)

Lemma A.9 and Lemma A.10.2 are rather technical and will be needed during the proof. We first formulate the above steps in terms of definitions and lemmata, before we proceed with the proof of Theorem 5.5.

To simplify the atomic treatment of subcommands in $Stut_{\mathcal{V}}$ of commands in $Slice_{\mathcal{V}}$ in inductive arguments, we introduce the language $Slice^{+}_{\mathcal{V}}$ (note the resemblance to the definition of $Mgl_{\mathcal{V}}$). We define the set $Slice^{+}_{\mathcal{V}}$ by the following grammar:

$$L ::= P \mid P; Id_l := Exp; L \mid P; \mathsf{if}\ B\ \mathsf{then}\ L_1\ \mathsf{else}\ L_2; L \mid P; \mathsf{while}\ B\ \mathsf{do}\ L_1; L \mid P; \mathsf{fork}(L_1 V); L\,,$$

where L, L_1, L_2 are placeholders for commands in $Slice^{+}_{\mathcal{V}}$, V is a placeholder for a command vector in $\vec{Slice}^{+}_{\mathcal{V}}$, and P is a placeholder for a command in $Stut_{\mathcal{V}} \cup \{\epsilon\}$. By a straightforward induction one proves that $Slice_{\mathcal{V}} \subseteq Slice^{+}_{\mathcal{V}}$.

Definition A.4. The binary relation $\doteq$ on $\vec{Slice}^{+}_{\mathcal{V}}$ is defined as the reflexive, symmetric and transitive closure of the relation inductively defined in Figure A.1. We call commands $V, V' \in \vec{Mgl}_{\mathcal{V}}$ with $V \doteq V'$ *resembling*.

Lemma A.8. *Let $C, C_1, C_2 \in Mgl_{\mathcal{V}}$. Then the following assertions hold:*

1. *$C \hookrightarrow' C' : S$ implies $S \in Mgl_{\mathcal{V}} \cap Slice_{\mathcal{V}}$ and $var(S) \subseteq var(C)$.*

2. *$\mathcal{U}(\{C_1 \overset{?}{\simeq}_L C_2\}) \neq \emptyset$ and $C_i \hookrightarrow' C'_i : S_i$ for $i = 1, 2$ implies $S_1 \doteq S_2$.*

Proof. 1. It is easy to see that $S \in Slice_{\mathcal{V}}$ holds, and we concentrate on containment in $Mgl_{\mathcal{V}}$. We proceed by induction on the term structure of $C \in Mgl_{\mathcal{V}}$.

Suppose $C \in Pad_{\mathcal{V}} \cap Mgl_{\mathcal{V}}$. By definition of $Mgl_{\mathcal{V}}$, C has a terminal meta-variable. Then clearly $S \in Mgl_{\mathcal{V}}$ as it contains a terminal meta-variable and no assignments. Furthermore, $var(S) \subseteq var(C)$ is fulfilled.

Suppose now $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : L$ and $C \hookrightarrow' C' : S$. Then by definition of $\hookrightarrow'$ we have $P \hookrightarrow' P' : S_0$, $C_1 \hookrightarrow' C_1' : S_1$, $C_2 \hookrightarrow' C_2' : S_2$ and $C_3 \hookrightarrow' C_3' : S_3$. By induction hypothesis, $S_0, S_1, S_2, S_3 \in Mgl_{\mathcal{V}}$ and the condition on the meta-variables holds. By definition of $Mgl_{\mathcal{V}}$, $S = S_0; \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; S_3 \in Mgl_{\mathcal{V}}$, and $var(S) \subseteq var(C)$ holds.

Suppose now $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : H$. We have $C \hookrightarrow' C' : S$, hence by definition of $\hookrightarrow'$ we have $P \hookrightarrow' P' : S_0$, $C_1 \hookrightarrow' C_1' : S_1$, $C_2 \hookrightarrow' C_2' : S_2$ and $C_3 \hookrightarrow' C_3' : S_3$ and also $S_1 \doteqdot_L S_2 :: \eta$ for some η. By induction hypothesis, $S_1, S_2, S_3 \in Mgl_{\mathcal{V}}$ and $var(S_i) \subseteq var(C_i)$. Hence (S_1, S_2) contains every meta-variable at most once. With Lemma 5.5.4 we see that $\eta S_1 \in Mgl_{\mathcal{V}}$, and every meta-variable occurs at most once in ηS_1. From Lemma 5.5.3, it follows that $dom(\eta) \cup ran(var(\eta))$ is a subset of the meta-variables in S_1 and S_2 and hence $var(S) \subseteq var(C)$ is fulfilled for $S = (S_0; \mathsf{skip}; \eta S_1); S_3$. $S_0; \mathsf{skip} \in Stut_{\mathcal{V}}$, and so by the definition of $Mgl_{\mathcal{V}}$, $S_0; \mathsf{skip}; \eta S_1 \in Mgl_{\mathcal{V}}$. By a straightforward induction, one shows that $D_1; D_2 \in Mgl_{\mathcal{V}}$ whenever $D_1, D_2 \in Mgl_{\mathcal{V}}$, and we see that $S = (S_0; \mathsf{skip}; \eta S_1); S_3$ and hence in $Mgl_{\mathcal{V}}$.

The cases for the other constructors follow along the same lines as the low conditional.

2. Let $\sigma C_1 \doteqdot_L \sigma C_2$. By symmetry and transitivity of $\doteqdot_L$, we conclude $\sigma C_1 \doteqdot_L \sigma C_1$ and $\sigma C_2 \doteqdot_L \sigma C_2$. With the help of Lemma A.9, we obtain $C_1'', C_2'' \in Slice_{\mathcal{V}}$ with $S_1 \doteq C_1'' \doteqdot_L \sigma C_1 \doteqdot_L \sigma C_2 \doteqdot_L C_2'' \doteq S_2$. Lemma A.10.1 shows that $C_1'' \doteqdot_L C_2''$ implies $C_1'' \doteq C_2''$, and by transitivity of $\doteq$ we get $S_1 \doteq S_2$. □

Lemma A.9. *Let $C \in Mgl_{\mathcal{V}}$ with $\sigma \in \mathcal{U}(\{C \doteqdot_L C\})$ and $C \hookrightarrow' C' : S$. Then there is a $C'' \in Slice_{\mathcal{V}}$ with $C'' \doteqdot_L \sigma C$ and $C'' \doteq S$.*

Proof. We proceed by structural induction on $C \in Mgl_{\mathcal{V}}$.

Suppose $C \in Pad_{\mathcal{V}}$. Choose C'' as σC, where all assignments to high variables are replaced by skips. We have $C'' \in Stut_{\mathcal{V}}$. We also have $S \in Stut_{\mathcal{V}}$, and so $C'' \doteq S$.

Suppose $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : L$. We have $\sigma C \simeq_L \sigma C$, hence by definition of $\simeq_L$ we obtain $\sigma P \simeq_L \sigma P$ and $\sigma C_i \simeq_L \sigma C_i$ for $i = 1, 2, 3$. From the precondition $C \hookrightarrow' C' : S$ and the definition of $\hookrightarrow'$ we get $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1, 2, 3$. We apply induction hypothesis to the corresponding command-pairs and obtain P'', C_i'' with $P'' \simeq_L \sigma P$, $P'' \doteq S_0$, and $C_i'' \simeq_L \sigma C_i$, $C_i'' \doteq S_i$ for $i = 1, 2, 3$. Then we can conclude $C'' = P''; \mathsf{if}\ B\ \mathsf{then}\ C_1''\ \mathsf{else}\ C_2''; C_3'' \simeq_L \sigma C$, as well as $C'' \doteq S$.

Suppose $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : H$. We have $\sigma C \simeq_L \sigma C$, so by definition of $\simeq_L$ we obtain $\sigma P \simeq_L \sigma P$, $\sigma C_1 \simeq_L \sigma C_1$ $\sigma C_3 \simeq_L \sigma C_3$ and $\sigma C_1 \simeq_L \sigma C_2$. By symmetry and transitivity of $\simeq_L$ we conclude $\sigma C_2 \simeq_L \sigma C_2$. From the precondition $C \hookrightarrow' C' : S$ and the definition of $\hookrightarrow'$ we get $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1, 2, 3$. We apply induction hypothesis to the corresponding command-pairs and obtain P'', C_i'' with the desired properties for $i = 1, 2, 3$. Define C'' as $P''; \mathsf{skip}; C_1''; C_3'' \in \mathit{Slice}_{\mathcal{V}}$. We have $C'' \simeq_L \sigma C$ by Lemma A.7. On the other hand, $S = S_0; \mathsf{skip}; \eta S_1; S_3$ for some preserving substitution η. We are left to show $S \doteq C''$. By hypothesis, we know $C_1'' \doteq S_1$ and $C_3'' \doteq S_3$. A straightforward induction shows that $S_1 \doteq \eta S_1$ for every preserving η. Another straightforward induction shows that $\doteq$ is a congruence with respect to sequential composition and so we conclude $C'' = P''; \mathsf{skip}; C_1''; C_3'' \doteq S_0; \mathsf{skip}; \eta S_1; S_3 = S$.

The other constructors follow along the same lines as the low conditional. □

Lemma A.10. *Let $S_1, S_2 \in \mathit{Slice}_{\mathcal{V}}$. Then the following holds.*

1. *$S_1 \simeq_L S_2$ implies $S_1 \doteq S_2$.*

2. *If $S_1, S_2 \in \mathit{Mgl}_{\mathcal{V}}$ and $S_1 \doteq S_2$ then there is an η with $S_1 \overset{?}{\simeq}_L S_2 :: \eta$.*

Proof. We prove Assertion 1 by induction on the structure of S_1, where we make use of the fact that $\mathit{Slice}_{\mathcal{V}} \subseteq \mathit{Slice}^{+}_{\mathcal{V}}$. If $S_1 \in \mathit{Stut}_{\mathcal{V}}$, then by definition of $\simeq_L$ and the precondition $S_2 \in \mathit{Slice}_{\mathcal{V}}$ we know that $S_2 \in \mathit{Stut}_{\mathcal{V}}$, and hence $S_1 \doteq S_2$. If $S_1 = P_1; \mathsf{if}\ B_1\ \mathsf{then}\ S_{1,1}\ \mathsf{else}\ S_{1,2}; S_{1,3}$ with $B_1 : L$, then by definition of $\simeq_L$ we know that $S_2 = P_2; \mathsf{if}\ B_2\ \mathsf{then}\ S_{2,1}\ \mathsf{else}\ S_{2,2}; S_{2,3}$ with $P_1, P_2 = \epsilon$ or $P_1 \simeq_L P_2$, and $S_{1,i} \simeq_L S_{2,i}$ for $i = 1, 2, 3$ and $B_1 \equiv B_2$. By definition of $\doteq$, $P_1 \doteq P_2$ holds, and by induction hypothesis we see $S_{1,i} \doteq S_{2,i}$ for $i = 1, 2, 3$. By definition of $\doteq$ we conclude $S_1 \doteq S_2$. The other constructors follow in a similar fashion.

We prove Assertion 2 by induction on the term structure of S_1:

Suppose $S_1 \in \mathit{Stut}_{\mathcal{V}}$. Then by definition of $\doteq$, S_2 must also be in $\mathit{Stut}_{\mathcal{V}}$. $S_1, S_2 \in \mathit{Mgl}_{\mathcal{V}}$, hence they have terminal meta-variables. A simple induction on the length of S_1 shows that $S_1 \simeq_L S_2 :: \eta$ is derivable.

Suppose $S_1 = P_1; \mathsf{if}\ B_1\ \mathsf{then}\ S_{1,1}\ \mathsf{else}\ S_{1,2}; S_{1,3}$, where $P_1, S_{1,1}, S_{1,2}, S_{1,3}$ are elements of $Mgl_{\mathcal{V}} \cap Slice_{\mathcal{V}}$. By definition of $\doteq$ and $Mgl_{\mathcal{V}}$ we get $S_2 = P_2; \mathsf{if}\ B_2\ \mathsf{then}\ S_{2,1}\ \mathsf{else}\ S_{2,2}; S_{2,3}$ with $P_2, S_{2,1}, S_{2,2}, S_{2,3} \in Mgl_{\mathcal{V}} \cap Slice_{\mathcal{V}}$ and $P_1 \doteq P_2$, $S_{1,i} \doteq S_{2,i}$ for $i = 1,2,3$, and $B_1 \equiv B_2$. We apply induction hypothesis to obtain $P_1 {\simeq}^?_L P_2 :: \sigma_0$ and $S_{1,i} {\simeq}^?_L S_{2,i} :: \sigma_i$, for $i = 1,2,3$. We can conclude $S_1 {\simeq}^?_L S_2 :: \sigma$ with $\sigma = \bigcup_{i=0}^{3} \sigma_i$ by definition of the unification calculus. The other cases follow along the same lines. □

Theorem 5.5. Let $V \in \vec{Com}$, $\overline{V}, W \in \vec{Com}_{\mathcal{V}}$, W be a lifting of V, and $V \rightharpoonup \overline{V}$.

1. If there is a preserving substitution σ with $\sigma W \simeq_L \sigma W$, then $\overline{V} \hookrightarrow' V' : S$ for some $V', S \in \vec{Com}_{\mathcal{V}}$.

2. If $W \hookrightarrow W' : S$ for some $W', S \in \vec{Com}_{\mathcal{V}}$ then $\overline{V} \hookrightarrow' V' : S'$ for some $V', S' \in \vec{Com}_{\mathcal{V}}$.

Proof. 1. Let W be an arbitrary lifting of V. From Theorem 5.4, it follows that $\sigma W \simeq_L \sigma W$ implies $\mathcal{U}(\overline{V} \simeq_L \overline{V}) \neq \emptyset$. From Lemma 5.4, we see that $\overline{V} \in Mgl_{\mathcal{V}}$. Restricting ourselves to commands in $Com_{\mathcal{V}}$ for the moment and substituting C for $\overline{V}$, it suffices to show the assertion

$$\exists \sigma . \sigma C \simeq_L \sigma C \Rightarrow C \hookrightarrow' C' : S$$

for all $C \in Mgl_{\mathcal{V}}$. We prove this assertion by induction on the term structure of C.

If $C \in Pad_{\mathcal{V}}$, we always have $C \hookrightarrow' C' : S$.

If $C = P; Id := Exp; C_1$, with $Id : L$ and $\sigma C \simeq_L \sigma C$, then we have $\sigma P \simeq_L \sigma P$, $\sigma C_1 \simeq_L \sigma C_1$, and $Exp : L$ by definition of $\simeq_L$. By applying induction hypothesis we obtain $P \hookrightarrow' P' : S_0$ and $C_1 \hookrightarrow' C_1' : S_1$. By definition of $\hookrightarrow'$ this implies $C \hookrightarrow' P'; Id := Exp; C_1' : S_0; Id := Exp; S_1$.

If $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : L$, and $\sigma C \simeq_L \sigma C$, then we have $\sigma P \simeq_L \sigma P$ and $\sigma C_i \simeq_L \sigma C_i$ for $i = 1,2,3$. By applying induction hypothesis, we obtain $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2,3$. By definition of $\hookrightarrow'$, this implies $C \hookrightarrow' P'; \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'; C_3' : S_0; \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2; S_3$

If $C = P; \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2; C_3$ with $B : H$, and $\sigma C \simeq_L \sigma C$, then we have $\sigma P \simeq_L \sigma P$ and $\sigma C_i \simeq_L \sigma C_i$ for $i = 1,3$. Furthermore we have $\sigma C_1 \simeq_L \sigma C_2$, from which we get $\sigma C_2 \simeq_L \sigma C_2$ by transitivity and symmetry of $\simeq_L$. Induction hypothesis yields $P \hookrightarrow' P' : S_0$ and $C_i \hookrightarrow' C_i' : S_i$ for $i = 1,2,3$. Every meta-variable occurs at most once in C, hence the same holds true for the subterms C_1, C_2.

Lemma A.8 shows that $S_1, S_2 \in Mgl_{\mathcal{V}} \cap Slice_{\mathcal{V}}$, $S_1 \doteq S_2$, and every meta-variable occurs at most once in (S_1, S_2). Lemma A.10.2 implies that there is an η with $S_1 \overset{?}{\simeq}_L S_2 :: \eta$. η is a unifier of S_1, S_2, as Lemma 5.5 shows. We conclude that $C \hookrightarrow' P'; \text{if } B \text{ then } \eta C_1' \text{ else } \eta C_2'; C_3' : S_0; \text{skip}; \eta S_1; S_3$, which is what we wanted.

The cases for the other constructors follow along the same lines as the low conditional. The assertion can then simply be lifted to command vectors.

2. By a straightforward induction over the derivation tree, it follows that $W \hookrightarrow W' : S$ implies $W' = \sigma W$ for a preserving substitution σ. From Lemma 5.1 $W' \simeq_L S$ follows. By symmetry and transitivity of $\simeq_L$, we obtain $\sigma W \simeq_L \sigma W$. The assertion now follows directly from part 1 of Theorem 5.5.

□

Appendix B

Code

B.1 Integer Multiplication

```
// Multiplication of 6-bit integers in GEZEL
// Computes "result=m_in*n_in", termination is signaled by "done"

dp prod(in  m_in, n_in : ns(6); out result : ns(12); out done : ns(1)) {

 reg i : ns(4);
 reg p : ns (12);
 reg n,m : ns(6);

 sfg init  {p=0;
            m=m_in;
            n=n_in;
            i=0;}
 sfg shift {p = p << 1;
            i=i+1;
            n = n << 1;}
 sfg add   {p = p + m;}
 sfg cont  {result=0;
            done=0;}
 sfg term  {result = p;
            done=1;}
}
```

```
fsm ctl_prod(prod) {
 initial s0;
 state s1,s2,s3;

 @s0 (init,cont) -> s1;
 @s1 if (i<6) then
      if (n[5]) then (shift,cont) -> s2;
                else (shift,cont) -> s1;
      else(term) -> s3;
 @s2 (add,cont) -> s1;
 @s3 (term) -> s3;
}
```

B.2 Finite Field Exponentiation

```
// Exponentiation in GF(2^6) in GEZEL
// Computes "result=x_in^a_in", termination is signaled by "done"

dp exp(in  x_in, a_in : ns(6); out done : ns(1); out result : ns(6)){

 reg x : ns(6);
 reg a : ns(6);
 reg m : ns(6);
 reg p,q,s : ns(6);
 reg i : ns(4);
 reg j : ns(4);

 sfg init  {x=x_in;
            a=a_in;
            m=0b000011;}  // T^6+T+1: irreducible polynomial over F2
 sfg sig0  {p=1;}         // serves as field polynomial
 sfg sig1  {q=0;}
 sfg sig2  {i=6;}
 sfg sig3  {j=6;}
 sfg sig4  {i=i-1;}
 sfg sig5  {j=j-1;}
 sfg sig6  {s=p;}
 sfg sig7  {p=q;}
 sfg sig8  {s=x;}
 sfg sig9  {q=q<<1;}
 sfg sig10 {s=s<<1;}
 sfg sig11 {a=a<<1;}
 sfg sig12 {q=q^m;}
 sfg sig13 {q=q^p;}
 sfg cont  {done=0;
             result=0;}
 sfg term  {done=1;
             result=x;}
}

fsm ctl_exp(exp) {
 initial s1;
 state s2,s3,s4,s5,s6,s8,s9,s10,s11,s12,end;

 @s1 (init,sig0,sig2,cont) -> s2;
 @s2 if (i==0)
        then (term) -> end;
        else (sig1,sig3,sig6,cont) -> s3;
 @s3 if (j==0)
        then (sig7,cont) -> s8;
        else (cont) -> s4;
 @s4 if (q[5])
        then (sig9,cont) -> s5;
        else (sig9,cont) -> s6;
 @s5 (sig12,cont) -> s6;
 @s6 if (s[5])
        then (sig5,sig10,sig13,cont) -> s3;
        else (sig5,sig10,cont) -> s3;
 @s8 if (a[5])
```

```
        then (sig1,sig3,sig8,cont) -> s9;
        else (sig4,sig11,cont) -> s2;
 @s9 if (j==0)
        then (sig4,sig7,sig11,cont) -> s2;
        else (cont)-> s10;
 @s10 if (q[5])
        then (sig9,cont) -> s11;
        else (sig9,cont) -> s12;
 @s11 (sig12,cont) -> s12;
 @s12 if (s[5])
        then (sig5,sig10,sig13,cont) -> s9;
        else (sig5,sig10,cont) -> s9;
 @end (term) -> end;
}
```

Bibliography

[1] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys via Branch Prediction. In *Proc. Cryptographer's Track at RSA Conference (CT-RSA '07)*, LNCS 4377, pages 225–242. Springer, 2007.

[2] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Improving Brumley and Boneh Timing Attack on Unprotected SSL Implementations. In *Proc. 12th ACM Conference on Computer and Communications Security (CCS '05)*, pages 139–146. ACM, 2005.

[3] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache-Based Remote Timing Attack on the AES. In *Proc. Cryptographer's Track at RSA Conference (CT-RSA '07)*, LNCS 4377, pages 271–286. Springer, 2007.

[4] Johan Agat. Transforming out Timing Leaks. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 40–53. ACM, 2000.

[5] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM-Channels. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*, LNCS 2523, pages 29–45. Springer, 2002.

[6] Robert B. Ash. *Information Theory*. Dover Publications Inc., 1990.

[7] Aslan Askarov and Andrei Sabelfeld. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *Proc. IEEE Symposium on Security and Privacy (S & P '07)*, pages 207–221. IEEE Computer Society, 2007.

[8] Franz Baader and Wayne Snyder. Unification Theory. In *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[9] Gilles Barthe, Pedro D'Argenio, and Tamara Rezk. Secure Information Flow by Self-Composition. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW '04)*, pages 100–114. IEEE Computer Society, 2004.

[10] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing Timing Leaks through Transactional Branching Instructions. *Electr. Notes Theor. Comput. Sci.*, 153:33–55, 2006.

[11] G.P. Basharin. On a Statistical Estimate for the Entropy of a Sequence of Independent Random Variables. *Theory Probab. Appl.*, 47:333–336, 1959.

[12] David Basin, Stefan Friedrich, and Marek Gawkowski. Bytecode Verification by Model Checking. *Journal of Automated Reasoning*, 30(3-4):399–444, 2003.

[13] Tugkan Batu, Sanjoy Dasgupta, Ravi Kumar, and Ronitt Rubinfeld. The Complexity of Approximating Entropy. In *Proc. 34th ACM Symposium on Theory of Computing (STOC' 02)*, pages 678–687. ACM, 2002.

[14] Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, second edition, 1998.

[15] Dan Boneh and D. Brumley. Remote Timing Attacks are Practical. In *Proc. 12th USENIX Security Symposium*, 2003.

[16] Gérard Boudol and Ilaria Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theor. Comput. Sci.*, 281(1-2):109–130, 2002.

[17] David Brumley and Dan Boneh. Remote Timing Attacks are Practical. *Computer Networks*, 48(5):701–716, 2005.

[18] Christian Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zürich, 1997.

[19] Julien Cathalo, Francois Koeune, and Jean-Jacques Quisquater. A New Type of Timing Attack: Application to GPS. In *Proc. 5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '03)*, LNCS 2779, pages 291–303. Springer, 2003.

[20] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *Proc. CRYPTO '99*, LNCS 1666, pages 398–412. Springer, 1999.

[21] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template Attacks. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*, LNCS 2523, pages 13–28. Springer, 2002.

[22] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative Information Flow, Relations and Polymorphic Types. *J. Log. Comput.*, 18(2):181–199, 2005.

[23] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[24] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in Information Flow. In *Proc. 18th IEEE Computer Security Foundations Workshop (CSFW '05)*, pages 31–45. IEEE Computer Society, 2005.

[25] Ellis Cohen. Information Transmission in Sequential Programs. In *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[26] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[27] Mads Dam. Decidability and Proof Systems for Language-based Noninterference Relations. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 67–78. ACM, 2006.

[28] Ádám Darvas, Reiner Hähnle, and Dave Sands. A Theorem Proving Approach to Analysis of Secure Information Flow. In *Proc. 2nd International Conference on Security in Pervasive Computing*, LNCS 3450, pages 193 – 209. Springer, 2005.

[29] Marc Davio, Jean-Pierre Deschamps, and André Thayse. *Digital Systems with Algorithm Implementation*. John Wiley & Sons, Inc., 1983.

[30] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.

[31] Jean-Francois Dhem, Francois Koeune, Philippe-Alexandre Leroux, Patrick Mestre, Jean-Jacques Quisquater, and Jean-Louis Willems. A Practical Implementation of the Timing Attack. In *Proc. International Conference on Smart Card Research and Applications (CARDIS '98)*, LNCS 1820, pages 167–182. Springer, 1998.

[32] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. Information Flow Analysis in a Discrete-Time Process Algebra. In *Proc. 13th IEEE Computer Security Foundations Workshop (CSFW '00)*, pages 170–184. IEEE Computer Society, 2000.

[33] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *Proc. 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES '01)*, LNCS 2162, pages 251–261. Springer, 2001.

[34] Yves Geissbühler. Quantifying Information Leaks, Semester Thesis, ETH Zurich, 2006.

[35] Roberto Giacobazzi and Isabella Mastroeni. Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 186–197. ACM, 2004.

[36] Roberto Giacobazzi and Isabella Mastroeni. Timed Abstract Non-Interference. In *Proc. International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS '05)*, LNCS 3829, pages 289–303. Springer, 2004.

[37] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proc. IEEE Symposium on Security and Privacy (S& P '82)*, pages 11–20. IEEE Computer Society, 1982.

[38] James W. Gray. Toward a Mathematical Foundation for Information Flow Security. *Journal of Computer Security*, 1(3-4):255–294, 1992.

[39] Daniel Hedin and David Sands. Timing Aware Information Flow Security for a JavaCard-like Bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1):163–182, 2005.

[40] Alexander Herold and Jörg Siekmann. Unification in Abelian Semigroups. *Journal of Automated Reasoning*, 3:247–283, 1987.

[41] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[42] Paris C. Kanellakis and Scott A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[43] Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proc. 16th Annual International Cryptology Conference (CRYPTO '96)*, LNCS 1109, pages 104–113. Springer, 1996.

[44] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proc. 19th Annual International Cryptology Conference (CRYPTO '99)*, LNCS 1666, pages 388–397. Springer, 1999.

[45] Robert König, , Ueli Maurer, and Stefano Tessaro. Abstract Storage Devices. *ArXiv e-prints*, 706, June 2007.

[46] Boris Köpf and David Basin. Timing-Sensitive Information Flow Analysis for Synchronous Systems. In *Proc. 11th European Symposium on Research in Computer Security (ESORICS '06)*, LNCS 4189, pages 243–262. Springer, 2006.

[47] Boris Köpf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks (to appear). In *Proc. 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, 2007.

[48] Boris Köpf and Heiko Mantel. Eliminating Implicit Information Leaks by Transformational Typing and Unification. In *Proc. 3rd International Workshop on Formal Aspects in Security and Trust (FAST'05)*, LNCS 3866, pages 42–62. Springer, 2006.

[49] Boris Köpf and Heiko Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *International Journal of Information Security*, 6:107–131, March 2007.

[50] Audrey Lim. Detecting Timing Leaks in Hardware, Semester Thesis, ETH Zurich, 2006.

[51] Gavin Lowe. Quantifying Information Flow. In *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 18–31. IEEE Computer Society, 2002.

[52] Heiko Mantel. Preserving Information Flow Properties under Refinement. In *Proc. IEEE Symposium on Security and Privacy (S & P '01)*, pages 78–91. IEEE Computer Society, 2001.

[53] Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität des Saarlandes, July 2003.

[54] James L. Massey. Guessing and Entropy. In *Proc. IEEE International Symposium on Information Theory (ISIT '94)*, page 204. IEEE Computer Society, 1994.

[55] J. D. McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *Proc. IEEE Symposium on Security and Privacy (S& P '94)*, pages 79–93. IEEE Computer Society, 1994.

[56] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[57] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power Analysis Attacks of Modular Exponentiation in Smartcards. In *Proc. 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES '99)*, LNCS 1717, pages 144–157. Springer, 1999.

[58] Silvio Micali and Leonid Reyzin. Physically Observable Cryptography (Extended Abstract). In *Proc. 1st Theory of Cryptography Conference (TCC '04)*, LNCS 2951, pages 278–296. Springer, 2004.

[59] Johnathan K. Millen. Covert Channel Capacity. In *Proc. IEEE Symposium on Security and Privacy (S& P '87)*, pages 60–66. IEEE Computer Society, 1987.

[60] Jonathan Millen. 20 Years of Covert Channel Modeling and Analysis. In *Proc. IEEE Symp. on Security and Privacy (S& P '99)*. IEEE Computer Society, 1999.

[61] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Math. Computation*, 44:519–521, 1985.

[62] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Proc. Cryptographer's Track at RSA Conference (CT-RSA '06)*, LNCS 3860, pages 1–20. Springer, 2006.

[63] Robert Paige and Robert E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 6(16):973–989, 1987.

[64] Alessandra Di Pierro, Chris Hankin, Igor Siveroni, and Herbert Wiklicky. Tempus Fugit: How To Plug It. *Journal of Logic and Algebraic Programming*, 72:173–190, 2007.

[65] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate Non-Interference. In *Proc.15th IEEE Computer Security Foundations Workshop (CSFW '02)*, pages 3–17. IEEE Computer Society, 2002.

[66] John O. Pliam. On the Incomparability of Entropy and Marginal Guesswork in Brute-Force Attacks. In *Proc. 1st International Conference in Cryptology in India (INDOCRYPT '00)*, LNCS 1977, pages 67–79. Springer, 2000.

[67] Peter Puschner and Alan Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[68] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Couter-Measures for Smard Cards. In *Proc. International Conference on Research in Smart Cards (E-SMART '01)*, LNCS 2140, pages 200–210. Springer, 2001.

[69] Christian Rechberger and Elisabeth Oswald. Practical Template Attacks. In *Proc. 5th International Workshop on Information Security Applications (WISA '04), Revised Papers*, LNCS 3325, pages 443–457. Springer, 2004.

[70] Alejandro Russo, John Hughes, David Naumann, and Andrei Sabelfeld. Closing Internal Timing Channels by Transformation. In *Proc. 11th Asian Computing Science Conference (ASIAN '06)*, LNCS. Springer, 2006.

[71] Alejandro Russo and Andrei Sabelfeld. Securing Interactions between Threads and the Scheduler. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW '06)*, pages 177–189. IEEE Computer Society, 2006.

[72] Peter Ryan, John D. McLean, Jonathan K. Millen, and Virgil D. Gligor. Noninterference, Who Needs It? In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 237–238. IEEE Computer Society, 2001.

[73] Andrei Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *Proc. 4th International Conference on Perspectives of System Informatics (PSI '01)*, LNCS 2244, pages 225–239. Springer, 2001.

[74] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-Flow Security. *IEEE J. Selected Areas in Communication*, 21(1):5–19, 2003.

[75] Andrei Sabelfeld and David Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proc. 13th IEEE Computer Security Foundations Workshop (CSFW '00)*, pages 200–215. IEEE Computer Society, 2000.

[76] Andrei Sabelfeld and David Sands. A PER Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.

[77] Patrick Schaumont and Doris Ching. GEZEL Language Reference. http://rijndael.ece.vt.edu/gezel2/index.php/GEZEL_Language_Reference.

[78] Patrick Schaumont, Doris Ching, and Ingrid Verbauwhede. An Interactive Codesign Environment for Domain-Specific Coprocessors. *ACM Transactions on Design Automation for Electronic Systems*, 11(1):70–87, 2006.

[79] Patrick Schaumont and Ingrid Verbauwhede. The Descriptive Power of GEZEL. Technical report, UCLA Electrical Engineering Department, 2005. http://rijndael.ece.vt.edu/gezel2/download/descrgezel.pdf.

[80] Fred B. Schneider. Enforceable Security olicies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[81] Claude E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[82] Igor Siveroni. Filling Out the Gaps: A Padding Algorithm for Transforming Out Timing Leaks. *Electr. Notes Theor. Comput. Sci.*, 153(2):241–257, 2006.

[83] Geoffrey Smith. Probabilistic Noninterference through Weak Probabilistic Bisimulation. In *Proc. 16th IEEE Computer Security Foundations Workshop (CSFW '03*, pages 3–13. IEEE Computer Society, 2003.

[84] Geoffrey Smith and Dennis Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Proc. 25th ACM Symposium on Principles of Programming Languages (POPL '98)*, pages 355–364. ACM, 1998.

[85] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A Formal Practice-Oriented Model for the Analysis of Side-Channel Attacks. Cryptology ePrint Archive, Report 2006/139, 2006. http://eprint.iacr.org/.

[86] François-Xavier Standaert, Eric Peeters, Cédric Archambeau, and Jean-Jacques Quisquater. Towards Security Limits in Side-Channel Attacks. In *Proc. 8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES '06)*, LNCS 4249, pages 30–45, 2006.

[87] Terkel Tolstrup. *Language-based Security for VHDL.* PhD thesis, Technical University of Denmark, 2007.

[88] Terkel Tolstrup and Flemming Nielson. Analyzing for Absence of Timing Leaks in VHDL, 2006. 6th International Workshop on Issues in the Theory of Security (WITS '06)". No formal proceedings.

[89] Terkel Tolstrup, Flemming Nielson, and H. Nielson. Information Flow Analysis for VHDL. In *Proc 8th International Conference on Parallel Computing Technologies (PaCT '05)*, LNCS 3606, pages 79–98. Springer, 2005.

[90] Hiroshi Unno, Naoki Kobayashi, and Akinori Yonezawa. Combining Type-Based Analysis and Model Checking for Finding Counterexamples Against Non-interference. In *Proc. Workshop on Programming Languages and Analysis for Security (PLAS '06)*, pages 17–26. ACM, 2006.

[91] Dennis Volpano. Safety Versus Secrecy. In *Proc. 6th International Symposium on Static Analysis (SAS '99)*, LNCS 1694, pages 303–311. Springer, 1999.

[92] Dennis Volpano and Geoffrey Smith. Probabilistic Noninterference in a Concurrent Language. In *Proc. 11th IEEE Computer Security Foundations Workshop (CSFW '98)*, pages 34–43. IEEE Computer Society, 1998.

[93] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[94] Christian Waldvogel and James L. Massey. The Probability Distribution of the Diffie-Hellman Key. In *Proc. Workshop on the Theory and Application of Cryptographic Techniques (ASIACRYPT '92)*, LNCS 718, pages 492–504. Springer, 1992.

[95] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

[96] J. Todd Wittbold and Dale M. Johnson. Information Flow in Nondeterministic Systems. In *Proc. IEEE Symposium on Security and Privacy (S& P '90)*, pages 144–161. IEEE Computer Society, 1990.

[97] Steve Zdancewic. Challenges for Information-flow Security. First International Workshop on Programming Language Interference and Dependence (PLID '04). No formal proceedings, 2004.

Curriculum Vitae

Personal Information

- born on December 5, 1976 (Immenstadt i. Allgäu)
- citizen of Germany

Academic Record

• ETH Zurich, Switzerland Doctoral studies in computer science	March 2003 – November 2007
• University of Konstanz, Germany M.Sc. studies in mathematics	October 1997 – December 2002
• Universidad de Chile, Santiago Exchange semester	August 1999 – March 2000
• Universidade de Campinas, Brazil Exchange semester	March 2000 – July 2000

Relevant Experience

• Assistant in computer science, ETH Zurich	2003 – 2007
• Student assistant in computer science, University of Konstanz	2000 – 2001
• Software engineer, Compal GmbH, Allensbach	1998

Merit-based Scholarships

• Studienstiftung des Deutschen Volkes	2000 – 2002
• Deutscher Akademischer Austauschdienst	1999 – 2000