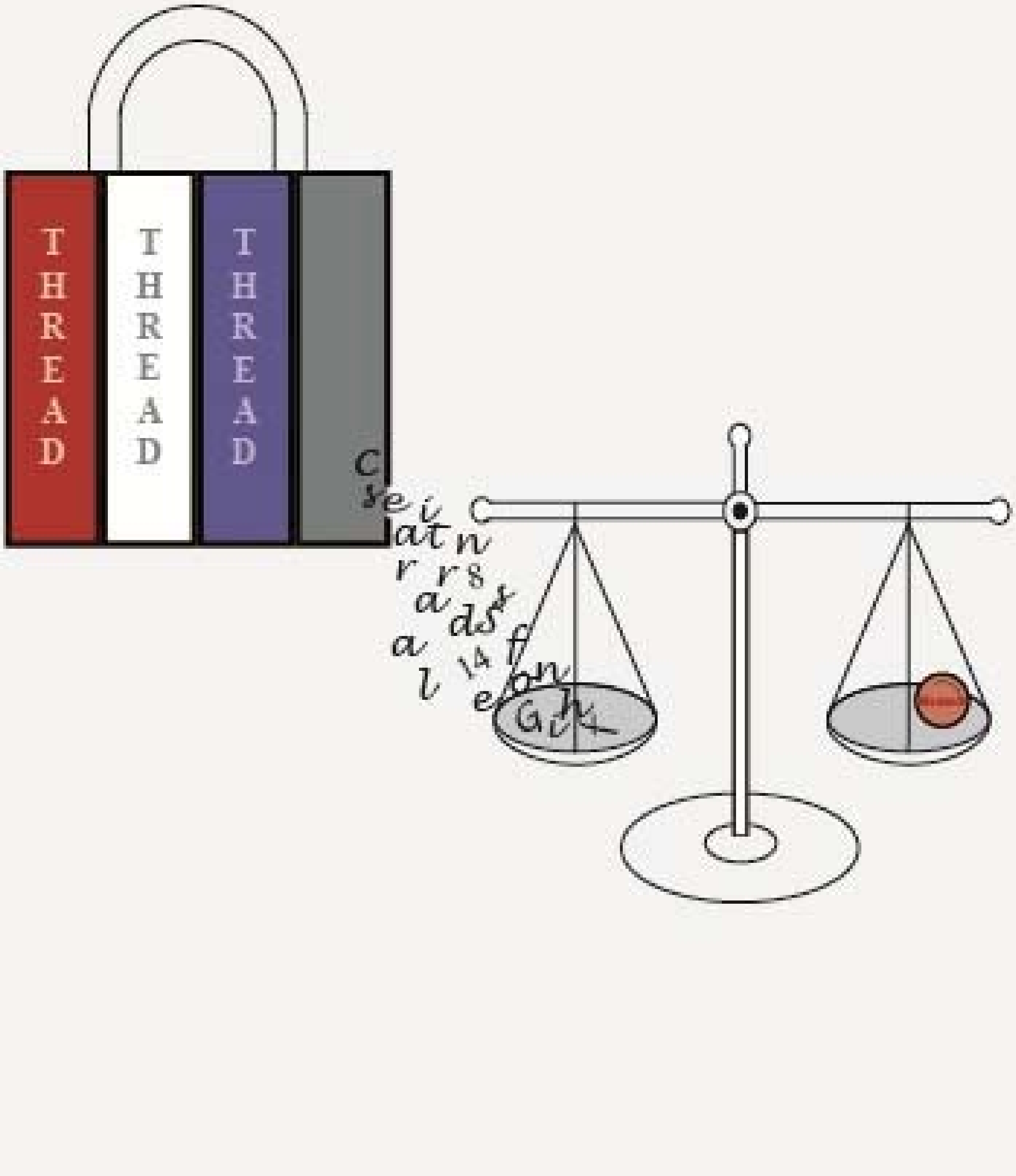
Ngô Minh Trí



Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs

Qualitative and Quantitative Information Flow Analysis for Multi-threaded Programs

Tri Minh Ngo

Graduation committee:

Chairman:	Prof. dr. Hans Wallinga
Promotors:	Prof. dr. Jaco van de Pol
Co-promotor	Dr. Marieke Huisman

Members:	
Prof. dr. Sandro Etalle	Technische Universiteit Eindhoven
Prof. dr. Wan Fokkink	Vrije Universiteit Amsterdam
Prof. dr. ir. Joost-Pieter Katoen (PDEng)	RWTH Aachen University
Prof. dr. Catuscia Palamidessi	INRIA and Ecole Polytechnique
Prof. dr. David Sands	Chalmers University of Technology

CTIT Ph.D. Thesis Series No. 14-305
Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217 – 7500 AE Enschede

IPA Dissertation Series No. 2014-05
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Netherlands Organisation for Scientific Research
The work in this thesis was supported by the SLALOM project (Security by Logic for Multithreaded applications), funded by NWO grant 612.067.802.

ISBN 978-90-365-3652-3
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 14-305)
Available online at `http://dx.doi.org/10.3990/1.9789036536523`

Typeset with LaTeX
Printed by Gildeprint
Cover design by Tri Minh Ngo

QUALITATIVE AND QUANTITATIVE INFORMATION FLOW ANALYSIS FOR MULTI-THREADED PROGRAMS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof. dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, April 17th, 2014 at 16:45 o'clock.

by

Tri Minh Ngo

born on 14 August 1982
in Danang, Vietnam

This dissertation has been approved by:

Prof. dr. Jaco van de Pol (promotor)
Dr. Marieke Huisman (co-promotor)

To my parents

Acknowledgements

Four and a half years ago, I took a flight from Vietnam to the Netherlands for a PhD position interview, and at the airport, I found that my luggage was lost. Unfortunately, it was Sunday; and all shops were closed. Thus, I came to the interview in shorts and a T-shirt of which Mark Timmer later told me that it was crappy. Nobody would offer me this position, I thought.

The bad luck did not stop. The battery of my laptop was dead, and the power adapters cable was in the luggage. I had to use Marieke's laptop to present my Master work. The interview was not so good. Jaco asked me why I chose Delft University of Technology to do my Master, and I answered that since it was the best university of technology in the Netherlands. Oh, no! it was not a good answer, and try to look smarter for the next questions, I thought. However, the luck came in the end. Therefore, first of all, I would like to thank Jaco and Marieke for offering me this position; in other words, for offering me four wonderful and unforgettable years.

There are no proper words to express my gratitude and respect for my daily supervisor, Marieke. You have taught me many things. You taught me how to give a talk. You taught me how to structure and write a paper. Whenever I have an accepted paper and if the reviewer mentions that the paper is well-written, it is because I learned the basics from you. You taught me how to be more patient in research, and not to be satisfied too soon with the early results. Nothing is perfect and everything can be improved. Besides, I also thank you for giving me the freedom to explore a variety of topics, and motivating me to become an independent researcher.

My sincere thanks must also go to my supervisor, Jaco van de Pol. I am grateful for your involvement in my thesis work. By giving useful comments and detailed corrections, you helped me to judge the skill of scientific writing, which led to significant improvements in my thesis.

During my PhD time, I do not remember how many times I came to Stefan

Blom's office to ask for his help. Stefan, I owe you a lot. You helped me to come up with some interesting ideas, and guided me through the implementation of our algorithms. I am also very grateful to Mariëlle Stoelinga. It is fun to develop algorithms to verify our confidentiality properties with you. Our collaborations resulted in two papers that are presented in Chapter 5 and Chapter 6 of this thesis. I also learned a lot from you, and you are an inspiration on how to make things as simple as possible.

I would like to thank the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) who funded my work via the SLALOM project (Security by Logic for Multi-threaded Applications).

In addition, I would also like to take this opportunity to express my appreciation to Catuscia Palamidessi and Kostas Chatzikokolakis for many fruitful discussions during my visit to your group. I was so lucky that I attended the FOSAD 2012 summer school where Catuscia gave a lecture about quantitative analysis of information flow. This lecture has had a strong impact on the direction of my research.

I am also very grateful to the members of the committee, for their time and for their valuable comments on the manuscript.

I also send special thanks to Marina Zaharieva-Stojanovski and Stefano Schivo, who have been the most loyal friends. You guys made my PhD life enjoyable and a lot easier. You are simply friends indeed. You always stay by my side, until the last moment of my PhD journey: being my paranymphs! This certainly makes me feel confident during my defense. Stefano, maybe it is now the time to say sorry for your suffering of being my office-mate in almost three years. Sorry for being bossy sometimes (please confirm that it is only *occasional* :)).

I would like to thank all my colleagues at the FMT group at the University of Twente, for creating a very nice working environment, for many social and sport events we had together. You are brilliant friends and colleagues who inspired me over these four years. Arend Rensink, thank you for creating a nice social environment in/outside the group such that we have chances to get together. Gijs Kant, you are a good companion on any social and sport event. Your carrot cake is one of my favorite cakes ever. Mark Timmer, thank you for being who you are: smart, nice, friendly and helpful (I hope these adjectives are enough to describe you, or do you want more? :)). Waheed Ahmad, thank you for joining us in the run, and wearing a suit to the Christmas dinner even when you did not want to. Thank you for cooking me a Pakistan dinner. It was a little bit spicy; but after drinking almost a full bottle of orange juice, I think I *did* enjoy the dinner. My acknowledgments are not complete if I do not mention our wonderful

Scrum team: Lesley Wevers, Saeed Darabi, Afshin Amighi, Marina, Stefan, and Wojciech Mostowski. I think I will miss our Scrum meetings for a long time. Now I have a habit to ask myself the same question every morning "what did I do yesterday?", and put an effort to make an impression that yesterday was a very productive day. I would also like to mention Tom van Dijk, Alfons Laarman, and Bugra M. Yildiz for... everything :); your names should be in my acknowledgments.

I would also like to acknowledge Joke Lammerink, not only for the administrative support during the time I worked in Enschede, but also for your kind care, and Axel Belinfante, for your help with the technical problems.

Besides, I also thank all students of the Vietnamese community in Enschede for having created warm, relaxed and friendly activities. I would also like to pay high regards to all my school and college teachers in Vietnam, especially my father — my first Mathematics teacher. You gave me the love of Mathematics and Science, and are the inspiration to my pursuit of knowledge.

I save the best for last. I dedicate this thesis to my parents and my brother, who always give me support and encouragement. You are the motivation for me to reach this far. I love you very much, and this sometimes cannot be expressed by words.

Ngo Minh Tri,
Enschede, 2014.

Abstract

In today's information-based society, guaranteeing information security plays an important role in all aspects of life: governments, military, companies, financial information systems, web-based services etc. With the existence of Internet, Google, and shared-information networks, it is easier than ever to access information. However, it is also harder than ever to protect the security of sensitive information. If an attacker can access important information, he can bring down a company or even harm people's lives. Thus, there are growing challenges of how best to keep private information processed by computing systems secure.

With the trend of multiple cores on a chip and parallel systems like general-purpose graphic processing units, applications implemented in a *multi-threaded fashion* are becoming the standard. Protecting the confidentiality of information manipulated by multi-threaded programs is an important problem, but also a challenge. Firstly, since the program execution involves the *scheduler* — to decide the ordering of executed threads — data behave in an unpredictable way; and thus, it is difficult to predict what an attacker can observe during the execution. Secondly, with the help of more powerful computing techniques, the attackers are more and more powerful, i.e., they can observe the traces of public data during the execution, and are even able to choose the scheduler to limit the set of possible program traces. Many researchers are concerned with this challenge, but most of the approaches are not sufficient, or very restrictive. The goal of this thesis is to propose more suitable and practically efficient methods to analyze information flow of multi-threaded programs.

Firstly, we formalize two *qualitative* confidentiality properties, (1) one for *non-deterministic* programs, where we do not take into account the probabilistic behavior of programs and schedulers, and (2) another one for *probabilistic* programs, where we assume to have knowledge about the probability of scheduling events. These two properties are *scheduler-specific*, i.e., if data traces of the program execution satisfy these properties, the program is guaranteed not to leak information *under the scheduler* used to deploy the program. We compare these formalizations with the existing proposals in the literature, and show that our definitions better approximate the intuitive understanding of confidentiality, which unfortunately cannot be formalized directly.

Secondly, we propose *verification methods* to verify our information flow properties, i.e., logic-based and efficient algorithmic verification methods. These methods not only give precise and efficient verifications for confidentiality properties, but also are relevant outside the security context. Our approaches have two advantages: (1) many other formalizations of confidentiality in the literature can also be verified by minor modifications of our algorithms, and (2) we can synthesize attacks for insecure programs, based on *counter-example generation techniques*. Since the verification is precise, if it fails, a counter-example can be produced, describing a possible attack on the security of the program. This idea of synthesizing attacks for information flow properties of multi-threaded programs has not been previously published in the literature. We also develop a tool which contains these techniques, and show its practical application on some case studies.

Counter-examples give us the reasons why a program fails a confidentiality requirement. However, in same cases, it is also interesting to know the *quantity* of the information flow that has been revealed. A *quantitative* security policy offers a richer security policy than the traditional *qualitative* properties, since the amount of leakage can be used to decide whether we can tolerate the *minor* leakage.

Classical quantitative information flow analysis often considers a system as an *information-theoretic channel*, where private data are the only input and public data are the output. First of all, this thesis extends this classical context by considering systems where the attacker is able to influence the initial values of public data, which should also be considered as an input of the channel. We adapt the classical view of information-theoretic channels in order to quantify information flow of programs that contain both private *and* public inputs.

Additionally, we show that our measure can also be used to reason about the case where a system operator *on purpose* adds noise to the output, instead of always producing the correct output. The noisy outcomes are used to reduce the correlation between the output and the input, and thus to increase the remaining uncertainty of the attacker about the secret. However, even though the noisy outcomes enhance the security, they reduce the reliability of the program. We show how given a certain noisy output policy, the increase in security and the decrease in reliability can be quantified.

Finally, this thesis presents a *novel model* of analysis for multi-threaded programs where the attacker is able to select the scheduling policy. This model does not follow the traditional information-theoretic channel setting. In this analysis, we first study what extra information an attacker can get if he knows the scheduler's choices, and then integrate this information into the transition system modeling the program execution. Via a case study, we compare this approach with the traditional information-theoretic models, and show that this approach gives more intuitive-matching results.

Table of Contents

Acknowledgements **vii**

Abstract **xi**

1 Introduction **1**

1.1 How to be secure . . . 1

1.2 Qualitative information flow analysis . . . 3

1.2.1 Confidentiality for multi-threaded programs . . . 4

1.2.2 Property verification and attack synthesis . . . 7

1.3 Quantitative information flow analysis . . . 8

1.3.1 Classical quantitative security analysis . . . 9

1.3.2 Quantitative security analysis for programs with low input and noisy output . . . 10

1.3.3 Quantitative security analysis for multi-threaded programs with the effect of schedulers . . . 12

1.4 Main contributions . . . 14

1.5 Organization of the thesis . . . 15

2 Preliminaries **17**

2.1 Basics . . . 17

2.2 Kripke structures . . . 18

2.3 Probabilistic Kripke structures . . . 19

2.4 Schedulers . . . 20

2.5 Stuttering-free Kripke structures and stuttering equivalence . . . 21

2.6 Probability space . . . 22

I Qualitative Information Flow Properties 25

3 Scheduler-Specific Observational Determinism 27
3.1 Introduction . . . 27
3.2 Observational determinism in the literature . . . 28
3.2.1 Existing definitions of observational determinism . . . 28
3.2.2 Shortcomings of these definitions . . . 30
3.3 Scheduler-specific observational determinism . . . 35
3.3.1 Properties of SSOD . . . 37
3.3.2 Limitations of SSOD . . . 38
3.4 Probabilistic noninterference in the literature . . . 40
3.5 Scheduler specific probabilistic observational determinism . . . 43
3.6 Scheduler-independent observational determinism . . . 45
3.7 Conclusions . . . 46

II Qualitative Verification and Attack Synthesis 47

4 Logic-based Verification 49
4.1 Introduction . . . 49
4.2 Characterization of stuttering equivalence and program model . . . 51
4.2.1 State properties . . . 52
4.2.2 Program model . . . 54
4.3 Logical characterization of stuttering equivalence and SSOD . . . 57
4.3.1 LTL and CTL . . . 57
4.3.2 Characterization of stuttering equivalence . . . 59
4.3.3 Temporal-logic characterization of SSOD . . . 61
4.4 Conclusions . . . 64

5 Algorithmic Verification 65
5.1 Introduction . . . 65
5.2 Simplified SSOD . . . 67
5.3 Verification of SSOD-1K . . . 68
5.3.1 Algorithm . . . 68
5.3.2 Overall complexity . . . 77
5.4 Verification of SSOD-2K . . . 77
5.4.1 Algorithm . . . 78
5.4.2 Overall complexity . . . 79
5.5 Verification of SSPOD-1 . . . 80

5.5.1 Algorithm 80
5.5.2 Overall correctness 85
5.5.3 Overall complexity 87
5.6 Verification of SSPOD-2 87
5.6.1 Algorithm 87
5.6.2 Overall correctness 89
5.6.3 Overall complexity 90
5.7 Related work 90
5.8 Conclusions 93

6 Attack Synthesis **95**
6.1 Introduction 95
6.2 Attack synthesis for SSOD-1K 96
6.3 Attack synthesis for SSOD-2K 101
6.4 Attack synthesis for SSPOD-1 104
6.5 Attack synthesis for SSPOD-2 104
6.6 Conclusions 105

7 Implementation and Case Studies **107**
7.1 Introduction 107
7.2 Implementation 107
7.3 Case study 1: a possibilistic model 110
7.4 Case study 2: a probabilistic model 115
7.5 Conclusions 120

III Quantitative Information Flow Analysis **121**

8 Programs with Low Input and Noisy Output **123**
8.1 Introduction 123
8.2 Basic settings for the analysis 124
8.3 Classical models of quantitative security analysis 125
8.3.1 Entropy 126
8.3.2 Quantity of information leakage 128
8.4 Shortcomings of the classical models 129
8.4.1 Counter-intuitive and conflict results 129
8.4.2 Leakage in intermediate states 131
8.5 Analytical model for programs that contain low input 132
8.5.1 Leakage of programs with low input 132

8.5.2 Case studies . . . 134
8.6 Noisy output . . . 137
8.6.1 Adding noise to the output . . . 137
8.6.2 Negative information flow . . . 138
8.7 Noisy-output policy . . . 139
8.7.1 Design a policy . . . 140
8.7.2 Example . . . 142
8.8 Related work . . . 144
8.9 Conclusions . . . 147

9 Multi-threaded Programs with the Effect of Schedulers 149
9.1 Introduction . . . 149
9.2 Program model . . . 151
9.3 Leakage of a program trace . . . 151
9.4 Leakage of a multi-threaded program . . . 153
9.5 A case study . . . 154
9.5.1 Comparison . . . 156
9.6 Technique for computing leakage . . . 157
9.7 Related work . . . 157
9.8 Conclusions . . . 158

10 Conclusions 161
10.1 Thesis summary . . . 161
10.2 Future work . . . 163

List of papers by the author 166

Bibliography 167

Samenvatting 179

Chapter 1

Introduction

1.1 How to be secure

We are living in an information-based society, where information is an important strategic resource. Thus, guaranteeing information security plays a crucial role in every aspect of life. Governments, military, companies, financial information systems, as well as web-based services, e.g., mail, shopping, and business-to-business transactions, all want to keep a good deal of information secret. For example, companies should protect salary information of their employees, or business plans, or any other information that gives them a competitive edge. Web-based services need to protect their customers' personal information when they perform on-line functions such as banking, shopping, or social networking. When more and more sensitive information is stored, processed electronically, and transmitted across networks, the risks of unauthorized access increases. If important information falls into the wrong hands, it can wreck lives, bring down businesses, and even commit harm. Thus, we are presented with growing challenges related to how to protect valuable private information in the best way. This thesis aims to deal with these challenges, i.e., *to keep secret information manipulated by computing systems secure.* In this thesis, unless otherwise stated, the term security refers to *confidentiality*.

Securing the data manipulated by computing systems has been a challenge in the past years. Several methods to limit the information disclosure exist today, such as *access control*, and *cryptography*. For example, companies include some form of access control to protect their files from being read or modified by unauthorized users. Web-based services protect their customers' information by limiting the places where information might appear (in databases, log files, backups, printed receipts etc.), and also by restricting access to the places where it is stored. A credit card transaction on the Internet requires the credit card number to be encrypted during transmission.

These are important and useful approaches, of course, but they have a fundamental limitation, i.e., they can prevent confidential information from being read or modified by unauthorized users, but they do not regulate the information propagation after it has been released. For example, access control prevents unauthorized file access, but is insufficient to control how the data is used afterwards. Similarly, cryptography provides the means to exchange information privately across a non-secure channel, but no guarantee about the confidentiality of private data is given after it is decrypted. Thus, neither access control nor encryption provide a *complete* solution to protect confidentiality of information systems.

To ensure confidentiality for an information system, it is necessary to show that the system *as a whole* enforces a confidentiality policy, i.e., by analyzing how information flows within the system. The analysis must show that information controlled by a confidentiality policy cannot flow to a location where that policy is violated. Thus, the confidentiality policy we wish to enforce is an *information flow policy*, and the method that enforces them is an *information flow analysis*. An information flow policy is a standard way to apply the principle of *end-to-end* design to the specification of computer security requirements. Therefore, we expect the guaranteed security specification to be an *end-to-end* security policy, i.e., not only preventing unauthorized access to information, but also tracking how information flows during program executions [78].

Basically, information flow analysis is to track and regulate the information flow of a system during its execution to prevent the flow of private information to unauthorized users/attackers. If the program passes the analysis, then the system's execution does not contain insecure information flow. The analysis can be done either *dynamically*, e.g., by runtime monitoring or test execution, or *statically*, e.g., by data-flow analysis or model checking. Dynamic analysis marks data with labels describing their security levels, and then propagates those labels to all derivatives of the data to check whether a violation occurs, while static analysis analyzes the source code of the program that processes the data to determine whether it respects the information flow policy.

However, dynamic analysis cannot be precise, since confidentiality is a property concerning all possible execution paths, while dynamic analysis only has information about the single *current* execution [78, 77]. The static approach is a *more promising* way of enforcing information flow policies, since it considers all possible data traces; and thus, it can control information flow with high precision [78]. This thesis follows the static approach, which can be classified into *qualitative* and *quantitative information flow analysis*. Qualitative information flow analysis checks whether an application leaks secret information, and quantitative analysis determines how much secret information has been leaked in case the application is rejected by qualitative analysis.

Many systems for which confidentiality is important are implemented in a multi-threaded fashion where multiple activities can be executed concurrently, e.g., web-based services, databases and operating systems. With the increasing popularity of multiple cores on a chip and massively parallel systems like general-purpose graphic processing units, multi-threading is becoming the standard. However, to guarantee confidentiality for multi-threaded programs is a challenge, since data of a multi-threaded program often behave unpredictably during the execution, and thus, it is difficult to predict what an attacker can observe. While many researchers are concerned with this challenge with various different approaches having been proposed, efficient information flow analysis techniques for multi-threaded programs are still lacking.

This thesis focuses on *static* information flow analysis for *multi-threaded programs*, i.e., to determine whether, and how much, private information has been leaked via public data.

1.2 Qualitative information flow analysis

Information flow is the flow of information from one variable to another variable in a program. In information flow analysis, each variable is assigned a security level. The basic model comprises two distinct levels: *low* and *high*, meaning, respectively, *publicly observable* information, and *private* information. Qualitative information flow analysis prohibits any information flow from a high security level to a low security level[1]. For example, the following program,

$$\texttt{if}\ (S > \texttt{0})\ \texttt{then}\ O := \texttt{0}\ \texttt{else}\ O := \texttt{1},$$

where S is a private variable and O is a public variable, is rejected by qualitative security properties, since we can learn information about S from the value of O.

[1]This model can be generalized in an obvious way, i.e., security levels can be viewed as a *lattice* with information flowing only *upwards* in the lattice.

Many applications such as Internet banking, e-commerce, and medical information systems etc. need to enforce strict protection of private data, e.g., credit card details, medical records etc. The success of these applications depends for a large part on the confidentiality guarantees that can be given to clients. If private data is not absolutely protected, users refuse to use such applications. Thus, it is necessary that these applications satisfy qualitative confidentiality properties, i.e., private information cannot be derivable from public data. Using formal means to establish confidentiality is a promising way to gain users' trust. Of course, there are many challenges related to this.

1.2.1 Confidentiality for multi-threaded programs

Different notions of qualitative confidentiality properties are proposed in the literature. Many researchers are concerned with defining and refining variations of *noninterference* — a fundamental qualitative confidentiality property that is often used for *sequential* programs [38, 92]. Noninterference states that a program is considered secure if the set of possible *final values of public variables* are independent of the *initial values of private variables* [92, 83][2]. An open challenge is to establish a suitable *formalization* of confidentiality for multi-threaded programs, since noninterference is *not appropriate* for a *multi-threaded* setting. This is due to two reasons. First of all, due to the exchange of intermediate results during the execution of a multi-threaded program, we have to take into account the leakage in intermediate states. Consider the following multi-threaded program, where $S \in H$ (set of high variables) and $O \in L$ (set of low variables).

Example 1.1

$$\begin{array}{l} O := \mathtt{0};\\ \big(\{\mathtt{if}\,(O = \mathtt{1})\ \mathtt{then}\,(O := S)\ \mathtt{else\ skip}\} \parallel O := \mathtt{1};\big)\\ O := \mathtt{1}; \end{array}$$

From now on, for notational convenience, let C_1 and C_2 denote the left and right operands of the parallel composition operator $\parallel$. Executing this program, we obtain the following traces $T_{|_O}$ of O, depending on which thread is picked first.

[2]There exist many definitions of noninterference. We refer to the definition of Volpano and Smith [92].

$$T_{|O} = \begin{cases} [0,1,1] & \text{execute } C_1 \text{ first} \\ [0,1,S,1] & \text{execute } C_2 \text{ first} \end{cases}$$

According to the definition of noninterference, this program is secure, since the final value of O is independent of the initial value of S. However, this program leaks the entire secret, since the attacker can access S via an intermediate state on the public data trace when C_2 is executed first.

Thus, the definition of noninterference which considers only leaks in final states is not appropriate to ensure confidentiality for multi-threaded programs. Instead, for multi-threaded programs, we have to require that private data are never revealed throughout the whole *execution traces*, i.e., the sequences of states that occur during the program execution [96, 80].

Secondly, because of the interactions between threads, data traces of a multi-threaded program depend on the scheduling policy that is used to deploy the program. Thus, for multi-threaded programs, we have to consider the *refinement attack* where an attacker chooses an appropriate scheduler to refine the set of possible program traces; and thus, secret information can be revealed from this limited set of traces.

Thus, new methods have to be developed for an observational model where an attacker is able to access the program source code, observe traces of public data, and limit the set of possible program traces by selecting an appropriate scheduler. This thesis proposes two confidentiality properties for multi-threaded programs, one for *non-deterministic* programs, where we do not take into account the probabilistic behavior of programs and schedulers, and one for *probabilistic* programs, where we assume to have knowledge about the probability of scheduling events.

Non-deterministic multi-threaded programs. Different proposals exist that attempt to establish a confidentiality property for the multi-threaded setting. We follow the approach advocated by Roscoe [75] that for a multi-threaded program, not to leak information about private data, behavior that can be observed by an attacker should be deterministic, and thus, it cannot be influenced by private data. The only way information can flow from private data to public data is when public data behave differently with different private data. To capture this, the notion of *observational determinism* has been introduced. *Intuitively*, observational determinism expresses that a multi-threaded program is secure when its *publicly observable traces* are deterministic and independent of its private data. Several *formal* definitions are proposed in the literature, e.g., by [96, 48, 87], but none of them captures this intuition exactly, i.e., they accept insecure programs, since their formalizations of deterministic behavior are not precise. Besides, these definitions also claim that they are scheduler-independent, i.e, they are resistant to refinement attacks. However, this claim is not correct, i.e., with an appropriate scheduler, the attacker can derive secret information from an accepted program.

Taking into account the effect of schedulers on confidentiality, this thesis proposes a definition of *scheduler-specific observational determinism* (SSOD). Basically, a program respects SSOD if (SSOD-1) each public variable has to evolve deterministically on traces, i.e., traces of each public variable are *stuttering equivalent*, and (SSOD-2) the relative orderings of public-variable updates on traces are coincidental. SSOD is scheduler-specific, since traces model the runs of a program under a particular scheduler. When the scheduling policy changes, some traces *cannot occur*, and also, some new traces *might appear*; thus the new set of traces may not respect our conditions. Notice that this definition does not consider the probabilistic behavior of programs and scheduling policies.

Probabilistic multi-threaded programs. SSOD is a *non-deterministic* secure information flow property: it only considers the nondeterminism that is possible in an execution, but it does not consider the probability that an execution will happen. When a scheduler's behavior is *probabilistic*, some threads might be executed more often than others, which opens up the possibility of probabilistic attacks. To prevent information leakage under probabilistic attacks, several notions of probabilistic noninterference have been proposed by Volpano et al., Sabelfeld and Sands, and Smith [93, 80, 82]. However, these definitions have limitations, i.e., they accept leaky programs, while rejecting many secure ones. Therefore, this thesis also introduces the notion of *scheduler-specific probabilistic observational determinism* (SSPOD). This definition extends SSOD, and makes it usable in a larger context.

SSPOD formalizes a confidentiality property for multi-threaded programs executed under probabilistic schedulers. Basically, a probabilistic program respects SSPOD if (SSPOD-1) each public variable individually behaves deterministically with probability 1, and (SSPOD-2) the relative ordering of public-variable updates on traces are probabilistically coincidental.

Scheduler-independent confidentiality. Besides, we consider it very important that security of a given program is robust w.r.t. any particular scheduler used; otherwise security guarantees may be destroyed by a slight change in the scheduling policy. Therefore, this thesis also derives a definition of *scheduler-independent* observational determinism. Intuitively, considering all possible interleavings of the threads of a multi-threaded program, if all traces of all public variables behaves deterministically, the program is secure w.r.t. any scheduling policy used to deploy the program.

1.2.2 Property verification and attack synthesis

Besides formalizing confidentiality properties, this thesis also discusses how to verify them. While various, subtly different approaches to formalize multi-threaded confidentiality have been proposed, efficient verification techniques for these properties are still lacking. Classical approaches to check information flow properties are typically based on type systems: if a program can be typed, it ensures secure information flow. Type systems are efficient, and support compositional verification. However, they also have several drawbacks. First of all, they are often imprecise, and insensitive to control flow. Secondly, type systems for multi-threaded programs often aim to prevent information leakage from the thread timing behavior of executions; and thus, to achieve this goal, type systems are often very restrictive. This restrictiveness makes practical programming impossible. Finally, the extensibility of type systems is very poor: each variant of the information flow policy or each new feature added to the programming language requires a modification of the type system and its soundness proof [15].

Logic-based verification. Instead, logic-based verification approaches are more flexible, and also offer a more general mechanism to enforce a variety of information flow policies, without the need to prove soundness repeatedly. This thesis discusses a method to encode the information flow property as a temporal logic property. To do this, we implement the idea of *self-composition* — a construction where a program is composed with its copy and each program copy keeps an independent memory [34, 15]. Basically, we construct a program model that executes the program to be verified twice, in parallel with itself. This program model enables us to characterize the confidentiality property as a logical property; and thus, the information flow verification problem can be translated into a model-checking problem. This approach offers us a way to reuse existing model checkers to verify information flow properties for programs.

Algorithmic verification. Besides reusing existing verification tools, we also propose more efficient specialized algorithms to verify our information flow properties. These algorithms not only give a precise and efficient verification method for confidentiality, but also are relevant outside the security context. We would like to stress that other formalizations of observational determinism [96, 48, 87] can also be verified by minor modifications of our algorithms. In comparison to the logical verification approach, the program model in the algorithmic approach is simpler, which makes the verification of large systems more practical.

Attack synthesis. Another advantage of using model-checking techniques to verify information flow properties is that we can synthesize attacks for insecure programs, based on counter-example generation techniques. Since the verification algorithm is precise, if it fails, a counter-example can be produced, describing a possible attack on the security of the program. This thesis describes how the verification algorithms can be instrumented to produce these counter-examples. We believe that our idea of applying counter-example generation to synthesize attacks for information flow properties has not previously been mentioned in the qualitative confidentiality theory for multi-threaded programs.

We also develop a tool which contains these algorithmic techniques, and provide case studies to show the feasibility of the algorithmic approaches and the practical capability of the tool.

This thesis introduces scheduler-specific observational determinism properties for multi-threaded programs. Additionally, it proposes precise and efficient verification techniques to check whether a program satisfies these security requirements. For *rejected* programs, this thesis proposes attack-synthesis techniques that describe a possible attack on a security hole of programs.

1.3 Quantitative information flow analysis

As discussed above, qualitative information flow analysis absolutely forbids any flow of information. Thus, qualitative analysis does not distinguish between two programs ($C1$) $O := S$ and ($C2$) $O := S \mod 2$, where S is a private variable and O is a public output. Both $C1$ and $C2$ are rejected, since they reveal secret information. The qualitative confidentiality properties only tell whether a program is completely secure or not completely secure, i.e., they only make binary decisions.

Qualitative security analysis is essential for applications where private data need strict protection. However, many practical programs require the ability to *intentionally* violate qualitative information flow properties by leaking *minor* information. Such systems include password checkers (PWC), cryptographic operations etc. For instance, when an attacker tries a string to guess the password: even when the attacker makes a wrong guess, secret information has been leaked, i.e., it reveals information about what the real password is *not*. Similarly, encrypting some private data would seem to make them public. Thus, there is a flow of information from the plain-text to the cipher-text, since the cipher-text depends on the plain-text. These applications are not accepted by qualitative security properties.

Standard qualitative security policies are incapable of expressing the desired security properties for these systems. These violations necessitate a richer security policy than the traditional qualitative properties. An approach that has recently become an active research topic in the computer security community is *quantitative* information flow analysis [64, 28, 22, 62, 61, 99, 84, 9]. Basically, this approach relaxes the absolute confidentiality properties by quantifying the information flow and determining how much secret information has been leaked, i.e., expressing the amount of leakage in quantitative terms. A quantitative theory of information flow offers a method to compute *bounds* on how much information is leaked. This information can be used to decide whether we can tolerate *minor* leakage. Quantifying information flow also provides a way to judge whether one application leaks more information than another, although both are insecure. For example, a reasonable quantitative security analysis would assign a higher value of leakage to $C1$ than to $C2$, since an attacker is able to learn the entire content of S in $C1$, while $C2$ only allows him to learn one bit of S. Thus, a quantitative security policy can be seen as a *generalization* of an absolute one, since it can provide properties that go *beyond the binary output* of a qualitative approach.

1.3.1 Classical quantitative security analysis

Classical quantitative analysis models the program execution as a *channel* in the information-theoretic sense, where the secret S is the *only* input and the observable O is the output [4]. An attacker, by observing O, might be able to derive information about S. The quantitative security analysis then concerns the amount of private data that an attacker is able to learn. The analysis is based on the notion of *entropy*. The entropy of a random private variable expresses the *uncertainty* of an attacker about its value, i.e., how *difficult* it is for an attacker to discover its value. The leakage of a program is typically defined as the *difference* between the secret's *initial uncertainty*, i.e., the uncertainty of the attacker about the private data before the program execution, and the secret's *remaining uncertainty*, i.e., the uncertainty of the attacker after observing the program's public outcomes, i.e.,

$$\text{Information leakage} = \text{Initial uncertainty - Remaining uncertainty.}$$

1.3.2 Quantitative security analysis for programs with low input and noisy output

This thesis discusses how to quantitatively analyze information flow of an application where an attacker is able to influence the initial values of its public variables. For example, in *PWC*, the string an attacker tries to guess the password is the *low input*. Many real-world applications, e.g., login systems, *PWC*, or banking systems fall in this category. Making a suitable quantitative analysis of information flow for programs containing *low* input is more difficult than it might seem [30, 49]. The key point is how to model such programs, since a wrong model results in counter-intuitive quantities of information flow.

The common sense of the information-theoretic channel is that the secret is the *only* input. However, for programs where an attacker can set up the initial low values based on his knowledge about the program code and private data, the initial low values are also input of the channel. Thus, the channel that models the program now has two different kinds of inputs, i.e., the secret and the initial low values. This makes the traditional form of channel *invalid* when quantifying information flow of such programs.

To apply the traditional channel to this situation, we consider the initial low values as *parameters* of the channel. In particular, we consider all possible sets of initial low values, and for each set, we construct a channel corresponding to these low values. Each channel is seen as a *test*, i.e., the attacker sets up the low parameters to test the system. Since the attacker knows the program code, he knows which test would help him to gain the most information. Therefore, the leakage of the program with low input is defined as *the maximum leakage over all possible tests.*

To make our model of quantitative security analysis suitable for both sequential and multi-threaded programs, firstly, we consider also the leakage in intermediate states, instead of just the leakage in the final states. Basically, the output of our channel is a set of public-data traces obtained from the program execution. Secondly, we assume that the attacker cannot choose schedulers. In the next section, we discuss a different model of analysis aiming for multi-threaded programs where the attacker is able to select an appropriate scheduler to control the set of program traces. The model for multi-threaded programs does not follow the traditional information-theoretic channel setting.

A new measure for the remaining uncertainty. The existing approaches of quantitative information theory do not agree on a unique measure to quantify information flow. Past works have proposed several entropy measures to compute program's leakage. Several researchers base their analysis on *Shannon entropy* and *Rényi's min-entropy* with Smith's version of conditional min-entropy [64, 28, 22, 62, 61, 99, 84, 9]. Basically, the Shannon entropy of a random variable X is a lower bound of the *expected* number of *guesses* that are needed to determine correctly the value of X, while the min-entropy represents the measure of success to guess the value of X by just *one single try.*

However, for some scenarios, these measures are in conflict, i.e., Shannon-entropy measures judge some programs more dangerous than others, while min-entropy measures give the opposite results. Thus, the literature admits that there is *no unique* measure that is likely to suit all cases: some measures will be more appropriate for the analysis in certain *threat models* [7].

This thesis follows the *one-try guessing model*, i.e., after observing the public outcome, the attacker is allowed to guess the value of S by only one try. This threat model is suitable to many security situations, i.e., the system will trigger an alarm when an attacker makes a wrong guess. For this threat model, the most established approach to quantify and reason about information flow is based on Rényi's min-entropy with Smith's definition of conditional min-entropy [84]. However, we show that in some cases, Cachin's version of conditional min-entropy [21] might be a more reasonable measure for the notion of remaining uncertainty, i.e., it gives results that better match the intuition than Smith's version. Therefore, this thesis proposes to consider Cachin's version as a new measure for quantifying information flow. We believe that this measure has not previously been used in the theory of quantitative information flow.

Noisy-output policy. The literature argues that by observing public outcomes of the execution, the attacker gains more knowledge about private data. Thus, the observable outcomes would reduce the initial uncertainty of the attacker on the secret; and hence, the value of leakage cannot be negative. However, we show that this non-negativeness property of leakage does not always hold, for example, in case the output of the program contains noise. The idea is that to enhance the security, the system operator might *secretly* add noise to the output, i.e., instead of always producing the exact outcomes, the program might sometimes report noisy ones. The noisy-output policy makes the outcomes of program more random, and thus, it reduces the correlation between the output and the input. As a consequence, the noisy outcomes might mislead the attacker's belief about the secret, and thus, increase the final uncertainty. Therefore, the value of leakage might be negative. We believe that this property might open the door for a new understanding of what the measure of uncertainty should be.

Adding noise to the output enhances the security, but it reduces the program's *reliability*, i.e., the probability that the program produces the correct outcomes. Totally random output might achieve the best confidentiality, but these outcomes are practically useless. Thus, it is clear that a noisy-output policy should consider the balance between confidentiality and reliability.

This thesis discusses how to construct an efficient noisy-output policy such that the attacker cannot derive secret information from the public outcomes, while a certain level of reliability is still preserved. Since the policy is kept secret, i.e., we do not want the attacker to find out that the system has been modified, the policy needs to respect some properties of the system. In this way, the noisy-output policy would help to protect the system effectively, while it still preserves the program's function at the same time. To the best of our knowledge, the analysis for systems containing noisy output, and the idea of noisy-output policy have not been discussed in the literature before.

1.3.3 Quantitative security analysis for multi-threaded programs with the effect of schedulers

Since the outcomes of multi-threaded programs depend on the scheduling policy, to obtain a model of the *complete analysis* for multi-threaded programs, it is necessary to study what extra information an attacker can get if he knows the scheduler's choices. Therefore, this thesis also discusses a *novel model* of analysis for multi-threaded programs where the attacker is able to select an appropriate scheduler to control the set of program traces. In this analysis, we model the execution of a multi-threaded program under the control of a probabilistic scheduler by a probabilistic Kripke structure[3]. The probabilities

[3]Probabilistic Kripke structure can be seen as a discrete-time Markov chain.

of the transitions are given by the scheduler that is used to deploy the program. States denote the probability distributions of private data S.

Therefore, the program execution can be seen as a *distribution transformer*. During the execution, the distribution of private data transforms from the initial distribution to the final distributions over traces. The distributions of private data at the initial and the final state of a trace can be used to define the *initial* uncertainty of the attacker about the secret information, and his *final* uncertainty, after observing the public data trace, respectively. Consequently, we define the *leakage of an execution trace*, i.e., the leakage given by a sequence of publicly observable data obtained during the execution of the program, as the difference between the *initial uncertainty* and the *final uncertainty.*

We denote the initial and the final uncertainty of an attacker by Rényi's min-entropies of the initial and final distributions of private data, respectively. Notice that in this model of analysis, the notion of *final* uncertainty is slightly different from the notion of of *remaining* uncertainty in the channel-based approach. While the *remaining* uncertainty depends only on the public outcomes of the execution, our notion of final uncertainty depends on the observables along the trace, and also on the program commands (chosen by the scheduler) that result in such observables. Both notions of *initial* and *final* uncertainty are computed by the *same* notion of entropy, i.e., Rényi's min-entropy, while the notion of *remaining* uncertainty is computed by the *conditional* min-entropy.

Since the execution of a multi-threaded program always results in a set of traces, *the leakage of a program* is then defined as the *expected* value of the leakage-trace values. Via a case study, we demonstrate how the leakage of a multi-threaded programs is measured. We also compare our approach with the existing channel-based analysis models. We show that our approach gives a more accurate way to study quantitatively the security property of multi-threaded programs.

This thesis discusses how to estimate the quantity of information leakage for programs that contain low input and noisy output. It also introduces a new measure for the notion of remaining uncertainty. The analysis for multi-threaded programs that takes into account the effect of schedulers is also investigated.

1.4 Main contributions

In summary, the main contributions of this thesis to the field of information flow analysis are as follows,

- Qualitative information flow analysis:
  - We introduce the notions of scheduler-specific observational determinism that are formalizations of the secure information-flow requirements for multi-threaded programs. We show that our formalizations approximate the intuitive notion of security more precisely than the earlier definitions of observational determinism, which either accept insecure programs, or are overly restrictive. Besides, our definitions are also the only ones to consider the effect of schedulers on confidentiality.
  - We propose precise methods — logic-based and algorithmic verification techniques — to verify secure information flow properties. The verification uses a combination of new and existing algorithms. Since these properties are fundamental concepts in the theory of concurrent and distributed systems, the algorithms are also applicable in a broader situation, outside the security context.
  - The advantage of using model-checking algorithms is that they can generate counter-examples when the verification fails. We extend our algorithms for this purpose, i.e., presenting counter-examples to synthesize information leaking attacks.
  - We are implementing a tool, named LTSmin-CHECK, that contains the proposed algorithmic techniques. The feasibility of the algorithmic method and the capability of the tool are shown via practical case studies.
- Quantitative information flow analysis:
  - We discuss how to analyze quantitatively the information flow of a popular kind of programs — the ones that contain low input. For such programs, we adapt the traditional information-theoretic channel by considering the initial low values as parameters of the channel.
  - We show that the value of information flow might be negative in case the system operator adds noise to the outcomes, i.e., the noise misleads the attacker's belief about the secret, and thus, it increases

the final uncertainty. We believe that this property would change the way people often think about the measure of uncertainty.

– We propose a new measure for the notion of remaining uncertainty, based on Cachin's definition of conditional min-entropy. This new measure matches the real leakage values in many cases. This thesis also discusses how to design an efficient noisy-output policy, which generates noisy outcomes, while still guaranteeing a high overall reliability.

– We propose a novel approach for estimating the leakage of multi-threaded programs. This approach takes into account the observable data in intermediate states, and also the effect of the scheduler. We believe that this method gives us a more accurate way to study the quantitative security of multi-threaded programs. Thus, we consider this work as an important contribution in the field of quantitative security analysis for multi-threaded programs.

1.5 Organization of the thesis

This thesis consists of 10 chapters, which are basically grouped into three main parts. This introduction aside, the following is a brief summary of the contents of each chapter.

Chapter 2 provides the necessarily mathematical backgrounds for this thesis, including the definitions of Kripke structure, scheduler and stuttering equivalence.

Part 1: Qualitative Information Flow Properties

Chapter 3 discusses the limitations of existing confidentiality formalizations, and then presents two formalizations that overcomes these shortcomings: scheduler-specific observational determinism (SSOD) for non-deterministic programs and scheduler-specific probabilistic observational determinism (SSPOD) for probabilistic programs. Finally, a scheduler-independent confidentiality property is also derived.

Part 2: Qualitative Verification and Attack Synthesis

Chapter 4 shows how an information flow property can be verified by a logic-based verification method. Concretely, this chapter shows that SSOD can be characterized by a temporal logic formula, and thus, existing standard verification tools can be used to prove or disprove this property.

Chapter 5 presents an algorithmic verification approach for both SSOD and SSPOD properties.

Chapter 6 introduces an attack synthesis method for insecure programs, based on counter-example generation techniques.

Chapter 7 discusses the practical implementation of the proposed algorithms, and case studies.

Part 3: Quantitative Information Flow Analysis

Chapter 8 discusses how to quantify information flow of programs that contain public input and noise at the output.

Chapter 9 presents a quantitative security analysis model for multi-threaded programs.

Chapter 10 concludes this thesis by summarizing its contributions, and also sketches directions for future work.

Chapter 2

Preliminaries

This chapter provides concepts and notations that are used throughout the remainder of this thesis. We first give definitions of (probabilistic) Kripke structures that are used to model semantics of (probabilistic) programs. The notion of schedulers that are used to deploy programs is also introduced. Finally, we define the notion of stuttering equivalence that is used to formally define the deterministic behavior of traces.

2.1 Basics

Sequences. Let $\mathbf{X}$ be an arbitrary set. The sets of all *finite sequences*, and all *finite/infinite sequences* of elements from $\mathbf{X}$ are denoted by $\mathbf{X}^*$, and $\mathbf{X}^\omega$, respectively. The empty sequence is denoted by ε. Given a sequence $\sigma \in \mathbf{X}^*$, we denote its last element by $\mathit{last}(\sigma)$. A sequence $\rho \in \mathbf{X}^*$ is called a *prefix* of $\sigma \in \mathbf{X}^\omega$, denoted $\rho \sqsubseteq \sigma$, if there exists another sequence $\rho' \in \mathbf{X}^\omega$ such that $\rho\rho' = \sigma$.

Probability distributions. A *probability distribution* μ over a set $\mathbf{X}$ is a function $\mu \in \mathbf{X} \to [0, 1]$, such that the sum of the probabilities of all elements is 1, i.e., $\sum_{x \in \mathbf{X}} \mu(x) = 1$. If $\mathbf{X}$ is uncountable, then $\sum_{x \in \mathbf{X}} \mu(x) = 1$ implies that $\mu(x) > 0$ only for countably many $x \in \mathbf{X}$. We denote by $\mathcal{D}(\mathbf{X})$ the set of all probability distributions over $\mathbf{X}$.

The *support* of a distribution $\mu \in \mathcal{D}(\mathbf{X})$ is the set $\mathit{supp}(\mu) = \{x \in \mathbf{X} \mid \mu(x) > 0\}$ of all elements with a positive probability. For an element $x \in \mathbf{X}$, we denote by $\mathbf{1}_x$ the probability distribution that assigns probability 1 to x and 0 to all other elements. The distribution is *uniform* when it assigns equal probability to all elements.

2.2 Kripke structures

Kripke structures [57] are a standard way to model programs' semantics [41]. Basically, Kripke structures are graphs where nodes represent states of the system and edges represent transitions between states. Each state may enable several transitions, modeling different execution orders to be determined by a scheduler. State labels equip each state with *relevant information* about that state. For technical convenience, our Kripke structures label states with arbitrary-valued variables from a set Var, rather than with only Boolean-valued atomic propositions. Thus, each state c is labeled by a function (valuation) $V(c) : \mathit{Var} \to \mathit{Val}$ that assigns a value $V(c)(v) \in \mathit{Val}$ to each variable $v \in \mathit{Var}$. We assume that Var is partitioned into sets of low (public) variables L and high (private) variables H, i.e., $\mathit{Var} = L \cup H$, with $L \cap H = \emptyset$.

Definition 2.1 (Kripke structure) *A* Kripke structure *(KS) $\mathcal{A}$ is a tuple $\langle \mathcal{S}, I, \mathit{Var}, \mathit{Val}, V, \to \rangle$ consisting of (i) a set $\mathcal{S}$ of* states, *(ii) an initial state $I \in \mathcal{S}$, (iii) a finite set of variables Var, (iv) a countable set of values Val, (v) a* labeling function $V : \mathcal{S} \to (\mathit{Var} \to \mathit{Val})$, *(vi) a* transition relation $\to \subseteq \mathcal{S} \times \mathcal{S}$. *We assume that $\to$ is non-blocking, i.e., $\forall c \in \mathcal{S}.\, \exists c' \in \mathcal{S}.\, c \to c'$.*

Given a set $\mathit{Var}' \subseteq \mathit{Var}$, the *projection* $\mathcal{A}|_{\mathit{Var}'}$ of $\mathcal{A}$ on Var', restricts the labeling function V to labels in Var'. Thus, we obtain $\mathcal{A}|_{\mathit{Var}'}$ from $\mathcal{A}$ by replacing V with $V|_{\mathit{Var}'} : \mathcal{S} \to (\mathit{Var}' \to \mathit{Val})$.

Semantics of programs. A program C over a variable set Var can be expressed as a KS $\mathcal{A}^C$ in a standard way: The states of $\mathcal{A}^C$ are tuples $\langle C, s \rangle$ consisting of a program fragment C and a valuation $s : \mathit{Var} \to \mathit{Val}$. The transition relation $\to$ follows the small-step semantics of C. If a program terminates in a state c, we include a special transition $c \to c$, i.e., a self-loop, ensuring that $\mathcal{A}^C$ is non-blocking. In the remainder of this thesis, we leave out the superscript C whenever this is clear from the context.

Paths and traces. A *path* π in an arbitrary KS $\mathcal{A}$ is an infinite sequence $\pi = c_0 c_1 c_2 \ldots$ such that (i) $c_i \in \mathcal{S}, c_0 = I$, and (ii) for all $i \in \mathbb{N}$, $c_i \to c_{i+1}$. We define $\mathit{Path}(\mathcal{A})$ as the set of all *infinite* paths of $\mathcal{A}$; and $\mathit{Path}^*(\mathcal{A}) = \{\pi' \sqsubseteq \pi \mid \pi \in \mathit{Path}(\mathcal{A})\}$ as the set of all *finite* paths in $\mathit{Path}(\mathcal{A})$.

The *trace* T of a path π records the valuations along π. Formally, $T = \mathit{trace}(\pi) = V(c_0)V(c_1)V(c_2)\ldots$. Trace T is a *lasso* iff it ends in a loop, i.e., if $T = T_0 \ldots T_i(T_{i+1} \ldots T_n)^\omega$, where $(T_{i+1} \ldots T_n)^\omega$ denotes a loop. We write $c \Downarrow T$ iff c is the start state of T.

Let $\mathit{Trace}(\mathcal{A})$ denote the set of all *infinite* traces of $\mathcal{A}$. We use $T_{\gg i}$ to denote the *suffix* of T starting with T_i, i.e., $T_{\gg i} = T_i, T_{i+1}, T_{i+2}, \ldots$, and $T_{\ll i}$ to denote the *prefix* of T up to the index i, i.e., $T_{\ll i} = T_0, T_1, \ldots, T_i$.

Two states c and c' are *low-equivalent*, denoted $c \sim_L c'$, iff $V(c)_{|L} = V(c')_{|L}$. Over a trace T, we let $T_{|l}$ and $T_{|L}$ denote the projections of T on a low variable l and the set of low variables L, respectively.

2.3 Probabilistic Kripke structures

Probabilistic Kripke structures (PKS) can be used to model semantics of probabilistic multi-threaded programs. PKSs are like standard Kripke structures, except that each transition $c \to \mu$ leads to a probability distribution μ over the next states, i.e., the probability to end up in state c' is $\mu(c')$. Each state may enable several probabilistic transitions, modeling different execution orders to be determined by a scheduler. Our PKSs also label states with arbitrary-valued variables from a set Var.

Definition 2.2 (Probabilistic Kripke structure) *A* PKS $\mathcal{A}$ *is a tuple* $\langle \mathcal{S}, I, \mathit{Var}, \mathit{Val}, V, \to \rangle$ *consisting of (i) a set* $\mathcal{S}$ *of* states, *(ii) an initial state* $I \in \mathcal{S}$*, (iii) a finite set of variables* Var*, (iv) a countable set of values* Val*, (v) a* labeling function $V : \mathcal{S} \to (\mathit{Var} \to \mathit{Val})$*, (vi) a* transition relation $\to \subseteq \mathcal{S} \times \mathcal{D}(\mathcal{S})$. *We assume that* $\to$ *is non-blocking, i.e.,* $\forall c \in \mathcal{S}.\, \exists \mu \in \mathcal{D}(\mathcal{S}).\, c \to \mu$.

A PKS is *fully probabilistic* if each state has at most one outgoing transition, i.e., if $c \to \mu$ and $c \to \mu'$, then $\mu = \mu'$.

Semantics of probabilistic programs. A probabilistic program C over a variable set Var can be expressed as a PKS $\mathcal{A}$ in a standard way. Probabilities of transitions are assigned by the scheduler that is used to deploy the program. If a program terminates in a state c, we include a special transition $c \to \mathbf{1}_c$, ensuring that $\mathcal{A}$ is non-blocking.

Notice that we use the same notation $\mathcal{A}$ for KSs and PKSs. However, it is clear from the context that if C is non-deterministic, C is modeled as a KS $\mathcal{A}$; otherwise, $\mathcal{A}$ is a PKS.

Paths and traces. A *path* π in a PKS $\mathcal{A}$ is an infinite sequence $\pi = c_0c_1c_2\dots$ such that (i) $c_i \in \mathcal{S}, c_0 = I$, and (ii) for all $i \in \mathbb{N}$, there exists a transition $c_i \to \mu$ with $\mu(c_{i+1}) > 0$. The definition of *traces* in PKSs is the same as for KSs.

2.4 Schedulers

A multi-threaded program executes threads from the set of non-terminated threads, i.e., the live threads. During the execution, a *non-deterministic* scheduling policy repeatedly decides which thread is picked to proceed next, while a *probabilistic* scheduling policy decides with which probability the thread is selected. A scheduler is a function that implements a scheduling policy [80]. To make our security property applicable for many schedulers, we give a general definition. We allow a scheduler to use the full history of computation to make decisions.

Non-deterministic schedulers. Given a path ending in some state c, a non-deterministic scheduler δ, which determines a set of the possible successor states Q, is formally defined as follows,

Definition 2.3 (Non-deterministic scheduler) *A* non-deterministic scheduler δ *for a KS* $\mathcal{A} = \langle \mathcal{S}, I, \mathit{Var}, \mathit{Val}, V, \to \rangle$ *is a function* $\delta : \mathit{Path}^*(\mathcal{A}) \to 2^{\mathcal{S}}$*, such that, for all finite paths* $\pi \in \mathit{Path}^*(\mathcal{A})$*, if* $\delta(\pi) = Q \subseteq \mathcal{S}$ *then* $\mathit{last}(\pi)$ *can make a transition to each* $c \in Q$*.*

Probabilistic schedulers. Given a path ending in some state c, a probabilistic scheduler chooses probabilistically which of the transitions enabled in c to execute. Since each transition results in a distribution, a probabilistic scheduler returns a distribution of distributions[1].

Definition 2.4 (Probabilistic scheduler) *A* probabilistic scheduler δ *for a PKS* $\mathcal{A} = \langle \mathcal{S}, I, \mathit{Var}, \mathit{Val}, V, \to \rangle$ *is a function* $\delta : \mathit{Path}^*(\mathcal{A}) \to \mathcal{D}(\mathcal{D}(\mathcal{S}))$*, such that, for all finite paths* $\pi \in \mathit{Path}^*(\mathcal{A})$*,* $\delta(\pi)(\mu) > 0$ *implies* $\mathit{last}(\pi) \to \mu$*.*

[1]Thus, we assume a discrete probability distribution over the uncountable set $\mathcal{D}(\mathcal{S})$; only the countably many transitions occurring in $\mathcal{A}$ can be scheduled with a positive probability.

The effect of a scheduler δ on $\mathcal{A}$ can be described by $\mathcal{A}_\delta$: the set of states of $\mathcal{A}_\delta$ is obtained by unrolling the paths in $\mathcal{A}$, i.e., $\mathcal{S}_{\mathcal{A}_\delta} = Path^*(\mathcal{A})$, such that states of $\mathcal{A}_\delta$ contain a full history of execution. Besides, the unreachable states of $\mathcal{A}$ under the scheduler δ are removed by the transition relation $\rightarrow_\delta$. These terms are formally defined as follows.

Definition 2.5 *Given $\mathcal{A} = \langle \mathcal{S}, I, Var, Val, V, \rightarrow \rangle$, and let δ be a scheduler for $\mathcal{A}$.*

For the non-deterministic scenario*, the Kripke structure associated to δ is $\mathcal{A}_\delta = \langle Path^*(\mathcal{A}), I, Var, Val, V_\delta, \rightarrow_\delta \rangle$, where $V_\delta : Path^*(\mathcal{A}) \times Var \rightarrow Val$ is given by $V_\delta(\pi) = V(last(\pi))$, and the transition relation is given by $\pi \rightarrow_\delta \pi c$ iff $c \in \delta(\pi)$, i.e., $\mathcal{A}_\delta$ can transition from a path π to a path πc if δ enables scheduling state c after π.*

For the probabilistic scenario*, the probabilistic Kripke structure associated to δ is $\mathcal{A}_\delta = \langle Path^*(\mathcal{A}), I, Var, Val, V_\delta, \rightarrow_\delta \rangle$, where $V_\delta : Path^*(\mathcal{A}) \times Var \rightarrow Val$ is given by $V_\delta(\pi) = V(last(\pi))$, and the transition relation is given by $\pi \rightarrow_\delta \mu$ iff $\mu(\pi c) = \sum_{\nu \in supp(\delta(\pi))} \delta(\pi)(\nu) \cdot \nu(c)$ for all π, c.*

In the probabilistic scenario, since all nondeterministic choices in $\mathcal{A}$ have been resolved by δ, $\mathcal{A}_\delta$ is fully probabilistic, and can be considered as a Markov chain. The probability $P(\pi)$ given to a finite path $\pi = \pi_0\pi_1 \ldots \pi_n$ is determined by $\delta(\pi_0)(\pi_1) \cdot \delta(\pi_0\pi_1)(\pi_2) \cdots \delta(\pi_0\pi_1 \ldots \pi_{n-1})(\pi_n)$. The probability of a finite trace T is obtained by adding the probabilities of all paths associated with T.

2.5 Stuttering-free Kripke structures and stuttering equivalence

Stuttering steps and *stuttering equivalence* [73] are the basic ingredients of our confidentiality properties.

Definition 2.6 (Stuttering-free KS) *A* stuttering step *is a transition $c \rightarrow c'$ that leaves the labels unchanged, i.e., $V(c') = V(c)$. A KS is called* stuttering free*, if $c \rightarrow c'$ and $V(c) = V(c')$ imply $c = c'$ and c is a final state, i.e., stuttering steps are only allowed as self-loops in final states.*

In probabilistic scenarios, a transition stutters if, with positive probability, at least one of the reached states has the same label. Similar to a stuttering-free KS, a *stuttering-free* PKS allows only stuttering transitions as self-loops in final states.

Definition 2.7 (Stuttering-free PKS) *A* stuttering step *is a transition $c \to \mu$ with $V(c) = V(c')$ for some $c' \in supp(\mu)$. A PKS is called* stuttering-free *if for all stuttering steps $c \to \mu$, we have that $\mu = \mathbf{1}_c$, and no other transition is enabled, i.e., if $c \to \mu'$, this implies $\mu = \mu'$.*

The key ingredient in the various definitions of observational determinism is trace equivalence up to stuttering, or up to stuttering and prefixing. The formal definition of stuttering equivalence given below is based on [73, 48]. It uses the auxiliary notion of *stuttering equivalence up to indexes i and j.*

Definition 2.8 (Stuttering equivalence) *Traces T and T' are* stuttering equivalent up to i and j, *written $T \sim_{i,j} T'$, iff we can partition $T_{\ll i}$ and $T'_{\ll j}$ into n blocks such that elements in the p^{th} block of $T_{\ll i}$ are equal to each other and also equal to elements in the p^{th} block of $T'_{\ll j}$ (for all $p \leq n$). Corresponding blocks may have different lengths.*

Formally, $T \sim_{i,j} T'$ iff there are sequences $0 = k_0 < k_1 < k_2 < \ldots < k_n = i + 1$ and $0 = g_0 < g_1 < g_2 < \ldots < g_n = j + 1$ such that for each $0 \leq p < n$, and for any $k_p \leq v < k_{p+1}$ and $g_p \leq w < g_{p+1}$, $T_v = T'_w$ holds.

T and T' are stuttering equivalent*, denoted $T \sim T'$, iff $\forall i.\ \exists j.\ T \sim_{i,j} T' \wedge \forall j.\ \exists i.\ T \sim_{i,j} T'$.*

Basically, two sequences are stuttering equivalent if they are the same after we remove adjacent occurrences of the same label, e.g., $(\mathbf{aaabcccd})^\omega$ and $(\mathbf{abbcddd})^\omega$. Stuttering-equivalence defines an equivalence relation, i.e., it is reflexive, symmetric and transitive [73, 48].

A set X is *closed under stuttering equivalence* if $T \in X \wedge T \sim T'$ imply $T' \in X$.

We say that T and T' are equivalent up to stuttering and prefixing, written $T \sim_p T'$, iff T is stuttering-equivalent to a prefix of T' or vice versa, i.e., $\exists i.\, T \sim T'_{\ll i} \vee T_{\ll i} \sim T'$ [96, 48]. For example, two sequences $\mathbf{aaabccc}(\mathbf{d})^\omega$ and $\mathbf{abbcddd}(\mathbf{e})^\omega$ are equivalent up to stuttering and prefixing.

2.6 Probability space

A probability space $(\Omega, \mathcal{F}, \mathbf{P})$ is defined in a standard way [88], where Ω is the *sample space*, $\mathcal{F}$ is a set of *events*, and $\mathbf{P}$ is the unique measure on $\mathcal{F}$ [86]. The sample space Ω is a set of all possible outcomes of the probabilistic experiment. Events are defined as sets of outcomes, i.e., subsets of the sample space; thus, $\mathcal{F}$ is a set of all events to which probabilities are assigned by the probability measure $\mathbf{P}$. Formally, a probability space is a tuple $(\Omega, \mathcal{F}, \mathbf{P})$, where

1. $\Omega \neq \emptyset$ is the sample space
2. $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is the set of events, such that
   - $\Omega \in \mathcal{F}$
   - $E \in \mathcal{F}$ implies $\Omega \setminus E \in \mathcal{F}$
   - $E_i \in \mathcal{F}$ for $i = 1, 2, \dots$ implies $\bigcup_{i=1}^{\infty} E_i \in \mathcal{F}$
3. $\mathbf{P} \colon \mathcal{F} \to [0, 1]$ is the probability measure, such that
   - $\mathbf{P}(\Omega) = 1$
   - $\mathbf{P}(\bigcup_{i=1}^{\infty} E_i) = \sum_{i=1}^{\infty} \mathbf{P}(E_i)$, if $E_j \cap E_k = \emptyset$ for all $j \neq k$.

A probability space can be used to describe the behavior of a probabilistic program. The main idea is that *infinite* paths are often assigned probability 0. For example, consider the program `while` (`true`) `do` `x` := `0` $\|$ `x` := `1` under a uniform scheduler. For this example, the number of possible paths is infinite. Each path is also infinite, and has probability 0. However, the probability of a certain set of paths is *nonzero*. For instance, the probability of the set of paths that first execute $x := 0$ is $\frac{1}{2}$. Therefore, instead of assigning probabilities to individual paths, the function $\mathbf{P}$ assigns probabilities to certain sets of paths, collected in the family $\mathcal{F}$ of *measurable sets*.

Thus, given $\mathcal{A}_\delta$, we can associate a probability space $(\Omega, \mathcal{F}, \mathbf{P}_\delta)$ over its sets of traces. Following the definition, we set $\Omega = (\mathit{Var} \to \mathit{Val})^\omega$, $\mathcal{F}$ contains all sets of traces, and $\mathbf{P}_\delta : \mathcal{F} \to [0, 1]$ is a probability measure — given by the scheduler δ — on $\mathcal{F}$, i.e., given a set $\mathbf{X} \in \mathcal{F}$, $\mathbf{P}_\delta(\mathbf{X})$ is the probability that a trace inside $\mathbf{X}$ occurs.

Part I

Qualitative Information Flow Properties

Chapter 3

Scheduler-Specific Observational Determinism

3.1 Introduction

This part discusses how to obtain appropriate *formalizations* of secure information flow properties for multi-threaded programs. We start with the definition of *noninterference*, a classical approach to establish a qualitative information flow property for *sequential* programs. Noninterference claims that a program is secure if changing the *initial* values of private variables cannot vary its set of possible *final* publicly observable data [92, 83]. However, as shown in Chapter 1, the requirement of *deterministic* public data only at the *final* states is not sufficient to guarantee confidentiality for multi-threaded programs. During the execution of a multi-threaded program, threads interact with each other; and thus, the behavior of *intermediate* results should also be taken into account, as shown in Example 1.1.

Roscoe [75] was the first to state the importance of *trace determinism* to guarantee secure information flow of multi-threaded programs. Intuitively, the execution of a multi-threaded program contains no information flow from private data to public data when its *publicly observable traces* behave deterministically, i.e., independent of its private data. Many researchers have attempted to formalize this intuitive idea, and call this *observational determinism*, but none of these formalizations capture it precisely [96, 48, 87]. Formalizing an appropriate deterministic behavior for traces of multi-threaded programs is not easy. Firstly, the outcomes of multi-threaded programs depend on the scheduling policy. Thus, we have to consider *refinement attacks*, i.e., attacks where the attacker uses an appropriate scheduler to refine the set of possible program traces. Secondly, we do not want the deterministic requirement to be too restrictive, such that many practical applications would be rejected. Thirdly, if we consider probabilistic scenarios, where the attacker has knowledge about the probability of scheduling events, we need to also take into account *probabilistic attacks*.

Organization of the chapter. Section 3.2 starts this chapter with a discussion about limitations of the existing formal definitions of observational determinism. Then, Section 3.3 presents our formalization that overcomes these shortcomings. Section 3.4 extends the context to probabilistic settings, discussing the existing confidentiality properties that cope with probabilistic attacks. Section 3.5 presents our probabilistic version of observational determinism. While these two formalizations are scheduler-specific, i.e., they only guarantee security under a particular scheduler, Section 3.6 gives a formalization that is scheduler-independent. Finally, Section 3.7 concludes this chapter.

Origins of the chapter. The definition of scheduler-specific observational determinism was first published in the proceedings of the *2011 International Conference on Formal Verification of Object-Oriented Software* (FoVeOOS'11) (revised selected papers) [47] and also in a corresponding technical report (extended version) [46]. Later, this definition appeared in the *Journal of Computer Security* (JCS) (A special issue) [69]. The definition of scheduler-specific probabilistic observational determinism was introduced afterwards, in the proceedings of the *5th International Conference on Engineering Secure Software and Systems* (ESSoS'13) [68], and also in a corresponding technical report [70].

3.2 Observational determinism in the literature

3.2.1 Existing definitions of observational determinism

The idea of *observational determinism* is based on the classical confidentiality property for sequential programs, called *non-interference*. Non-interference states that a program is secure iff the final states of the execution behave deterministically w.r.t. the private data, as formally stated below.

Given a program C, and any two initial low-equivalent states I and I', i.e., $I \sim_L I'$, C is secure, if $I \Downarrow T$ and $I' \Downarrow T'$, then $\mathit{last}(T) \sim_L \mathit{last}(T')$ [38].

Since for multi-threaded programs, we should also consider the leakage in intermediate states, observational determinism generalizes non-interference by requiring that the whole traces T and T' are deterministic w.r.t. private data.

To formalize the above determinism idea, the first formalization proposed by Zdancewic and Myers [96] states that a program is observationally deterministic iff starting in any two initial low-equivalent states, any two traces of each low variable are equivalent up to *stuttering and prefixing*. In 2006, Huisman, Worah and Sunesen [48] thought this formalization was weak, i.e., it accepts insecure programs. They strengthened it by requiring that any two traces of each low variable are *stuttering equivalent* [48]. In 2008, Terauchi argued that considering traces of each low variable individually was not strong enough to reject insecure programs; thus, he proposed another variant of observational determinism, requiring that all traces should be equivalent up to *stuttering and prefixing* w.r.t. *all* low variables. The main difference between this definition and the two previous ones is that instead of dealing with each low variable separately, this one considers traces of all low variables together. However, this formalization of deterministic behavior is also not precise. The next section discusses these definitions' shortcomings in detail.

Notice that these approaches consider the program execution with *all possible interleavings* of threads, i.e., the authors implicitly assume to use a uniform scheduler to deploy multi-threaded programs. Thus, in the remainder of this thesis, we use the notation $\mathcal{A}$ to denote an execution of a multi-threaded program where all possible interleavings are considered. Otherwise, when a specific scheduler δ has been used to deploy the program, we use the notation $\mathcal{A}_\delta$.

Given a program C, and any two initial low-equivalent states I and I', let $\mathcal{A}$ and $\mathcal{A}'$ denote two Kripke structures that model the executions of C from I and I', respectively. Formally, a program C is *observationally deterministic*, according to

- Zdancewic and Myers [96]: iff any two traces of each low variable are stuttering and prefixing equivalent, i.e.,

$$\forall T \in \mathit{Trace}(\mathcal{A}),\ T' \in \mathit{Trace}(\mathcal{A}'), l \in L.\ T_{|_l} \sim_p T'_{|_l}.$$

- Huisman et al. [48]: iff any two traces of each low variable are stuttering equivalent, i.e.,

$$\forall T \in \mathit{Trace}(\mathcal{A}),\ T' \in \mathit{Trace}(\mathcal{A}'), l \in L.\ T_{|_l} \sim T'_{|_l}.$$

- Terauchi [87]: iff any two traces are stuttering and prefixing equivalent w.r.t. all low variables, i.e.,

$$\forall T \in \mathit{Trace}(\mathcal{A}),\, T' \in \mathit{Trace}(\mathcal{A}').\ T\,_{|L} \sim_p T'\,_{|L}.$$

These definitions all claim that they are *scheduler-independent*. Zdancewic and Myers, followed by Terauchi, allow prefixing. This has advantage that it removes the obligation to consider program termination. In particular, Terauchi's definition is stronger than Zdancewic and Myers' definition as it requires equivalence on traces of *all* low variables instead of on traces of *each* low variable. The definition of Huisman et al. is stronger than the definition of Zdancewic and Myers, as it only allows stuttering equivalence.

3.2.2 Shortcomings of these definitions

Unfortunately, all these definitions have shortcomings. Let us first briefly mention them, before illustrating them all by examples. Zdancewic and Myers argue that prefixing is a sufficiently strong requirement, as this only causes external termination leaks of one bit of information [96]. However, Huisman et al. show that the termination leaks might reveal more than just one bit of information [48]. To avoid the termination channel, Huisman et al. require stuttering equivalence between traces, instead of stuttering and prefixing equivalence as in [96]. Further, as observed by Terauchi, an attacker might derive secret information by observing the relative ordering of low-variable updates [87]. Thus, it is not sufficient to require that only each low variable individually behaves deterministically for a program to be secure. As a consequence, Terauchi requires that all traces should be stuttering and prefixing equivalent w.r.t. all low variables. However, because of allowing prefixing, Terauchi's definition still accepts leaky programs. Moreover, the requirement that traces have to agree on updates to all low variables as a whole is overly restrictive, as illustrated later.

In addition, these definitions of observational determinism claim that they are scheduler-independent. However, we show that this claim is not correct. All these definitions accept programs that behave insecurely under some specific schedulers.

All these shortcomings are illustrated below by several examples. In all examples, we assume an observational model where an attacker can access the program code, observe the traces of public data, and is able to limit the set of possible program traces by using an appropriate scheduler.

How allowing prefixing might reveal information

Consider the following program, from Huisman et al. [48].

Example 3.1 *Suppose $S \in H$ and $O1, O2 \in L$.*

```
$O1 := 0;\ O2 := 0;$
while $(S > 0)$ do $\{O1 := O1 + 1;\ S := S - 1\};$
$O2 := 1$
```

If we execute this program from several low-equivalent initial states for different values of S, we obtain the following traces. A trace is denoted by a sequence of low-value states, containing the values of low variables in order, i.e., $(O1,\ O2)$.

$$\begin{array}{l}
\text{Case } S = 1 : T_{|L} = [(0,0),(1,0),(1,1)] \\
\text{Case } S = 2 : T_{|L} = [(0,0),(1,0),(2,0),(2,1)] \\
\text{Case } S = 3 : T_{|L} = [(0,0),(1,0),(2,0),(3,0),(3,1)] \\
\text{Case } S = 4 : T_{|L} = [(0,0),(1,0),\cdots,(4,0),(4,1)] \\
\ldots
\end{array}$$

Since traces of each low variable are stuttering and prefixing equivalent, this program is observationally deterministic according to Zdancewic and Myers. However, this program is *insecure*, because the final value of $O1$ reveals the initial value of S. This illustrates that allowing prefixing can reveal serious secret information via termination leaks.

Terauchi strengthens the definition of Zdancewic and Myers by requiring that the traces need to agree on the updates to low variables as a whole, instead of just to individual one. Therefore, Example 3.1 is rejected by Terauchi. However, Terauchi's definition still accepts programs that leak partial information, because of allowing prefixing, as illustrated by the following example.

Example 3.2 *Suppose S is a Boolean, 0 or 1, and consider a uniform scheduler,*

```
$O1 := 0; O2 := 0;$
$\{$if $(O1 = 1)$ then $(O2 := S)$ else skip$\} \parallel O1 := 1$
```

$$\text{Case } S = 0 : T_{|L} = \begin{cases} [(0,0),(1,0)] & \text{execute } C_1 \text{ first} \\ [(0,0),(1,0),(1,0)] & \text{execute } C_2 \text{ first} \end{cases}$$

$$\text{Case } S = 1 : T_{|L} = \begin{cases} [(0,0),(1,0)] & \text{execute } C_1 \text{ first} \\ [(0,0),(1,0),(1,1)] & \text{execute } C_2 \text{ first} \end{cases}$$

According to Zdancewic and Myers, and Terauchi, this program is observationally deterministic. However, when $S = 1$, we can terminate in a state where $O2 = 1$. Thus, if the value of $O2$ changes, an attacker can conclude surely that $S = 1$; thus partial information is still leaked because of prefixing.

Zdancewic and Myers, and Terauchi allow prefixing, since they consider that termination can only reveal one bit of information, and technically, prefixing simplifies the type systems that verify the property. However, we think that prefixing might cause serious leakage, as shown in Example 3.1, and therefore, only stuttering equivalence should be allowed.

How the relative ordering of updates might reveal information

Zdancewic and Myers (followed by Huisman et al.) consider traces of each low variable *independently*. Thus, this cannot avoid attacks based on the observation of the relative ordering of low-variable updates. Consider the following program, from Terauchi [87].

Example 3.3

```
O1 := 0; O2 := 0;
if (S > 0)  then {O1 := 1; O2 := 1}
            else {O2 := 1; O1 := 1}
```

If we execute this program, we get traces of the following shapes.

$$\begin{array}{ll} \text{Case } S > 0: & T_{|L} = [(0,0),(1,0),(1,1)] \\ \text{Case } S \leq 0: & T_{|L} = [(0,0),(0,1),(1,1)] \end{array}$$

Attackers can learn information about S by observing whether $O1$ is updated before or after $O2$. This shows that it is not sufficient to require the deterministic behavior of each low variable for a program to be secure. Terauchi solves this by requiring the determinism of traces of all low variables, but this results in an overly restrictive definition of observational determinism, as illustrated next.

How too strong conditions might reject too many programs

The restrictiveness of Terauchi's definition stems from the fact that no variation in the relative ordering of updates is allowed. This rejects many harmless programs, as follows.

Example 3.4

$$\begin{array}{l} O1 := \mathtt{0};\ O2 := \mathtt{0}; \\ O1 := \mathtt{3} \parallel O2 := \mathtt{4} \end{array}$$

$$\begin{array}{ll} C_1 \text{ is executed first:} & T_{|_L} = [(0,0),(3,0),(3,4)] \\ C_2 \text{ is executed first:} & T_{|_L} = [(0,0),(0,4),(3,4)] \end{array}$$

This program is rejected by Terauchi, since not all traces are stuttering and prefixing equivalent w.r.t. all low variables.

How scheduling policies might be exploited by attackers

The authors of [96, 48, 87] do not consider the effect of schedulers on the execution of multi-threaded programs, i.e., all possible interleavings of threads are enabled. However, the security of a multi-threaded program depends strongly on the scheduler's behavior, since it decides how the threads are deployed. In practice, the scheduler may vary from execution to execution. Under a specific scheduler, some traces *cannot occur*. Since we assume that an attacker knows the program's source code, if he uses an appropriate scheduler, secret information might be revealed from the limited set of possible traces. This sort of attack is called refinement attack [78, 16], since the choice of scheduling policy refines the set of possible program traces.

The existing definitions of observational determinism claim that they are scheduler-independent. However, this claim is not correct, as illustrated by the following examples.

Example 3.5 *Consider the following program,*

$$\begin{array}{l} O := \mathtt{0}; \\ \{\{\mathtt{if}\ (S > \mathtt{0})\ \mathtt{then}\ \mathtt{sleep(n)}\};\ O := \mathtt{1}\} \parallel O := \mathtt{0} \end{array}$$

where `sleep(n)` *abbreviates* `n` *consecutive skip commands.*

Under a uniform scheduler, the initial value of S cannot be derived. However, suppose we execute this program using a *round-robin* scheduling policy, i.e., the scheduler that picks a thread, and then proceeds to run that thread for m steps, before giving control to the next thread. If $m < n$, we obtain traces of the following shapes.

$$\begin{array}{ll} \text{Case } S \leq 0: & T_{|_L} = \begin{cases} [(0),(1),(0)] & \text{execute } C_1 \text{ first} \\ [(0),(0),(1)] & \text{execute } C_2 \text{ first} \end{cases} \\ \text{Case } S > 0: & T_{|_L} = \begin{cases} [(0),\ldots,(0),(1)] & \text{execute } C_1 \text{ first} \\ [(0),\ldots,(0),(1)] & \text{execute } C_2 \text{ first} \end{cases} \end{array}$$

Thus, only when $S \leq 0$, we can terminate in a state where $O = 0$. When $S > 0$, traces only terminate in a state where $O = 1$, since the round-robin scheduler does not let C_1 finish the `sleep` command before giving the turn to C_2. Thus, the final value of O might reveal whether S is positive or not. However, this program is accepted by the definitions of Zdancewic and Myers, and Terauchi; and thus, it shows that Zdancewic and Myers, and Terauchi's definitions are not scheduler-independent.

In this example, there is an encoding of a *timing leak* into an implicit flow (by using an appropriate scheduler). Hence, this attack is also called *internal observable timing* attack [96, 80, 76]. During the execution of a multi-threaded program, without access to a clock, the attacker is still able to learn information about the private data from observing the internal timing of actions. This makes this attack highly dangerous. Often a timing leak does not manifest itself when a scheduler is completely uniform, but only when a more deterministic scheduling policy is used.

Huisman et al. reject the above example, because of prefixing. However, the following example shows that the definition of Huisman et al. is also not scheduler-independent.

Example 3.6 *Consider a program consisting of 3 threads as follows,*

$$O1 := 0;\ O2 := 0;$$

$$\{\texttt{if}\ (S > 0)\ \texttt{then}\ O1 := 1\ \texttt{else}\ O2 := 1\}$$
$$\|\ \{O1 := 1; O2 := 1\}$$
$$\|\ \{O2 := 1; O1 := 1\}$$

This program is secure under a uniform scheduler, and it is accepted by the definitions of Zdancewic and Myers, and Huisman et al. An attacker can not derive secret information by observing the values of low variables, since the changes of each low variable does not depend on the high variable S. In addition, secret information cannot be derived from the observation of relative ordering of updates also, since whether $O1$ or $O2$ is updated first does not depend on the value of S.

However, when an attacker chooses a scheduler which always executes the *leftmost* thread first, he gets only two different kinds of traces: when $S > 0$, $T_{|L} = [(0,0),(1,0),(1,1),\ldots]$; otherwise, $T_{|L} = [0,0),(0,1),(1,1),\ldots]$.

According to this set of traces, this program is still accepted by the definitions of Zdancewic and Myers, and Huisman et al. but the secret information is revealed by observing whether $O1$ is updated before $O2$, i.e., when $O1$ is updated before $O2$, the attacker knows that $S > 0$.

To conclude, the examples above show that the existing definitions of observational determinism accept programs that reveal private data, since they allow equivalence up to prefixing, as in the definitions of Zdancewic and Myers, and Terauchi, or do not consider the relative ordering of updates, as in the definitions of Zdancewic and Myers, and Huisman et al. Besides, the definition of Terauchi is overly restrictive, rejecting many secure programs. In addition, all these definitions are not scheduler-independent. They accept programs behaving insecurely under a specific scheduling policy. This is our motivation to propose a new definition of observational determinism that is scheduler-specific. We would like that this definition on the one hand rejects any leaky program, but on the other hand, is less restrictive on harmless programs, in comparison to Terauchi's definition. Besides, having a confidentiality property that is parametric over the scheduler allows us to quantify which scheduler (or classes of schedulers) can be used to securely execute a program.

3.3 Scheduler-specific observational determinism

A *non-deterministic* multi-threaded program is secure w.r.t. a particular scheduler iff no secret information can be derived from the observation of public-data traces, or from the observation of ordering of public data updates. This is captured formally by the definition of *scheduler-specific observational determinism.*

As shown in [96], to be secure, a multi-threaded program must enforce a deterministic order on the accesses to a single low variable, i.e., the sequence of operations performed at a low variable is deterministic. Therefore, SSOD's first condition requires that any two traces of each low variable starting in any two initial low-equivalent states I_1 and I_2 are stuttering equivalent. This condition ensures that no secret information can be derived from public-data traces, since when each low variable individually evolves deterministically, the public values are independent of the private data. Notice that requiring only the *existence* of a single matching low-variable trace is not sufficient, as in the following example.

Example 3.7 *Consider the following program, where S is a Boolean,*

$$\begin{array}{l}\texttt{if}\ (S)\ \texttt{then}\ \{O := \texttt{0}; O := \texttt{1}\} \parallel O := \texttt{0} \\ \qquad\quad \texttt{else}\ \{O := \texttt{0}; O := \texttt{1}\} \parallel \{O := \texttt{0}; O := \texttt{0}\}\end{array}$$

This program leaks information under a uniform scheduler, since when S is 1, O is more likely to contain 1 than 0 in final states. However, there always *exists* a matching trace for O. Therefore, we require instead that *all* traces of each low variable are deterministic. This deterministic property of traces also avoids *cache attacks*, i.e., attacks that exploit the timing behavior of threads via the cache to derive secret information [96].

Notice that the transition relation of the Kripke structure is non-blocking, i.e., there is a self-loop at each final state. Thus, the attacker cannot detect termination. However, the termination leaks, which might reveal more than one bit of information [10], are avoided by requiring stuttering equivalence between traces, instead of stuttering and prefixing equivalence as in [96, 87].

SSOD also requires that, given any two initial low-equivalent states I and I', for every trace starting in I, there *exists* a trace that is stuttering equivalent w.r.t. *all* low variables, starting in I'. This existential condition on traces, which depends strongly on the scheduler used to execute the program, makes SSOD scheduler-specific. This second condition of SSOD ensures that any difference in the relative ordering of updates is coincidental, thus, no information can be deduced from it.

Formally, SSOD is defined as follows,

Definition 3.1 (SSOD) *Given a scheduler δ, a program C respects SSOD w.r.t. L and δ, iff for any two initial low-equivalent states I and I',*

SSOD-1 $\forall T \in \mathit{Trace}(\mathcal{A}_\delta),\ T' \in \mathit{Trace}(\mathcal{A}'_\delta), l \in L.\ T_{|_l} \sim T'_{|_l}$

SSOD-2 $\forall T \in \mathit{Trace}(\mathcal{A}_\delta).\ \exists T' \in \mathit{Trace}(\mathcal{A}'_\delta).\ T_{|_L} \sim T'_{|_L}$

where $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ denote two Kripke structures corresponding to I and I', respectively. Program C is scheduler-specific observational deterministic *w.r.t. a set of schedulers Δ if it is so w.r.t. any scheduler $\delta \in \Delta$.*

In contrast to other formalizations of observational determinism, our SSOD explicitly considers the effect of the scheduler that is used to execute the program. Since traces model the runs of a program under a particular scheduler, when the scheduling policy changes, some traces *cannot occur*, and also, some new traces *might appear*; thus the new set of traces may not respect our requirements.

Notice that the question which classes of schedulers appropriately model real-life attacks is orthogonal to our results: our definition is parametric on the scheduler.

Simplified SSOD. Notice that we can simplify SSOD by replacing SSOD-1 with SSOD-1A as follows. Intuitively, SSOD-1A requires that, given a Kripke structure $\mathcal{A}_\delta$ that corresponds to any initial state, after projecting on any $l \in L$, all traces are stuttering equivalent,

SSOD-1A $\forall l \in L.\ T, T' \in \mathit{Trace}(\mathcal{A}_\delta).\ T_{|_l} \sim T'_{|_l}.$

SSOD-1A and SSOD-2 are equivalent to SSOD-1 and SSOD-2.

Theorem 3.1 *If a program is* scheduler-specific observational deterministic *w.r.t. L and a scheduler δ, then*

$$\textit{SSOD-1}\ \&\ \textit{SSOD-2} \Leftrightarrow \textit{SSOD-1A}\ \&\ \textit{SSOD-2}.$$

Proof:

1. SSOD-1 & SSOD-2 $\Rightarrow$ SSOD-1A

   Given any two traces $T1$, $T2 \in \mathit{Trace}(\mathcal{A}_\delta)$, and any $T' \in \mathit{Trace}(\mathcal{A}'_\delta)$. According to SSOD-1, $\forall l \in L$, $T1_{|_l} \sim T'_{|_l} \wedge T2_{|_l} \sim T'_{|_l}$. Therefore, we can conclude that $\forall l \in L.\ T1_{|_l} \sim T2_{|_l}$.

2. SSOD-1A & SSOD-2 $\Rightarrow$ SSOD-1

   Given any traces $T \in \mathit{Trace}(\mathcal{A}_\delta)$, $T' \in \mathit{Trace}(\mathcal{A}'_\delta)$. According to SSOD-2, there exists a trace $T'' \in \mathit{Trace}(\mathcal{A}'_\delta)$ such that $T_{|_L} \sim T''_{|_L}$. If $T_{|_L} \sim T''_{|_L}$, then $\forall l \in L.\ T_{|_l} \sim T''_{|_l}$. According to SSOD-1A, $\forall l \in L$, $T'_{|_l} \sim T''_{|_l}$, then $\forall l \in L.\ T_{|_l} \sim T'_{|_l}$.

$\square$

3.3.1 Properties of SSOD

To summarize, we explicitly list different properties of SSOD.

Property 1 (Deterministic public behavior.) *If a program is accepted by SSOD, no secret information can be derived from the observable data traces. SSOD-1 requires that the low variables individually evolve deterministically, and thus, private data cannot affect their values. Besides, by not allowing prefixing on traces, SSOD also prevents the execution from leaking information via non-deterministic behavior of public data in final states, i.e., the termination channel.*

Property 2 (Deterministic relative ordering of updates) *If a program is accepted by SSOD, the relative ordering of updates is independent from private data. This is ensured by SSOD-2: there always exists a matching trace w.r.t. all low variables (for any possible low-equivalent initial state).*

Property 3 (Less restrictive on harmless programs) *Compared with Terauchi's definition, SSOD is more permissive. SSOD differs from Terauchi's definition in one important aspect: the existential condition on traces. SSOD-2 releases the requirement that all traces have to agree on the relative ordering of low-variable updates.*

Example 3.4 and 3.6, which are secure, are accepted by our definition instantiated with a uniform scheduler, but rejected by Terauchi's definition. However, SSOD is strong enough to reject Example 3.3, where the relative ordering of updates reveals information about S.

SSOD-2 is scheduler-specific. For example, the insecure Examples 3.1, 3.2, 3.3, and 3.5 are rejected by SSOD with the given scheduler. The program in Example 3.6 is secure under a uniform scheduler, and it is accepted by our definition instantiated accordingly. However, it is insecure under more deterministic schedulers that have a smaller set of possible traces. For example, if it is executed under a scheduler that always chooses the leftmost thread first, it is rejected by SSOD.

3.3.2 Limitations of SSOD

Even though the properties above illustrate that SSOD captures observational determinism well, there are still some limitations. In particular, SSOD cannot avoid *external timing attacks*, and it cannot distinguish between a nondeterministic evolution of a low variable that depends on secret data and the one that does not. Notice that these limitations also apply to other definitions of observational determinism. Besides, since we do not take into account the probabilistic behavior of programs and schedulers for SSOD, thus, *probabilistic attacks* are not relevant to this property.

External observable timing attacks. All definitions of observational determinism, including ours, do not consider externally observable timing attacks. External timing observations are those made by an observer who can time the execution with mechanisms external to the computing system. This makes the capacity of attackers too powerful to prevent an attack. Consider the following program and its execution traces.

Example 3.8

```
$O := 0;$
if $(S > 0)$  then $O := 7$
              else { sleep(100); $O := 7$ }
```

Case $S > 0 :$ $\quad T_{|_O} = [(0), (7)]$
Case $S \leq 0 :$ $\quad T_{|_O} = [(0), (0), \ldots, (0), (7)]$

This program is accepted by all definitions of observational determinism. However, if the program-execution time is considered low-observable, an attacker with a stop watch can learn information about S, by simply measuring the execution time.

In order to handle these attacks, the security property must be very restrictive, ensuring also that the execution time is independent of private data. A definition requiring *equivalence* instead of *stuttering equivalence* might handle this formally, but would impose severe restrictions on how the program is actually executed on the hardware. Therefore, we do not consider this further here.

Rejecting all non-deterministic programs. A consequence of SSOD-1 (and all other formalizations of observational determinism) is that programs that do not contain private data, but whose behavior are non-deterministic are also rejected. For example, the harmless program $O := 0 \,\|\, O := 1$ is rejected. A reason to reject this program comes from the observation that it is *impossible to distinguish* between the low traces of $O := 0 \,\|\, O := 1$, and the low traces of

```
if $(S > 0)$ then {$O := 0;$ $O := 1$} else {$O := 1;$ $O := 0$},
```

which leaks secret information. Therefore, observational determinism rejects any program that is non-deterministic in its low behavior, even when it does not contain any private data. However, as shown by the literature [96], this deterministic property of low variables is necessary to avoid cache attacks.

Notice that the program $O1 := 3 \,\|\, O2 := 4$ in Example 3.4 is considered secure, since it writes to two different locations.

Probabilistic attacks. SSOD is a *possibilistic* secure information-flow property: it only considers the non-determinism that is possible in an execution, but it does not consider the probability that an execution will happen. When the scheduler's behavior is *probabilistic*, some threads might be executed more often than others, which opens up the possibility of a probabilistic attack, as in the following example.

Example 3.9

```
O1 := 0; O2 := 0;
if (S > 0) then {O1 := 1 ∥ O2 := 1} else {O1 := 1 ∥ O2 := 1}
```

Consider a scheduler that, when $S > 0$, picks thread $O1 := 1$ first with probability 3/4; otherwise, it chooses threads with equal probabilities. With this scheduler, we can learn information about S from the probabilities of public-data traces. However, the program is still accepted by SSOD w.r.t. this scheduler, since SSOD only considers the existence of traces, not their probabilities.

To extend our proposal of the confidentiality property to a larger context, we consider also programs that have probabilistic behaviors. For such programs, to prevent information leakage under probabilistic attacks, several notions of probabilistic noninterference have been proposed [93, 80, 82]. The following sections discuss these properties, together with their limitations, and then introduce a probabilistic version of observational determinism.

3.4 Probabilistic noninterference in the literature

The first to come up with a notion of probabilistic noninterference are Volpano and Smith [93]. Their definition is based on a lock-step execution of probability distributions on states. Given any two distributions $\mu_1, \mu_2 \in \mathcal{D}(\mathcal{S})$, these distributions are indistinguishable from the low observer's point of view, denoted $\mu_1 \cong \mu_2$, iff after projecting out high variables, they are the same w.r.t. *commands* and *low variables*. Let M denote the stochastic transition matrix where the element m_{ij} of M is the probability of a transition from state i to state j. Given a distribution μ, every following distribution μ' is computed as $\mu' = \mu \cdot M$. A program C is *probabilistically secure* according to Volpano and Smith, if for any two initial distributions $\mu_1 \cong \mu_2$, we have that $\mu_1 \cdot M \cong \mu_2 \cdot M$. Thus, intuitively, given any two initial low-equivalent states, the traces of the program execution starting in these two initial states, after projecting out high variables, are exactly *the same*.

As shown by Sabelfeld and Sands, this definition is too restrictive, because of the requirement that commands should be *syntactically equal* [80]. For example, any two distributions with *syntactically different commands* will be rejected even if they are indistinguishable w.r.t. low variables and probabilities. Consider the following example (from [80], or any program with conditionals and no low variable).

Example 3.10

$$\texttt{if}\ (S > \texttt{0})\ \texttt{then}\ S := S + \texttt{1}\ \texttt{else}\ S := S - \texttt{1}$$

Intuitively, the low behavior of this program is the same w.r.t. any initial value of S, since this program does not modify O. However, consider two states where $(S = 3, O = 0)$, and $(S = -3, O = 0)$. Let μ_1 and μ_2 denote two distributions as follows,

$$\mu_1 = \{\langle \texttt{if}\ (S > \texttt{0})\ \texttt{then}\ S := S + \texttt{1}\ \texttt{else}\ S := S - \texttt{1}, (3, 0)\rangle \mapsto 1\}$$

(it says that the probability of being in state $\langle\, \texttt{if}\ (S > \texttt{0})\ \texttt{then}\ S := S + \texttt{1}\ \texttt{else}\ S := S - \texttt{1},\ (3, 0)\,\rangle$ is 1) and

$$\mu_2 = \{\langle \texttt{if}\ (S > \texttt{0})\ \texttt{then}\ S := S + \texttt{1}\ \texttt{else}\ S := S - \texttt{1}, (-3, 0)\rangle \mapsto 1\}\,.$$

Here, we have $\mu_1 \cong \mu_2$. Now, $\mu_1 \cdot M = \{\langle S := S + \texttt{1}, (3, 0)\rangle \mapsto 1\}$ and $\mu_2 \cdot M = \{\langle S := S - \texttt{1}, (-3, 0)\rangle \mapsto 1\}$. After projecting out the high variable, different sets of commands imply that $\mu_1 \cdot M \ncong \mu_2 \cdot M$. Thus, the definition of Volpano and Smith rejects this secure program.

Therefore, Sabelfeld and Sands propose another definition of probabilistic noninterference. This definition is based on a *probabilistic low-bisimulation* that reflects the equivalence on the program's probabilistic behavior visible for attackers [80]. This definition also takes into account the role of schedulers on confidentiality, i.e., it is scheduler-specific. This criterion requires that given any two initial low-equivalent states, for any trace that starts in an initial state, *there exists* a trace that starts in the other initial state, and passes through the same equivalence classes of states *at the same time*, with *the same probability.*

Aiming to be a probabilistic- and timing-sensitive security property, this definition is very restrictive w.r.t. timing, i.e., it cannot accommodate threads whose running time depends on high variables, as a harmless program of the following shape:

$O\texttt{1} := \texttt{0};\ O\texttt{2} := \texttt{0};$
$\texttt{if}\ (S\texttt{1} > \texttt{0})\ \texttt{then}\ \{O\texttt{1} := \texttt{3};\ O\texttt{1} := \texttt{3};\ O\texttt{2} := \texttt{4}\}\ \texttt{else}\ \{O\texttt{1} := \texttt{3};\ O\texttt{2} := \texttt{4}\}$

Besides, probabilistic noninterference by Sabelfeld and Sands also puts restrictions on unreachable states, e.g.,

$$O := \texttt{1};\ \texttt{if}\ (O = \texttt{0})\ \texttt{then}\ O := S\ \texttt{else skip}$$

is secure but rejected. This definition defines two program commands to be probabilistically low-bisimilar iff given *any two low-equivalent states* — including unreachable states, these two commands are indistinguishable in probabilistic low-behavior. Thus, probabilistic noninterference rejects the above program, since it considers also the case when the conditional statement is executed from an unreachable state, e.g., where O equals 0, see [19].

To overcome the limitations of the probabilistic low-bisimulation, Smith proposed to use a *weak* probabilistic bisimulation [82]. Weak probabilistic bisimulation allows two traces to be equivalent if they reach the same outcome, but one runs slower than the other. This definition does not consider unreachable states, and is more tolerant w.r.t. timing.

However, weak probabilistic bisimulation still demands that any two bisimilar states have to reach states in the same equivalence classes with the same probability. This probabilistic condition is restrictive, since even when the probabilities of the two matching traces are the same, i.e., trace occurrences do not depend on high variables, weak probabilistic bisimulation still rejects the program. This is illustrated via the following situation.

Suppose we have an observationally deterministic program whose executions from the initial state **a** are given below. This program is rejected by (weak) probabilistic bisimulation, since basically, these Kripke structures are not probabilistic bisimilar. However, no secret information can be derived from probabilistic attacks, since the probabilities of trace occurrences are identical.

$$\begin{tikzcd}
& & \mathbf{a} \arrow[dl, "\frac{1}{10}"'] \arrow[d, "\frac{1}{10}"] \arrow[dr, "\frac{1}{10}"] \arrow[drr, "\frac{7}{10}"] & & \\
& \mathbf{b} \arrow[dl] & \mathbf{b} \arrow[dl, "\frac{1}{2}"'] \arrow[dr, "\frac{1}{2}"] & \mathbf{b} \arrow[dr] & \mathbf{c} \\
\mathbf{d} & \mathbf{e} & & \mathbf{f} & \mathbf{g}
\end{tikzcd}
\qquad
\begin{tikzcd}
& & \mathbf{a} \arrow[dl, "\frac{3}{10}"'] \arrow[dr, "\frac{7}{10}"] & \\
& \mathbf{b} \arrow[dl, "\frac{1}{3}"'] \arrow[d, "\frac{1}{6}"'] \arrow[dr, "\frac{1}{6}"] \arrow[drr, "\frac{1}{3}"] & & \mathbf{c} \\
\mathbf{d} & \mathbf{e} & \mathbf{f} & \mathbf{g}
\end{tikzcd}$$

In addition, all bisimulation-based definitions mentioned above do not require the determinism of each low variable. As mentioned before, we insist that a multi-threaded program must enforce a deterministic order on the access to every low variable.

In comparison between two bisimulation-based formalizations, Sabelfeld and Sands' definition is more restrictive. However, it can be used to construct a confidentiality property that is closed under both sequential and parallel composition, while Smith's weak bisimulation (and also our formalization presented below) do not have this property.

Summarizing, all these existing definitions of probabilistic confidentiality are not satisfactory for probabilistic multi-threaded programs. Therefore, we coin the notion of *scheduler-specific probabilistic-observational determinism*. It ensures that accepted programs do not leak secret information, while being significantly less restrictive on secure programs, e.g., w.r.t. timing behavior.

3.5 Scheduler specific probabilistic observational determinism

A *probabilistic* program is secure w.r.t. a particular scheduler iff no secret information can be derived from the observation of public-data traces, from the ordering of public-data updates, and also from the probabilities of traces. This is captured *formally* by the definition of *scheduler-specific probabilistic-observational determinism* (SSPOD).

As discussed before, to be secure, a multi-threaded program must enforce a deterministic order on the accesses to a single low variable. Consider a trace that stutters forever in a *non-final* stuttering loop. Since a non-final loop must contain at least one state having a transition that goes out of the loop, this loop contains a transition with a probability less than 1. Thus, the probability of a trace to end up in a non-final stuttering loop is 0, Therefore, SSPOD-1 requires that for any initial state, traces of each low variable that do not end in a non-final stuttering loop are stuttering equivalent with probability 1. Basically, SSPOD-1 requires the determinism of each low variable in the probabilistic context.

SSPOD also requires that, given any two initial low-equivalent states I and I', for every trace starting in I, there *exists* a trace that is stuttering equivalent w.r.t. *all* low variables, starting in I', and the probabilities of these two matching traces are the same. As mentioned before, this existential condition ensures that any difference in the relative ordering of updates is coincidental. In addition, SSPOD also guarantees that no private information can be derived from the probabilistic distribution of traces, since indistinguishable traces occur with the same probabilities.

Let $(\Omega, \mathcal{F}, \mathbf{P}_\delta)$ denote the probability space of a PKS $\mathcal{A}_\delta$ with an initial state

I. If X is a set of traces that end in a non-final stuttering loop, and are closed under stuttering equivalence, $\mathbf{P}_\delta[X]$ equals 0. SSPOD is formally defined as follows.

Definition 3.2 (SSPOD) *Given a scheduler δ, a program C respects SSPOD w.r.t. L and δ, iff for any initial state I,*

SSPOD-1 *For any $l \in L$, let $X \in \mathcal{F}$ be any set of traces closed under stuttering equivalence w.r.t. l, we have $\mathbf{P}_\delta[X] = 1$ or $\mathbf{P}_\delta[X] = 0$.*

SSPOD-2 *For any initial state I' that is low-equivalent with I, for all sets of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence w.r.t. L, we have $\mathbf{P}_\delta[X] = \mathbf{P}'_\delta[X]$, where $(\Omega, \mathcal{F}, \mathbf{P}'_\delta)$ denote the probability space of $\mathcal{A}_\delta$ with initial state I'.*

Program C is scheduler-specific probabilistic-observational deterministic w.r.t. a set of schedulers Δ if it is so w.r.t. any scheduler $\delta \in \Delta$.

In addition to the properties of SSOD, SSPOD has another property,

Property 4 (Deterministic probabilistic public behavior.) *If a program is accepted by SSPOD, the probabilistic behavior of its public-data traces is independent of the private data.*

For example, Example 3.9, with the given scheduler, is rejected by SSPOD, since it reveals secret information under a probabilistic attack.

Besides, SSPOD is less restrictive than the definition of Sabelfeld and Sands [80], since it does not require the probability equality involved in each execution step. SSPOD is also less restrictive w.r.t timing, e.g., SSPOD accepts

$O1 := \mathtt{0};\ O2 := \mathtt{0};$
$\mathtt{if}\ (S1 > \mathtt{0})\ \mathtt{then}\ \{O1 := \mathtt{3};\ O1 := \mathtt{3};\ O2 := \mathtt{4}\}\ \mathtt{else}\ \{O1 := \mathtt{3};\ O2 := \mathtt{4}\}$

SSPOD also does not put restrictions on unreachable states, i.e., it accepts

$$O := \mathtt{1};\ \mathtt{if}\ (O = \mathtt{0})\ \mathtt{then}\ O := S\ \mathtt{else\ skip},$$

that is rejected by probabilistic noninterference.

Notice that, in fact, SSPOD-1 is weaker than SSOD-1. SSPOD-1 only requires stuttering equivalence on the traces that do not end in a non-final stuttering loop, while in SSOD, the property must hold on all possible traces.

3.6 Scheduler-independent observational determinism

The next question is whether there exists a scheduler and a security specification such that if a program is accepted by this security condition, it is secure under any scheduling policy. This motivates our formalization of *scheduler-independent* observational determinism.

The execution of a program under a uniform scheduler contains all possible interleavings of threads. Thus, given any scheduling policy δ, the set of possible program traces under δ is a subset of the set of program traces under a uniform scheduler. Thus, if each trace produced from one initial states under a uniform scheduler matches with every trace produced from the other initial low-equivalent state w.r.t. all low variables, then with any other scheduler, the traces of the execution satisfy both SSOD or SSPOD. For example, the program $O := 3 \parallel S := 5 \parallel$ `skip` is secure w.r.t. any scheduling policy.

Therefore, the formal definition of scheduler-independent observational determinism can be stated as follows,

Definition 3.3 (Scheduler-independent observational determinism) *A program C is* scheduler-independent observationally deterministic *w.r.t. L iff for any two initial low-equivalent states I and I', the following condition is satisfied.*

$$\forall T \in \mathit{Trace}(\mathcal{A}),\, T' \in \mathit{Trace}(\mathcal{A}').\ T\,_{|_L} \sim T'\,_{|_L}.$$

Remember that $\mathcal{A}$ denotes a program execution where all possible interleavings are considered.

The following theorem states that if a program is accepted by Definition 3.3, it is also δ-specific observationally deterministic under any other scheduler δ.

Theorem 3.2 *Given a uniform scheduler, for any two initial low-equivalent states I and I', if* all possible *traces starting in I and I' are stuttering equivalent w.r.t. all low variables, this program is secure w.r.t. any scheduling policy.*

Proof: Any scheduling policy is a refinement of a uniform scheduling policy. Consider any scheduler δ. If a program satisfies Definition 3.3, any two traces are stuttering equivalent w.r.t. all low variables. Thus, SSOD-2 and SSPOD-2 follow directly. Besides, any traces are also stuttering equivalent w.r.t. each low variable. Thus, SSOD-1A and SSPOD-1 are also respected. Therefore, the program is δ-specific observationally deterministic. $\square$

3.7 Conclusions

This chapter introduced the notions of scheduler-specific observational determinism via two formalizations, i.e., SSOD formalizes the confidentiality property for multi-threaded programs, while SSPOD is the formalization for probabilistic multi-threaded programs. These properties take into account the effect of schedulers on execution traces. Thus, given a scheduling policy δ, if a program is accepted by SSOD, or SSPOD if the program is probabilistic, instantiated for this scheduler, the program execution under the scheduler δ does not leak secret information. SSOD and SSPOD capture the intuition of the confidentiality properties for multi-threaded programs better than other formalizations of observational determinism in the literature.

This chapter also presented a scheduler-independent confidentiality property. If a program is accepted by this security requirement, it is robust w.r.t. refinement attacks, i.e., the program execution does not leak secret information under any scheduling policy.

After having presented these improved definitions of observational determinism, the next chapters discuss how these properties can be verified automatically.

Part II

Qualitative Verification and Attack Synthesis

Chapter 4

Logic-based Verification

4.1 Introduction

While various, subtly different approaches to formalize the confidentiality property for multi-threaded programs have been proposed, efficient *verification* techniques for these properties are still lacking. Classical approaches to verify information flow properties are typically based on *type systems*: if a program can be typed, it ensures secure information flow. Except Huisman et al, to check confidentiality properties for multi-threaded programs, Zdancewic and Myers, Sabelfeld et al., Terauchi, Smith [96, 80, 87, 82] use type systems. Type systems are efficient, i.e., they are often *polynomial* in the size of the program [87]. However, type-based approaches are imprecise, and insensitive to control flow. Type-based approaches are not suitable to verify our information flow properties for multi-threaded programs, for several reasons:

- Firstly, type systems for multi-threaded programs often aim to prevent information leakage from the thread timing behavior of a program, e.g., secret information should not be derivable from the observation of the internal timing of actions [96]. To achieve this goal, type systems are often very restrictive. This restrictiveness makes programming in practice very difficult; many intuitively secure programs are rejected by many type-based approaches, i.e., $S := O; O := S$.

- Secondly, it is difficult to enforce stuttering equivalence via type-based approaches without being overly restrictive [87]. Stuttering equivalence makes the problem of verification more difficult, since we have to check whether the state changes on traces match each other, forever.

- Thirdly, type systems are not very flexible, i.e., they need to be redefined and proven sound for each modification to the information security policy [15].
- Finally, type systems are often used to verify *universal* properties — properties that hold on all execution traces — or bisimulation-based properties. They are not suitable to verify existential properties as the ones in our formalizations of observational determinism.

Therefore, recent work on adopting techniques from model checking [16, 34, 48] is emerging as an alternative approach to gain better precision. This thesis develops different methods to verify our information flow properties by combining newly developed and existing model-checking algorithms. This chapter introduces a *logic-based verification* method which implements the idea of *self-composition*, while the next chapter develops *algorithmic verification* techniques.

Self-composition is a technique to transform the verification of information flow properties into a standard safety verification problem [15, 34]. The basic idea is that we compose a program C with a copy, denoted C', i.e., we execute C and C' in parallel, and consider $C \| C'$ as a single program (called a *self-composed program*). Program C' is a copy of C, but all variables are renamed to make them distinguishable from the variables in C [15, 34]. In this composed model, the two programs C and C' are still distinguished; and therefore, we can express the information flow property as a property over the execution traces of the self-composed program $C \| C'$.

Organization of the chapter. Concretely, this chapter shows that the definition of scheduler-specific observational determinism (SSOD) can be characterized by a temporal-logic formula. The essence of observational determinism is stuttering equivalence on execution traces. Thus, firstly, Section 4.2 investigates the characteristics of stuttering equivalence, and discusses which *extra information* is needed to characterize it in temporal logic. Based on the idea of self-composition and the *extra information*, Section 4.3 defines a program model, and then gives a temporal-logic formula that characterizes stuttering equivalence. SSOD is then expressed in terms of this logic characterization. This results in a conjunction of an LTL and a CTL formula. Both formulas are evaluated over a Kripke structure associated to the self-composed program. This section also shows that the validity of these formulas is equivalent to SSOD.

Thus, the characterization as a model-checking problem is sound and complete. Finally, Section 4.4 concludes the chapter. Notice that Section 5.7 will discuss related work in both logic-based and algorithmic verification approaches.

Origins of the chapter. This logic-based verification method was published in the proceedings of the *2011 International Conference on Formal Verification of Object-Oriented Software* (FoVeOOS'11) (revised selected papers) [47], and also in a corresponding technical report (extended version) [46].

4.2 Characterization of stuttering equivalence and program model

First, we look at the characteristics of stuttering equivalence. Let symbols **a**,**b**,**c**, etc. represent states in traces. Given $T \sim T'$ as follows,

index:	0	1	2	3	4	5	...
$T =$	**a**	**b**	**c**	**d**	**d**	**d**	...
number of state changes in T:	0	1	2	3	3	3	
$T' =$	**a**	**a**	**b**	**b**	**c**	**d**	...
number of state changes in T':	0	0	1	1	2	3	

The top row indicates indexes of states. The row below each trace indicates the numbers of state changes, counted from the first state, that happened in a trace. Based on this example, we make some general observations about stuttering equivalence:

- Consider a state change that occurs *first* in trace T, e.g., at index i. The first property is that this state change will also occur later in trace T', at some index $j \geq i$. For example, the state change from **b** to **c** occurs first in T at T_2, while in T', this change occurs at T'_4.

- The second property is that for any index r between such a *first* and *second* occurrence of a state change, i.e., $i \leq r < j$, the total number of state changes at state T'_r is *strictly smaller* than the total number of state changes at T_r. For example, for any r such that $2 \leq r < 4$, the number of state changes at T'_r is smaller than the number of state changes at T_r, e.g., at T'_3, the total number of state changes is 1 ($\mathbf{a} \rightarrow \mathbf{b}$) while at T_3, the total number of state changes is 3 ($\mathbf{a} \rightarrow \mathbf{b}$, $\mathbf{b} \rightarrow \mathbf{c}$, and $\mathbf{c} \rightarrow \mathbf{d}$). And similarly for $r = 2$.

- Similarly for any change that occurs *first* in trace T'.

These properties characterize exactly stuttering equivalence, and form the basis for our temporal-logic characterization later.

4.2.1 State properties

To characterize stuttering equivalence between two traces in temporal logic, we construct a *program model* based on the idea of self-composition. We compose a program C with its copy C' in parallel, and consider $C \| C'$ as a single program. Via this model, we are able to express stuttering equivalent property as a temporal-logic formula over a *composed trace* of the self-composed program. As a convention, we use T^1 and T^2 to denote the two *component traces*. Thus, the i^{th} state of the composed trace contains both T^1_i and T^2_i. The essence of stuttering equivalence is that any state change occurring in one trace also occurs in the other trace. Therefore, we extend the state with *extra information* that allows us to determine for a particular state: (1) whether the current state is different from the previous one, (2) whether a change occurs *first* or *second*, and (3) how many state changes have already happened.

State changes

To determine whether a state change occurs, we need to know the previous state. Therefore, we define a *memorizing transition relation*, remembering the previous state of each transition.

Definition 4.1 (Memorizing transition relation) *Let* $\rightarrow \subseteq (\mathcal{S} \times \mathcal{S})$ *be a* transition relation. *The memorizing transition relation* $\rightarrow_m \subseteq (\mathcal{S} \times \mathcal{S}) \times (\mathcal{S} \times \mathcal{S})$ *is defined as:* $(c, c') \rightarrow_m (d, d') \Leftrightarrow c \rightarrow d \wedge d' = c.$

Thus, (c, c') makes a memorizing transition to (d, d') if (1) c makes a transition to d in the original system, and (2) d' remembers the old state c. We use accessor functions *current* and *old* to access components of a memorized state, such that $current(c, c') = c \wedge old(c, c') = c'$. A state change can be observed by comparing the old and current components of a single state.

The order of state changes

To determine whether a state change occurs for the first time in trace or has already occurred in the other trace, we use a queue of states, denoted $\mathcal{Q}$. Its contents represents the *difference* between the two traces. We have the following operations and queries on a queue: *add*, adds an element to the end of the queue, *remove*, removes the first element of the queue, and *first*, returns the first element of the queue. In addition, we use an extra state component *lead*, which indicates which component trace added the last state in $\mathcal{Q}$, i.e., $lead = m$ ($m = 1, 2$), if the last element in $\mathcal{Q}$ was added from T^m. Initially, the queue is empty (denoted ε), and *lead* is 0.

The rules to add/remove a state to/from the queue are the following. Whenever a state change occurs for the first time in T^m, the *current* state is added to the queue and *lead* becomes m. When this change occurs later in the other trace, the element will be removed from the queue. When a state change in one trace does not match with the change in the other trace, both $\mathcal{Q}$ and *lead* become *undefined*, denoted $\perp$, indicating a *blocked* queue. If $q = \perp$ (and $lead = \perp$), two component traces are not stuttering equivalent, and thus, we do not need to check the remainders of the traces. Therefore, when $\mathcal{Q}$ and *lead* are $\perp$, operations *add* and *remove* are not defined.

Formally, these rules for adding and removing are defined as follows. Initially, $\mathcal{Q}$ is ε and *lead* is 0. Whenever $\mathcal{Q} \neq \perp$ and $T_i^m \neq T_{i-1}^m$ ($m = 1, 2$),

- if $lead = 3 - m$ and $T_i^m = first(\mathcal{Q})$, then $remove(\mathcal{Q})$. If $\mathcal{Q} = \varepsilon$, set $lead = 0$.
- if $lead = m$ or $lead = 0$, then $add(\mathcal{Q}, T_i^m)$, and set $lead = m$.
- otherwise, set $\mathcal{Q} = \perp$ and $lead = \perp$.

Example 4.1 *Consider two traces T^1 and T^2 with the same initial states, as follows.*

$T^1 =$	**a**	**a**	**b**	**c**	**d**	**d**	...
$T^2 =$	**a**	**b**	**b**	**c**	**c**	**e**	...
$\mathcal{Q}$	ε	**b**	ε	ε	**d**	$\perp$	$\perp$
index	0	2	0	0	1	$\perp$	$\perp$

Initially, $\mathcal{Q} = \varepsilon$ and $lead = 0$. A state change from **a** to **b** occurs first in T^2, at T_1^2. Thus, the current component **b** is added to $\mathcal{Q}$. Thus, $\mathcal{Q}$ would be **b** and *lead* becomes 2. In T^1, the same state change occurs at T_2^1. Since *lead* is 2, it indicates that this is not a *first-occurring* change. Since $T_2^1 = first(\mathcal{Q})$ — the same change has occurred in the other component trace — the element **b** in $\mathcal{Q}$ is removed. Thus, $\mathcal{Q}$ becomes ε and *lead* becomes 0.

At T_3^1 and T_3^2, state changes occur simultaneously in both T^1 and T^2. Assume that we consider the change in T^1 first. Because the current *index* is 0, we add $\mathbf{c}$ to $\mathcal{Q}$. Now, $\mathcal{Q}$ becomes $\mathbf{c}$, and *lead* becomes 1. Then we consider the change in T^2. Since $\mathit{lead} \neq 2$ and $T_3^2 = \mathit{first}(q) = \mathbf{c}$, we remove $\mathbf{c}$ from $\mathcal{Q}$. The updated $\mathcal{Q}$ is ε and *lead* is 0. If we had considered the change in T^2 first, we would have obtained the same result.

If the state changes in T^1 and T^2 do not match, as is the case for T_4^1 and T_5^2, both q and *leading* become $\perp$.

The number of state changes

To determine the number of state changes that have happened, we extend states with counters $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$. Initially, both $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$ are 0, and whenever a state change occurs, i.e., $T_i^m \neq T_{i-1}^m$ $(m = 1, 2)$, $\mathit{nr\_ch}^m$ is increased by one. Thus, the number of state changes at T_i^1 and T_i^2 can be determined via the values of $\mathit{nr\_ch}^1$ and $\mathit{nr\_ch}^2$, respectively.

4.2.2 Program model

Based on the Kripke structure, we define a program model over which a temporal-logic formula should hold. Given a program C and the valuations s and s' of two initial low-equivalent states, i.e., $s = V(I)$ and $s' = V(I')$, we take the parallel composition of C and its copy C'. In this model, the valuation of $C \parallel C'$ can be considered as the product of the two separate valuations s and s', ensuring that variables from the two program copies are disjoint, and thus variable updates are done locally, i.e., not affecting the memory of the other program copy. We define elements of the program model as follows.

Composed states. A state of a composed trace is of the form $(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, \langle C_3 \parallel C_4, (s_3, s_4)\rangle, \chi)$, where $\langle C_3 \parallel C_4, (s_3, s_4)\rangle$ remembers the old state (via the memorizing transition relation defined in Definition 4.1), and χ is extra information, as discussed above, of the form $(\mathit{nr\_ch}^1, \mathit{nr\_ch}^2, \mathcal{Q}, \mathit{lead})$. We define accessor functions conf_1, conf_2, and *extra* to extract $(\langle C_1, s_1\rangle, \langle C_3, s_3\rangle)$, $(\langle C_2, s_2\rangle, \langle C_4, s_4\rangle)$, and χ, respectively.

Thus, in this model, the two original program copies are distinguished, and the updates of program copies are done locally. Therefore, if $\mathcal{T}$ is a trace of the composed model, we can decompose it into two individual traces by functions Π_1 and Π_2, respectively, defined as $\Pi_m = \mathit{map}(\mathit{conf}_m)$. Thus, let $\mathcal{T}_i = (\langle C_1 \parallel C_2, (s_1, s_2)\rangle, \langle C_3 \parallel C_4, (s_3, s_4)\rangle, \chi)$ be a state of the composed trace, then $(\Pi_1(\mathcal{T}))_i = (\langle C_1, s_1\rangle, \langle C_3, s_3\rangle)$ and $(\Pi_2(\mathcal{T}))_i = (\langle C_2, s_2\rangle, \langle C_4, s_4\rangle)$.

$$\frac{lead = 2 \quad c = \mathit{first}(\mathcal{Q}) \quad nr\_ch^{1'} = nr\_ch^1 + 1 \quad \mathcal{Q}' = \mathit{remove}(\mathcal{Q}) \quad lead' = 1}{(nr\_ch^1, nr\_ch^2, \mathcal{Q}, lead) \overset{c}{\rightarrow} (nr\_ch^{1'}, nr\_ch^{2'}, \mathcal{Q}', lead')}$$

$$\frac{lead \in \{0,1\} \quad lead' = 1 \quad nr\_ch^{1'} = nr\_ch^1 + 1 \quad \mathcal{Q}' = \mathit{add}(\mathcal{Q}, c)}{(nr\_ch^1, nr\_ch^2, \mathcal{Q}, lead) \overset{c}{\rightarrow} (nr\_ch^{1'}, nr\_ch^{2'}, \mathcal{Q}', lead')}$$

$$\frac{lead \notin \{0,1\} \quad c \neq \mathit{first}(\mathcal{Q}) \quad nr\_ch^{1'} = nr\_ch^1 + 1 \quad \mathcal{Q}' = \bot \quad lead' = \bot}{(nr\_ch^1, nr\_ch^2, \mathcal{Q}, lead) \overset{c}{\rightarrow} (nr\_ch^{1'}, nr\_ch^{2'}, \mathcal{Q}', lead')}$$

Figure 4.1: Definition of $\rightarrow$

The current state of the program copy m can be extracted by a function Γ_m, defined as $\Gamma_m = \mathit{map}(\mathit{current}) \circ \Pi_m$. Thus, $(\Gamma_1(\mathcal{T}))_i = \langle C_1, s_1 \rangle$ and $(\Gamma_2(\mathcal{T}))_i = \langle C_2, s_2 \rangle$. Finally, $\mathit{extra}(\mathcal{T}_i)(x)$ denotes the value of the extra information x at $\mathcal{T}_i$, for $x \in \{nr\_ch^1, nr\_ch^2, \mathcal{Q}, lead\}$.

Transition relation. The transition relation $\rightarrow_\chi$ is defined as the composition of a relation on the operational semantics and a relation on the extra information. More precisely, the first component is the memorizing transition relation $\rightarrow_m$ (cf. Definition 4.1), derived from the transition relation induced by the operational semantics of a program executed under a scheduler δ. The second component is a relation $\rightarrow \subseteq \chi \times \mathit{Conf} \times \chi$ that describes how the extra information evolves, following the rules in Figure 4.1. Notice that $\rightarrow$ is parametric on the concrete equality relation used. Concretely, $\rightarrow_\chi$ is defined by rules such as:

$$\frac{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2) \rightarrow_m (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, \langle C_1 \parallel C_2, (s_1, s_2)\rangle) \quad \chi \overset{\langle C_1', s_1' \rangle}{\rightarrow} \chi'}{(\langle C_1 \parallel C_2, (s_1, s_2)\rangle, c_2, \chi) \rightarrow_\chi (\langle C_1' \parallel C_2, (s_1', s_2)\rangle, \langle C_1 \parallel C_2, (s_1, s_2)\rangle, \chi')}$$

Rules for when C_1 terminates, i.e., $\langle C_1, s_1 \rangle \rightarrow \langle \epsilon, s_1 \rangle$, and the symmetric counterparts for C_2 are defined similarly.

Notice that above, we studied stuttering equivalence in a generic way, where two traces could make a state change simultaneously. However, in our self-composed program model, in every step, either C or C', but not both, is able to make a transition. Therefore, for any trace $\mathcal{T}$, state changes do not happen simultaneously in both $\Pi_1(\mathcal{T})$ and $\Pi_2(\mathcal{T})$. This also means that it never happens that in one step, both *add* and *remove* are applied simultaneously on the queue.

Atomic propositions and their valuation. Next, we define *atomic propositions* of our program model, together with their valuation. Notice that their valuation is parametric on the concrete equality relation used. Below, when characterizing SSOD, we instantiate this in different ways, i.e., to define stuttering equivalence on traces of each low variable, and on traces of all low variables, respectively.

For each $m = 1, 2$, we define the following atomic propositions:

- $\mathit{first\_change}^m$ denotes that a state change occurs for the first time in the program copy m.
- $\mathit{second\_change}^m$ denotes that a state change occurs in the program copy m, while the program copy $3 - m$ has already made this change.
- $\mathit{nr\_ch}^m < \mathit{nr\_ch}^{3-m}$ denotes that the number of state changes made by the program copy m is less than the total number of state changes made by the program copy $3 - m$.

The valuation function λ for these atomic propositions is defined as follows. Let $\mathfrak{c}$ denote a state of the composed trace.

$$\begin{array}{rcl}
\mathit{first\_change}^m \in \lambda(\mathfrak{c}) & \Leftrightarrow & \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and} \\
 & & \mathit{extra}(\mathfrak{c})(\mathit{lead}) = m \text{ or } \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 0 \\
\mathit{second\_change}^m \in \lambda(\mathfrak{c}) & \Leftrightarrow & \mathit{current}(\mathit{conf}_m(\mathfrak{c})) \neq \mathit{old}(\mathit{conf}_m(\mathfrak{c})) \text{ and} \\
 & & \mathit{extra}(\mathfrak{c})(\mathit{lead}) = 3 - m \text{ and} \\
 & & \mathit{current}(\mathit{conf}_m(\mathfrak{c})) = \mathit{first}(\mathit{extra}(\mathfrak{c})(\mathcal{Q})) \\
\mathit{nr\_ch}^m < \mathit{nr\_ch}^{3-m} \in \lambda(\mathfrak{c}) & \Leftrightarrow & \mathit{extra}(\mathfrak{c})(\mathit{nr\_ch}^m) < \mathit{extra}(\mathfrak{c})(\mathit{nr\_ch}^{3-m})
\end{array}$$

Program model. Using the above definitions, we define a program model, encoding the behavior of a self-composed program under a scheduler δ. The temporal-logic characterizations will be expressed over this model.

Definition 4.2 (Program model) *Given a scheduler δ, let C be a program, and s_1 and s_2 be two low-equivalent* variable *valuations. The* program model *$\mathcal{M}^{\delta}_{C,s_1,s_2}$ is defined as $(\Sigma, \rightarrow_{\chi}, AP, \lambda, I)$ where:*

- Σ *denotes the set of all states, including the extra information,*
- $I = \{\langle C \parallel C', (s_1, s_2)\rangle\}$ *is the initial state of the composed trace.*
- *Other components have been defined before.*

Now, we state a useful property of the program model that is used below to prove correctness of the characterizations. The property states that the atomic proposition $\mathit{second\_change}^m$ only holds if in an earlier state, $\mathit{first\_change}^{3-m}$ was true. Intuitively, a state change can only be considered as a *second-time* change, if the same change has occurred before. Formally, this is expressed as follows.

Lemma 1

$$\mathcal{T}_j \models \mathit{second\_change}^m \Rightarrow \exists i.\, i < j.\ \mathcal{T}_i \models \mathit{first\_change}^{3-m}.$$

Proof: This follows directly from the construction of the queue and the valuation of the corresponding atomic propositions. □

4.3 Logical characterization of stuttering equivalence and SSOD

4.3.1 LTL and CTL

For convenience, we give a brief overview of those parts of LTL and CTL that are needed for the characterization below. For more details, we refer to [51].

LTL stands for *linear time temporal logic.* Its connectives allow us to refer to the future. It models time as a sequence of states, extending infinitely into the future. Let AP denote a set of Boolean-valued atomic propositions.

Definition 4.3 (LTL syntax) *Given an atomic proposition $a \in AP$, a formula ϕ in LTL is defined as follows,*

$$\phi ::= \ a \mid \neg\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \rightarrow \phi' \mid \mathsf{G}\phi \mid \phi\,\mathsf{U}\phi'.$$

A *state* satisfies an LTL formula ϕ if *all execution traces* starting in that given state satisfy ϕ. Thus, LTL implicitly quantifies *universally* over all traces.

The connective G is called '*Globally*', while U is called '*Until*'. The formula $\mathsf{G}\phi$ holds on a trace if ϕ holds in every state along that trace. The formula $\phi\,\mathsf{U}\phi'$ holds on a trace if (1) ϕ holds continuously *until* ϕ' holds, and (2) ϕ' holds in some future state.

The semantics of LTL is defined w.r.t. a program model $\mathcal{M}$.

Definition 4.4 (LTL semantics) *Let $\mathcal{M}$ be a program model, c a state, T an execution trace of $\mathcal{M}$, and ϕ an LTL formula. We write $\mathcal{M}_c \models \phi$, i.e., $\mathcal{M}_c$ satisfies a formula ϕ, if for every trace T of $\mathcal{M}$ starting in c, we have $T \models \phi$.*

The satisfaction relation $T \models \phi$, where ϕ is among $\{\neg\phi, \phi \wedge \phi', \phi \vee \phi', \phi \rightarrow \phi'\}$, is defined as the standard satisfaction relation between the state c and the formula, i.e., $c \models \phi$. $T \models a$ iff $a \in AP(c)$, where $AP(c)$ is the set of atomic propositions of AP that are valid in c. $T \models \mathsf{G}\,\phi$ iff $\forall i \geq 0, T_{\gg i} \models \phi$. $T \models \phi\,\mathsf{U}\,\phi'$ iff $\exists k \geq 0.\, T_{\gg k} \models \phi'$ and $\forall 0 \leq i < k, T_{\gg i} \models \phi$.

CTL is a *branching time logic*. It models time as a tree-like structure.

Definition 4.5 (CTL syntax) *Given an atomic proposition $a \in AP$, a formula ϕ in CTL is defined as follows,*

$$\begin{aligned}\phi ::= \quad & a \mid \neg\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \rightarrow \phi' \mid \\ & \mathsf{AG}\,\phi \mid \mathsf{EG}\,\phi \mid \mathsf{A}[\phi\,\mathsf{U}\,\phi'] \mid \mathsf{E}[\phi\,\mathsf{U}\,\phi'].\end{aligned}$$

CTL allows us to quantify explicitly over traces. Quantifiers A and E denote '*all traces*' and '*exists a trace*', respectively. Notice that every temporal operator (G, U) has to be associated with a unique path quantifier (A, E). Intuitively, a formula $\mathsf{E}\psi$, where $\psi = \mathsf{G}\,\phi$ or $\psi = \phi\,\mathsf{U}\,\phi'$, expresses that there exists an execution trace for which the formula ψ holds, while $\mathsf{A}\psi$ expresses that ψ holds for all traces.

The semantics of CTL is also defined w.r.t. a program model $\mathcal{M}$.

Definition 4.6 (CTL semantics) *Let $\mathcal{M}$ be a program model, c a state, T an execution path of $\mathcal{M}$, and ϕ a CTL formula. The satisfaction relation $\mathcal{M}_c \models \phi$ is defined as usual for $a, \neg\phi, \phi \wedge \phi', \phi \vee \phi', \phi \rightarrow \phi'$, i.e., the satisfaction relation between the state c and the formula: $c \models \phi$. $\mathcal{M}_c \models \mathsf{E}\psi$, where $\psi = \mathsf{G}\phi$ or $\psi = \phi\,\mathsf{U}\,\phi'$, iff $\exists T.\, c \Downarrow T \wedge T \models \psi$ (for $T \models \psi$, refer to Definition 4.4). $\mathcal{M}_c \models \mathsf{A}\psi$ iff $\forall T.\, c \Downarrow T \Rightarrow T \models \psi$.*

4.3.2 Characterization of stuttering equivalence

Based on the characteristics of stuttering equivalence and the program model above, the stuttering equivalence property between two component traces is characterized by the following LTL formula ϕ, that holds on a composed trace.

$$\phi = \mathsf{G}\Big(\bigwedge_{m\in\{1,2\}} \mathit{first\_change}^m \Rightarrow \mathit{nr\_ch}^{3-m} < \mathit{nr\_ch}^m \;\; U\; \mathit{second\_change}^{3-m}\Big).$$

This formula expresses the characteristics of stuttering equivalence: any state change occurring in one component trace will occur later in the other component trace; and in between these changes, the number of state changes at intermediate states in the latter trace is strictly smaller than in the first.

We prove formally that ϕ characterizes stuttering equivalence precisely.

Theorem 4.1 *Let $\mathcal{T}$ be a composed trace that can be decomposed into T^1 and T^2 with $T^1_0 = T^2_0$, then $T^1 \sim T^2 \Leftrightarrow \mathcal{T} \models \phi$.*

Proof: **Case 1:** $T^1 \sim T^2 \Rightarrow \mathcal{T} \models \phi$

Following the LTL semantics, we have to show the following:

$$\begin{aligned} &\forall i.\ \forall m \in \{1,2\}.\ \mathcal{T}_i \models \mathit{first\_change}^m \Rightarrow \\ &\quad \exists j.\ \ \mathcal{T}_j \models \mathit{second\_change}^{3-m} \wedge \\ &\qquad (\forall r.\ i \le r < j.\ \mathcal{T}_r \models \mathit{nr\_ch}^{3-m} < \mathit{nr\_ch}^m) \end{aligned}$$

Suppose without loss of generality that at an index i, a state change occurs *first* in trace T^1, thus $T^1_{i-1} \neq T^1_i$. Since $T^1 \sim T^2$, there exists a j such that $T^1 \sim_{i,j} T^2$.

Thus, there are two sequences $0 = k_0 < k_1 < k_2 < \ldots < k_n = i+1$ and $0 = g_0 < g_1 < g_2 < \ldots < g_n = j+1$ (for some $n \geq 0$) such that for each $0 \le p < n$:

$$T^1_{k_p} = T^1_{k_p+1} = \cdots = T^1_{k_{p+1}-1} = T^2_{g_p} = T^2_{g_p+1} = \cdots = T^2_{g_{p+1}-1}.$$

If we partition $T^1{}_{\ll i}$ and $T^2{}_{\ll j}$ in such a way that states in two adjacent blocks are different, then it is easy to see that the total number of state changes at states in the p^{th} block is p.

Since $T^1_{i-1} \neq T^1_i$, then $k_{n-1} = i$. Thus, $T^1_i = T^1_{k_{n-1}} = T^2_{g_{n-1}}$, and also $T^1_{i-1} = T^1_{k_{n-1}-1} = T^2_{g_{n-1}-1}$. Thus, the same change must occur *second* at $T^2_{g_{n-1}}$ in T^2.

Besides, we show that at all intermediate states, the number of changes in the first trace is greater than the number of changes in the second trace. This follows directly, since for any r such that $i \leq r < g_{n-1}$, T^1_r is in the q^{th} block, with $q \geq n-1$, while T^2_r is in q'^{th} block with $q' < n-1$, thus the number of changes at T^1_r is q, the number of changes at T^2_r is q' and $q > q'$.

Case 2: $\mathcal{T} \models \phi \Rightarrow T^1 \sim T^2$

To show $T^1 \sim T^2$, it is sufficient to show $\forall i.\ \exists j.\ T^1 \sim_{i,j} T^2$, and vice versa. W.l.o.g. we show the first conjunct, by induction on i.

Base case $i = 0$: Take $j = 0$. It follows immediately that $T^1 \sim_{0,0} T^2$.

Induction step:

Assume that $T^1 \sim_{i,j} T^2$. We have to show that $\exists h.\ T^1 \sim_{i+1,h} T^2$. If $T^1_i = T^1_{i+1}$, we take $h = j$, and we are done. If $T^1_i \neq T^1_{i+1}$, there are two cases.

Case 2.1: $i \leq j$

Because $T^1 \sim_{i,j} T^2$, there are two sequences $0 = k_0 < k_1 < k_2 < \ldots < k_n = i+1$ and $0 = g_0 < g_1 < g_2 < \ldots < g_n = j+1$ (for some $n \geq 0$) such that for each $0 \leq p < n$:

$$T^1_{k_p} = T^1_{k_p+1} = \cdots = T^1_{k_{p+1}-1} = T^2_{g_p} = T^2_{g_p+1} = \cdots = T^2_{g_{p+1}-1}.$$

In this case, the state change $T^1_i \neq T^1_{i+1}$ occurs *first* in T^1. Since ϕ holds, there exists an h ($h \geq j+1$) such that $T^1_{i+1} = T^2_h \neq T^2_{h-1}$.

If $h = j+1$, it is trivial that $T^1 \sim_{i+1,h} T^2$.

If $h > j+1$, we prove that for all r such that $j \leq r < h$, $T^2_j = T^2_r$. We prove this by contradiction. Assume that there exists a state change that occurs between the indexes j and h in trace T^2 (different from the change occurring in h), i.e., at index r', $j+1 \leq r' < h$. This is the n^{th} state change in T^2. Because ϕ holds, the same state change has to also occur in trace T^1, and the total number of state changes at $T^1_{r'}$ must be strictly smaller than the total number of state changes at $T^2_{r'}$. However, the total number of state changes at $T^1_{r'}$ is $\geq n$, thus we have a contradiction.

Therefore, we can conclude that $T^2_j = T^2_{j+1} = \cdots = T^2_{h-1}$. Choose $k_{n+1} = i+2$, $g_{n+1} = h+1$, $k_n = i+1$ and $g_n = h$ such that for each $0 \leq p < n+1$:

$$T^1_{k_p} = T^1_{k_p+1} = \cdots = T^1_{k_{p+1}-1} = T^2_{g_p} = T^2_{g_p+1} = \cdots = T^2_{g_{p+1}-1}.$$

Thus $T^1 \sim_{i+1,h} T^2$.

Case 2.2: $i > j$

In this case, the change $T^1_i \neq T^1_{i+1}$ can be either a *first* change or a *second* change.

If it is a first change, by a similar argument, we can conclude that there exists a h such that $T^1 \sim_{i+1,h} T^2$.

If it is a second change, according to Theorem 1, there must be a first change in T^2 that matches with this change. We assume that the first change occurs at T^2_h where $h > j$.

If $h = j + 1$, we are done.

If $h > j + 1$, we prove that for all r such that $j \leq r < h$, $T^2_j = T^2_r$. We prove this by contradiction. Assume that a state change occurs between the indexes j and h in trace T^2 (different from the change occurring in h), i.e., at index r', $j + 1 \leq r' < h$. Because ϕ holds, the same state change will also occur in trace T^1. However, this change must occur after the change at state T^1_i, and this is a contradiction, since ϕ holds.

Thus, $T^1 \sim_{i+1,h} T^2$. □

4.3.3 Temporal-logic characterization of SSOD

Based on the results above, this section defines a temporal-logic formula characterizing SSOD. The formula consists of two parts: one that expresses stuttering equivalence between traces of each low variable individually, and one that expresses stuttering equivalence between traces of all low variables. Both are instantiations of the formula characterizing stuttering equivalence defined above.

Atomic propositions. To support the characterization of stuttering equivalence in different ways, we define different atomic propositions. To characterize stuttering equivalence over traces of each low variable, we use atomic propositions $\mathit{first\_change}^m_l$, $\mathit{second\_change}^m_l$, and ${\mathit{nr\_ch}_l}^m < \mathit{nr\_ch}^{3-m}_l$ for each $l \in L$. To characterize stuttering equivalence over traces of all low variables, we use atomic propositions $\mathit{first\_change}^m_L$, $\mathit{second\_change}^m_L$, and ${\mathit{nr\_ch}_L}^m < \mathit{nr\_ch}^{3-m}_L$.

The formal definitions of these atomic propositions are defined in the previous section, where equality is instantiated as $=_l$ and $=_L$, respectively.

Logic characterization of SSOD. A program C is observationally deterministic under a scheduler δ, iff for any two initial low-equivalent valuations s_1

and s_2, the following formula holds on traces of $\mathcal{M}^{\delta}_{C,s_1,s_2}$.

$$\left(\bigwedge_{l \in L} \phi_l\right) \wedge \phi_L, \text{ where}$$

$$\begin{aligned}
\phi_l &= \mathsf{G}\Big(\bigwedge_{m \in \{1,2\}} \mathit{first\_change}^m_l \Rightarrow \\
&\qquad \mathit{nr\_ch}_l{}^{3-m} < \mathit{nr\_ch}^m_l \;\mathsf{U}\; \mathit{second\_change}^{3-m}_l\Big) \\
\phi_L &= \mathsf{AG}\Big(\bigwedge_{m \in \{1,2\}} \mathit{first\_change}^m_L \Rightarrow \\
&\qquad \mathsf{E}(\mathit{nr\_ch}_L{}^{3-m} < \mathit{nr\_ch}^m_L \;\mathsf{U}\; \mathit{second\_change}^{3-m}_L)\Big)
\end{aligned}$$

Notice that ϕ_l is an LTL and ϕ_L a CTL formula.

For a program with N low variables, we have $N+1$ verification tasks: N tasks relate to traces of each low variable, and one task relates to traces of all low variable. For each task, we instantiate the extra information χ and the equality relation differently.

Theorem 4.2 *Given program C, and two low-equivalent valuations s_1 and s_2, C is observationally deterministic under δ iff*

$$\mathcal{M}^{\delta}_{C,s_1,s_2} \models \left(\bigwedge_{l \in L} \phi_l\right) \wedge \phi_L.$$

Proof: In the model of a self-composed program, the variables of the two program copies are disjoint, and the updates of each program copy are done locally, thus the stuttering equivalent property of the two component traces are unchanged as if they are produced from the executions of C starting in two initial valuations s_1 and s_2 separately, without composition.

Case 1: SSOD-1 $\Leftrightarrow \mathcal{M}^{\delta}_{C,s_1,s_2} \models \bigwedge_{l \in L} \phi_l$

In Theorem 4.1, we show a temporal-logic formula that characterizes stuttering equivalence between the two component traces. Using Theorem 4.1, instantiated with the atomic propositions defined above, it is easy to see that SSOD-1 is characterized exactly by the formula $\left(\bigwedge_{l \in L} \phi_l\right)$.

Case 2: SSOD-2 $\Leftrightarrow \mathcal{M}^{\delta}_{C,s_1,s_2} \models \phi_L$

Case 2.1 SSOD-2 $\Rightarrow \mathcal{M}^{\delta}_{C,s_1,s_2} \models \phi_L$

The condition SSOD-2 says that for any trace starting in s_1, there exists a trace starting in s_2 such that these two traces are low stuttering equivalent. It means that in a self-composed model, with respect to any low change in a component trace, there always *exists* a *composed trace* such that there is a match in the other *component trace*. Following the first case in the proof of Theorem 4.1, instantiated with the atomic propositions defined in 4.2.2, we can conclude that ϕ_L holds.

Case 2.2: $\mathcal{M}^{\delta}_{C,s_1,s_2} \models \phi_L \Rightarrow$ **SSOD-2**

Consider $\mathcal{M}^{\delta}_{C,s_1,s_2}$. Any path $\mathcal{T}$ of the model can be decomposed into T^m and T^{3-m} ($m \in \{1,2\}$). To satisfy SSOD-2, it is sufficient to show that for any given path $\mathcal{T}1$, for any i, there always exists a path $\mathcal{T}2$ such that $\mathcal{T}2_{\ll i} = \mathcal{T}1_{\ll i}$ ($\mathcal{T}1$ and $\mathcal{T}2$ have the same prefix) and $\exists j.\ T1^m{}_{|_L} \sim_{i,j} T2^{3-m}{}_{|_L}$, and similarly for $T1^{3-m}{}_{|_L}$. W.l.o.g. we show the first conjunct, by induction on i.

Base case $i = 0$: Take $j = 0$, then $T1^m{}_{|_L} \sim_{0,0} T2^{3-m}{}_{|_L}$, for any $\mathcal{T}2$.

Induction step: Assume that $T1^m{}_{|_L} \sim_{i,j} T1^{3-m}{}_{|_L}$. We have to show that $\exists \mathcal{T}2.\ \mathcal{T}2_{\ll i+1} = \mathcal{T}1_{\ll i+1}$, and $\exists h.\ T1^m{}_{|_L} \sim_{i+1,h} T2^{3-m}{}_{|_L}$.

If $T1^m_i{}_{|_L} = T1^m_{i+1}{}_{|_L}$, we take $\mathcal{T}2 = \mathcal{T}1$ and $h = j$, we are done.

If $T1^m_i{}_{|_L} \neq T1^m_{i+1}{}_{|_L}$, there are two cases.

Case 2.2.1: It is a *first* change

Since ϕ_L holds, there exists a trace $\mathcal{T}2$ such that $\mathcal{T}2_{\ll i+1} = \mathcal{T}1_{\ll i+1}$, and there also exists an h ($h \geq i+1$) such that $T1^m_{i+1}{}_{|_L} = T2^{3-m}_h{}_{|_L} \neq T2^{3-m}_{h-1}{}_{|_L}$. Due to the construction of the queue and the definitions of atomic propositions, $T1^m{}_{|_L}$ and $T2^{m-3}{}_{|_L}$ must be stuttering equivalent up to $i+1$ and h. If they are not, a contradiction occurs. According to ϕ_L, the total number of changes at $T2^{3-m}_r$ ($i+1 \leq r < h$) must be strictly *smaller than* the total number of changes at $T1^m_r$. Assume that at $T2^{3-m}_{r'}$ ($i+1 < r' < h$), there is a change that have not occurred in $T1^m{}_{\ll i+1}$. According to ϕ_L, the total number of changes at $T1^m_{r'}$ must be strictly *smaller than* the total number of changes at $T2^{3-m}_{r'}$, thus we have a contradiction.

Case 2.2.2: It is a *second* change

Follow the same argument in the proof of Theorem 4.1 — instantiated with the atomic propositions defined above — on trace $\mathcal{T}1$, we are done. □

4.4 Conclusions

This chapter discussed how an observational determinism property can be verified via a logic-based verification approach. Implementing the idea of self-composition, SSOD properties over traces of the executions of C from two initial states are reduced to a logic property over traces of a single execution of the composed program $C \| C'$, where C' is a copy of C. This approach has the advantage that we can reuse existing standard verification tools to prove or disprove information flow properties.

A disadvantage of the logic-based verification approaches is that the program model is often very complex; and thus, they are not very feasible to verify large systems. Therefore, the remainder of this part introduces another approach, based on an algorithmic verification technique, to verify information flow properties. The algorithmic method has a less complex program model; and thus, it is often more practical for real applications.

Chapter 5

Algorithmic Verification

5.1 Introduction

Developing a logic-based verification technique has the advantage that we can use an existing model checker to verify the information flow property. However, since the program model for the logical characterization is rather complex, it is difficult to handle large state-space systems. Therefore, instead of relying on existing model-checking tools, this chapter develops more efficient specialized verification methods for our confidentiality properties.

To verify scheduler-specific observational determinism (SSOD), a property that characterizes secure information flow for non-deterministic multi-threaded programs, we first extract a Kripke structure that models the program execution under the control of the given scheduler. Remember that SSOD imposes two conditions: (SSOD-1) all individual public variables have to evolve deterministically, and (SSOD-2) the relative ordering of updates of public variables is coincidental, i.e., there always exists a matching trace.

We verify SSOD-1 by reducing it to the question whether all traces of each public variable of a Kripke structure under the given scheduler are stuttering equivalent. An algorithm to verify all-trace stuttering equivalence is implemented by checking whether there exists a functional bisimulation between the Kripke structure and a witness trace.

To verify SSOD-2, we first remove stuttering steps, and then determinize two Kripke structures that model the program executions starting in two initial low-equivalent states. Next, we check whether these two deterministic and stuttering-free Kripke structures are strongly bisimilar. Our verification is based on the well-known fact that in deterministic and stuttering-free Kripke structures, trace equivalence and strong bisimulation coincides [36].

Using a similar approach, scheduler-specific probabilistic observational determinism (SSPOD), a confidentiality property for probabilistic multi-threaded programs, can also be verified algorithmically. SSPOD requires two conditions: (SSPOD-1) all traces of each low variable are stuttering equivalent with probability 1, and (SSPOD-2) for every trace considering all publicly visible variables, there always exists a matching trace with equal probability.

To verify SSPOD, firstly, the executions of the probabilistic program under the given scheduler are modeled as probabilistic Kripke structures. To check SSPOD-1, we first remove all stuttering loops, except self-loops in final states, i.e., we remove all traces that stutter forever and have probability 0. With a stuttering-loop-free probabilistic Kripke structure, SSPOD-1 is satisfied if all traces of each public variable are stuttering equivalent. The verification of all-trace stuttering equivalence here is similar to the verification of SSOD-1. SSPOD-2 is implemented by removing stuttering steps, thereby reducing this property verification to checking of probabilistic-language equivalence [89, 35, 52].

The time complexity of the algorithms to verify SSOD-1 and SSPOD-1 is linear in the size of the Kripke structure that models the program execution. Due to the determinization step, the checking of SSOD-2 might be exponential in the number of states of the Kripke structures. However, this worst-case complexity is only reached in a very few cases [63]. The average complexity is mostly far lower than exponential. The complexity of checking SSPOD-2 is polynomial in the size of the Kripke structure.

This algorithmic approach gives a precise verification method for information flow properties. The proposed algorithms are general, and also applicable in other, non-security related contexts, since stuttering equivalence is a fundamental concept in the theory of concurrent and distributed systems.

Organization of the chapter. To present algorithms in a general way, Section 5.2 first simplifies the notations of SSOD, and then Section 5.3 and Section 5.4 give efficient specialized algorithms to verify its two conditions. The algorithms to verify two conditions of SSPOD are presented in Section 5.5 and Section 5.6, respectively. Section 5.7 discusses related work, while Section 5.8 draws conclusions.

Origins of the chapter. The algorithms to verify SSOD were published in the *Journal of Computer Security* (JCS) [69]. The algorithms to verify SS-POD were published in the proceedings of the *5th International Conference on Engineering Secure Software and Systems* (ESSoS'13) [68], and also in a corresponding technical report [70].

5.2 Simplified SSOD

For the convenience, we remind two formalizations of SSOD-1 and SSOD-2.

SSOD-1 $\forall T \in \mathit{Trace}(\mathcal{A}_\delta),\ T' \in \mathit{Trace}(\mathcal{A}'_\delta), l \in L.\ T_{\,|_l} \sim T'_{\,|_l}$

SSOD-2 $\forall T \in \mathit{Trace}(\mathcal{A}_\delta).\ \exists T' \in \mathit{Trace}(\mathcal{A}'_\delta).\ T_{\,|_L} \sim T'_{\,|_L}$

As shown in Section 3.3, we can simplify SSOD by replacing SSOD-1 with SSOD-1A. Basically, SSOD-1A requires that, given a Kripke structure $\mathcal{A}_\delta$ that corresponds to any initial state, after projecting on any $l \in L$, all traces are stuttering equivalent.

SSOD-1A $\forall l \in L.\ T, T' \in \mathit{Trace}(\mathcal{A}_\delta).\ T_{\,|_l} \sim T'_{\,|_l}.$

The replacement of SSOD-1 by SSOD-1A simplifies a lot the algorithms to verify SSOD, in both time and space complexity. For instance, we do not need to model two program executions starting in two different initial states, and then combine these two Kripke structures together to verify SSOD-1.

To present algorithms in a general way, we simplify the notations of SSOD-1A and SSOD-2. Let $\mathcal{A}_{\delta\,|_l}$ and $\mathcal{A}_{\delta\,|_L}$ represent the projections of $\mathcal{A}_\delta$ on the label sets l and L, respectively. Since $\{T_{\,|_l} \mid T \in \mathit{Trace}(\mathcal{A}_\delta)\} = \mathit{Trace}(\mathcal{A}_{\delta\,|_l})$ and $\{T_{\,|_L} \mid T \in \mathit{Trace}(\mathcal{A}_\delta)\} = \mathit{Trace}(\mathcal{A}_{\delta\,|_L})$, properties SSOD-1A and SSOD-2 can be easily reformulated over these Kripke structures, as follows.

SSOD-1K $\forall l \in L.\ T, T' \in \mathit{Trace}(\mathcal{A}_{\delta\,|_l}).\ T \sim T',$

SSOD-2K $\forall T \in \mathit{Trace}(\mathcal{A}_{\delta\,|_L}).\ \exists T' \in \mathit{Trace}(\mathcal{A}'_{\delta\,|_L}).\ T \sim T'.$

Thus, SSOD-1K requires that for each $l \in L$, all traces of $\mathcal{A}_{\delta\,|_l}$ are stuttering equivalent, while SSOD-2K requires stuttering trace equivalence between $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$. Notice that the below algorithms work only on finite Kripke structures, i.e., we assume that data domains are finite, and schedulers use finite memory.

5.3 Verification of SSOD-1K

Given a program C, and a scheduler δ, SSOD-1K requires that after projecting $\mathcal{A}_\delta$ on any low variable l, all traces must be stuttering equivalent. To verify this, we pick one arbitrary trace, and then ensure that all other traces are stuttering equivalent to this trace.

5.3.1 Algorithm

Concretely, for each $l \in L$, we carry out the following steps. (Figures 5.1, and 5.2 on page 74 provide an elaborate illustration of these steps.)

```
Algorithm 1: SSOD-1K on $l$
1: Project $\mathcal{A}_\delta$ on $l$, yielding $\mathcal{A}_{\delta\,|_l}$.
2: Identify all divergent states of $\mathcal{A}_{\delta\,|_l}$. A state $c$ of $\mathcal{A}$ is divergent if
   there exists a trace such that all states following $c$ are equivalent to $c$.
3: Check whether all traces of $\mathcal{A}_{\delta\,|_l}$ are stuttering equivalent by:
    3.1: Choose a witness trace by:
    3.1.1: Take an arbitrary lasso $T$ of $\mathcal{A}_{\delta\,|_l}$.
    3.1.2: Remove stuttering steps and minimize $T$ by strong bisimulation
           reduction.
    3.2: Check stuttering trace equivalence between $\mathcal{A}_{\delta\,|_l}$ and $T$ by
         checking if there exists a functional bisimulation between them.
```

Notice that the below algorithms that implement the above steps can be applied to any Kripke structure, i.e., independent of the scheduling policy. Therefore, instead of the notation $\mathcal{A}_\delta$ indicating a *specific* scheduler, we define these algorithms over *arbitrary* Kripke structures $\mathcal{A}$.

Step 1 is done by labeling every state of $\mathcal{A}$ with the value of l in that state.

Step 2 identifies all divergent states of a Kripke structure $\mathcal{A}$. Before describing the algorithm for Step 2, we first derive two lemmas that follow directly from the definition of a divergent state (given in Step 2 of Algorithm 1).

Lemma 2 *If state c has a stuttering loop or a self-loop, c is divergent.*

Lemma 3 *Assume that state c has no self-loop or no stuttering loop. Then, state c is divergent iff c has a divergent equivalent successor.*

Step 2 is implemented by exploring the state space of a Kripke structure $\mathcal{A}$ and determining whether each reachable state is divergent or not. The state space of $\mathcal{A}$ is explored in a breadth first search order (BFS). Let $\mathit{Pred}(\mathcal{A}, c)$ and $\mathit{Succ}(\mathcal{A}, c)$ denote the set of all *direct* predecessors and successors of c, respectively, i.e., $\mathit{Pred}(\mathcal{A}, c) = \{b \in \mathcal{S} \mid b \to c\}$ and $\mathit{Succ}(\mathcal{A}, c) = \{d \in \mathcal{S} \mid c \to d\}$. Let $c \sim_V c'$ denote that c and c' have the same valuation, i.e., $V(c) = V(c')$.

The algorithm to identify divergent states of $\mathcal{A}$ uses two queues: $\mathcal{Q}$ and $\mathit{Non\_Diver\_Q}$, and two maps: *Divergent* and *Checked*. The queue $\mathcal{Q}$ stores the set of *frontier* states of the exploration. Initially, *Divergent* indicates the number of stuttering transitions of each state, i.e., $\mathit{Divergent}[c] = 3$ indicates that c is equivalent to three of its successors. During the execution, the algorithm changes the values in *Divergent*. When the algorithm terminates, the value of $\mathit{Divergent}[c]$ will indicate whether c is divergent or not, i.e., iff $\mathit{Divergent}[c] = 0$, c is *non-divergent*. The queue $\mathit{Non\_Diver\_Q}$ stores non-divergent states, i.e., all reachable state c such that $\mathit{Divergent}[c] = 0$. The map *Checked* indicates whether a state has been checked or not, i.e., *true* or *false*.

Algorithm 2 works as follows. States of $\mathcal{A}$ are explored by a BFS (lines 6-12 in the algorithm). For each explored state c, the number of its direct stuttering transitions is stored in $\mathit{Divergent}[c]$ (line 11). If c has no outgoing stuttering transition, i.e., $\mathit{Divergent}[c] = 0$, it is clear that c is non-divergent (line 12).

For any c such that $\mathit{Divergent}[c] = 0$, for each predecessor b of c such that c is reached from b by a stuttering transition, the algorithm decreases the value $\mathit{Divergent}[b]$ by 1 (lines 13-18). The idea is to remove the number of *non-divergent* equivalent successors out of the value $\mathit{Divergent}[b]$ of a state b. Finally, when *Divergent* is stable, the value $\mathit{Divergent}[b]$ will indicate whether b is divergent or not, i.e., if b has a divergent equivalent successor, i.e., $\mathit{Divergent}[b] \neq 0$, b is divergent (due to Lemma 3).

For example, assume that b is a direct predecessor of a non-divergent c, and b has *only* one stuttering transition, which goes to c, i.e., $\mathit{Divergent}[b] = 1$. Since c is non-divergent, according to Lemma 3, b is also non-divergent. Due to the algorithm, the value of $\mathit{Divergent}[b]$ is decreased by 1; and thus becomes 0, which indicates that b is non-divergent.

Algorithm 2: Identify Divergent States ($\mathcal{A}$)

```
//  Initialization
1.    for all states $c \in \mathcal{S}$ do
2.        $\mathit{Checked}[c] := \mathit{false}$;
3.        $\mathit{Divergent}[c] := 0$;
4.    $\mathcal{Q} := \mathit{empty\_queue}()$; $\mathit{enqueue}(\mathcal{Q}, \mathit{init\_state})$;
```

```
5.       $Non\_Diver\_Q := empty\_queue();$
//   Explore state space by BFS
6.       $enqueue(\mathcal{Q}, init\_state);$ $Checked[init\_state] := true;$
7.       while $!empty(\mathcal{Q})$ do
8.             $current := dequeue(\mathcal{Q});$
9.             for all states $c \in Succ(\mathcal{A}, current)$ and $\neg Checked[c]$ do
10.                  $enqueue(\mathcal{Q}, c);$ $Checked[c] := true;$
     //   Record the number of stuttering successors
11.            $Divergent[current] := |\{c \in Succ(\mathcal{A}, current) \mid c \sim_V current\}|;$
12.            if $Divergent[current] = 0$ then $enqueue(Non\_Diver\_Q, current);$
//   Propagate non-divergence backwards
13.      while $!empty(Non\_Diver\_Q)$ do
14.            $current := dequeue(Non\_Diver\_Q);$
15.            for all states $b \in Pred(\mathcal{A}, current)$ do
16.                  if $b \sim_V current$ and $Divergent[b] \neq 0$ then
17.                       $Divergent[b] := Divergent[b] - 1;$
18.                       if $Divergent[b] = 0$ then $enqueue(Non\_Diver\_Q, b);$
//   Normalize divergence
19.      for all states $c \in \mathcal{S}$ do
20.           if $Divergent[c] \neq 0$ then $Divergent[c] := true$
21.                                    else $Divergent[c] := false;$
```

Theorem 5.1 *Algorithm 2 identifies divergent states of a Kripke structure $\mathcal{A}$.*

Proof: Algorithm 2 always terminates, since $\mathcal{A}$ is finite. We show that the *second while* loop correctly determines the divergent property of each state. We first discuss its loop invariant *Inv*:

If a state c is divergent, $Divergent[c] \neq 0$.

Initially, the value of $Divergent[c]$ is the number of stuttering successors of c. If c is divergent, according to the definition of divergent state, it must have at least one stuttering successor, i.e., $Divergent[c] \neq 0$. Thus, clearly, *Inv* holds upon the first entry of the loop.

We show that the invariant is preserved by every iteration of the loop. Assume that *Inv* holds before the loop body. Consider a divergent state c. Due to the invariant, $Divergent[c] \neq 0$ holds before the execution of the loop.

Suppose that the value of $Divergent[c]$ has been decreased by 1 in the iteration, i.e., c has a stuttering successor d with $Divergent[d] = 0$.

- Case c has a self-loop. In that case, c has at least two stuttering successors: one of them is itself. Therefore, the invariant is preserved after the iteration, since $\mathit{Divergent}[c] \geq 2$ before the iteration.

- Case c has a stuttering loop. States in a stuttering loop always have at least one stuttering successor. Since they are connected in a loop, their *Divergent* values never become 0. Since $\mathit{Divergent}[d] = 0$, d must be outside the stuttering loop. Thus, c has at least two stuttering successors before the iteration. The invariant is preserved.

- If c has no self-loop or no stuttering loop, according to Lemma 2, c must have a divergent stuttering successor e. Since the invariant is true before the iteration, $\mathit{Divergent}[e] \neq 0$. Hence, d and e are not the same state. Thus, c has at least two stuttering successors. The invariant is preserved.

Thus, *Inv* is a loop invariant. Therefore, *Inv* holds after termination: if c is divergent, $\mathit{Divergent}[c] \neq 0$. In other words, any c such that $\mathit{Divergent}[c] = 0$ is non-divergent.

Additionally, we show the following *post-condition*: **if the algorithm terminates, and if c is non-divergent, then** $\mathit{Divergent}[c] = 0$.

If c is non-divergent, all traces starting in c must pass a state that is not equivalent to c. Let $\mathcal{C}$ denote the set of *equivalent* states that are reachable from c only via *stuttering transitions*. To define $\mathcal{C}$ formally, we define the *stuttering-closure* $\mathit{Stut}(Q)$ of a subset $Q \subseteq \mathcal{S}$,

$$\begin{aligned} &Q_0 = Q, \\ &\forall n \geq 0.\, Q_{n+1} = \{c' \in \mathcal{S} \mid \exists c \in Q_n.\, c \to c' \wedge c \sim_V c'\}. \end{aligned}$$

We define $\mathit{Stut}(Q) = \cup_n Q_n$. This is formally defined as an infinite union, but it is actually a finite union; since there are at most a finite number of states in $\mathcal{S}$. Therefore, formally, $\mathcal{C} = \mathit{Stut}(c)$.

There must exist a state $c' \in \mathcal{C}$ such that initially, $\mathit{Divergent}[c'] = 0$, i.e., c' only connects to states outside $\mathcal{C}$. Otherwise, a contradiction occurs: assume that initially, $\forall c' \in \mathcal{C}$, $\mathit{Divergent}[c'] \neq 0$, i.e., all states have at least a stuttering successor. Since the set $\mathcal{C}$ is finite, if all states of $\mathcal{C}$ have stuttering successors, there must exist a stuttering loop inside $\mathcal{C}$. Hence, c is divergent due to Lemma 2; and this is a contradiction, since we assume that c is non-divergent.

Let D_0 denote the set of states $c' \in \mathcal{C}$ such that *initially*, $\mathit{Divergent}[c'] = 0$. Notice that Algorithm 2 changes the *Divergent* value of states. Let $\mathbf{D_0}$ denote the set of states of $\mathcal{C}$ such that *currently*, their *Divergent* values are 0. Initially, $\mathbf{D_0} = D_0$.

Consider the set of direct predecessors of D_0. We claim that there must exist a state c'' in this set of predecessors such that initially, $\mathit{Divergent}[c'']$ is equal to the number of its successors c' with $\mathit{Divergent}[c'] = 0$, i.e, $\mathit{Divergent}[c''] = |\{c' \in \mathit{Succ}(\mathcal{A}, c'') \mid c' \sim_V c'' \wedge \mathit{Divergent}[c'] = 0\}|$. If not, the contradiction occurs by the same argument, i.e., there exists a stuttering loop. Let D_1 denote the set of direct predecessors c'' of D_0 such that *initially*, $\mathit{Divergent}[c''] = 1$. According to the algorithm, the *Divergent* values of states in D_1 are decreased by 1. Thus, after a few iterations, the *Divergent* values of all states in D_1 become 0, i.e., $\mathbf{D_0} = D_0 \cup D_1$.

Therefore, by a similar argument, the *Divergent* values of other states, e.g., the predecessors of D_0 with the *Divergent* value 2, also become 0; and so on. The set $\mathbf{D_0}$ gradually grows, and at the termination, $\mathbf{D_0} = \mathcal{C}$. Therefore, when the algorithm terminates, if c is non-divergent, $\mathit{Divergent}[c] = 0$. This is also equivalent that if $\mathit{Divergent}[c] \neq 0$, c is divergent.

Notice that the two queues $\mathcal{Q}$ and *Non_Diver_Q* ensure that each state and each edge of $\mathcal{A}$ are processed at most once in each *while* loop. Thus, the time complexity of Algorithm 2 is linear in the size of $\mathcal{A}$. □

Step 3: Step 3.1.1 is implemented via a classical *cycle-detection* algorithm based on depth-first search. The initial state of a lasso is also the initial state of the Kripke structure. The algorithm essentially proceeds by picking arbitrary next steps, and terminates when it hits a state that was picked before.

Algorithm 3: Lasso T of $\mathcal{A}$

```
for all states $c \in \mathcal{S}$ do $\mathit{Visit}[c] := \mathit{false}$;
$\mathit{index} := 0$;
$\mathit{current} := \mathit{init\_state}$;
for (; ; ) do
    $T[\mathit{index}] := \mathit{current}$;       // Implement $T$ as an array
    $\mathit{index} := \mathit{index} + 1$;
    if $\mathit{Visit}[\mathit{current}] = \mathit{true}$ then break;
    $\mathit{Visit}[\mathit{current}] := \mathit{true}$;
    $\mathit{current} :=$ some state $c \in \mathit{Succ}(\mathcal{A}, \mathit{current})$;
return($T$, position of $\mathit{current}$ in $T$);
```

In Algorithm 3, we use a map *Visit* to indicate the visited states of $\mathcal{A}$, i.e., *Visit*[*current*] = *true* indicates that *current* has been visited before. Clearly, Algorithm 3 returns a trace of $\mathcal{A}$. Moreover, it always terminates, since $\mathcal{A}$ is finite and there is a self-loop at every final state.

Step 3.1.2 is done via the standard *strong bisimulation reduction* [17]. For example, the *minimal* form of a lasso $\mathbf{abb}(\mathbf{cb})^\omega$ is $\mathbf{a}(\mathbf{bc})^\omega$. This minimal lasso is called the *witness trace*. Except the final state, all states of the witness trace are set to be non-divergent.

Step 3.2 checks stuttering trace equivalence between a Kripke structure $\mathcal{A}$ and the witness trace T by checking if there exists a *functional* bisimulation between them, i.e., a bisimulation that is a function, thus mapping each state in $\mathcal{A}$ to a single state in T. This is done by exploring the state space of $\mathcal{A}$ in a breadth-first search and building the mapping *Map* during exploration. We name each state in T by a unique symbol $u \in \mathcal{U}$, i.e., u_i denotes T_i. Let $Succ(T, u)$ denote the successor of u on T.

We map the initial state of $\mathcal{A}$ to u_0, i.e., $Map[init\_state] = u_0$ (line 4 in Algorithm 4). Each iteration of the algorithm examines the successors of the state stored in the variable *current* (lines 6-8). Assume that *Map*[*current*] is u, consider a successor $c \in Succ(\mathcal{A}, current)$. The *potential_map* of c is u if $current \to c$ is a stuttering transition; otherwise, it is $Succ(T, u)$ (line 9). The algorithm returns *false*, i.e., *continue* = *false*, if (i) c and *potential_map* have different valuations, (ii) c and *potential_map* have different divergent values, or (iii) c has been checked before, but its mapped state is not *potential_map* (line 10, 11, and 13).

If none of these cases occurs, and c was not checked before, c is added to $\mathcal{Q}$, and mapped to *potential_map* (line 12). Basically, a state c of $\mathcal{A}$ is mapped to u, i.e., $Map[c] = u$, iff the trace from the initial state to state c in $\mathcal{A}$ and the prefix of T up to u are stuttering equivalent.

In the algorithm, $final(\mathcal{A}, c)$ denotes that c is a final state in $\mathcal{A}$; and $final(T, u)$ denotes that u is the final state in T. Algorithm 4 also uses a queue $\mathcal{Q}$ of *frontier* states. The termination of Algorithm 4 follows the termination of BFS over a finite $\mathcal{A}$.

```
Algorithm 4: All-Trace Stuttering Equivalence ($\mathcal{A}$, $T$)
1.    for all states $c \in \mathcal{S}$ do $Map[c] := \bot$;
2.    $continue := true$;
3.    $\mathcal{Q} := empty\_queue()$; $enqueue(\mathcal{Q}, init\_state)$;
4.    $Map[init\_state] := u_0$;      // $u_0$ is $T_0$
5.    while $!empty(\mathcal{Q}) \wedge continue$ do
6.        $current := dequeue(\mathcal{Q})$;
```

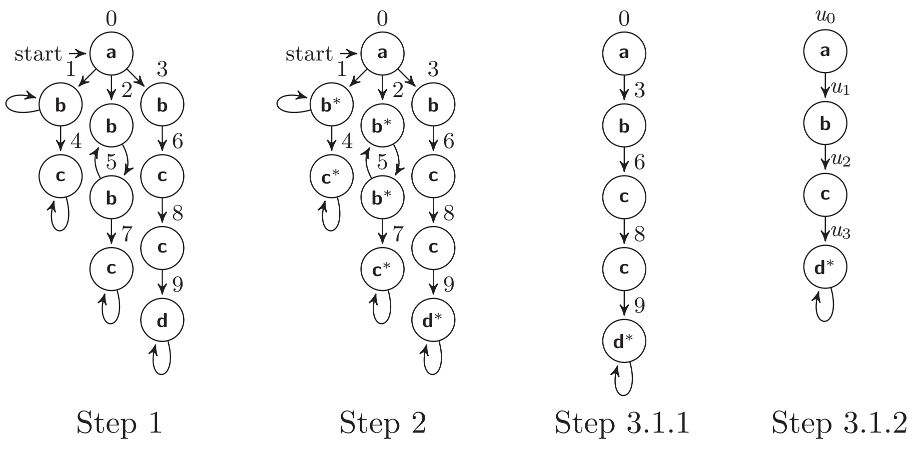
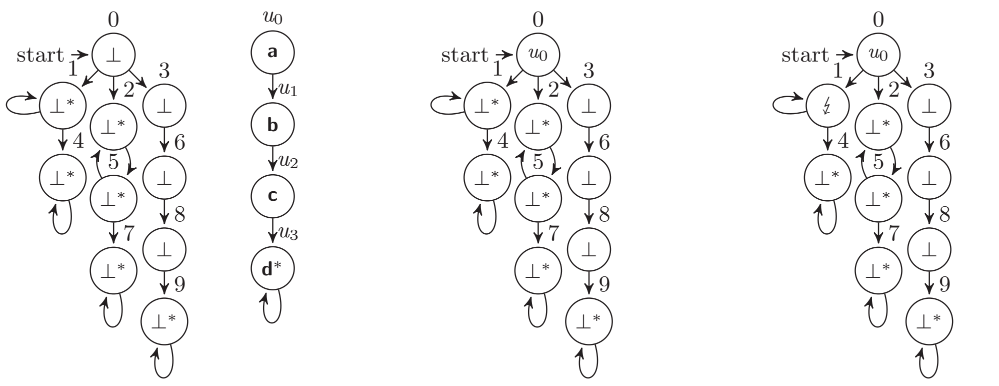


Figure 5.1: Step 1 - Step 3.1 of Algorithm 1

```
7.              $u := Map[current]$;
8.              for all states $c \in Succ(\mathcal{A}, current)$ do
9.                  $potential\_map := (c \sim_V current) \,?\, u : Succ(T, u)$;
10.                 case $c \not\sim_V potential\_map \rightarrow continue := false$;
11.                 ▯ $Divergent[c] \neq Divergent[potential\_map] \rightarrow$
                                $continue := false$ ;
12.                 ▯ $Map[c] = \perp \rightarrow enqueue(\mathcal{Q}, c);\ Map[c] := potential\_map$;
13.                 ▯ $Map[c] \neq potential\_map \rightarrow continue := false$;
14.     return $continue$;
```

Example 5.1 *Figure 5.1 illustrates Step 1 - Step 3.1 on a Kripke structure $\mathcal{A}$ consisting of 10 states, numbered from 0 to 9. Step 1 shows a projection of $\mathcal{A}$ on a low variable l where the symbols **a**, **b**, **c** etc. denote state contents, i.e., states with the same value of l are represented by the same symbol. Step 2 identifies divergent states by *. Step 3.1 takes an arbitrary trace of $\mathcal{A}$, and then minimizes it. Each state of the witness trace T is denoted by a unique symbol u_i. Figure 5.2 illustrates Step 3.2. Initially, all states of $\mathcal{A}$ are mapped to a special symbol $\perp$ that indicates* unchecked *states. To keep states readable, we skip the valuation. Next, state 0 is enqueued, and mapped to u_0. In the next step, the algorithm examines all unchecked successors of state 0, i.e., states 1, 2, 3. Each of them follows a non-stuttering step, thus their potential_maps are all u_1. State 1 is divergent while potential_map is not, thus, continue = false.*



$\mathcal{Q} = \emptyset$ Before mapping $\mathcal{Q} = [0]$ Map state 0 $\mathcal{Q} = [1, 2, 3]$ Violation in state 1

Figure 5.2: Step 3.2 of Algorithm 1 (i.e., Algorithm 4)

SSOD-1K fails, since there exists a trace that stutters in state 1 *forever, and thus, $\mathcal{A}$ and T are not stuttering trace equivalent. The algorithm terminates.*

As a first step towards proving correctness, we prove that Algorithm 4 ensures the following loop invariant.

Theorem 5.2 *Algorithm 4 preserves the following loop invariant Inv:*
If continue, then $\forall c \in \mathcal{S}$ such that $Map[c] = u$, the trace from init_state to c and the prefix of T up to u are stuttering equivalent, and if $\neg$continue, then there exists a trace of $\mathcal{A}$ that is not stuttering equivalent to T.

Proof: Clearly, *Inv* holds upon the first entry of the loop, since initially, *continue* holds, and only *init_state* is mapped to the initial state of T, i.e., u_0.

We show that the invariant is preserved by every iteration of the loop. Assume that *Stm* holds before the loop body. If *continue* does not hold, then the loop is not executed, and the algorithm ends. The invariant is preserved.

Otherwise, *continue* holds. The invariant before the loop body states that the trace from *init_state* to *current* and the prefix of T up to u are stuttering equivalent. Now consider a successor c of *current*. We distinguish the following cases:

Case $c \not\sim_V u$ **and** $c \not\sim_V Succ(T, u)$**.** Let *potential_map* denote the mapping candidate of c. It is u if $c \sim_V$ *current*; otherwise, it is $Succ(T, u)$. If $c \not\sim_V u$

and $c \not\sim_V Succ(T, u)$, then $c \not\sim_V potential\_map$. Thus, *continue* becomes *false*. The invariant is preserved, since any trace that goes from *current* to c is *not* stuttering equivalent to T.

Case $c \sim_V u$ **or** $c \sim_V Succ(T, u)$**.** Thus, $c \sim_V potential\_map$. Now, we consider the following cases:

Case $Divergent[c] \neq Divergent[potential\_map]$**.** If c is a divergent state of $\mathcal{A}$, then there must be a trace that stutters in c forever, while T can evolve from *potential_map* to a state with a different valuation (or vice versa). Thus, these two traces are not stuttering equivalent. Hence, *continue* becomes *false*; and the invariant is preserved.

Case $Divergent[c] = Divergent[potential\_map]$**.**

Case c **is unchecked.** Thus, $Map[c] = \perp$. State c is added to $\mathcal{Q}$, and becomes a frontier state. Moreover, it is mapped to *potential_map*. It is easy to see that the trace from *init_state* to c and the prefix of T up to *potential_map* are stuttering equivalent. Hence, the invariant is preserved.

Case c **is checked before.** Thus, $Map[c] \neq \perp$.

Case $Map[c] = potential\_map$**.** State c has been explored before; the algorithm does not explore it further. Since *continue* and *Map* are not updated, the invariant is preserved.

Case $Map[c] \neq potential\_map$**.** Thus, *continue* becomes *false*. The invariant is preserved, since there exist two traces that both lead to c and in these two traces, c is mapped to two different states of T; thus, one of these two trace is not stuttering equivalent to T.

□

Theorem 5.3 *Algorithm 4 returns true iff there exists a functional bisimulation between $\mathcal{A}$ and T.*

Proof: If Algorithm 4 returns *false*, it follows directly from the invariant that no functional bisimulation exists. If it returns *true*, due to the loop invariant, we can conclude that for any trace of $\mathcal{A}$, e.g., $T1$, there exists a prefix of the witness T that is stuttering equivalent to $T1$. We show that $T1$ is actually stuttering equivalent to the whole T.

Case $T1$ ends with a final state c. Assume that Algorithm 4 maps c to *potential_map*. Since the algorithm is divergent-sensitive, and in the witness trace T, the only divergent state is the final state, thus *potential_map* is also the final state of T. Therefore, $T1$ and T are stuttering equivalent.

Case $T1$ ends with a non-stuttering loop that starts and ends in state c. Thus, state c is investigated twice, and in the second visit, its corresponding mapped state (of T) must be the same as its mapped state in the first visit; otherwise, the algorithm returned *false*. Hence, the c's mapped state is also the start and end of a loop that terminates T. Thus, $T1$ and T are stuttering equivalent.

$\square$

Notice that our definition of *scheduler-independent* observational determinism can also be verified via Algorithm 1. First, we project both $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ (modeling the program executions starting in two initial low-equivalent states I and I') on the set L, yielding $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$. Next, we combine $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$ together, yielding $\mathcal{A}^+_{\delta\,|_L}$, and then apply **Step 2** and **Step 3** of Algorithm 1.

5.3.2 Overall complexity

Step 1 labels every state of $\mathcal{A}$ by the value of l in that state. This is done in time complexity $O(n)$, where n is the number of states of $\mathcal{A}$. Step 2 is based on BFS, thus its time complexity is $O(n+m)$, where m is the number of transitions of $\mathcal{A}$. The time complexity of Step 3.1 to find a witness trace is $O(m)$. The core of Step 3.2 is also BFS, whose running time is $O(n+m)$. Therefore, for a single low variable l, the total time complexity of the verification of SSOD-1K is linear in the size of $\mathcal{A}$, i.e., $O(n+m)$, and for any initial state, the total complexity of the verification (for all $l \in L$) is $|L|O(n+m)$. If we put restrictions on the initial inputs, i.e., the number of initial states is finite, the verification of SSOD-1K is feasible in practice (see Chapter 7).

5.4 Verification of SSOD-2K

SSOD-2K requires that, given two Kripke structures $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ that model the executions of a program C from any two initial low-equivalent states I and I', if we project them on the set of low variables L, $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$ are stuttering trace equivalent.

To verify SSOD-2, our algorithm first transforms Kripke structures into two equivalent ones, without *stuttering steps*, and then *determinizes* them[1]. It is well-known that, for deterministic and stuttering-free Kripke structures, trace equivalence and strong bisimilarity coincide [36].

5.4.1 Algorithm

We verify SSOD-2K by combining several existing algorithms.

Algorithm 5: SSOD-2K

```
1: Project both $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ (modeling the executions starting in $I$ and $I'$)
   on the set $L$, yielding $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$.
2: Remove all stuttering steps from $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$, yielding stuttering-free
   Kripke structures $\mathcal{A}^{sf}_{\delta}\,|_L$ and $\mathcal{A}'^{sf}_{\delta}\,|_L$.
3: Re-establish self-loops for final states of $\mathcal{A}^{sf}_{\delta}\,|_L$ and $\mathcal{A}'^{sf}_{\delta}\,|_L$.
4: Determinize $\mathcal{A}^{sf}_{\delta}\,|_L$ and $\mathcal{A}'^{sf}_{\delta}\,|_L$, yielding deterministic stuttering-free
   $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$.
5: Combine $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$, yielding $\mathcal{R}^{+}_{\delta}\,|_L$, and then compute all bisimila-
   rity equivalence classes of $\mathcal{R}^{+}_{\delta}\,|_L$.
6: Check if $I$ and $I'$ are in the same bisimilarity equivalence class.
```

Step 1 is done by labeling every state of a Kripke structure with the set of low values L in that state. To remove the stuttering steps in **Step 2**, we compute the stuttering closure of each state, using the standard all-pair shortest path algorithm [33], and then collapse these components into a single state. To ensure that the transition relation remains non-blocking, **Step 3** re-establishes the self-loops for final states. Notice that $\mathcal{A}$ and $\mathcal{A}^{sf}$ are stuttering trace equivalent, since traces of $\mathcal{A}^{sf}$ are traces of $\mathcal{A}$, but all stuttering steps in traces have been removed.

The determinization of a Kripke structure in **Step 4** is obtained via the well-known subset construction. Due to the property of determinization of finite automata, $\mathcal{A}^{sf}$ and $\mathcal{R}$ are trace equivalent. Notice that the determinization is based on *low events*, i.e., operations of changing the values of low variables. Thus, a state of $\mathcal{R}$ is a group of states in the original Kripke structure $\mathcal{A}$ that are reached via *the same low operation*. Therefore, states of $\mathcal{R}$ have the same label as their inside *component* states.

[1]We refer to the determinization concept in automata theory [44].

Step 5: Computing bisimilarity equivalence classes. For deterministic stuttering-free Kripke structures, trace equivalence and strong bisimilarity coincide. To verify strong bisimilarity of $\mathcal{R}$ and $\mathcal{R}'$, we first take the union of state spaces of the two Kripke structures, denoted $\mathcal{R}^+$, and use the classic algorithm for computing bisimilarity by Paige and Tarjan [71]. This is a standard algorithm, and readers can refer to [71] for a detailed description of the algorithm. However, since this step strongly relates to the chapter where we synthesize attacks for insecure programs, we represent briefly the algorithm's main idea.

The algorithm for computing bisimilarity equivalence classes exploits the well-known *partition-refinement* technique [71]. The main idea of this technique is to partition the state space into disjoint blocks of states, and repeatedly refine this partition: whenever we find that states of a block are not equivalent, we split the block into separate blocks. If the partition is stable, i.e., it is not necessary to refine it further, we terminate.

A *partition* $\mathcal{P}$ of S is a collection $\{Q_i\}_{i\in\mathcal{I}}$ of nonempty subsets of S such that $\bigcup_i Q_{i\in\mathcal{I}} = S$ and for any $i' \neq i : Q_{i'} \cap Q_i = \emptyset$. The elements of a partition are called *blocks*. Given a partition $\mathcal{P}$, we say that another partition $\mathcal{P}'$ *refines* $\mathcal{P}$, if any block of $\mathcal{P}'$ is included in a block of $\mathcal{P}$. We define $\mathcal{P}$-equivalence as follows: $c \sim_{\mathcal{P}} c' \Leftrightarrow \exists Q \in \mathcal{P}.\, c \in Q \wedge c' \in Q$, i.e., intuitively, c and c' are in the same block.

The *initial* partition $\mathcal{P}_0$ is constructed by categorizing states with the same valuation into blocks, i.e., $c \sim_{\mathcal{P}_0} c' \Leftrightarrow V(c) = V(c')$. In the refinement step, we split a block Q into two disjoint subblocks, one collects all states that are able to reach another block Q', while the other collects all states that cannot reach Q'. In this case, we call Q' a *splitter* of Q. Partition $\mathcal{P}$ is *stable* w.r.t. a block Q' if there is no block $Q \in \mathcal{P}$ such that Q' is a splitter of Q. $\mathcal{P}$ is *stable* if it is stable w.r.t. all its blocks.

Step 6: Inclusion check. Finally, we check if two initial states I and I' are in the same bisimilarity equivalence class. This will indirectly answer whether $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$ are bisimilar, i.e., they are bisimilar if two initial states fall into the same block, otherwise they are not.

5.4.2 Overall complexity

The stuttering closures in Step 2 can be computed in $O(n^3)$ using the all-pair shortest path algorithm. However, improved algorithms for doing so run in $O(n^{2.376})$ [32].

The algorithm by Paige and Tarjan computes the partition corresponding to strong bisimilarity in $O(m \cdot \log n)$ [71]. Thus, the complexity of verifying SSOD-2K is dominated by the determinization in Step 4, which is exponential in the number of states. However, Melichar et al [63] shows that the exponential complexity of determinization is often reached in only the extreme cases. The average complexity is mostly far lower than exponential. In [59], Lamperti et al. introduce an algorithm called *incremental subset construction* for determinizing a non-deterministic finite automata that is far more efficient than the traditional determinization by subset construction. Therefore, we believe that the verification of SSOD-2K is feasible in practice.

5.5 Verification of SSPOD-1

Following this algorithmic approach, this section discusses how the algorithms to verify SSOD can be adapted to verify SSPOD, a secure information flow property in the probabilistic context.

5.5.1 Algorithm

Given a program C, and a scheduler δ, SSPOD-1 requires that after projecting $\mathcal{A}_\delta$ on any low variable l, all traces that do not stutter forever in a non-final stuttering loop must be stuttering equivalent with probability 1. Thus, the verification of SSPOD-1 can be done via the following algorithm. Concretely, for each $l \in L$, we carry out the following steps.

```
Algorithm 6: SSPOD-1 on $l$
    1: Project $\mathcal{A}_\delta$ on $l$, yielding $\mathcal{A}_{\delta\,|_l}$.
    2: Remove all stuttering loops in $\mathcal{A}_{\delta\,|_l}$.
    3: Re-establish self-loops for final states of $\mathcal{A}_{\delta\,|_l}$. This yields a
       stuttering-loop free PKS, denoted $\mathcal{R}_{\delta\,|_l}$.
    4: Check whether all traces of $\mathcal{R}_{\delta\,|_l}$ are stuttering equivalent by:
        4.1: Choose a *witness* trace by:
            4.1.1: Take an arbitrary lasso $T$ of $\mathcal{R}_{\delta\,|_l}$.
            4.1.2: Remove stuttering steps and minimize $T$ by strong bisi-
                   mulation reduction.
        4.2: Check stuttering trace equivalence between $\mathcal{R}_{\delta\,|_l}$ and $T$ by
             checking if there exists a functional bisimulation between them.
```

We assume that data domains are finite, and schedulers use finite memory. Thus, Algorithm 6 works only on finite fully probabilistic PKSs — all determinism has been resolved by the scheduler — which can be viewed as Markov chains.

Algorithm 6 works, since we transform the probabilistic SSPOD-1 property into a possibilistic one. Key insight is that the probability of a trace that stutters forever in a *non-final* stuttering loop is 0. Therefore, after removing all non-final stuttering loops, it is sufficient to determine whether all traces are stuttering equivalent. Notice that a stuttering loop is non-final if it contains at least a state with an outgoing transition that can reach a non-equivalent state.

To perform **Step 1**, we label every state with the value of l in that state. To remove the stuttering loops in **Step 2**, we use a classical algorithm for finding strongly connected components w.r.t. stuttering steps [1], and collapse these components into a single state. To ensure that the transition relation remains non-blocking, **Step 3** re-establishes self-loops for final states. **Step 4.1.1** and **Step 4.1.2** are the same as the corresponding ones of Algorithm 1 to verify SSOD-1K.

Step 4.2 checks stuttering trace equivalence between a PKS $\mathcal{A}$ and the witness trace T by checking if there exists a functional bisimulation between them. Similarly to the verification of SSOD-1K, this is done by exploring the state space of $\mathcal{A}$, and building the mapping *Map* during the exploration. The main difference between the following algorithm (Algorithm 7) and Algorithm 4 is that in Algorithm 7, we do not check the divergent property of states, since Step 2 of Algorithm 6 has removed all of them, except the final states. Thus, in Algorithm 7, instead, we check whether a state is a final state or not.

Consider a state c and its *potential_map*. The algorithm returns *false*, i.e., *continue* = *false*, if (i) c and *potential_map* have different valuations, (ii) c is a final state of $\mathcal{A}$, while *potential_map* is not the final state of T, or (iii) c has been checked before, but its mapped state is not *potential_map*. If none of these cases occurs, and c was not checked before, c is added to $\mathcal{Q}$, and mapped to *potential_map*.

Algorithm 7: All-Trace Stuttering Equivalence($\mathcal{A}$, T) (SSPOD version)

```
for all states $c \in \mathcal{S}$ do $Map[c] := \bot$;
$continue := true$;
$\mathcal{Q} := empty\_queue()$; $enqueue(\mathcal{Q}, init\_state)$;
$Map[init\_state] := u_0$;        // $u_0$ is $T_0$
while $!empty(\mathcal{Q}) \wedge continue$ do
```

```
        $current := dequeue(\mathcal{Q})$;
        $u := Map[current]$;
        for all states $c \in Succ(\mathcal{A}, current)$ do
            $potential\_map := (c \sim_V current)\ ?\ u : Succ(T, u)$;
              case  $c \not\sim_V potential\_map \rightarrow continue := false$;
                []    $final(\mathcal{A}, c) \wedge \neg final(T, potential\_map) \rightarrow continue := false$;
                []    $Map[c] = \bot \rightarrow enqueue(\mathcal{Q}, c)$;
                                        $Map[c] := potential\_map$;
                []    $Map[c] \neq potential\_map \rightarrow continue := false$;
    return $continue$;
```

Example 5.2 *Figure 5.3 illustrates Step 1 - Step 4.1 on a PKS $\mathcal{A}$ consisting of 10 states. Step 1 projects $\mathcal{A}$ on a low variable l. Step 2 removes all stuttering loops, while Step 3 re-establishes self-loops for final states. Step 4.1 takes an arbitrary trace of $\mathcal{A}$, and then minimizes it. Each state of the witness trace T is denoted by a unique symbol u_i. Figure 5.4 illustrates Step 4.2. Initially, all states of $\mathcal{A}$ are mapped to a special symbol $\bot$. Next, state 0 is enqueued, and mapped to u_0. Next, Algorithm 7 examines all unchecked successors of state 0, i.e., states 1, 2, 3. Each of them follows a non-stuttering step; thus their potential_maps are all u_1. Since states 1, 2, and 3 have the same valuation as potential_map, i.e., **b**, they are all enqueued, and mapped to u_1. Next, the successor of state 1, i.e., state 4, is considered. The transition $1 \rightarrow 4$ is non-stuttering; thus potential_map $= u_2$. State 4 has the same valuation as potential_map, but it is a final state of $\mathcal{A}$, while potential_map is not the final state of T. Therefore, continue $=$ false. The PKS $\mathcal{A}$ and the witness trace T are not stuttering trace equivalent, since there exists a trace that stutters in state 4 forever. Hence, the algorithm terminates.*

Before proving the correctness of Algorithm 7, we first discuss its loop invariant.

Theorem 5.4 *Algorithm 7 preserves the following loop invariant:*

> *If continue, then $\forall c \in \mathcal{S}$ such that $Map[c] = u$, the trace from init_state to c and the prefix of T up to u are stuttering equivalent, and if $\neg$continue, then there exists a trace of $\mathcal{A}$ that is not stuttering equivalent to T.*

Proof: The proof of this theorem and the one we gave for Theorem 5.2 on page 75 are very much the same. However, instead of checking the divergent

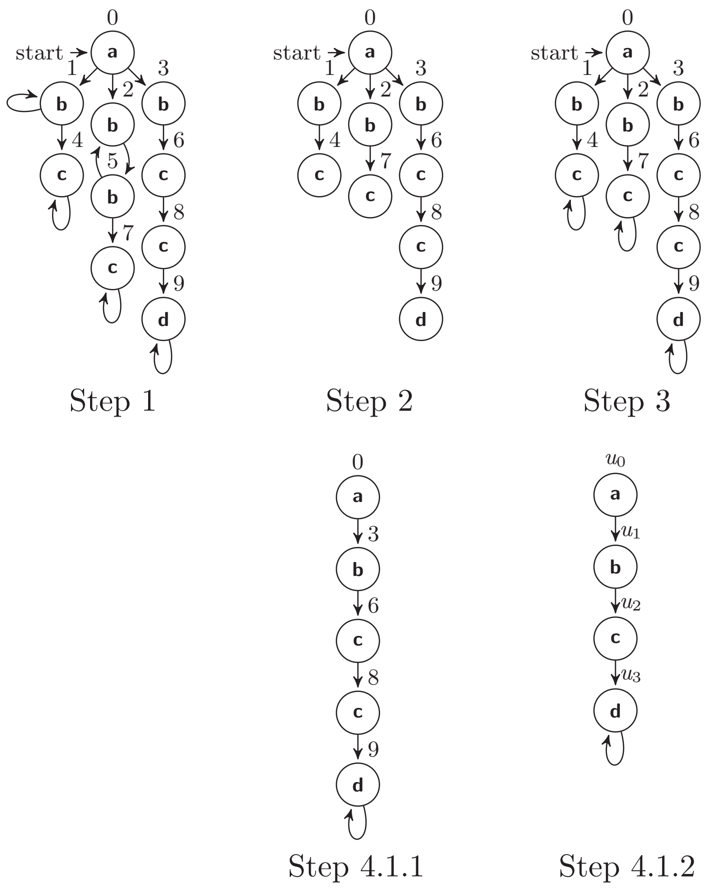


Figure 5.3: Step 1 - Step 4.1 of Algorithm 6

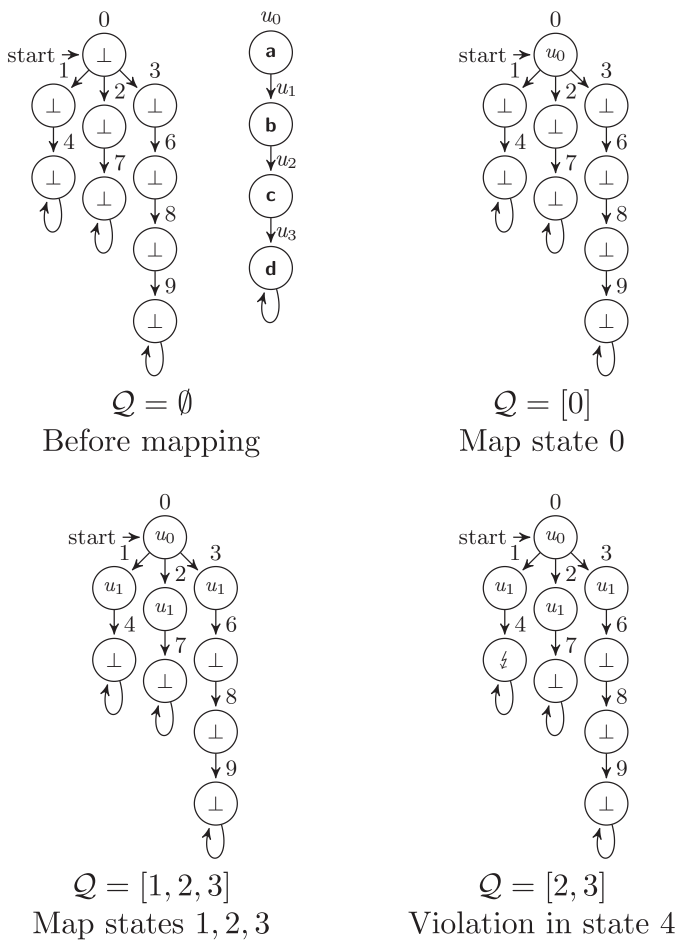


Figure 5.4: Step 4.2 of Algorithm 6

property as in the proof for Theorem 5.2, we check whether c is a final state while *potential_map* is not. That is:

Case $\mathit{final}(\mathcal{A}, c) \wedge \neg\mathit{final}(T, \mathit{potential\_map})$**.** Thus, *continue* becomes *false*. The invariant is preserved, since if c is a final state of $\mathcal{A}$, then there must be a trace that stutters in c forever, while T can make a step from *potential_map* to another state with a different valuation, due to the fact that T is stuttering-free.

Notice that in case $c \sim_V$ *potential_map*, and c is not a final state while *potential_map* is, the check still continues, since the traces of $\mathcal{A}$ are *not stuttering-free*, and the trace from the initial state to c is still stuttering equivalent to T. □

Theorem 5.5 *Algorithm 7 returns true iff there exists a bisimulation between a PKS $\mathcal{A}$ and T.*

Proof: See the proof for Theorem 5.3 on page 76. □

5.5.2 Overall correctness

Step 1 only changes the state labels of a PKS. Thus, the probability space of the PKS is unchanged. Hence, after projecting $\mathcal{A}_\delta$ on l, we can reformulate SSPOD-1 in terms of $\mathcal{A}_{\delta\,|_l}$. Let $(\Omega, \mathcal{F}, \mathbf{P}_{\delta,l})$ denote the probability space of $\mathcal{A}_{\delta\,|_l}$. First, we reformulate SSPOD-1, which talks about the traces of $\mathcal{A}_\delta$, in terms of the traces of the projected $\mathcal{A}_{\delta,l}$

Theorem 5.6 *For any $l \in L$, and for a set of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence, if $\mathbf{P}_{\delta,l}[X] = 1$ or $\mathbf{P}_{\delta,l}[X] = 0$, then SSPOD-1 holds.*

Proof: Let T be a trace of $\mathcal{A}_\delta$, and let $T_{\,|_l}$ denote the projection of T on l. First, notice that $\{T_{\,|_l} \mid T \in \mathit{Trace}(\mathcal{A}_\delta)\} = \mathit{Trace}(\mathcal{A}_{\delta\,|_l})$. Let $X \subseteq \mathit{Trace}(\mathcal{A}_\delta)$ such that X is closed under stuttering equivalence w.r.t. l. Clearly, also $X \subseteq \mathit{Trace}(\mathcal{A}_{\delta\,|_l})$, and X is closed under stuttering equivalence. Moreover, the probability space of $\mathcal{A}_\delta$ is preserved under the projection on l. Thus, if for any l, $\mathbf{P}_{\delta,l}[X] = 1$ or $\mathbf{P}_{\delta,l}[X] = 0$, then $\mathbf{P}_\delta[X] = 1$ or $\mathbf{P}_\delta[X] = 0$, respectively. Thus, SSPOD-1 holds. □

The key step (Step 2) of Algorithm 6 is the reduction of a *probabilistic* property to a *non-probabilistic* property, i.e., after removing all stuttering loops — removing all sets of traces X such that X stutters forever, and $\mathbf{P}_{\delta,l}[X] = 0$ —

if all traces of $\mathcal{A}_{\delta\,|_l}$ are stuttering equivalent, then $\mathbf{P}_{\delta,l}[X] = 1$. Thus, SSPOD-1 holds. The correctness of this step follows from a result from Baier and Kwiatkowska [14]:

> whenever *all fair traces* of a PKS fulfill a certain property φ, then φ holds *with probability* 1.

In our context, we define the fairness of traces w.r.t. *non-stuttering transitions.* A non-stuttering transition is *enabled* in a state T_i iff there exists a finite sequence of transitions from T_i that leads to T_j such that $V(T_j) \neq V(T_i)$. A non-stuttering transition is said to be *taken* in a state T_i of T iff $\exists j > 0.\, T_i \neq T_{i+j}$. A trace is *strongly fair* w.r.t. non-stuttering transitions if given that a non-stuttering transition is enabled *infinitely* often, it is taken *infinitely* often. Thus, a trace that stutters in a non-final stuttering loop forever is *unfair.* Let *Fair*$(\mathcal{A})$ denote the set of *fair* traces of *Trace*$(\mathcal{A})$. Applying the result from [14], we obtain:

Theorem 5.7 *Given a finite $\mathcal{A}_{\delta\,|_l}$. Consider a set of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence, and do not stutter forever in a non-final stuttering loop, if $\forall T, T' \in Fair(\mathcal{A}_{\delta\,|_l}).\, T \sim T'$, then $\mathbf{P}_{\delta,l}[X] = 1$.*

We show that after removing all stuttering loops, and re-establishing self-loops for final states, the set of fair traces of $\mathcal{A}$ is preserved.

Theorem 5.8 *Given a PKS $\mathcal{A}$. Let $\mathcal{R}$ denote the PKS that is obtained after removing all stuttering loops, and re-establishing self-loops for final states, $Fair(\mathcal{A}) = Trace(\mathcal{R})$.*

Proof: Let $\mathcal{L}oop$ be a non-final stuttering loop of $\mathcal{A}$. Since $\mathcal{L}oop$ is non-final, it contains at least a state with an outgoing transition that leads to a non-stuttering transition. From the definition of fair traces, any T that is *trapped* in $\mathcal{L}oop$ forever is *unfair.* Hence, removing all stuttering loops, and re-establishing self-loops for final states, preserve the set of fair traces of $\mathcal{A}$. □

Combining these results, we obtain.

Theorem 5.9 *For any $l \in L$, if all traces of $\mathcal{R}_{\delta\,|_l}$ are stuttering equivalent, then SSPOD-1 holds.*

5.5.3 Overall complexity

Step 1 labels every state of $\mathcal{A}$ by the value of l in that state. This is done in time complexity $O(n)$, where n is the number of states of $\mathcal{A}$. Step 2 uses an $O(m)$-algorithm to find the strongly connected components, where m is the number of transitions of $\mathcal{A}$. The time complexity of Step 4.1 is also $O(m)$. The core of Step 4.2 is the BFS algorithm, whose running time is $O(n+m)$. Therefore, for a single low variable l, the total time complexity of the verification is linear in the size of $\mathcal{A}$, i.e., $O(n+m)$, and for any initial state, the total complexity of the verification of SSPOD-1 (for all $l \in L$) is $|L| \cdot O(n+m)$.

5.6 Verification of SSPOD-2

5.6.1 Algorithm

SSPOD-2 states that, given a program C, for any two initial low-equivalent states I and I', if we project on the set of low variables L, the probabilistic languages arising from the executions of I and I' should be the same. A number of efficient algorithms for checking equivalence between probabilistic languages have been developed, the classical ones in [27, 89], and the improved variants in [35, 52]. However, none of the existing algorithms exactly fit our purposes, since either they do not abstract from stuttering steps [89, 35, 52], or they consider a different variation of probabilistic language inclusion [27].

Therefore, to verify SSPOD-2, our algorithm first transforms the PKS into an equivalent one, without *stuttering steps*, and then we use an efficient algorithm from Kiefer et al. [52] to check probabilistic-language equivalence. Concretely, we carry out the following steps.

```
Algorithm 8: SSPOD-2

1: Project both $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ on the set $L$, yielding $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$.
2: Remove all stuttering steps from $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$, yielding stuttering-
   free PKSs $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$.
3: Check the equivalence of the stuttering-free probabilistic languages
   between $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$, using Kiefer et al. [52].
```

Step 1 and **Step 2** of this algorithm are just the same as the ones of Algorithm 5 to verify SSOD-2K. To make this thesis self-contained, we present the main idea of the algorithm to check the equivalence of stuttering-free probabilistic languages in **Step 3** [52]. The basic idea of this algorithm is based on a result from [74]:

> Two probabilistic automata with a *combined number* of states n are equivalent iff they have the same *n-bounded language*, i.e., each word of length at most n is accepted with the same probability by both probabilistic automata, where each word denotes a sequence of computational operations.

Thus, we can represent the n-bounded language of a PKS by a *polynomial* in which each monomial presents an *input word* of the language, and the *coefficient* of the monomial represents the weight of the word, i.e., the probability of the execution of that sequence of operations. Two PKSs have the same probabilistic language iff both polynomials yield the same value for any word that is chosen independently and randomly. This method reduces a language equivalence problem to a polynomial identity testing.

Following this algorithm, we extend the notion of a PKS with a non-empty set of *alphabet* $\mathcal{E}$, where each alphabet symbol $\sigma \in \mathcal{E}$ denotes a computation operation that results in a state change. Let n denote the number of states of a *stuttering-free* PKS $\mathcal{R}$. We assign to each alphabet symbol $\sigma \in \mathcal{E}$ a transition matrix M of size $n \times n$, where $M(\sigma)[i, j]$ is the probability that $\mathcal{R}$ moves from state c_i to state c_j after executing σ. Let α denote an initial (row) vector of size n, i.e., setting $\alpha = (1, 0, \ldots, 0)$ to represent the initial state of $\mathcal{R}$. To present final states, we use n-dimensional (column) vectors η such that $\eta[i] = 1$ if c_i is a final state, and 0 otherwise.

The PKS $\mathcal{R}$ assigns each word $w = \sigma_1 \cdots \sigma_k$ a weight $\mathcal{R}(w)$, where $\mathcal{R}(w) = \alpha \cdot M(\sigma_0) \cdots M(\sigma_k) \cdot \eta$. Two PKSs $\mathcal{R}$ and $\mathcal{R}'$ over the same alphabet $\mathcal{E}$ are said to be probabilistic-language equivalent iff $\mathcal{R}(w) = \mathcal{R}'(w)$ for any word $w \in \mathcal{E}^*$.

Given two PKSs $\mathcal{R}$ and $\mathcal{R}'$, where n and n' are their numbers of states, respectively. Let $\mathbf{x} = \{x_{\sigma,i} : \sigma \in \mathcal{E}, 0 \leq i < n + n'\}$ be a family of variables. Each monomial $x_{\sigma_1,1}x_{\sigma_2,2} \cdots x_{\sigma_k,k}$ is associated with a word $w = \sigma_1 \cdots \sigma_k$ of length $k \leq n + n'$. Kiefer et al. represent the $n + n'$-bounded language of $\mathcal{R}$ by a polynomial $P^{(\mathcal{R})}(\mathbf{x})$, as follows.

$$P^{(\mathcal{R})}(\mathbf{x}) = \sum_{k=0}^{n+n'} \sum_{w \in \mathcal{E}^k} \mathcal{R}(w) \cdot x_{\sigma_1,1}x_{\sigma_2,2} \cdots x_{\sigma_k,k}$$

The polynomial $P^{(\mathcal{R}')}(\mathbf{x})$ is defined similarly. Thus, $P^{(\mathcal{R})} \equiv P^{(\mathcal{R}')}$ iff $\mathcal{R}$ and $\mathcal{R}'$ are probabilistic-language equivalent [52].

To test the equivalence of $P^{(\mathcal{R})}$ and $P^{(\mathcal{R}')}$, the algorithm selects a value for each variable $x_{\sigma_i,i}$ independently and uniformly at random from a set of rationals of size $K(n + n')$ — K is a constant. If $P^{(\mathcal{R})} \equiv P^{(\mathcal{R}')}$, then both polynomials will yield the same value. Otherwise, if $P^{(\mathcal{R})} \not\equiv P^{(\mathcal{R}')}$, the polynomials will yield different values with probability at least $\frac{K-1}{K}$. Notice that the following algorithm is in a backward direction, starting with the final state vector η and pre-multiplying by transition matrices. Readers can refer to [52] for the proof of correctness.

```
Algorithm 9: Probabilistic Language Equivalence ($\mathcal{R}, \mathcal{R}'$)
if $\alpha \cdot \eta \neq \alpha' \cdot \eta'$ then
    return: $\mathcal{R}$ and $\mathcal{R}'$ are not equivalent;
$\nu := \eta$; $\nu' := \eta'$;
for $i$ from 1 to $n + n'$ do
    Choose a random vector $r \in \{1, 2, \cdots, K(n + n')\}^{\mathcal{E}}$;
    $\nu := \sum_{\sigma \in \mathcal{E}} r_\sigma \cdot M(\sigma) \cdot \nu$;
    $\nu' := \sum_{\sigma \in \mathcal{E}} r_\sigma \cdot M'(\sigma) \cdot \nu'$;
    if $\alpha \cdot \nu \neq \alpha' \cdot \nu'$ then
        return: $\exists w$ with length $i$ such that $\mathcal{R}(w) \neq \mathcal{R}'(w)$;
return: $\mathcal{R}$ and $\mathcal{R}'$ are equivalent with probability at least $\frac{K-1}{K}$;
```

5.6.2 Overall correctness

After projecting both $\mathcal{A}_\delta$ and $\mathcal{A}'_\delta$ on L, we can reformulate SSPOD-2 in terms of $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$. Let $(\Omega, \mathcal{F}, \mathbf{P}_{\delta,L})$ and $(\Omega, \mathcal{F}, \mathbf{P}'_{\delta,L})$ denote the probability space of $\mathcal{A}_{\delta\,|_L}$ and $\mathcal{A}'_{\delta\,|_L}$, respectively.

Theorem 5.10 *SSPOD-2 holds iff for all sets of traces $X \in \mathcal{F}$ that are closed under stuttering equivalence, we have $\mathbf{P}_{\delta,L}[X] = \mathbf{P}'_{\delta,L}[X]$.*

Proof: Step 1 labels states of a PKS by the set of L-values. The probability space of the PKS is unchanged after this step. Thus, let $X \subseteq \mathit{Trace}(\mathcal{A}_\delta)$ such that X is closed under stuttering equivalence w.r.t. L. Clearly, also $X \subseteq \mathit{Trace}(\mathcal{A}_{\delta\,|_L})$, and X is closed under stuttering equivalence. Moreover, $\mathbf{P}_{\delta,L}[X] = \mathbf{P}_\delta[X]$. Thus, SSPOD-2 holds. □

Let $\mathcal{R}$ denote a stuttering-free PKS obtained by applying Step 2 on a given $\mathcal{A}$. Let $Tr_\mathcal{A}$ and $Tr_\mathcal{R}$ be probabilistic transition functions of $\mathcal{A}$ and $\mathcal{R}$, respectively. Step 2 removes all stuttering steps by changing $Tr_\mathcal{A}$ to $Tr_\mathcal{R}$, given by the following equations.

$$Tr_{\mathcal{R}}(c, c') = \begin{cases} Tr_{\mathcal{A}}(c, c') & \text{if } V(c) \neq V(c') \\ \sum_{c'':V(c)=V(c'')} Tr_{\mathcal{A}}(c, c'')\ Tr_{\mathcal{R}}(c'', c') & \text{otherwise.} \end{cases}$$

Thus, for non-stuttering steps, $Tr_{\mathcal{A}}$ and $Tr_{\mathcal{R}}$ are the same; for stuttering steps, $Tr_{\mathcal{R}}$ *accumulates* the probabilities of moving to c' via some stuttering steps $c \to c''$. Thus, $Tr_{\mathcal{R}}$ transforms the transition probabilities of stuttering steps in $\mathcal{A}$ into the transition probabilities of non-stuttering steps in $\mathcal{R}$. Therefore, removing stuttering steps does not change the probabilities of sets of traces that are closed under stuttering equivalence.

Theorem 5.11 *Let $X \in \mathcal{F}$ be a set of traces that are closed under stuttering equivalence, then $\mathbf{P}_{\mathcal{A}}[X] = \mathbf{P}_{\mathcal{R}}[X]$.*

Combining all results, it is obvious that to check SSPOD-2, we can check for probabilistic language equivalence between $\mathcal{R}_{\delta\,|_L}$ and $\mathcal{R}'_{\delta\,|_L}$.

5.6.3 Overall complexity

Step 1 is done in time complexity $O(n)$, where n is the number of states of two PKSs. Step 2 essentially calculates probability of reachability, and is defined as a system of n linear equations over n variables. This equation system can be solved in $O(n^3)$. Step 3 can be done in $O(nm)$, where m is the number of transitions [52]. Thus, the overall complexity is $O(n^3)$ for each pair of initial states I and I'.

5.7 Related work

This section reviews the existing approaches of analyzing information flow of multi-threaded programs, and compares them with our approach. Analyzing information flow of a program can be done *dynamically* or *statically. Dynamic analysis* attempts to analyze information flow within a program while it is executing. An interesting class of dynamic analysis is dynamic runtime monitoring [60, 81, 95, 40]. This approach follows the control flow of a program execution so that calculation of control dependences can be more accurate than other dynamic approaches. However, this approach is often computational and storage overhead, and also weak at identifying implicit information flows, in comparison with static analysis [78].

In general, dynamic approaches do not *precisely* enforce the confidentiality property, e.g., noninterference, since (1) they only have information about how a program behaves in a single execution, while confidentiality is a property concerning all possible execution paths, and (2) they do not also have sufficient information to predict future steps of the execution. Recently, the improved dynamic approaches in [11, 12, 79, 13] show that the purely dynamic runtime monitoring can enforce *termination-insensitive* noninterference, i.e., noninterference that ignores information flow related to the termination behavior of the program. However, as argued in Section 3.2, a security property that does not consider termination leaks is weak, since termination leaks might be serious. Also notice that the termination behavior is hard to control dynamically [77]. Thus, to control this behavior, it is not surprising that a dynamic mechanism would have to be extremely conservative.

Static analysis, such as *type systems*, which attempts to analyze information flow prior to the program execution, avoids many problems associated with dynamic analysis. Static analysis also has the advantage of having much more information about the program behavior than what can be observed from a single execution, as in dynamic analysis. Another significant advantage of static techniques is their ability to accurately track implicit information flows.

In comparison between dynamic and static analysis, Russo et al. [77] show that dynamic information flow enforcement is only more permissive than static analysis in the *flow-insensitive* confidentiality properties. Notice that our properties are *flow-sensitive*. For flow-sensitive security properties, Russo et al. show that any purely dynamic monitor fails to be more permissive than type systems.

To verify confidentiality for multi-threaded programs, Zdancewic and Myers, Sabelfeld et al., Terauchi, Smith, Kobayashi, and Russo et al. [96, 80, 87, 82, 53, 76] use type systems. However, Sabelfeld et al., and Russo et al. consider explicitly the role of schedulers in their analysis. Russo et al. restrict the allowed interactions between threads and the scheduler; thus, this allows them to present a compositional security type system which guarantees confidentiality for a wide class of schedulers [76]. Notice that Russo et al. propose a security specification that is similar to noninterference, just considering final outcomes of the execution.

Type-based approaches are efficient, since they are often *polynomial* in the size of the program. However, as discussed before, they are not suited to verifying existential properties and stuttering equivalence, as in SSOD and SSPOD. Just as we expected, since the formalizations of SSOD and SSPOD are more complicated than other confidentiality properties discussed in Chapter 3, the verifications of our notions have higher worse-case complexity — due to the high complexity of the verifications of SSOD-2 and SSPOD-2. However, if we apply our algorithmic techniques to verify other notions of observational determinism [96, 48, 87], which do not contain the existential properties, the time complexity is only *linear* in the size of the Kripke structure modeling the program.

Huisman et al. also characterize the information flow properties as temporal-logic formulas [48], based on self-composition. In particular, Huisman et al. characterize observational determinism in CTL*, using a special non-standard synchronous composition operator, and also in the polyadic modal μ-calculus (a variation of the modal μ-calculus) [48]. In later work, Huisman and Blondeel [45] have shown that this approach is feasible, by developing an encoding in Concurrency WorkBench, and applying this on several small examples. However, the encoding is very *ad hoc*, and this is not a general solution to the problem of efficiently verifying information flow properties, as in our approach.

The way self-composition is used in this chapter and in [48], with a temporal-logic characterization, also bears resemblance with temporal-logic characterizations of strong bisimulation [72].

Also based on the idea of self-composition, Darvas et al. [34] characterize non-interference by a *general program logic.* This paper shows that program logics based on simple safety and liveness properties are inadequate for this purpose. Thus, they propose to use dynamic logic. The feasibility of this approach is investigated by showing that a general purpose theorem proving tool for software verification, such as the KeY tool, can be used to analyze information flow of a program. Notice that dynamic logic is applicable only to sequential deterministic programs, where the information flow property considers only the final state of the execution.

Alur et al. [2] enrich the traditional tree model, whose paths correspond to possible executions of the system, with labeled edges that capture observational indistinguishability between nodes. They convert the tree into a Kripke structure, where both nodes and edges have labels, and interpret temporal logics over it. This enriched model is expressive enough to specify information flow properties in temporal logics. In this approach, a specification describes a set of correct trees; and thus, the verification of secrecy requirements of software systems reduces to a membership problem.

In later work, Černý and Alur [91] consider a weaker class of properties, so-called conditional confidentiality properties for sequential programs, and develop a sound automated analysis for this, based on a combination of under- and over-approximations. Basically, the proposed verification method analyzes a program to produce a logical formula that characterizes the confidentiality requirement. The resulting formula can be discharged by using existing SMT tools. The practical impact of this work is illustrated by applying it on Java2ME.

Following the algorithmic approach, Giffhorn et al. [37] propose a verification method based on program dependence graphs to check observational determinism. A program dependence graph models information flow through a program, where nodes are program statements, and edges are data dependences or control dependences. This verification method has a rather high time complexity, i.e., $O(n^3)$, while our algorithm to check all-trace stuttering equivalence has a linear time complexity in the size of the graph modeling the program. Notice that the definition of observational determinism in this paper is very similar to SSPOD-1. However, in their approach, there is no *global security classification* of variables, i.e., a variable at one program point may contain a low value, but at another point a high value. Thus, their trace definition is based on *low operations*, i.e., read or write on a low variable, instead of low values as in the traditional approaches.

Finally, Van der Meyden and Zhang [90] develop algorithmic verification techniques on state-based models for a number of different *noninterference* notions, and characterize the computational complexity of the associated verification problems. This work also shows that noninterference in deterministic finite state systems can be reduced to a *safety* property; thus it is verifiable in polynomial time by existing model checkers.

Notice that Groote and Vaandrager [39] propose algorithms for the Relational Coarsest Partition with stuttering problem that is closely related to the problem of deciding stuttering equivalence on finite Kripke structures. The algorithms of Groote and Vaandrager are based on the partition refinement technique. These algorithms can be adapted to verify SSOD-1 and SSPOD-1. Basically, our algorithms — presented in Chapter 5 — and their algorithms have the same time and space complexities, i.e., they are linear in the size of the automaton that models the program. However, when generating counter-examples, in practice, our algorithms are more efficient. The partition refinement technique can be used to indicate whether the automaton contains an *unmatched* state, but then, we have to use BFS to locate this unmatched state. Our algorithms are only based on BFS, and thus immediately return a counter-example.

5.8 Conclusions

This chapter discussed efficient algorithmic verifications of SSOD and SSPOD. The algorithmic method is based on a combination of novel and existing model-checking techniques, i.e., on techniques to check stuttering equivalence and trace equivalence of Kripke structures. The newly developed algorithms have a broad application. In particular, other formalizations of observational determinism [96, 48, 87] can also be verified by minor modifications of our algorithms, since all of them are based on the notion of stuttering equivalence. Besides, as mentioned before, the newly-developed algorithms are also relevant outside the security context, since stuttering equivalence is the fundamental concept of concurrent and distributed systems.

Notice that the algorithm to check SSPOD-1 borrows ideas from the one we developed for SSOD-1, but it solves a completely different problem: SSOD-1 checks if all traces are stuttering equivalent, while SSPOD-1 in essence checks if all traces are stuttering equivalent *with probability 1*.

The goal of the algorithms presented in this chapter is to determine whether a program reveals secret information during its execution. Thus, they only give binary answers: secure or insecure. However, these algorithms can also be adapted to give us more useful information, i.e., the reasons why the program fails a confidentiality requirement. This will be discussed in Chapter 6.

In comparison with the logic-based approach, the program models in the algorithmic method are far more simple. Thus, algorithmic verification approaches are capable of handling large state-space systems; which makes the verification of real-world applications feasible. Chapter 7 will discuss the feasibility of this approach and its practical capability on some case studies.

Chapter 6

Attack Synthesis

6.1 Introduction

In the previous chapter, we have seen how algorithmic techniques can be used to efficiently verify information flow properties. Another advantage of using algorithmic techniques is that we can *synthesize attacks* for insecure programs, based on counter-example generation techniques. If a program does not satisfy a confidentiality requirement, its execution might contain a security hole. Since the verification algorithm is precise, if it fails, a counter-example can be produced. The counter-example describes a possible attack on this security hole of the program; and thus, it reveals reasons why the program is rejected by the security policy. The counter-example also helps us to recognize which kind of scheduling policies could make the program reveal secret information. This chapter describes how the verification algorithms presented in Chapter 5 can be adapted to produce counter-examples for rejected programs.

Counter-example generation is a powerful technique in model checking, with many applications, such as diagnosis, scheduler synthesis, and debugging [29, 24, 43, 8]. However, to the best of our knowledge, this idea of applying counter-example generation to synthesize confidentiality attacks for multi-threaded programs has not previously been mentioned in the literature.

Organization of the chapter. Attack synthesis for SSOD is discussed in Section 6.2 and Section 6.3, while attack synthesis for SSPOD is in Section 6.4 and Section 6.5. Finally, Section 6.6 concludes the chapter.

Origins of the chapter. The idea and algorithms for attack synthesis for SSOD were published in the *Journal of Computer Security* (JCS) [69]. The algorithms for attack synthesis for SSPOD-1 are based on the ones developed for SSOD-1K. To synthesize attacks for SSPOD-2, we apply an existing algorithm from [52].

6.2 Attack synthesis for SSOD-1K

Algorithm 4 on page 68 checks whether all traces of a Kripke structure (KS) is stuttering equivalent to a witness trace. This section extends this algorithm, which results in Algorithm 10, to return a trace that is not stuttering equivalent to the witness trace in case the given KS is not stuttering-trace equivalent. Basically, Algorithms 4 and 10 are similar; however, when a state that violates SSOD-1K is found, Algorithm 10 does not terminate, as Algorithm 4 does, but continues until it finds an *unmatched trace.* The unmatched trace is returned via Algorithm 12 — **Trace Return** — given below.

State c is denoted as an *unmatched* state if it is not equivalent to its corresponding state $potential\_map$. State c is also *unmatched* if c is equivalent to $potential\_map$, but c is divergent, while $potential\_map$ is not. Whenever an *unmatched* state is found, SSOD-1K is not satisfied. A counter-example trace is the trace from the initial state I to the unmatched state. Since the search is based on BFS, the trace is the shortest path from I to the unmatched state (line 11 in Algorithm 10). Notice that if c is equivalent to $potential\_map$, and c is *not divergent* but $potential\_map$ is, it is clear that $\mathcal{A}$ does not satisfy SSOD-1K. However, the trace from I up to c is not a counter-example, since it is still stuttering equivalent to the witness trace T. Hence, in this case, state c is treated as a *normal* state, i.e., if c is an unchecked state, it is still assigned the label $potential\_map$, and the algorithm continues until a real counter-example is found (lines 13-15).

Consider a situation that the check hits a state c that has been checked before. If $c \sim_V potential\_map$ but $Map[c] \neq potential\_map$, we consider that c is an *abnormal checked* state, denoted *abnormal* (lines 16-18). It means that there exist at least two traces that both lead to *abnormal*; and *abnormal* corresponds to two different states of T. Notice that one of them hits *abnormal* via *current*, thus we denote *current* as $pre\_abnormal$.

Figure 6.1 explains this situation. According to the algorithm, state 1 and state 2 are both mapped to u_1. Next, state 4 is also mapped to u_1, since it follows

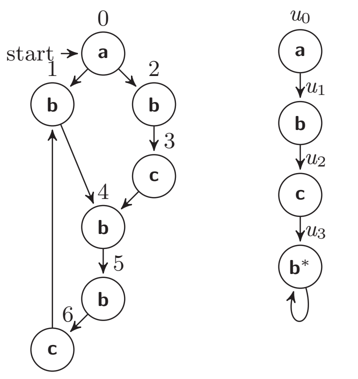


Figure 6.1: Abnormal state

state 1 via a stuttering step. In this check, we store *parent*[state 4] = state 1. Next, the algorithm examines the successor of state 2, i.e., state 3, and maps state 3 to u_2; since this is a non-stuttering transition and state 3 is equivalent to u_2.

When the algorithm examines the successor of state 3, which is state 4, the *potential_map* is u_3, since this is a non-stuttering transition. However, state 4 has been explored before, and its current assigned label is u_1. Since the current label of state 4 is different from *potential_map*, but state 4 is still equivalent to *potential_map*, state 4 is an *abnormal checked* state. Notice that if state 4 and *potential_map* are *not* equivalent, state 4 is an unmatched state. State 3 is the *pre_abnormal* state, since in this check, state 4 is its successor. However, *parent*[state 4] still indicates that state 1 is the parent of state 4, i.e., referring to the step of checking the successor of state 1.

When an *abnormal* is found, SSOD-1K is not satisfied. However, neither of the two traces from I — denoted by *init_state* in the algorithm — up to *abnormal*, e.g., **abb** and **abcb** in Figure 6.1, is a complete counter-example. Actually, one of them is a prefix of a real counter-example.

Therefore, the function **Trace Return** returns two traces: P_1 that is from *init_state* to *parent*[*abnormal*], e.g., $P_1 = \mathbf{ab}$, and P_2 that is from *init_state* to *pre_abnormal*, e.g., $P_2 = \mathbf{abc}$. The current BFS check is stopped. The new check starts from *abnormal* (given by Algorithm 13), in which P_1 is extended to a lasso. In this new check, if an unmatched state is found, we are done; otherwise, a complete lasso $T1$ is obtained, i.e., $T1 = P_1 + P_3$. *In Figure 6.1,* $T1 = \boldsymbol{a}(\boldsymbol{bbbc})^\omega$. If $T1$ is *not* stuttering equivalent to the witness trace T, it is a counter-example; otherwise, the counter-example is the lasso $T2$ that is an extension of $P_2 + P_3$. The reason is that three traces T, $T1$, and $T2$ cannot be all stuttering equivalent to each other. *In Figure 6.1,* $T1$ *is the counter-example.*

If no counter-example is found after all reachable states from the initial state have been explored, SSOD-1K is satisfied (see Algorithm 10).

Algorithm 10: Attack Synthesis for SSOD-1K ($\mathcal{A}$, T)

```
1.   for all states $c \in \mathcal{S}$ do $Map[c] := \bot$;
2.   $continue := true$;
3.   $\mathcal{Q} := empty\_queue()$; $enqueue(\mathcal{Q}, init\_state)$;
4.   $parent[init\_state] := root$;      // to indicate the start of traces
5.   $Map[init\_state] := u_0$;      // $u_0$ is $T_0$
6.   while $!empty(\mathcal{Q}) \wedge continue$ do
7.       $current := dequeue(\mathcal{Q})$;
8.       $u := Map[current]$;
9.       for all states $c \in Succ(\mathcal{A}, current)$ do
10.          $potential\_map := (c \sim_V current)\ ?\ u : Succ(T, u)$;
11.          Unmatched-State Check 1 $(c, potential\_map, current)$
12.          case
13.             $[\!]\ Map[c] = \bot \rightarrow\ enqueue(\mathcal{Q}, c)$;
14.                                 $parent[c] := current$;
15.                                 $Map[c] := potential\_map$;
16.             $[\!]\ Map[c] \neq potential\_map \rightarrow$
17.                     $continue := false$;
18.                     Abnormal-State Check $(current, c)$;
19.   return $continue$;
```

The function **Unmatched-State Check 1** $(c, potential\_map, current)$ returns a counter-example if c is an unmatched state (see Algorithm 11).

Algorithm 11: Unmatched-State Check 1 (c, $potential\_map$, $current$)

```
if $c \not\sim_V potential\_map$ or
   $Divergent[c] = true \wedge Divergent[potential\_map] = false$ then
            $parent[c] := current$;
            $continue := false$;
            Counter-Example = Trace Return$(init\_state, c)$;
```

The function **Trace Return** (a, b) returns a finite trace from state a to state b. Notice that the stack *trace* stores the trace backwards, i.e., a is the last element that enters the stack (see Algorithm 12).

Algorithm 12: Trace Return (a, b)

```
    $trace := empty\_stack()$;    // $trace$ is a stack
    PUSH $(trace, b)$;    // Add an item to the stack
    $parent\_value := parent[b]$;
    **while** $parent\_value \neq parent[a]$ **do**
        PUSH$(trace, parent\_value)$;
        $parent\_value := parent[parent\_value]$;
    **while** $!empty(trace)$ **do**
        **return** POP $(trace)$; // Remove an item from the top of the stack
```

The function **Abnormal-State Check** $(pre\_abnormal, abnormal)$ returns a counter-example when an *abnormal* state is found (see Algorithm 13).

Algorithm 13: Abnormal-State Check $(pre\_abnormal, abnormal)$

```
    Let $P_1$ = **Trace Return**$(init\_state, parent[abnormal])$;
    Let $P_2$ = **Trace Return**$(init\_state, pre\_abnormal)$;
// Except states on $P_1$, erase the labels and parents of other states
    **for** all states $c \in \mathcal{S} \wedge c \notin P_1$ **do**
        $Map[c] := \bot$;
        $parent[c] := \bot$;
// The new check starts from $abnormal$
    $current := abnormal$;
    $u := Map[abnormal]$;
    $c :=$ some state $\in Succ(\mathcal{A}, current)$;
// Extend $P_1$ to a lasso
    **while** $Map[c] = \bot$ **do**
        $potential\_map := (c \sim_V current)\ ?\ u : Succ(T, u)$;
    // if an unmatched state is found, we are done
        **Unmatched-State Check 1** $(c, potential\_map, current)$;
        $parent[c] := current$;
        $Map[c] := potential\_map$;
        $current := c$;
        $u := potential\_map$
        $c :=$ some state $\in Succ(\mathcal{A}, current)$
// Return $T_1$ if it is not stuttering equivalent to $T$
    **if** $Map[c] \neq potential\_map$ **then**
        Counter-Example = $P_1$ + **Trace Return**$(abnormal, c)$;
```

```
//  Return $T_2$ if $T_1$ is stuttering equivalent to $T$
    **if** $Map[c] = potential\_map$ **then**
        **if** $c \in P_1$ **then**
            Counter-Example = $P_2$ + **Trace Return**$(abnormal, c)$ +
                                      **Trace Return**$(c, abnormal)$;
        **else** Counter-Example = $P_2$ + **Trace Return**$(abnormal, c)$;
```

Notice that in case $c \in P_1$, P_2 + **Trace Return**$(abnormal, c)$ is not a complete lasso (see Figure 6.1 for a visual example), a counter-example should be P_2 + **Trace Return**$(abnormal, c)$ + **Trace Return**$(c, abnormal)$.

Theorem 6.1 *In case a Kripke structure $\mathcal{A}$ does not satisfy SSOD-1K, Algorithm 10 returns a trace that is not stuttering equivalent to the witness trace T.*

Proof: Algorithm 10 is a variant of Algorithm 4 whose correctness has been proved. Here we show that the algorithm produces a correct counter-example.

Case $c \not\sim_V potential\_map$**.** It is clear that the trace from *init_state* to c is an counter-example.

Case $Divergent[c] = true$ **and** $Divergent[potential\_map] = false$**.** Since c is a divergent state, there exists a trace that goes from *init_state* to c, and then stutters in c forever. State *potential_map* is not divergent, thus, it is not the final state of T. Since T is stuttering-free, it can evolve to a state that is not equivalent to *potential_map*. Therefore, the trace from *init_state* to c is an counter-example.

Otherwise,. Let P_1 = **Trace Return** $(init\_state, parent[abnormal])$, and P_2 = **Trace Return** $(init\_state, pre\_abnormal)$. Let $T1$ denote a lasso that is an extension of P_1 from *abnormal*, i.e., $T1 = P_1 + P_3$, where P_3 is the extension. While extending P_1, if an unmatched state is found, we are done. Otherwise, when the lasso T_1 is complete, i.e., when $Map[c] \neq \perp$, one of the two following scenarios occurs.

Case $Map[c] \neq potential\_map$**.** $T1$ is not stuttering equivalent to T, since c corresponds to two different states of T.

Case $Map[c] = potential\_map$**.** It is clear that T_1 is stuttering equivalent to a prefix of T, i.e., the prefix up to *potential_map*. We show that $T1$ is actually stuttering equivalent to the whole T. Given that c

has been checked before, then c is the start and end of a loop that terminates the lasso T_1.

Case c is not divergent. Since c is mapped to *potential_map* twice, *potential_map* is also the start and end of a loop that terminates T. Thus $T1 \sim T$.

Case c is a divergent state. Thus, *potential_map* is also divergent. Therefore, *potential_map* is the final state of T; then $T1 \sim T$.

Let T_2 denote an extension of $P_2 + P_3$ to a lasso. Since T is deterministic stuttering-free, and *abnormal* is mapped to two different states of T, $T1$ and $T2$ cannot be both stuttering equivalent to T. Since $T1 \sim T$, T_2 is the counter-example.

$\square$

6.3 Attack synthesis for SSOD-2K

Given two deterministic stuttering-free Kripke structures $\mathcal{R}$ and $\mathcal{R}'$, if traces of $\mathcal{R}$ and $\mathcal{R}'$ do not satisfy the SSOD-2K requirement, Algorithm 14 and **Parent Return** will output a trace of $\mathcal{R}$ that does not match with any trace of $\mathcal{R}'$, or vice versa.

It is clear that for $\mathcal{R}$ and $\mathcal{R}'$, any two strongly bisimilar states must be in the same block. We denote a *pair* of states of $\mathcal{R}$ and $\mathcal{R}'$ to be *valid* if they have the same label, but they are not in the same block of the *stable partition* $\mathcal{P}$ given by the algorithm computing bisimilarity equivalence classes in Section 5.4.

Given a *valid* pair (c, c') ($c \in \mathcal{S}_{\mathcal{R}}$, $c' \in \mathcal{S}_{\mathcal{R}'}$, where $\mathcal{S}_{\mathcal{R}}$, $\mathcal{S}_{\mathcal{R}'}$ are state sets of $\mathcal{R}$ and $\mathcal{R}'$, respectively). Similarly, two successors of (c, c') are also *valid* if they have the same label, but they are not strongly bisimilar. Therefore, from an initial state, a counter-example trace must pass states in a sequence of valid pairs until it hits a state such that the other trace (from the other initial state) cannot find a successor to form a *valid* pair. The shortest counter-example trace is found based on BFS.

Given two valid states $c \in \mathcal{S}_{\mathcal{R}}$ and $c' \in \mathcal{S}_{\mathcal{R}'}$. Let **SameBlock**$(c, c')$ be a predicate to check whether c and c' are in the same block of the given partition, i.e., **SameBlock** (c, c') $\{$**return** $(\exists Q \in \mathcal{P}. c \in Q \wedge c' \in Q)\}$. We formally define **Valid** (c, c') $\{$**return** $(c \in \mathcal{S}_{\mathcal{R}} \wedge c' \in \mathcal{S}_{\mathcal{R}'} \wedge V(c) = V(c') \wedge \neg$**SameBlock** $(c, c'))\}$, and the set *Valid_Succ*(c, c') of valid successors of (c, c') as follows,

$$\mathit{Valid\_Succ}(c, c') = \{(d, d') \mid c \in \mathcal{S}_{\mathcal{R}}, c' \in \mathcal{S}_{\mathcal{R}'}. c \to d, c' \to d' \wedge \textbf{Valid}\ (d, d')\}.$$

We also define a predicate **Unmatched Succ** on a valid pair (c, c') to (1) check whether either c or c' can take a transition to a state d such that none of the other state's successors is equivalent to d, and to (2) return d, if d exists. Formally,

```
Unmatched Succ $(c, c')$ {
        if ( $c \to d \wedge \nexists d' \in \mathcal{S}_{\mathcal{R}'}.\ c' \to d' \wedge V(c') = V(c)$) then return ($d$)
        if ( $c' \to d' \wedge \nexists d \in \mathcal{S}_{\mathcal{R}}.\ c \to d \wedge V(c) = V(c')$) then return ($d'$) }
```

Therefore, given the stable partition $\mathcal{P}$, the following algorithm explores the state spaces and returns the first state given by **Unmatched Succ**.

Algorithm 14: Invalid-State Search ($\mathcal{P}$)

```
$\mathcal{Q}_{\mathcal{R}}[0]$ := initial state of $\mathcal{R}$;
$parent_{\mathcal{R}}[\mathcal{Q}_{\mathcal{R}}[0]] := root$;
$\mathcal{Q}_{\mathcal{R}'}[0]$ := initial state of $\mathcal{R}'$;
$parent_{\mathcal{R}'}[\mathcal{Q}_{\mathcal{R}'}[0]] := root$;
$i := 0$;      // $i$: position of the latest items in both $\mathcal{Q}_{\mathcal{R}}$ and $\mathcal{Q}_{\mathcal{R}'}$
while (true) do
    Unmatched Succ ($\mathcal{Q}_{\mathcal{R}}[i], \mathcal{Q}_{\mathcal{R}'}[i]$);
    for each $(c, c') \in Valid\_Succ(\mathcal{Q}_{\mathcal{R}'}[i], \mathcal{Q}_{\mathcal{R}'}[i])$ do
        $\mathcal{Q}_{\mathcal{R}}[i+1] := c$;
        $parent_{\mathcal{R}}[\mathcal{Q}_{\mathcal{R}}[i+1]] := \mathcal{Q}_{\mathcal{R}}[i]$;
        $\mathcal{Q}_{\mathcal{R}'}[i+1] := c'$;
        $parent_{\mathcal{R}'}[\mathcal{Q}_{\mathcal{R}'}[i+1]] := \mathcal{Q}_{\mathcal{R}'}[i]$;
    $i := i + 1$;
```

Then **Parent Return** outputs the trace from the initial state up to the state returned by the algorithm.

Theorem 6.2 *In case $\mathcal{R}$ and $\mathcal{R}'$ do not satisfy SSOD-2K, Algorithm 14 and* ***Parent Return*** *return a trace of $\mathcal{R}$ that does not match with any trace of $\mathcal{R}'$, or vice versa.*

Proof: Two queues $\mathcal{Q}_{\mathcal{R}}$ and $\mathcal{Q}_{\mathcal{R}'}$ store pairs of valid successors reachable from two initial states. When the first unmatched state is found, the trace from the initial state to this state is returned. Due to BFS, this trace is one of the shortest paths from the initial state to it.

Notice that in case no state is returned by **Unmatched Succ** $(\mathcal{Q}_{\mathcal{R}}[i], \mathcal{Q}_{\mathcal{R}'}[i])$, for each successor of $\mathcal{Q}_{\mathcal{R}}[i]$, there must exist a successor of $\mathcal{Q}_{\mathcal{R}'}[i]$ such that these two states have the same label, and vice versa. If none of these pairs is *valid*, i.e., two states of any pair are bisimilar, then $\mathcal{Q}_{\mathcal{R}}[i]$ and $\mathcal{Q}_{\mathcal{R}'}[i]$ are also bisimilar. This is a contradiction, since $\mathcal{Q}_{\mathcal{R}}[i]$ and $\mathcal{Q}_{\mathcal{R}'}[i]$ are a *valid pair*. Thus, the set $\mathit{Valid\_Succ}(\mathcal{Q}_{\mathcal{R}}[i], \mathcal{Q}_{\mathcal{R}'}[i])$ is not empty; and the algorithm continues until an unmatched state is found. $\square$

Now, we have to derive the original traces of $\mathcal{A}$ that correspond to the trace of $\mathcal{R}$ given by Algorithm 14, since these original traces are the real counter-examples.

Derive the original traces of $\mathcal{A}$ from a trace of $\mathcal{R}$. Suppose that $\mathcal{R}$ is the corresponding deterministic Kripke structure of the stuttering-free $\mathcal{A}^{sf}$. We define a relation R between a transition of $\mathcal{A}^{sf}$ and a transition of $\mathcal{R}$, i.e., $\mathsf{R} \subseteq \rightarrow_{\mathcal{A}^{sf}} \times \rightarrow_{\mathcal{R}}$ as follows. Let *trn* denote a transition, *source*(*trn*) and *dest*(*trn*) the source and the destination state of *trn*.

$$\forall trn \in \rightarrow_{\mathcal{A}^{sf}}, trn' \in \rightarrow_{\mathcal{R}} \,.\; trn \,\mathsf{R}\, trn' \Leftrightarrow \begin{array}{l} V(source(trn)) = V(source(trn')) \wedge \\ V(dest(trn)) = V(dest(trn')). \end{array}$$

Given a trace T of $\mathcal{R}$, the following algorithm derives a set $\mathit{Trace}(T) = \{T'\}$ of $\mathcal{A}^{sf}$ corresponding to T. Let N denote the number of transitions in T. We rewrite T as an array of N transitions, i.e., $T = T[0], T[1], \ldots, T[N-1]$, where $T[i] = (T_i, T_{i+1})$. We implement an array *States* of size $N+1$, where each element *States*$[i]$ is a set of states of $\mathcal{A}^{sf}$, i.e., $\mathit{States}[i] \subseteq \mathcal{S}_{\mathcal{A}^{sf}}$.

```
Algorithm 15: Trace Map ($T$)
    $States[0]$ := initial state of $\mathcal{A}^{sf}$;
    for $i := 1$ to $N$ do
        $States[i] := \{c' \in \mathcal{S}_{\mathcal{A}^{sf}} \mid \exists c \in States[i-1].\ c \rightarrow_{\mathcal{A}^{sf}} c' \wedge (c, c')\ \mathsf{R}\ T[i-1]\}$;
    for each $c \in States[N]$ do
        $current := c$;
        for $i := N-1$ to $0$ do
            Take $c' \in States[i] \wedge (c', current)\ \mathsf{R}\ T[i]$;
            $T'[i] := (c', current)$;
            $current := c'$;
```

Traces of $\mathcal{A}$ can be derived easily, since we assumed that states are numbered and the transition relation between states is accessible.

6.4 Attack synthesis for SSPOD-1

The algorithms of attack synthesis for SSOD-1K can also be used for SSPOD-1. However, for SSPOD-1, we define an *unmatched* state in a slightly different way. State c is *unmatched* if it is not equivalent to its corresponding state *potential_map*. State c is also *unmatched* if c is equivalent to *potential_map*, but c is a final state, while *potential_map* is not.

Thus, if we replace Algorithm 11: **Unmatched-State Check 1** with the following algorithm, i.e., **Unmatched-State Check 2**, in Algorithm 10 and Algorithm 13, these two algorithms can be used to generate counter-examples for SSPOD-1.

```
Algorithm 16: Unmatched-State Check 2 ($c$, $potential\_map$, $current$)
    if $c \not\sim_V potential\_map$ or
       $final(\mathcal{A}, c) \wedge \neg final(T, potential\_map)$ then
            $parent[c] := current$;
            $continue := false$;
            Counter-Example = Trace Return($init\_state, c$);
```

Notice that, if c is equivalent to *potential_map*, and c is *not* a final state, but *potential_map* is, it is clear that $\mathcal{A}$ does not satisfy SSPOD-1. However, the trace from I up to c is not a counter-example, since it is still stuttering equivalent to the witness trace T. Hence, in this case, Algorithm 10 continues until a real counter-example is found.

6.5 Attack synthesis for SSPOD-2

Counter-examples for SSPOD-2 can be generated incrementally, starting with the empty string, and using Algorithm 9 as a selector to choose the next low operation (denoted by a symbol in the language) at each step to construct a counter-example. The following algorithm is also proposed by Kiefer et al., based on Algorithm 9 [52].

```
Algorithm 17: Language Equivalence with Counter-example ($\mathcal{R}, \mathcal{R}'$)
    $\nu_0 := \eta$; $\nu'_0 := \eta'$;
    for $i$ from 1 to $n + n'$ do
        Choose a random vector $r \in \{1, 2, \cdots, K(n + n')\}^{\mathcal{E}}$;
        $\nu_i := \sum_{\sigma \in \mathcal{E}} r_\sigma \cdot M(\sigma) \cdot \nu_{i-1}$;
        $\nu'_i := \sum_{\sigma\in\mathcal{E}} r_\sigma \cdot M'(\sigma) \cdot \nu'_{i-1}$;
        **if** $\alpha \cdot \nu \neq \alpha' \cdot \nu'$ **then**
            $w := \varepsilon$;     // An empty string
            $\theta := \alpha$; $\theta' := \alpha'$;
            **for** $j$ from $n + n'$ down to 1 **do**
                Choose $\sigma \in \mathcal{E}$ with $\theta \cdot M(\sigma) \cdot \nu_{j-1} \neq \theta' \cdot M'(\sigma) \cdot \nu'_{j-1}$;
                $w := w\sigma$;
                $\theta := \theta \cdot M(\sigma)$;
                $\theta' := \theta' \cdot M'(\sigma)$;
            **return**: $w$;
    **return**: $\mathcal{R}$ and $\mathcal{R}'$ are equivalent with probability at least $\frac{K-1}{K}$;
```

Based on the sequence of operations w, the real counter-example can be derived by the technique presented in Section 6.3.

6.6 Conclusions

This chapter presented counter-example generation techniques to synthesize attacks for our confidentiality properties. That is, if for a given scheduler, a program does not satisfy a confidentiality requirement, we generate program traces that reveal the reason why the confidentiality requirement is broken. This information can be used to identify a set of *safe* and *unsafe* schedulers, or to suggest a way to mend this security hole.

A counter-example gives us information about how a suitable scheduling can make a program execution leak secret information. However, it is also interesting to know how much information has been revealed. This will be discussed in Part 3 of this thesis.

The next chapter discusses the practical implementation of our algorithmic approaches to verify and derive counter-examples for information flow properties, together with case studies.

Chapter 7

Implementation and Case Studies

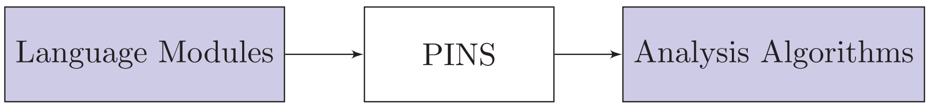
7.1 Introduction

The previous chapters discuss how we are able to check the security of a multi-threaded program in theory. This chapter discusses how we do it in *practice*, i.e., how we obtain the state space of the program execution under the control of a scheduler, and how we find that the execution contains an interleaving that can be exploited to derive secret information. The algorithms presented in Chapter 5 and Chapter 6 are implemented as a part of *the LTSmin tool set* [18]. We provide two case studies to show the feasibility of our algorithmic approaches and the practical capability of the implementation. The first case study is a non-deterministic multi-threaded program, from [37, 85], showing how the scheduling policy can be used to make a program execution leak secret information. The second case study is a probabilistic version of the dining cryptographer problem [23] showing how an attacker can derive secret information from the probabilistic outcomes.

7.2 Implementation

The proposed algorithmic techniques are implemented as a part of the LTSmin tool set, which is a tool set for symbolic, distributed and multi-core model checking and manipulating transition systems [18]. We choose LTSmin since firstly, our algorithms are applied to large state-space Kripke structures, which model the behavior of programs, and LTSmin provides means to store the transition systems in a compressed format. Secondly, our algorithms rely on stuttering-transition compression, bisimularity checking, and determinization techniques, and LTSmin incorporates implementations for all these features.

The concept of the LTSmin tool set is to provide a framework that combines the convenience of a variety of existing modeling languages with the power of different verification techniques. Since basically, the verification algorithms do not rely on a specific modeling language, LTSmin comes up with an intermediate interface, called PINS, that *separates* the responsibility for state-space *generation* from the state space *exploration* tools. The PINS interface connects language modules to analysis algorithms, as follows:

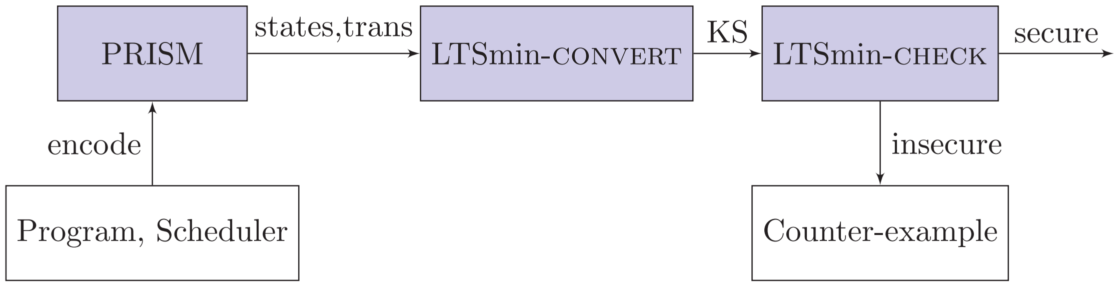


The key feature of LTSmin is its *modularity*: language front-ends, PINS layers, and algorithmic back-ends are completely decoupled. The language modules are responsible for translating the system specification to the PINS interface, i.e., a textual description of the system. The PINS interface is where the state space is generated. The underlying semantic model of PINS is a transition system (Kripke structure) with edge and state labels. Thus, via this interface, LTSmin abstracts away language-specific features. The analysis algorithms are purely based on this Kripke structure, and thus, are completely unaware of the details of specification languages. Besides, this interface also allows existing language modules to connect easily. Thus, LTSmin is a *language-independent* model checking tool set.

To model systems, we choose *the PRISM language*, since it supports modeling formally many types of systems that exhibit probabilistic behavior, and contain parallel composition between modules [58]. The PRISM language is a simple, high-level state-based language, based on a guarded command notation. It can be used to model discrete-time Markov chains (dtmc), continuous-time Markov chains (ctmc), Markov decision processes (mdp) and probabilistic timed automata (pta).

The fundamental components of a PRISM model are *modules*, i.e., a model is composed of a number of modules which can interact with each other. Modules can be used to model threads of a multi-threaded program. Thus, the PRISM model corresponding to a multi-threaded program is the parallel composition of modules representing threads.

Since the PRISM language allows to define precisely the way in which modules are composed in parallel, the scheduler's choices can be encoded by specifying at each step, which module(s) is (are) enabled to make a transition. Thus, the structure of our implementation can be illustrated via the following diagram:



The encoding of the program and the given scheduler is done manually. Based on the model, PRISM exports the set of reachable states, and the matrix representing the transition relations between states. Based on these two files, the LTSmin-CONVERT tool (an existing tool of the LTSmin tool set) constructs a Kripke structure modeling the operational execution of the program. The LTSmin-CHECK tool takes this Kripke structure as the input, and then verifies the required confidentiality properties. If the data violate one of the requirements, a counter-example is generated.

Notice that for probabilistic multi-threaded case studies modeled by the PRISM language, it is more convenient to generate the state space directly in PRISM. Thus, the implementation presented in this chapter does not include the PINS interface. However, for a system that can be modeled by other languages such as mCRL2, Promela, DVE, UPPAAL etc., PINS can be used to construct the Kripke structure for the LTSmin-CHECK tool.

Inside the LTSmin-CHECK tool, the algorithms to verify and derive counter-examples for SSOD-1K (Algorithm 1-4, Algorithm 10-13) and SSPOD-1 (Algorithm 6-7, Algorithm 16-17) are new implementations, while the implementation of SSOD-2 (Algorithm 5) is basically based on the existing features provided by the LTSmin tool set, such as stuttering-step removing, Kripke-structure determinization, and strong-bisimilarity checking. Our implementation is still ongoing with the implementation of the algorithms to verify and derive counter-examples for SSPOD-2 (Algorithm 8-9), and algorithms to derive counter-examples for SSOD-2 (Algorithm 14-15).

7.3 Case study 1: a possibilistic model

The following program, borrowed from [37, 85], consists of three threads that read a private value *PIN*, and then compute a public value *result*. Via this case study, we show that due to the interaction between threads and the control of an appropriate scheduler, the confidentiality of a multi-threaded program might be violated.

- Thread $C1$

```
while (mask != 0) do
     while (trigger0 = 0) do skip;      // wait
     result := result | mask;      // Bitwise 'or'
     trigger0 := 0;
     maintrigger := maintrigger + 1;
     if (maintrigger = 1) then trigger1 := 1;
```

- Thread $C2$

```
while (mask != 0) do
     while (trigger1 = 0) do skip;
       // Bitwise 'and' with the complement of mask
     result := result & ~mask;
     trigger1 := 0;
     maintrigger := maintrigger + 1;
     if maintrigger = 1 then trigger0 := 1;
```

- Thread $C3$

```
while (mask != 0) do
     maintrigger := 0;
     if (PIN & mask) = 0 then trigger0 := 1
      else trigger1 := 1;
     while (maintrigger != 2) do skip;
     mask := mask / 2;
```

In this program, three parallel-composed threads communicate via a shared memory, where, except *result*, all other variables are high. At first glance, this program seems not to leak information, since there is no explicit or implicit flow from the private *PIN* to the public *result*.

However, if the scheduling policy is fair, e.g., a uniform scheduler, and if this program starts in an initial state where:

$maintrigger = 0$, $trigger0 = 0$, $trigger1 = 0$, $result = 0$,
mask is a power of 2, and
PIN is an arbitrary natural number that is less than twice *mask*,

the value of *PIN* might be leaked. In this case, *mask* has the binary form $010\ldots0$; and thus, the outcome of *PIN* & *mask* will determine the value of the bit in *PIN* that corresponds to the bit 1 in *mask*. Depending on this bit, the ordering of the executions of $C1$ and $C2$ is determined. As a consequence, the corresponding bit in *result* is given the same value. Thus, eventually, the value of *PIN* will be copied into *result*.

Now, we show how we encode this multi-threaded program as a PRISM model. The corresponding PRISM model is composed of three modules, where each module models a thread. Each module contains a number of variables, whose values describe a state of the module. The global state of the whole model is determined by the local state of all modules in PRISM. The behavior of each module is described by a set of commands, having the following general form:

```
[] guard -> prob_1 :  update_1 + ...  + prob_n :  update_n;
```

The *guard* is a predicate over variables, including those belonging to other modules. Each update describes an enabled transition when the guard is satisfied. A transition is specified by giving new values to variables in the module, together with the probability of the transition.

In this model, when the guard is satisfied, the update is deterministic. Therefore, the command has a simpler form, i.e., `[] guard -> update;`

We encode the shared variables as `global` variables. To indicate which command line of a thread does the update, we introduce for each thread a local variable, named (program) *counter*. The PRISM model of the above program is given below, where we set $PIN = 23$ and initially, $mask = 16$.

```
dtmc

global result :  [0..64] init 0;
global trigger0 :  [0..1] init 0;
global trigger1 :  [0..1] init 0;
global maintrigger :  [0..2] init 0;
global mask :  [0..32] init 16;
const int pin=23;

module ThreadC1

   counter1 :  [1..6] init 1;

   [] counter1=1 & mask=0 -> true; // stutters here
   [] counter1=1 & mask!=0 -> (counter1'=2);
   [] counter1=2 & trigger0=0 -> true;
   [] counter1=2 & trigger0=1 -> (counter1'=3);
   [] counter1=3 & mod(floor(result/mask),2)=0 & result+mask <=64
                   -> (result'=result+mask) & (counter1'=4);
   [] counter1=3 & mod(floor(result/mask),2)=1 -> (counter1'=4);
   [] counter1=4 -> (trigger0'=0) & (counter1'=5);
   [] counter1=5 & maintrigger+1<3 -> (maintrigger'=maintrigger+1)
                   & (counter1'=6);
   [] counter1=6 & maintrigger=1 -> (trigger1'=1) & (counter1'=1);
   [] counter1=6 & maintrigger=0 -> (counter1'=1);

endmodule

module ThreadC2

   counter2 :  [1..6] init 1;

   [] counter2=1 & mask=0 -> true; // stutters here
   [] counter2=1 & mask!=0 -> (counter2'=2);
   [] counter2=2 & trigger1=0 -> true;
   [] counter2=2 & trigger1=1 -> (counter2'=3);
   [] counter2=3 & mod(floor(result/mask),2)=1
                   -> (result'=result-mask) & (counter2'=4);
   [] counter2=3 & mod(floor(result/mask),2)=0 -> (counter2'=4);
   [] counter2=4 -> (trigger1'=0) & (counter2'=5);
   [] counter2=5 & maintrigger+1<3 -> (maintrigger'=maintrigger+1)
                   & (counter2'=6);
   [] counter2=6 & maintrigger=1 -> (trigger0'=1) & (counter2'=1);
   [] counter2=6 & maintrigger=0 -> (counter2'=1);

endmodule

module ThreadC3

   counter3 :  [1..6] init 1;

   [] counter3=1 & mask=0 -> true; // stutters here
   [] counter3=1 & mask!=0 -> (counter3'=2);
   [] counter3=2 -> (maintrigger'=0) & (counter3'=3);
   [] counter3=3 & mod(floor(pin/mask),2)=0 -> (trigger0'=1) &
                   (counter3'=4);
   [] counter3=3 & mod(floor(pin/mask),2)=1 -> (trigger1'=1) &
                   (counter3'=4);
   [] counter3=4 & maintrigger!=2 -> true;
   [] counter3=4 & maintrigger=2 -> (counter3'=5);
   [] counter3=5 -> (mask'=floor(mask/2)) & (counter3'=1);

endmodule
```

Notice that to encode command line 3 in Thread $C1$, we first check whether the bit in *result* that corresponds to the bit 1 in *mask* has been set or not, i.e., by `mod(floor(result/mask),2)`, where `floor(x)` rounds `x` down to the nearest integer. If this bit is still 0, we set it by `result'=result+mask`; otherwise, we do nothing. Notice that before increasing a variable in the PRISM language, we need to ensure that this update does not give an *out-of-range* value, i.e., by checking `result+mask <=64` before the update. Command line 3 in Thread $C2$ is encoded similarly.

To model a uniform scheduler, we encode this example as a `dtmc` model, i.e., the model is constructed as the parallel composition of its modules, and in each step, each module is enabled with an equal probability [58].

PRISM shows that this model consists of 1000 states and 2414 transitions.

Based on the set of states and transition relations between states, the Kripke structure is constructed via LTSmin-CONVERT. Since this program is non-probabilistic, we check the SSOD-properties. As expected, LTSmin-CHECK rejects this program, since the traces of this model do not satisfy the SSOD-1K requirement.

LTSmin-CHECK also shows that the model contains two traces that are not stuttering equivalent w.r.t. the values of the public variable *result*, i.e., one trace consists of states: 0, 1, 2, 5, 12, 21, 22, 23, 13, 15, 26, 28, and another trace consists of states: 0, 1, 2, 5, 12, 21, 22, 23, 13, 15, 26, 28, 92. The tool also indicates that state 28 is *divergent*. These are the output data of the tool.

To get more information, we refer to data of these states in the *state* and *transition* files. States of the model are represented by the values of its variables, i.e., (`result, trigger0, trigger1, maintrigger, mask, counter1, counter2, counter3`), in order. Based on these states' data, we can derive *the interleaving order* that results in these two counter-example traces, as shown below.

State 0:	(0	0	0	0	16	1	1	1)	- Initial state
State 1:	(0	0	0	0	16	1	1	2)	- `ThreadC3 is picked`
State 2:	(0	0	0	0	16	1	1	3)	- `ThreadC3 is picked`
State 5:	(0	0	0	0	16	1	2	3)	- `ThreadC2 is picked`
State 12:	(0	0	0	0	16	2	2	3)	- `ThreadC1 is picked`
State 21:	(0	0	1	0	16	2	2	4)	- `ThreadC3 is picked`
State 22:	(0	0	1	0	16	2	3	4)	- `ThreadC2 is picked`
State 23:	(0	0	1	0	16	2	4	4)	- `ThreadC2 is picked`
State 13:	(0	0	0	0	16	2	5	4)	- `ThreadC2 is picked`
State 15:	(0	0	0	1	16	2	6	4)	- `ThreadC2 is picked`
State 26:	(0	1	0	1	16	2	1	4)	- `ThreadC1 is picked`
State 28:	(0	1	0	1	16	3	1	4)	- `ThreadC1 is picked`
State 92:	(16	1	0	1	16	4	1	4)	- `ThreadC1 is picked`

Notice that state 28 is divergent. When the model is in this state, if the scheduler picks `ThreadC3`, since `counter3` $= 4$ and `maintriger` $\neq 2$, the trace will stutter here, i.e., due to the execution of command line 4 in Thread $C3$. However, if the scheduler picks `ThreadC1` (as shown above), state 28 makes a transition to state 92 by the execution of command line 3 in Thread $C1$. In state 92, the value of *result* is 16. Hence, these two traces are not stuttering equivalent.

It should be stressed that this multi-threaded program leaks information only when the scheduling policy is *fair*. Unfair schedulers that give one thread *priority* over other threads would make this program secure, and thus, be accepted by the tool. Since the updates inside a module are enabled only when the guard is satisfied. Thus, the priority of a module over other modules can be modeled by setting suitable guards. Also notice that the running time of LTSmin-CHECK for this case study is less than a second.

7.4 Case study 2: a probabilistic model

For a probabilistic model, we consider the dining-cryptographer case study [23]. Three cryptographers gather around a table to have dinner at a restaurant. The waiter informs them that the meal has been paid by someone, who could be one of the cryptographers or their boss. The three cryptographers respect each other's right to make an anonymous payment, but would like to know if the boss has paid or not. So they decide to execute the following two-stage protocol [23]:

- In the first stage, each cryptographer flips an *unbiased* coin, and then informs the cryptographer *on the right* the outcome. Figure 7.1 illustrates this situation, for example, an edge from Cryptographer 1 to Cryptographer 2 shows that Cryptographer 1 can see the coin of Cryptographer 2.
- In the second stage, each cryptographer *publicly* states whether the two coins that he can see (his own coin and the left-hand neighbor's coin) are the same ('*agree*') or different ('*disagree*'). However, if he *actually paid* for the dinner, then he instead states an *opposite answer*, i.e., 'disagree' when the coins are the same, and 'agree' when the coins are different.

After the second stage, if the number of 'agrees's is odd, it implies that none of the cryptographers paid (so the boss must have paid). Otherwise, it would imply that one of the cryptographers paid. This protocol is *secure*, since in case one of the cryptographers has paid for the dinner, the others cannot find out the identity of that person [23].

To make this protocol *leak information*, we make a slight change to it: the coins are *biased*, i.e., heads (modeled as value 1) come up with probability 0.6, and tails (modeled as value 2) come up with probability 0.4. We consider the case that one of the cryptographers has paid for the dinner, and another one is the attacker, who tries to find out the payer's identity. This case study is encoded as the following PRISM model.

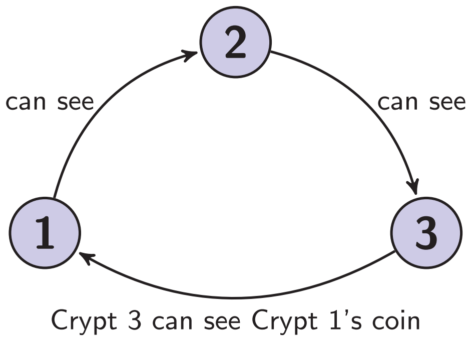


Figure 7.1: Dining cryptographers

```
dtmc

// constants used to indicate identities of cryptographers
const int p1 = 1;
const int p2 = 2;
const int p3 = 3;

// global variable which decides who pays
// (pay=i then cryptographer i paid)
global pay :  [1..3];

// parity outcomes
global par :  [0..1];

//agree : [0..1] - 0 : disagree, 1 : agree)
global agree1 :  [0..1];
global agree2 :  [0..1];
global agree3 :  [0..1];

// module for the first cryptographer
module Crypt1

   coin1 :  [0..2]; // value of its coin
   s1 :  [0..1]; // its status (0 = not done, 1 = done)
```

```
    // flip coin
    [] coin1=0 -> 0.6 :  (coin1'=1) + 0.4 :  (coin1'=2);

    // make statement (once relevant coins have been flipped)
    // agree (coins the same and does not pay)
    [] s1=0 & coin1>0 & coin2>0 & coin1=coin2 & (pay!=p1)
                        -> (s1'=1) & (agree1'=1);

    // disagree (coins different and does not pay)
    [] s1=0 & coin1>0 & coin2>0 & !(coin1=coin2) & (pay!=p1)
                        -> (s1'=1);

    // disagree (coins the same and pays)
    [] s1=0 & coin1>0 & coin2>0 & coin1=coin2 & (pay=p1)
                        -> (s1'=1);

    // agree (coins different and pays)
    [] s1=0 & coin1>0 & coin2>0 & !(coin1=coin2) & (pay=p1)
                        -> (s1'=1) & (agree1'=1);

    // update the bit par
    [] s1>0 -> (par'=parity);

endmodule

// module for the second cryptographer
module Crypt2

    coin2 :  [0..2]; // value of its coin
    s2 :  [0..1]; // its status (0 = not done, 1 = done)

    [] coin2=0 -> 0.6 :  (coin2'=1) + 0.4 :  (coin2'=2);

    [] s2=0 & coin2>0 & coi3>0 & coin2=coin3 & (pay!=p2)
                        -> (s2'=1) & (agree2'=1);
    [] s2=0 & coin2>0 & coin3>0 & !(coin2=coin3) & (pay!=p2)
                        -> (s2'=1);
    [] s2=0 & coin2>0 & coin3>0 & coin2=coin3 & (pay=p2)
                        -> (s2'=1);
```

```
   [] s2=0 & coin2>0 & coin3>0 & !(coin2=coin3) & (pay=p2)
                       -> (s2'=1) & (agree2'=1);

   [] s2>0 -> (par'=parity);

endmodule

// module for the third cryptographer
module Crypt3

   coin3 :  [0..2]; // value of its coin
   s3 :  [0..1]; // its status (0 = not done, 1 = done)

   [] coin2=0 -> 0.6 :  (coin2'=1) + 0.4 :  (coin2'=2);

   [] s3=0 & coin3>0 & coi1>0 & coin3=coin1 & (pay!=p3)
                       -> (s3'=1) & (agree3'=1);
   [] s3=0 & coin3>0 & coin1>0 & !(coin3=coin1) & (pay!=p3)
                       -> (s3'=1);
   [] s3=0 & coin3>0 & coin1>0 & coin3=coin1 & (pay=p3)
                       -> (s3'=1);
   [] s3=0 & coin3>0 & coin1>0 & !(coin3=coin1) & (pay=p3)
                       -> (s3'=1) & (agree3'=1);

   [] s3>0 -> (par'=parity);

endmodule

// set of initial states
// (cryptographers in their initial state, the high variable 'pay' can be anything)
init par=0 & coin1=0 & s1=0 & agree1=0 & coin2=0& s2=0
     & agree2=0 & coin3=0 & s3=0 & agree3=0 endinit

// parity of number of 'agree's (0 = even, 1 = odd)
formula parity = func(mod, agree1+agree2+agree3, 2);
```

Typically, a variable declaration specifies the initial value for that variable, as in Case study 1. The initial state of the model is then defined by the initial values of all variables. To specify a model that has *multiple* initial states, we use the `init...endinit` construct. Any state which satisfies the predicate between the `init` and `endinit` keywords is an initial state.

We show that this probabilistic protocol contains a probabilistic data channel. Without lack of generality, we assume that cryptographers inform statements about the coins' values in order, i.e., Cryptographer 1 - 2 - 3. Since the attacker is one of the cryptographers, he can observe the parity of the number of 'agrees's at each step. Assume that after following the protocol, the trace of `parity` is 1 - 0 - 0. Based on this trace, it is trivial to see that the cryptographers's statements about the coins' values (in order) are: *agree* - *agree* - *disagree*.

Without lack of generality, we assume that Cryptographer 2 is the attacker. Based on the final `parity`, i.e., the number of 'agrees's is even, he knows that the boss did not pay; thus either Cryptographer 1 or Cryptographer 3 has paid for the dinner. Assume that the coin of Cryptographer 2 is *head*. Thus, the coin of Cryptographer 3 must be also *head*, since the answer of Cryptographer 2 is 'agree', and Cryptographer 2 can see the coin of Cryptographer 3. Cryptographer 2 cannot see the coin of Cryptographer 1; thus, he only knows the values of coins are either *head* - *head* - *head* or *tail* - *head* - *head*.

	Crypt 1	Crypt 2	Crypt 3	
Case 1:	head	head	head	$\Rightarrow$ Crypt 3 paid
Case 2:	tail	head	head	$\Rightarrow$ Crypt 1 paid

If the coins are *head* - *head* - *head*, Cryptographer 3 must be the payer, since he lied about his result. Similarly, if the coins are *tail* - *head* - *head*, Cryptographer 1 is the payer. Since the coins are biased, and heads come up with probability 0.6, it is more likely that Cryptographer 3 is the one who paid. This is a probabilistic leak.

PRISM indicates that the above model consists of 411 states and 1050 transitions. As expected, this model is rejected by LTSmin-CHECK. It is clear that SSPOD-1 is not satisfied, since traces of `parity` are not all stuttering equivalent. For example, if the coins are *tail* - *tail* - *head*, and if Cryptographer 1 is the payer, the answers of the cryptographers will be *disagree* - *disagree* - *disagree*. Thus, the corresponding trace of `parity` will be 0 - 0 - 0, which is not stuttering equivalent to the traces given above.

Notice that looking only at the final states of `parity` cannot derive useful information, since the final `parity` is always 0. The running time of LTSmin-CHECK for this case study is also less than a second.

7.5 Conclusions

This chapter discussed the *feasibility* of our algorithmic approaches, which is illustrated by two case studies. Via case studies, we show how to model (probabilistic) multi-threaded systems by the PRISM language, and how the tool not only checks the confidentiality properties of the system, but also describes possible attacks on security holes if the systems are *insecure*. All verifications were done within seconds. The implementation is still ongoing, and we also plan to apply the implementation to larger programs.

Chapter 5 presents verification algorithms that check whether the program execution under the control of a given scheduler is secure. In case the program leaks information, counter-example algorithms in Chapter 6 provide us the information about how and why a program fails a security requirement. Besides, it is also necessary to know *how much information* has been revealed during the execution. This will be discussed in the next part of this thesis.

Part III

Quantitative Information Flow Analysis

Chapter 8

Programs with Low Input and Noisy Output

8.1 Introduction

Qualitative information flow analysis aims to determine whether a program leaks private information or not. Thus, these absolute security properties always reject a program if it leaks any information. *Quantitative* information flow analysis offers a more general security policy, since it gives a method to tolerate a *minor* leakage, i.e., by computing how much information has been leaked and comparing this with a threshold. By adjusting the threshold, the security policy can be applied for different applications, and in particular, if the threshold is 0, the quantitative policy is seen as a qualitative one.

Classical quantitative information flow analysis uses concepts from information theory to model information flow [64, 28, 22, 62, 61, 99, 84, 9]. It considers a system as an *information-theoretic channel* with private data as the *only* input and public data as the output. The classical analysis studies the amount of private data that an attacker can learn from the public-data observation.

This chapter extends the classical context by considering systems where an attacker is able to influence the initial values of public data. This is a popular kind of systems with many real-world applications, e.g., login systems, password checkers, or banking system. For these systems, the initial low values are *also* the input of the channel that models the system, i.e., these systems contain both *private* and *initial public inputs*. Therefore, the classical approaches are not appropriate to analyze this scenario. This chapter adapts the classical view of information-theoretic channels to analyze quantitatively the security of systems containing both *high* and *low* inputs. Our analysis also proposes a new measure of information leakage that is based on Cachin's definition of conditional-min entropy.

Additionally, we show that our measure also can be used to reason about the case where a system operator *on purpose* adds noise to the output, instead of always producing the correct output. The noisy outcome is used to reduce the correlation between the output and the input, and thus, to increase the remaining uncertainty about the secret. However, even though adding noise to the output enhances security, it *reduces* the reliability of the program. We show how given a certain noisy output policy, the increase in security and the decrease in reliability can be quantified.

Organization of the chapter. Section 8.2 presents our setting for the analysis. Then, Section 8.3 and Section 8.4 discuss the classical analysis and its shortcomings. Section 8.5 presents our quantitative security analysis model for programs that contain both low and high input, and its application. The next two sections discuss when negative information flow is expected, and how to construct and evaluate a noisy-output policy. Section 8.8 discusses related work, while Section 8.9 concludes.

Origins of the chapter. The content of this chapter was published in the proceedings of the *6th International Conference on Engineering Secure Software and Systems* (ESSoS'14) [67].

8.2 Basic settings for the analysis

To aim for simplicity and clarity, rather than full generality, following the traditional approaches [84, 4], our models of analysis are based on the following basic settings. First, we assume that programs always terminate, and the attacker knows the source code of the program. We restrict to programs with just a single *high security input* S and a single *low security input* L_{in}. Since the high security output is irrelevant, programs only give a *low security outcome* O. Our goal is to quantify how much information about S is deduced by the attacker who can influence L_{in}, and observe the traces of O. We also assume that the sets of possible values of data are finite, as in the traditional approaches.

Secondly, we assume that there is a *priori, publicly-known* probability distribution on the high values. We also assume that data at the same security level are indistinguishable in the *security meaning*. Thus, a system that leaks the last 9 bits of private data is considered to be just as dangerous as a system that leaks the first 9 bits[1].

Finally, we consider the *one-try guessing model*, i.e., after observing the public outcomes, the attacker is allowed to guess the value of S by only one try. This model of attack is suitable to many security situations where systems trigger an alarm if an attacker makes a wrong guess. For the password checker, this one-try guessing model can be understood as that an attacker is only allowed to try once. If the entered string is not the correct password, the system will block the account.

Notice that these restrictions aim to demonstrate our core idea. However, the analysis might adapt to more complex situations easily after some trivial modifications.

8.3 Classical models of quantitative security analysis

Classical quantitative security analysis [64, 28, 22, 62, 61, 99, 84, 9] proposes to use information theory as a setting to model information flow, and to define the quantity of information leakage. The common idea of these approaches is that a system is considered as a channel in the information-theoretic sense.

Formally, an information-theoretic channel $\mathcal{M}$ is a triple $\mathcal{M} = (\mathbf{X}, \mathbf{Y}, M)$, where $\mathbf{X}$ represents a finite set of secret values as the input, $\mathbf{Y}$ represents a finite set of observable outcomes as the output, and M is a $|\mathbf{X}| \times |\mathbf{Y}|$ channel matrix that contains the conditional probabilities $p(y|x)$ for each $x \in \mathbf{X}$ and $y \in \mathbf{Y}$. Thus, each entry of M is a real number between 0 and 1, and each row sums to 1.

Summarizing, classical quantitative security analysis models a program as a standard *input-output* channel with the secret S as the input and the public outcome O as the output. The analysis studies how much information about S an attacker might learn from the output O.

[1]These assumptions follow the traditional analysis.

8.3.1 Entropy

The quantitative analysis of information flow is based on the notion of *entropy*. The entropy of a random private variable expresses the *uncertainty* of an attacker about its value, i.e., how *difficult* it is for an attacker to discover its value. Thus, the entropy of a private value expresses the uncertainty of an attacker about its correct value. So far, most of the approaches were based on Shannon entropy [64, 28, 22, 62, 61, 99] and Rényi's min-entropy with Smith's version of conditional min-entropy [84, 9]. Various definitions of entropy are given as follows.

Let X and Y denote two discrete random variables. Let $p(X = x)$ denote the probability that $X = x$, and let $p(X = x|Y = y)$ denote the *conditional* probability that $X = x$ when $Y = y$.

Shannon entropy

Definition 8.1 *The Shannon entropy of a random variable X is defined as,*

$$\mathcal{H}_{Shannon}(X) = -\sum_{x\in\mathbf{X}} p(X = x)\log p(X = x),$$

where the base of the logarithm is set to 2 [4].

Definition 8.2 *The conditional Shannon entropy of a random variable X given Y is,*

$$\mathcal{H}_{Shannon}(X|Y) = \sum_{y\in\mathbf{Y}} p(Y = y)\mathcal{H}_{Shannon}(X|Y = y),$$

where $\mathcal{H}_{Shannon}(X|Y = y) = -\sum_{x\in\mathbf{X}} p(X = x|Y = y)\log p(X = x|Y = y)$.

It is possible to prove that $0 \leq \mathcal{H}_{Shannon}(X|Y) \leq \mathcal{H}_{Shannon}(X)$. The minimum value of $\mathcal{H}_{Shannon}(X|Y)$ is 0, if X is completely determined by Y. The maximum value of $\mathcal{H}_{Shannon}(X|Y)$ is $\mathcal{H}_{Shannon}(X)$, when X and Y are independent [4].

Min-entropy

Definition 8.3 *The Rényi's min-entropy of a random variable X is defined as [84]: $\mathcal{H}_{Rényi}(X) = -\log \max_{x\in\mathbf{X}}\ p(X = x)$.*

Notice that Shannon entropy and Rényi's min-entropy coincide on uniform distributions.

Rényi did not define the notion of *conditional min-entropy*, and there are different definitions of this notion.

Definition 8.4 (Smith's version of conditional min-entropy [84]) *The conditional min-entropy of a random variable X given Y is,*

$$\mathcal{H}_{Smith}(X|Y) = -\log \sum_{y \in \mathbf{Y}} p(Y=y) \cdot \max_{x \in \mathbf{X}} p(X=x|Y=y).$$

Various researchers, including Cachin, have considered the following definition:

Definition 8.5 (Cachin's version of conditional min-entropy [21]) *The conditional min-entropy of a random variable X given Y is,*

$$\mathcal{H}_{Cachin}(X|Y) = -\sum_{y \in \mathbf{Y}} p(Y=y) \cdot \log \max_{x \in \mathbf{X}} p(X=x|Y=y).$$

Example 8.1 *Given a discrete random variable X with the priori distribution $\pi = \{p(X=x_1) = \frac{2}{3}, p(X=x_2) = \frac{1}{6}, p(X=x_3) = \frac{1}{6}\}$. Thus,*

$$\begin{aligned} \mathcal{H}_{Shannon}(X) &= -(\tfrac{2}{3}\log\tfrac{2}{3} + \tfrac{1}{6}\log\tfrac{1}{6} + \tfrac{1}{6}\log\tfrac{1}{6}), \text{ and} \\ \mathcal{H}_{Rényi}(X) &= -\log\tfrac{2}{3}. \end{aligned}$$

Suppose that the channel (matrix) M is as follows,

M	y_1	y_2
x_1	1/2	1/2
x_2	1/6	5/6
x_3	0	1

The channel M and the distribution π determine the joint probability matrix J, where $J[x_i, y_j] = \pi(x_i) \cdot M[x_i, y_j]$.

M	y_1	y_2
x_1	1/3	1/3
x_2	1/36	5/36
x_3	0	1/36

The joint probability matrix J determines the distribution of Y, i.e., $p(Y = y_j) = \sum_{\forall x_i} J[x_i, y_j]$. Thus, $p(Y = y_1) = \frac{1}{3} + \frac{1}{36} + 0 = \frac{13}{36}$, and $p(Y = y_2) = \frac{1}{3} + \frac{5}{36} + \frac{1}{6} = \frac{23}{36}$.

Since $p(X = x_i|Y = y_j) = \frac{J[x_i,y_j]}{p(Y=y_j)}$, we have

$$p(X = x_1|Y = y_1) = \frac{12}{13},\ p(X = x_2|Y = y_1) = \frac{1}{13},\ p(X = x_3|Y = y_1) = 0,\ \textit{and}$$

$$p(X = x_1|Y = y_2) = \frac{12}{13},\ p(X = x_2|Y = y_2) = \frac{5}{23},\ p(X = x_3|Y = y_3) = \frac{6}{23}.$$

Therefore, various measures of conditional entropy are given as follows,

$$\begin{aligned}
\mathcal{H}_{Shannon}(X|Y) &= -\tfrac{13}{36} \cdot (\tfrac{12}{13} \log \tfrac{12}{13} + \tfrac{1}{13} \log \tfrac{1}{13}) \\
&\quad - \tfrac{23}{36} \cdot (\tfrac{12}{23} \log \tfrac{12}{23} + \tfrac{5}{23} \log \tfrac{5}{23} + \tfrac{6}{23} \log \tfrac{6}{23}) \\
\mathcal{H}_{Smith}(X|Y) &= -\log(\tfrac{13}{36} \cdot \tfrac{12}{13} + \tfrac{23}{36} \cdot \tfrac{12}{23}) \\
\mathcal{H}_{Cachin}(X|Y) &= -(\tfrac{13}{36} \log \tfrac{12}{13} + \tfrac{23}{36} \log \tfrac{12}{23})
\end{aligned}$$

8.3.2 Quantity of information leakage

Suppose that a program C is modeled as a channel matrix M with S as the input and O as the output. The leakage of C is defined as the *difference* between the uncertainty of the attacker about S *before* executing the program and his uncertainty *after* observing O. Let $\mathcal{H}(S)$ denote the initial uncertainty of the attacker, and $\mathcal{H}(S|O)$ denote the uncertainty after the program has been executed and the public outcomes are observed. Therefore, the quantity of information leakage of C is given by,

$$\mathcal{L}(C) = \mathcal{H}(S) - \mathcal{H}(S|O),$$

where $\mathcal{L}(C)$ denotes the leakage of C; $\mathcal{H}$ might be either Shannon entropy or min-entropy with Smith's version of conditional min-entropy. The quantity of information leakage is measured in bit.

In this chapter, we let $\mathcal{L}_{Shannon}$ denote the leakage computed with Shannon entropy, and $\mathcal{L}_{Smith}$ denote the leakage computed by using min-entropy with Smith's version of conditional min-entropy.

8.4 Shortcomings of the classical models

8.4.1 Counter-intuitive and conflict results

Many authors measure leakage using Shannon entropy. However, Smith shows that in the context of the *one-try guessing* model, a measure based on Shannon entropy does not always result in a very good operational security guarantee [84]. In particular, Smith argues that Shannon-entropy measure might be *counter-intuitive* by showing two programs C and C' such that by intuitive understanding, C leaks more information than C', but Shannon-entropy measures yield counter-intuitive results, i.e., $\mathcal{L}_{Shannon}(C) < \mathcal{L}_{Shannon}(C')$. For example, consider the two following programs $C1$ and $C2$ and their values of leakage, based on Shannon-entropy measure, from [84].

Example 8.2 (Program C_1)

$$\texttt{if}\,(S \mod 8 = 0)\ \texttt{then}\ O := S\ \texttt{else}\ O := 1;$$

Basically, C_1 copies S to O when S is a multiple of 8, otherwise, it sets O to 1. Assume that S is a 64-bit unsigned integer, $0 \le S \le 2^{64} - 1$, with the priori uniform distribution. Thus, the attacker's initial uncertainty is given by,

$$\begin{aligned}\mathcal{H}_{Shannon}(S) &= -\textstyle\sum_{s\in\{0,\dots,2^{64}-1\}} p(S=s)\log p(S=s)\\ &= -\textstyle\sum_{s\in\{0,\dots,2^{64}-1\}} \frac{1}{2^{64}}\log\frac{1}{2^{64}}\\ &= -2^{64}\frac{1}{2^{64}}\log\frac{1}{2^{64}} = 64 \quad .\end{aligned}$$

This program is deterministic, thus for each possible value of S, the execution of C_1 results in only one O. Among 2^{64} possible values of S, 2^{61} values give the output $O = S$. There are $2^{64} - 2^{61} = 7 \cdot 2^{61}$ inputs that map O to 1. Thus, $p(O = 1) = \frac{7\cdot 2^{61}}{2^{64}} = \frac{7}{8}$.

If the value O is different from 1, the secret is revealed, i.e., $p(S = s|O = s) = 1$. When O is equal to 1, the attacker only knows that S is not a multiple of 8, i.e., the last 3 bits of S are not all zeros. Thus, if $O = 1$, among $7 \cdot 2^{61}$ possible *uniformly-distributed* values of S, the probability that the guess of the secret value is correct is $p(S = s|O = 1) = \frac{1}{7\cdot 2^{61}}$. Therefore,

$$\begin{aligned}\mathcal{H}_{Shannon}(S|O) &= 2^{61}\cdot\frac{1}{2^{64}}(-\log 1) + \frac{7}{8}(-\log\frac{1}{7\cdot 2^{61}})\\ &= 55.83 \quad .\end{aligned}$$

Hence, according to Shannon-entropy measure, the leakage of C_1 is,

$$\mathcal{L}_{Shannon}(C1) = \mathcal{H}_{Shannon}(S) - \mathcal{H}_{Shannon}(S|O) = 64 - 55.83 = 8.17.$$

Example 8.3 (Program $C2$)

$$O := S \,\&\, (\texttt{0}\ldots\texttt{0.111.111.111})_{\texttt{b}};$$

where $(\texttt{0}\ldots\texttt{0.111.111.111})_{\texttt{b}}$ *is the 64-bit binary number such that the first 55 bits are* 0 *and the last 9 bits are* 1.

This program simply copies the last 9 bits of S into O. Shannon-entropy measure gives $\mathcal{L}_{Shannon}(C2) = 9$.

In $C1$, whenever the public outcome $O \neq 1$, the attacker obtains the secret S *completely*. Thus, the expected probability of guessing S correctly is greater than $2^{61} \cdot \frac{1}{2^{64}} = \frac{1}{8}$. In $C2$, for any value of O, the probability of guessing S correctly by one try is $\frac{1}{2^{55}}$, since the first 55 bits of S are still unknown. It means that, in the one-try threat model, intuitively, $C1$ is considered more dangerous than $C2$. However, the measure based on Shannon entropy judges $C2$ worse than $C1$.

For this reason, Smith develops an alternative theory of quantitative information flow based on min-entropy [84]. Smith defines uncertainty in terms of the *vulnerability* of S to be guessed in one try. The vulnerability of a random variable X is the *maximum* of the probabilities of the possible values of X. This approach seems to match the intuitive idea of the one-try guessing model, i.e., the attacker always chooses the value with the maximum probability.

However, also min-entropy measures might result in counter-intuitive values of leakage. Consider the two following programs C_3 and C_4, from [84, 97].

Example 8.4 (Program $C3$: Password Checker (*PWC*))

$$\texttt{if}\ (S = L_{in})\ \texttt{then}\ O := \texttt{1}\ \texttt{else}\ O := \texttt{0};$$

where S denotes the password, L_{in} the string entered by the attacker, and O the observable answer, i.e., right or wrong.

Assume that S is an unsigned integer with the priori uniform distribution. Thus, $\mathcal{H}_{Rényi}(S) = -\log \frac{1}{|S|} = \log |S|$, where $|S|$ is the number of possible values of S. If the output $O = 1$, the attacker obtains the password, i.e., $p(S = L_{in}|O = 1) = 1$. If $O = 0$, the probability that the attacker can guess the correct password by one try is $p(S = s|O = 0) = \frac{1}{|S|-1}$. Generally, the probability that the string L_{in} matches the password $p(O = 1)$ is $\frac{1}{|S|}$. Hence, $p(O = 0) = \frac{|S-1|}{|S|}$. Thus, according to Smith's conditional min-entropy, $\mathcal{H}_{Smith}(S|O) = -\log(\frac{1}{|S|} \cdot 1 + \frac{|S-1|}{|S|} \cdot \frac{1}{|S|-1}) = \log|S| - 1$. Therefore,

$$\mathcal{L}_{Smith}(C_3) = \mathcal{H}_{Rényi}(S) - \mathcal{H}_{Rényi}(S|O) = 1.$$

Example 8.5 (Program $C4$: Binary Search)

```
if $(S \geq L_{in})$ then $O$ := 1 else $O$ := 0;
```

where S is with the same size as in $C3$, and $L_{in} = |S|/2$ is a program parameter.

For this example, $p(O = 1) = p(O = 0) = \frac{1}{2}$, and $p(S = s|O = 1) = p(S = s|O = 0) = \frac{1}{|S|/2}$. Thus, $\mathcal{H}_{Smith}(S|O) = \log|S| - 1$. Therefore,

$$\mathcal{L}_{Smith}(C_4) = \mathcal{H}_{Rényi}(S) - \mathcal{H}_{Rényi}(S|O) = 1.$$

Thus, measure proposed by Smith does not distinguish between $C3$ and $C4$ by always judging that their leaks are the same. However, if $|S|$ is large, the probability that $S = L_{in}$ becomes so low in $C3$, i.e., $p(O = 1) = \frac{1}{|S|} \approx 0$. Thus, intuitively, $C3$ leaks almost nothing, since the chance of guessing S correctly after observing the outcome is $\frac{1}{|S|-1} \approx \frac{1}{|S|}$. Program $C4$ always leaks 1 bit of information, since the chance of the correct guessing based on the public outcome is $\frac{2}{|S|}$. Thus, program $C4$ should be judged more dangerous than $C3$. In his paper [84], Smith also admits that $C3$ and $C4$ should not be treated the same. It is trivial that the uncertainty of the password guessing decreases *slowly*, while in binary search, the uncertainty of the secret decreases very *rapidly*.

Notice that $\mathcal{L}_{Shannon}(C3) = 0$ and $\mathcal{L}_{Shannon}(C4) = 1$. This indicates that for these examples, Shannon-entropy measure matches the intuition. Therefore, we agree with Alvim et al. that no single leakage measure is likely to suit all cases [7].

8.4.2 Leakage in intermediate states

Classical approaches often consider only the leakage in the final states of the execution. However, to make a suitable model of analyzing quantitatively information flow, it is necessary to take into account also the leakage in intermediate states along the execution traces. Consider the following example where S is a 3-bit binary number.

Example 8.6 (Program $C5$) *Given that $(100)_b$ and $(011)_b$ are the binary forms of 4 and 3, respectively.*

$$\begin{array}{l} O := 0; \\ O := S \,\&\, (100)_b; \\ O := S \,\&\, (011)_b; \end{array}$$

Let $(s_3s_2s_1)_b$ denote the binary form of S. The analysis based only on the final-state observation judges that $C5$ leaks 2 bits of private data, i.e., $O = (s_3s_2s_1)_b \,\&\, (011)_b = (0s_2s_1)_b$. However, it is clear that this program leaks the secret *completely*, i.e., due to the leakage in the intermediate state, i.e., $O = (s_3s_2s_1)_b \,\&\, (100)_b = (s_300)_b$.

Thus, an appropriate model of quantitative security analysis needs to consider the leakage given by a sequence of publicly observable data obtained during the program execution. However, notice that the leakage in intermediate states does not always contribute to the overall leakage of a trace, as in the following example,

Example 8.7 (Program $C6$)

$$\begin{array}{l} O := 0; \\ O := S \,\&\, (001)_b; \\ O := S \,\&\, (011)_b; \end{array}$$

The overall leakage of this program trace is only 2 bits, since the leakage in the intermediate state, i.e., the last bit s_1, is included in the leakage in the final state. This example shows that leakage of a program trace is not simply the sum of the leakage of transition steps along the trace, as in the approach of Chen et al. [26].

8.5 Analytical model for programs that contain low input

8.5.1 Leakage of programs with low input

The only input of the information-theoretic channel is the secret. For programs where an attacker might influence the initial value of the low variable, the initial *low value* is also an *input* of the channel modeling the program. To apply the traditional channel for the analysis, we propose to model such a program by a set of channels. Each channel corresponds to the model of the program where the low input is assigned a certain value. Thus, in our approach, the initial low value is considered as a *parameter*. Since we assume that the low value set is finite, the set of channels is also finite.

We see a channel as a *test*. We run the analysis on the set of tests. Since the attacker knows the program code, and is also able to influence the initial low value, he knows which test would give him *more* secret information. Thus, we define the leakage of the program that contains low input as the maximum leakage over all tests.

Given a program C that contains a low input L_{in}. Let π denote the priori distribution on the possible values of the private data, and *LVal* denote the value set of L_{in}. Let $T_{|_O}$ denote a trace of O obtained from the execution of C. To define the leakage of C, our approach carries out the following steps.

```
Leakage of Programs with Low Input
1: Set up a test $(P, \pi, L_{in})$:
    1.1: Choose a value for $L_{in}$.
    1.2: Construct a channel where $S$ is the input, $L_{in}$ is the parameter
         of the channel, and the traces $T_{|_O}$ are the output.
2: Compute the leakage of the test $(P, \pi, L_{in})$:
        $\mathcal{L}(P, \pi, L_{in}) = \mathcal{H}_{Rényi}(S) - \mathcal{H}_{Cachin}(S|L_{in}, T_{|_O})$,
    where $\mathcal{H}_{Rényi}(S)$ is the min-entropy of $S$ corresponding to $\pi$.
3: Define the leakage of $P$ as: $\mathcal{L}(P, \pi) = \max_{L_{in} \in LVal} \mathcal{L}(P, \pi, L_{in})$.
```

Notice that Step **1** and **2** are repeated for all values of L_{in}.

Measures of uncertainty. Since we follow the one-try guessing model, the initial uncertainty is computed by Rényi's min-entropy of S with the priori distribution π. In our approach, we propose to use Cachin's version of conditional min-entropy (Definition 8.5) as a *new measure* for the remaining uncertainty, instead of Smith's version. To the best of our knowledge, this measure has not previously been used in the theory of quantitative information flow.

In the remainder of this chapter, to denote our measure, we use the notation $\mathcal{L}_{Cachin}$ to distinguish it from the measure proposed by Smith.

8.5.2 Case studies

Below, we analyze some case studies, and compare Smith's measure with our measure. We show that our measure agrees more with the intuition.

Password Checker. Consider the following *PWC*.

$$\texttt{if}\ (S = L_{in})\ \ \texttt{then}\ O := 1\ \texttt{else}\ O := 0;$$

where S might be A_1, A_2, or A_3, with $\pi = \{p(A_1) = 0.98, p(A_2) = 0.01, p(A_3) = 0.01\}$. Since the attacker tests L_{in} based on the value of S, there are 3 corresponding tests, i.e., $L_{in} = A_1$, $L_{in} = A_2$, or $L_{in} = A_3$. The leaks of the tests $L_{in} = A_2$ and $L_{in} = A_3$ are the same. Hence, we only analyze $L_{in} = A_1$ and $L_{in} = A_2$.

Before interacting with the *PWC*, the attacker believes that the password is A_1, since $p(A_1)$ dominates the other cases. Thus, in both tests, the attacker's *initial uncertainty* about S is $\mathcal{H}_{Rényi}(S) = -\log 0.98 = 0.02915$.

When $L_{in} = A_1$, the *PWC* is modeled by the following channel matrix M,

M	$O = 1$	$O = 0$
$S = A_1$	1	0
$S = A_2$	0	1
$S = A_3$	0	1

The channel M and the distribution π determine the joint probability matrix J, where $J[s, o] = \pi(s) \cdot M[s, o]$.

J	$O = 1$	$O = 0$
$S = A_1$	0.98	0
$S = A_2$	0	0.01
$S = A_3$	0	0.01

The joint probability matrix J determines a marginal distribution of O, i.e., $p(O = 1) = 0.98$ and $p(O = 0) = 0.02$.

Since $p(S = s|O = o) = \frac{J[s,o]}{p(o)}$, we have

$$p(S = A_1|O = 1) = 1,\ p(S = A_2|O = 1) = p(S = A_3|O = 1) = 0,\ \text{and}$$

$$p(S = A_1|O = 0) = 0,\ p(S = A_2|O = 0) = p(S = A_3|O = 0) = 0.5.$$

Therefore, $\mathcal{L}_{Cachin}(C, \pi, A_1) = 0.00915$, while $\mathcal{L}_{Smith}(C, \pi, A_1) = 0.01465$.

When $L_{in} = A_2$, we obtain the following channel,

M	$O = 1$	$O = 0$
$S = A_1$	0	1
$S = A_2$	1	0
$S = A_3$	0	1

Thus, $p(O = 1) = 0.01$ and $p(O = 0) = 0.99$, and

$$p(S = A_1|O = 1) = p(S = A_3|O = 1) = 0,\ p(S = A_2|O = 1) = 1, \text{ and}$$

$$p(S = A_1|O = 0) = 0.9899,\ p(S = A_2|O = 0) = 0,\ p(S = A_3|O = 0) = 0.0101.$$

Therefore, $\mathcal{L}_{Cachin}(C, \pi, A_2) = 0.01465$, while $\mathcal{L}_{Smith}(C, \pi, A_2) = 0.01465$.

The measure proposed by Smith judges that the leakage values of the two tests where $L_{in} = A_1$ and $L_{in} = A_2$ are the same. However, this contradicts the intuition. In the test $L_{in} = A_1$, if the *PWC* answers yes, it only helps the attacker to confirm something that he already believed to be certainly true. However, if the answer is $O = 0$, it does not help the attacker at all, i.e., he still does not know whether either A_2 or A_3 is more likely to be the password, since the posteriori probability $p(S = A_2|O = 0)$ is still equal to $p(S = A_3|O = 0)$.

Intuitively, the test $L_{in} = A_2$ helps the attacker to gain more secret information. If $O = 1$, it completely changes the attacker's *priori belief*, i.e., the password is not A_1, and it also confirms a very rare case, i.e., the password is A_2. If $O = 0$, this even *strengthens* what the attacker's belief about the secret, since the posteriori probability $p(S = A_1|O = 0) = 0.9899$ increases. The analysis should indicate that the test $L_{in} = A_2$ leaks more information than the test $L_{in} = A_1$.

Thus, in this example, our measure gives results that match more the intuition. The leakage of this *PWC* is defined as the leakage of the test $L_{in} = A_2$. This example also shows that the test in which the attacker sets the low input based on the value that he believes to be the secret is not always the "*best test*". Since the attacker knows the source code of the program and the priori distribution of the private data, he knows which test would give him the most information. This is the reason that we define the leakage of a program with low input as the maximum leakage over all tests.

In the general case, given $\pi = \{p(A_1) = a, p(A_2) = b, p(A_3) = c\}$, whenever $a > c$ and $b > c$, Smith's measure cannot distinguish between the test $L_{in} = A_1$ and $L_{in} = A_2$, while our measure can, and also agrees more with the intuition about what the leakage should be.

A Multi-threaded Program. Consider the following example,

Example 8.8

```
$O := 0;$
$\{$if $(O = 1)$ then $O := S/4$ else $O := S \bmod 2\} \parallel O := 1;$
$O := S \bmod 4;$
```

where S is a 3-bit unsigned integer with the priori uniform distribution.

The execution of this program results in the following traces, depending on whether thread C_1 or C_2 is picked first:

S	0		1		2		3		4		5		6		7	
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$T\|_O$	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1
	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3

Consider a uniform scheduler, i.e., a scheduler that picks threads with equal probability. It is clear that the last command $O := S \bmod 4$ always reveals the last 2 bits of S. The first bit might be leaked with probability $\frac{1}{2}$, depending on whether the scheduler picks thread C_2 first or not. Thus, with the uniform scheduler, intuitively, the real leakage of this program is 2.5 bits.

By observing the traces of O, the attacker is able to derive secret information. For example, if the trace is 0100, the attacker can derive S precisely, since this trace is produced only when $S = 0$. If the trace is 0010, the attacker can conclude that S is either 0 or 4 with the same probability, i.e., $\frac{1}{2}$. If the trace is 0111, the possible value of S is either 1 or 5, but with different probabilities, i.e., the chance that S is 5 is $\frac{2}{3}$. There are 6 traces such that the attacker is able to derive the value of S precisely from them. There are 4 traces such that the attacker is able to guess S correctly with the probability $\frac{1}{2}$, and 6 traces with the probability $\frac{2}{3}$. Therefore, $\mathcal{L}_{Cachin}(C, \pi) = 3 - (-(\frac{6}{16} \cdot \log 1 + \frac{4}{16} \cdot \log \frac{1}{2} + \frac{6}{16} \cdot \log \frac{2}{3})) = 2.53$, while $\mathcal{L}_{Smith}(C, \pi) = 3 - (-\log(\frac{6}{16} \cdot 1 + \frac{4}{16} \cdot \frac{1}{2} + \frac{6}{16} \cdot \frac{2}{3})) = 2.58$.

Consider a scheduler that picks thread C_2 first with probability $\frac{3}{4}$. With this scheduler, the real leakage of this program is 2.75. Our measure gives $\mathcal{L}_{Cachin}(C, \pi) = 2.774$, while $\mathcal{L}_{Smith}(C, \pi) = 2.807$. If the scheduler picks thread C_2 first with probability $\frac{1}{4}$, $\mathcal{L}_{Cachin}(C, \pi) = 2.271$, while $\mathcal{L}_{Smith}(C, \pi) = 2.321$. Of course in this case, the real leakage is 2.25. These results show that our measures are closer to the real leakage values.

These case studies show that our measure is more precise than the classical measure given by Smith's conditional min-entropy. The main difference between the two measures is *the position of* log in the expression of the remaining entropy. The idea of using logarithm is to express the notion of uncertainty in bits. Thus, the log should apply *only* to the probability of the guess, which represents the uncertainty of the attacker, as in our approach. Our measure distinguishes between the *probabilities of the observable* and the *probabilities of the guess* based on the observable. In Smith's measure, the logarithm applies to the combination of the two probabilities, and does not distinguish between them, which might cause imprecise results.

However, as a side remark, we emphasize that no unique measure is likely to be suitable for all cases. We believe that for some examples, measures based on Shannon entropy or Smith's version of conditional min-entropy might match better the real values of leakage. Thus, it would be interesting (as future work) to evaluate each measure to draw up a guideline indicating which measure to apply to which scenario.

8.6 Noisy output

8.6.1 Adding noise to the output

In relation to defining an appropriate measure for information flow quantification, this thesis also discusses a claim of the classical quantitative information flow theory, i.e., a quantitative measure of information leakage should return a *non-negative* value. The common idea is that the program's public-outcome observation would *enhance* the attacker's knowledge about the secret, and consequently *reduce* his initial uncertainty. Therefore, classical analysis often expects that the value of leakage should not be negative.

However, we think that this non-negativeness property does not always hold. For some applications, to enhance the confidentiality, the system operator adds noise *secretly* to the output, i.e., sometimes, the system might generate a wrong outcome, instead of the exact one, as in the following example.

Consider a deterministic system (program) C where the secret might be A_1, A_2, or A_3 with the priori uniform distribution $\pi = \{p(A_1) = p(A_2) = p(A_3) = \frac{1}{3}\}$. The system C might produce three low outcomes Z_1, Z_2, and Z_3 as described by the following channel matrix M.

M	$O = Z_1$	$O = Z_2$	$O = Z_3$
$S = A_1$	1	0	0
$S = A_2$	0	1	0
$S = A_3$	0	0	1

Since the attacker knows the program code, he is able to construct M in his database. The system is deterministic, i.e., the public outcomes are *totally* dependent on the secret; thus, the attacker can obtain the private data *entirely.* For example, if the outcome is Z_i, the attacker knows for sure that $S = A_i$.

To protect the private data, the system operator might mislead the attacker by a noisy-output policy (policy for short), i.e., adding noise to the output of the system via some output-perturbation mechanism based on randomization. Thus, the channel modeling the system is *secretly* changed by the policy, i.e., the system is now described by the following channel matrix M'.

M'	$O = Z_1$	$O = Z_2$	$O = Z_3$
$S = A_1$	$\frac{5}{6}$	$\frac{1}{6}$	0
$S = A_2$	0	$\frac{3}{4}$	$\frac{1}{4}$
$S = A_3$	$\frac{1}{3}$	0	$\frac{2}{3}$

The noisy-output policy is kept in secret, i.e., the attacker does not know that the security policy has been applied to the system. Thus, the attacker still thinks that the system is M, but in fact, the *real* system is M'.

Therefore, the posteriori distribution in the attacker's database is not correct anymore, e.g., for the outcome Z_i, $p(S = A_i|O = Z_i) = 1$ is not the real posteriori probability. The real posteriori distribution ensures that his guess is not 100% correct.

The noisy outcomes might mislead the attacker's belief about the secret, i.e., the attacker's final uncertainty is *increased.* As a consequence, the value of leakage might be *negative.* This idea is illustrated more by the following example.

8.6.2 Negative information flow

Password checker with noisy outcomes. Consider the example of PWC in Section 8.5.2. We assume that a system operator has changed its behavior *secretly*, i.e., the real PWC is a *probabilistic* PWC where some perturbation mechanism has been applied to the output,

$$\texttt{if}\ (S = L_{in})\ \texttt{then}\ \{O := 1\ {}_{0.9}\|\ O := 0\}\ \texttt{else}\ \{O := 0\ {}_{0.9}\|\ O := 1\};$$

In this version, the exact answers are reported with probability 0.9, i.e., when $S = L_{in}$, $O = 1$ is reported with probability 0.9, and $O = 0$ with probability 0.1. Consider the test $L_{in} = A_2$, the real channel M' is as follows,

M'	$O = 1$	$O = 0$
$S = A_1$	0.1	0.9
$S = A_2$	0.9	0.1
$S = A_3$	0.1	0.9

Notice that the attacker still thinks that the system is M, but in fact, the real system is M'. Based on π and M', the computation gives the real distribution of O, i.e., $p(O = 1) = 0.108$ and $p(O = 0) = 0.892$, and the real posteriori probabilities $p(S = A_2|O = 1) = 0.083$ and $p(S = A_1|O = 0) = 0.9887$.

Before observing the outcome, the attacker's belief that the password is A_1 has 98% chance of being correct. If the outcome is $O = 0$, the real posteriori probability gives the attacker's guess, i.e., $S = A_1$, a 98.87% chance of being correct. This is almost the same as the guess without the outcome observation. When $O = 1$, the attackers guesses $S = A_2$, since his database tells that this guess has the *highest chance* to be correct. However, the real posteriori probability ensures that his guess only has a 8.3% chance of being correct. Therefore, the outcomes of the program not only reveal *no* secret information, but also cause him to decide *wrongly*. Therefore, intuitively, this is a *negative information flow*.

As we expected, our measure indicates a negative leakage: $\mathcal{L}_{Cachin}(C, \pi, A_2)$ $= -\log 0.98 + (0.108 \log 0.083 + 0.892 \log 0.9887) = -0.37$, while $\mathcal{L}_{Smith}(C, \pi, A_2)$ $= -0.137$. Notice that the leakage is determined by the real probability of success, not by the probability in the attacker's database.

To the best of our knowledge, we think that this observation of negative information flow has not been reported in the literature. We believe that this property would change the classical view of how the measure of uncertainty should be, i.e., we do not need to *avoid* measures that do not guarantee the non-negativeness property.

8.7 Noisy-output policy

The noisy outcomes change the behavior of the system, i.e., they change the channel matrix M that models the system (the public channel matrix that the attacker also knows) to M' (the real channel matrix that is kept secret). The noisy outcomes should be generated in such a way that they change the original channel, but still preserve a *certain level* of reliability, e.g., the above probabilistic PWC works properly in 90% of the time. Totally random outcomes might achieve the best confidentiality, but these outcomes are practically useless. Besides, the noisy-output policy also needs to satisfy some *general requirements* that, on the one hand, help to mislead the attacker, i.e., the attacker does not know that the system has been changed by the policy; thus, he still uses the posteriori distributions based on M and π to make a guess, and on the other hand, reduce the leakage. This section discusses how to design such an efficient noisy-output policy.

8.7.1 Design a policy

Given a system C that is described by a channel matrix M of size $n \times m$, e.g., the set of secret input values is $\{A_1, \cdots, A_n\}$, and the set of observable outcomes is $\{Z_1, \cdots, Z_m\}$.

General requirements. Since the attacker knows π and M, he is able to compute the marginal distribution of the output. Thus, firstly, the *distribution of the output* has to be preserved by the channel matrix M', where the noise has been added. If the policy does not preserve this distribution, the attacker might find out that the channel M has been changed, and he will try to study the system before making a guess, i.e., trying to get the *real* program code of the system.

Secondly, for each outcome Z_i, assume that $p(S = A_j|Z_i)$ is the *maximum* posteriori probability, then $p'(S = A_j|Z_i)$ is also the *maximum* posteriori probability, i.e., the *maximum property* of the posteriori distributions has to be preserved. For example, if M gives a posteriori distribution where $p(S = A_j|Z_i) = 0.8$, then the real posteriori probability given by M' might be $p(S = A_j|Z_i) = 0.6$. Thus, if the outcome is Z_i, the attacker thinks that the guess $S = A_j$ has a 80% chance to be correct. However, in reality, this guess only has a 60% chance of success. Notice that $p(S = A_j|Z_i)$ does not need to be equal to $p'(S = A_j|Z_i)$.

The preservation of the maximum property of the posteriori distribution is necessary. Consider a uniform posteriori distribution $\{p(S = A_1|Z_i) = p(S = A_2|Z_i) = p(S = A_3|Z_i) = \frac{1}{3}\}$ in the attacker's database. Following the requirement, the posteriori distribution given by M' has to be also uniform. If we do not require this, then the real distribution might possibly be $\{p'(S = A_1|Z_i) = 0.2, p'(S = A_2|Z_i) = 0.7, p'(S = A_3|Z_i) = 0.1\}$. According to his database, the attacker might guess $S = A_2$, since all three guesses have the same chance of being correct. In this case, the real probability would increase the chance of success, and thus, increase the leakage.

Reliability. Reliability of a system is the probability that a system will perform its intended function during a specified period of observation time. Let $\mathcal{R}_i$ ($\mathcal{R}_i \leq 1$) denote the reliability corresponding to the secret value A_i, i.e., the probability that the system will produce correct outcomes when the secret is A_i. Thus, the *overall reliability* of the system C is $\mathcal{R}_P = \sum_i p(A_i) \cdot \mathcal{R}_i$. The noisy-output policy produces noise, and thus, it reduces the reliability of the system. Therefore, we require that a noisy-output should guarantee at least a certain level of reliability.

Noisy-output policy. We propose a simple policy that might reduce the unwanted information flow, while still preserving a certain level of reliability. The following policy only aims to demonstrate the core idea of what a noisy-output policy should be. The practical policy might be customized due to requirements of the application. Given a channel matrix M that models a system C. A noisy-output policy changes M to M' by choosing an appropriate set of $\{\mathcal{R}_1, \cdots, \mathcal{R}_n\}$.

Noisy-output policy

1: For each row i of M, multiply each entry of the row by the reliability *variable* $\mathcal{R}_i$. Choose *randomly* one of the smallest entries, and add the value $1 - \mathcal{R}_i$ to it. Denote this modified matrix by M'.

2: Choose an overall reliability value that the policy has to guarantee, e.g., $\mathcal{R}_{min}$. Establish an inequality: $\sum_i p(A_i) \cdot \mathcal{R}_i \geq \mathcal{R}_{min}$.

3: For any outcome Z_i, let $p(O = Z_i)$ denote the probability determined by π and M, and $p'(O = Z_i)$ determined by π and M', establish an equation: $p(O = Z_i) = p'(O = Z_i)$.

4: For each outcome Z_i, if $\forall k. p(S = A_j|Z_i) \geq p(S = A_k|Z_i)$, then establish the following condition: $\forall k.\, p'(S = A_j|Z_i) \geq p'(S = A_k|Z_i)$.

5: Solve these equations and inequalities. The set $\{\mathcal{R}_1, \cdots, \mathcal{R}_n\}$, are chosen in such a way that the leakage given by M' is close to zero, and the reliability of the system $\mathcal{R}_P$ is as high as possible.

Notice that in Step 1, the sum of all entries of a row has to be 1; thus we have to add the value $1-\mathcal{R}_i$ to one of its entries. Step 3 establishes a set of equations, i.e., $m-1$ *independent equations* that correspond to $m-1$ observable outcomes, that preserve the output distribution. We also obtain $(n-1)\cdot m$ inequalities in Step 4, that preserve the maximum property of the posteriori distributions.

There always exists a trivial solution $\mathcal{R}_1 = \cdots = \mathcal{R}_n = 1$, i.e., M and M' are identical. When there are multiple solutions, we choose one that gives low leakage, but a high overall reliability. However, this does not always happen. A solution that guarantees a very low value of leakage might also give a low reliability. In fact, a negative leakage, i.e., when the attacker decides wrongly based on the observable outcomes, is not always necessary. The goal of the policy is to ensure that the attacker cannot gain knowledge from the observable outcomes. Thus, $\mathcal{R}_1, \cdots, \mathcal{R}_n$ are chosen such that the leakage is close to zero and the overall reliability gets a high value. Next, we show an important property of our policy.

Theorem 8.1 *Given a priori distribution π and a channel matrix M, the channel matrix M' modified from M by a noisy-output policy always gives a leakage quantity that is not greater than the one given by M.*

Proof: For any outcome Z_i, assume that the maximum likely secret is A_j. Since $p'(Z_i) = p(Z_i)$ and $p'(S = A_j|Z_i) = \mathcal{R}_j \cdot p(S = A_j|Z_i)$, thus $-p'(Z_i)\log p'(S = A_j|Z_i) \geq -p(Z_i)\log p(S = A_j|Z_i)$. Therefore, the value of remaining uncertainty given by π and M' is greater than or equal to the one given by π and M. As a consequence, the corresponding leakage quantity is reduced. $\square$

8.7.2 Example

Section 8.6.1 presents an example idea with noisy outcomes. However, the outcomes in this example do not follow a policy. Thus, the general requirements are not preserved. This section illustrates how noisy outcomes are designed. We consider the same channel M with the same priori uniform distribution π.

Based on π and M, the attacker knows that $p(O = Z_1) = p(O = Z_2) = p(O = Z_3) = \frac{1}{3}$. If $O = Z_i$, the attacker will guess $S = A_i$, since $p(S = A_i|O = Z_i)$ is the maximum probability.

Following the idea of a noisy-output policy, to protect the secret, a system operator might mislead the attacker by adding noise to the output, i.e., the real system is M'.

M'	Z_1	Z_2	Z_3
$S = A_1$	$\mathcal{R}_1$	$1 - \mathcal{R}_1$	0
$S = A_2$	0	$\mathcal{R}_2$	$1 - \mathcal{R}_2$
$S = A_3$	$1 - \mathcal{R}_3$	0	$\mathcal{R}_3$

To preserve the output distribution, the following equations have to be satisfied:

$$\begin{aligned} \frac{1}{3}\mathcal{R}_1 + \frac{1}{3}(1 - \mathcal{R}_2) &= \tfrac{1}{3} \\ \frac{1}{3}(1 - \mathcal{R}_1) + \frac{1}{3}\mathcal{R}_2 &= \tfrac{1}{3} \end{aligned}$$

Simplifying them, we obtain $\mathcal{R}_1 = \mathcal{R}_2 = \mathcal{R}_3$.

The maximum property of the posteriori distributions determines that $\frac{1}{2} \leq \mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_2 \leq 1$. For this example, the reliability of the system is $\mathcal{R}_P = \mathcal{R}_1$, and $\mathcal{L}_{Cachin}(C, \pi) = \mathcal{L}_{Smith}(C, \pi) = \log 3\mathcal{R}_1$.

Thus, a high value of $\mathcal{R}_1$, which guarantees a high overall reliability, also gives a high value of leakage. If the goal is to reduce the leakage, we might choose $\mathcal{R}_1 = \mathcal{R}_2 = \mathcal{R}_3 = \frac{1}{2}$, which gives the smallest value of leakage, i.e., $\log \frac{3}{2}$, but also a very low reliability. If a high reliability is required, $\mathcal{R}_1 = \mathcal{R}_2 = \mathcal{R}_3 = \frac{2}{3}$ might be a good choice.

Consider the *PWC* example, for the test $L_{in} = A_2$, following the policy, we can choose $\mathcal{R}_1 = 0.995, \mathcal{R}_2 = 0.5, \mathcal{R}_3 = 0.99$ to have $\mathcal{L}_{Cachin}(C, \pi, A_2) = -0.00275$ with the reliability $\mathcal{R} = 0.99$. However, if we consider both tests, i.e., $L_{in} = A_1$ and $L_{in} = A_2$, $\mathcal{R}_1 = \mathcal{R}_2 = \mathcal{R}_3 = 1$.

As mentioned above, a noisy-output policy enhances the security, but reduces the reliability of a system, i.e., the system does not always work in a proper way. However, the drawback of the reduced reliability can be overcome. Consider a situation of the *PWC* in which an user or an attacker provides a *correct* password, but the system rejects it, and then blocks his account — according to the one-try model. If this context is for the attacker, it would be very nice, since the attacker does not have a chance to use the account again. If this context is for the real user, the situation is different from the one for the attacker: the user is still allowed to *reactivate* the account by contacting the company/website administrators and proving that he is the real owner of the account, while the attacker cannot do the same.

The other way around, i.e., when the system accepts a wrong password, is not nice for the security. This is the reason that the policy should guarantee a high reliability. Notice that in this scenario, the system accepts the login, but no private information has been leaked, since the attacker still does not know the correct password. Thus, in the next login, there is a *high chance* that the system will reject this wrong password. Moreover, to avoid the situation of accepting a wrong password, the system might also implement the *two-factor authentication*, i.e., in addition to asking for something that *only the user knows* (e.g., username, password, PIN), the system also requires something that *only the user has* (e.g., ATM card, smart card). The ATM scenario illustrates the basic concept of most two-factor authentication systems, i.e., without the combination of both ATM card and PIN verification, authentication does not succeed.

Finally, it should be stressed that we only sketch the main idea of a noisy-output policy. However, for practical applications, depending on the real security requirements, the above policy might be customized, e.g., in **Step 1**, instead of choosing *randomly* one of the smallest entries, and adding $1 - \mathcal{R}_i$ to it, the policy can add to *each of the smallest entries* a value such that the sum of all these values is equal to $1 - \mathcal{R}_i$.

8.8 Related work

Our proposal for programs with low input borrows ideas from Malacaria et al. [62] and Yasuoka et al. [94]. For programs that contain both high and low inputs, Malacaria et al. and Yasuoka et al. define the information leakage as follows,

$$\mathcal{L}(C) = \mathcal{H}(S|L_{in}) - \mathcal{H}(S|L_{in}, O).$$

In most cases, the high input S is independent of the low input L_{in}, thus we can rewrite the above expression as,

$$\mathcal{L}(C) = \mathcal{H}(S) - \mathcal{H}(S|L_{in}, O).$$

Malacaria et al. define their measure based on *Shannon entropy*, while Yasuoka et al. only consider the leakage in the *final states*.

Basically, Malacaria et al. and Yasuoka et al. *fix* the initial low value by assigning it a specific value, and then define the leakage of a program as the leakage of a single test, while we define it as the maximum leakage of all tests. Besides, in these works, how to model programs with low input is not mentioned. Since the analysis is based on the program model, a wrong channel model, as in the *accuracy-based* approaches discussed below, would result in counter-intuitive values of leakage.

Clarkson et al. [31] argue that the classical *uncertainty-based* analysis is not adequate to measure information flow by analyzing the leakage of the same PWC in Section 8.5.2. Clarkson et al. *fix* the value of the secret, i.e., the password, and also the initial low value, i.e., the string that the attacker believes to be the password, and then analyze information flow under that specific circumstance.

According to [31], the attacker always tries $L_{in} = A_1$, since he believes that the password is highly likely to be A_1 (with probability 0.98). When the real password is A_3, PWC produces the outcome $O = 0$ with probability 1. Therefore, the channel matrix that models this scenario looks like,

	$O = 1$	$O = 0$
$S = A_3$	0	1

Based on the outcome, the attacker concludes that A_2 and A_3 each has 0.5 chance of being the password. Thus, initially, the attacker is *quite certain* about the value of the password, but after observing the outcome, he is *rather uncertain* about it. Therefore, there is an *increase* in uncertainty, and as a consequence, the value of leakage is negative, i.e., the classical uncertainty-based analysis would interpret this negative value as an absence of information flow. However, Clarkson et al. argue that this result flatly contradicts the intuition. Since from interacting with PWC, the attacker gains more knowledge by learning that the password is not A_1. Thus, the information flow should be positive. Therefore, the authors suggest that the classical uncertainty-based measures are inadequate to quantify information flow. They propose another approach called *accuracy-based analysis.* This trend of research has been expanded in [30, 49, 50, 42].

However, the approach proposed by Clarkson et al. reports a leakage value that is *inconsistent* with the size of the flow. For example, consider the above example. Since cardinality of the password is 3, the maximum size needed to store the password is $\log 3 = 1.5849$ bits. However, the accuracy-based measure reports a leakage of 5.6438 bit [30]. Thus, the quantity of the secret information flow exceeds the size needed to store the secret.

We believe that there is a flaw in the way Clarkson et al. model the system. Clarkson et al. fix the value of the secret. Thus, this does not capture precisely the idea of an information-theoretic channel. The information-theoretic channel has the secret as the input, and the entropy of the input quantifies the uncertainty involved in predicting the value of the secret. Thus, if the value of the secret is fixed, it implies that the priori distribution on the possible values of the secret is not valid anymore, i.e., the secret is now a certain value with the absolute probability 1. As a consequence, the entropy of the input does not reflect the true meaning of the initial uncertainty. Also notice that in information theory, if the value of a random variable is fixed, then its amount of information is 0. For example, suppose one transmits 100 bits. If these bits are fixed, i.e., they are known ahead of transmission with the absolute probability, information theory indicates that no information has been transmitted. If, however, each bit is equally and independently likely to be 0 or 1, 100 bits have been transmitted. Therefore, in these approaches, a wrong channel model has led to misleading results, i.e., a negative uncertainty-based result, or a size-inconsistent accuracy-based result.

Alvim et al. discuss limitations of the classical information-theoretic channel by showing that it is not a valid model for *interactive systems* where secrets and observables can *alternate* during the computation and *influence* each other [6]. In [4], Alvim et al. also discuss the example of the password checker. They fix the password by assigning it a specific value, and then consider the initial low values as the only input to the channel. As discussed before, this idea does not reflect the true idea of the information-theoretic channel.

Köpf et al. also consider systems with low input, i.e., cryptosystems where the attacker can control the set of input messages [54]. However, their proposal is only for deterministic systems, i.e., for each input, the system produces only one output, while in our proposal, the output might be nondeterministic and probabilistic. Besides, Köpf et al. consider a different threat model, i.e., the *multiple-try* guessing model, and they put a restriction on the priori distribution of the secret, requiring it to be uniform.

Based on Smith's measure and the point of view of the probability of error — the probability that the attacker guesses the secret wrongly, Braun et al. [20] define the leakage as the difference between the probabilities of error of a priori guess and a posteriori guess, i.e., before and after observing the output. They define this difference in two different ways: one is *multiplicative* leakage, which coincides with Smith's measure apart from the absence of the logarithm, and another is *additive* leakage, which is new. This paper also shows that for these two measures, the worst case of leakage — the maximum leakage over all initial distributions of the secret — can be computed easily. Notice that this paper does not consider the low input, and the maximum leakage is defined by quantifying over the initial distributions, not over the initial low values, as in our proposal.

The idea of adding noise to the output comes from *differential privacy* control, i.e., the problem of protecting the privacy of database's participants when performing *statistical* queries [4, 5, 3]. The differential privacy control also uses some output-perturbation mechanism to report a noisy answer among the correct ones for the queries. Thus, while the attacker is still able to learn properties of the population as a whole, he cannot learn the value of an individual. To construct an efficient noisy-output policy for a statistical database, it is necessary to consider the balance between *privacy*, i.e., how difficult it is to guess the value of an individual, and *utility*, i.e., the capacity to retrieve accurate answers from the reported ones. In [55], Köpf et al. also explore a similar idea to cope with timing attacks for cryptosystems, i.e., randomizing each cipher-text before decryption. As a consequence, the strength of the security guarantee is enhanced, while the efficiency of the cryptosystem is decreased, since the execution time of the cryptographic algorithm is increased.

8.9 Conclusions

This chapter extended the classical quantitative information flow analysis by considering a context where programs also contain low input, i.e., an attacker can set up the initial low values of the program, based on his knowledge about the private data. We also introduced a new measure — based on the definition of conditional min-entropy by Cachin — for the notion of remaining uncertainty in the analysis.

This chapter also pointed out an interesting property of the analysis, namely: when the output contains noise, the value of leakage might be negative. This property gives rise to a proposal for a new measure of information flow. Finally, an efficient noisy-output policy, i.e., adding noise to the output but still preserving a high overall reliability, was also introduced.

Notice that this model of analysis fixes the scheduler, i.e., we assume that an attacker cannot choose the scheduler. It makes this analysis more suitable for sequential programs. Sequential programs contain no parallel operator; thus the scheduler is not necessary for such programs. For multi-threaded programs, as mentioned before, it is necessary to consider the effect of the scheduling policy. Thus, to apply this analysis to the multi-threaded setting, we should extend the observation of the channel not only to include traces, but also scheduling decisions at each step. However, this model of channels is complicated, since we then have to integrate many types of data into the channel. Instead, the next chapter presents a simple, more efficient analysis for multi-threaded programs.

Chapter 9

Multi-threaded Programs with the Effect of Schedulers

9.1 Introduction

This chapter proposes a model that aims to analyze quantitatively information flow of *multi-threaded programs*. In this model, we lift the restriction of the previous chapter that the attacker cannot control the scheduler's decisions. For multi-threaded programs, the effect of choices of the scheduler should not be ignored in the analysis, as illustrated by the following example

Example 9.1 *Consider the following program where S is a 2-bit secret variable with the priori uniform distribution,*

$$O := S/2 \parallel O := S \bmod 2.$$

Assume that the attacker executes this program with a uniform scheduler. Since S is a uniform 2-bit data, there are 4 possible traces $\{00, 01, 10, 11\}$ with the same probability of occurrence.

S	0		1		2		3	
$T_{\mid O}$	0	0	0	1	1	0	1	1
	0	0	1	0	0	1	1	1

If the obtained trace is either 00 or 11, the attacker knows for sure that S is 0, or 3, respectively. However, if the trace is 01 or 10, he is only able to guess that S is either 1 or 2, with the same probability of success. Thus, with this scheduler, the secret is not leaked completely.

However, if the attacker chooses a scheduler which always picks $O := S/2$ first, there are also 4 possible traces $\{00, 01, 10, 11\}$ with the same probability of occurrence; but in this case, the attacker can always derive the value of S correctly.

Thus, to obtain an appropriate model of quantitative information flow analysis for multi-threaded programs, we need to: (1) consider how public values in intermediate states help to reduce the attacker's initial uncertainty about the private data, (2) take into account the effect of schedulers on the overall leakage of the program, since the outcomes of a multi-threaded execution depend on the scheduler's choices.

The model of analysis presented in this chapter does not follow the traditional channel-based approaches. Instead, this analysis models the program execution under the control of a given scheduler by a PKS, where states denote the probability distributions of private data. Based on this program model, we define the leakage of an execution trace, i.e., the leakage that can be derived from a sequence of public data obtained during the program execution. The program leakage is then given as the *expectation* of the trace-leakage values.

Organization of the chapter. The remainder of this chapter starts with a description of the program model. This is a PKS where state labels are distributions of private data. This model describes the execution of a multi-threaded program under the control of a scheduler for the analysis. Based on this program model, in Section 9.3 and Section 9.4, we discuss how trace leakage and program leakage are computed. Section 9.5 provides a case study, and then compares our result with the existing channel-based approaches. Section 9.6 discusses a technique for computing the leakage. Section 9.7 discusses related work, while Section 9.8 concludes the chapter.

Origins of the chapter. The model of analysis presented in this chapter was published in the proceedings of the *11th International Workshop on Quantitative Aspects of Programming Languages and Systems* (QAPL'13) [66].

9.2 Program model

In this analysis, instead of the information-theoretic channel, we model the execution of a multi-threaded program under the control of a probabilistic scheduler by a variant of the PKS given in Definition 2.2. Basically, the following program model is a PKS, but it labels states with distributions of the private variable S, i.e., the labeling function $V : \mathcal{S} \to \mathcal{D}(S)$ assigns a distribution $\mu \in \mathcal{D}(S)$ to each state $c \in \mathcal{S}$, describing the attacker's knowledge about S at each state.

Definition 9.1 (Program model) *The program model is a PKS $\mathcal{A} = \langle \mathcal{S}, I, Var, V, \to \rangle$ consisting of (i) a set $\mathcal{S}$ of* states*, (ii) an initial state $I \in \mathcal{S}$, (iii) a finite set of variables $Var = H \cup L$, (iv) a* labeling function *$V : \mathcal{S} \to \mathcal{D}(S)$, where $S \in H$, and (v) a* transition relation *$\to \subseteq \mathcal{S} \times \mathcal{D}(\mathcal{S})$.*

The distribution of S changes from state to state along a trace, depending on the *public values* in the states and the *program commands* (chosen by the scheduler) that result in such observables. This idea is sketched by the following example.

Initially, given that the attacker knows that the value of S is in the set $\{0, 1, 2, 3\}$ without any priority, i.e., assuming a priori uniform distribution of S: $\pi = \{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}$. Consider the program:

$$O := S/2 \parallel O := S \mod 2.$$

The scheduler might choose either of the two threads to execute first, and either of them might result in the same O, e.g., $O = 1$. If the executed thread is $O := S/2$, the updated distribution is $\{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto \frac{1}{2}, 3 \mapsto \frac{1}{2}\}$; otherwise, it is $\{0 \mapsto 0, 1 \mapsto \frac{1}{2}, 2 \mapsto 0, 3 \mapsto \frac{1}{2}\}$.

Thus, the program is seen a *distribution transformer*, from the priori distribution of S in the initial state to the final distributions in the final states. By observing the traces of the distribution transformation, the attackers might be able to learn information about the initial private data. Our goal is to measure how much secret information that has been deduced by the attacker based on these observations.

9.3 Leakage of a program trace

Program $C6$ (Example 8.7) in Section 8.4.2 shows that the leakage of a trace is not simply the sum of leakage values of transition steps along the trace. This section addresses how we can compute a program-trace leakage.

Example 9.2 *Consider again $C6$,*

$$\begin{aligned} &O := 0; \\ &O := S \,\&\, (001)_b; \\ &O := S \,\&\, (011)_b; \end{aligned}$$

Let $(s_3 s_2 s_1)_b$ denote the binary form of S. We assume that the priori distribution of S is uniform. The execution of this program results in just one trace, i.e., $\langle(000)_b\rangle \longrightarrow \langle(00s_1)_b\rangle \longrightarrow \langle(0s_2s_1)_b\rangle$, where a state $\langle\rangle$ is represented by the public value O.

Assume that $s_2 = 1$ and $s_1 = 1$, the obtained trace is

$$\langle(000)_b\rangle \longrightarrow \langle(001)_b\rangle \longrightarrow \langle(011)_b\rangle.$$

At the initial state $\langle(000)_b\rangle$, the attacker's initial uncertainty is represented by the priori distribution of S, i.e., $\{0 \mapsto \frac{1}{8}, 1 \mapsto \frac{1}{8}, 2 \mapsto \frac{1}{8}, 3 \mapsto \frac{1}{8}, 4 \mapsto \frac{1}{8}, 5 \mapsto \frac{1}{8}, 6 \mapsto \frac{1}{8}, 7 \mapsto \frac{1}{8}\}$. At state $\langle(001)_b\rangle$, the attacker learns that the last bit of S is 1. Thus, the distribution of S changes, for example, S cannot be 0. Hence, the updated distribution at state $\langle(001)_b\rangle$ is $\{0 \mapsto 0, 1 \mapsto \frac{1}{4}, 2 \mapsto 0, 3 \mapsto \frac{1}{4}, 4 \mapsto 0, 5 \mapsto \frac{1}{4}, 6 \mapsto 0, 7 \mapsto \frac{1}{4}\}$. Similarly, at the final state $\langle(011)_b\rangle$, the updated distribution is $\{3 \mapsto \frac{1}{2}, 7 \mapsto \frac{1}{2}\}$[1].

Based on the final distribution of S, the attacker derives that the value of S is either 3 or 7. His uncertainty on secret information is reduced by the knowledge gained from the trace observation.

Since the program is a distribution transformer, the distribution of private data at the *initial state* of a trace can present the *initial uncertainty* of the attacker about the secret, and the distribution of private data at the *final state* can present his *final uncertainty*, after the trace has been observed. Thus, we can define the leakage of a program trace as,

Leakage of a program trace = Initial uncertainty - Final uncertainty.

Measure of uncertainty. Given a distribution of private data, the *best* strategy of the one-try guessing model is to choose the value with the maximum probability. 'Best' means that this strategy induces the smallest probability of guessing wrongly. Let $\mathbf{S}$ be the set of possible values of S, the value that affects the notion of uncertainty is $\max_{s\in\mathbf{S}} p(s)$. If $\max_{s\in\mathbf{S}} p(s) = 1$, the uncertainty must be 0, i.e., the attacker already knows the value of S. Thus, the *notion*

[1]We leave out the elements that have probability 0.

of uncertainty is computed as the *negation* of the logarithm of $\max_{s\in\mathbf{S}} p(s)$, i.e., uncertainty $= -\log\ \max_{s\in\mathbf{S}} p(s)$, where the negation is used to ensure the *non-negative uncertainty*[2].

This measure coincides with the notion of Rényi's min-entropy. Thus, given a distribution of S, the attacker's uncertainty about the secret in this analysis is: Uncertainty $= \mathcal{H}_{R\acute{e}nyi}(S)$.

Therefore, the leakage of a program trace T is,

$$\mathcal{L}(T) = \mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{i}}_T) - \mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{f}}_T),$$

where $\mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{i}}_T)$ is Rényi's min-entropy of S with the initial distribution, and $\mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{f}}_T)$ is Rényi's min-entropy of S with the final distribution, i.e., the distribution of the secret at the final state in T.

Thus, following our measure, in Example 8.7, $\mathcal{L}(T) = -\log\frac{1}{8} - (-\log\frac{1}{2}) = 2$. This value matches the intuitive understanding that $C6$ leaks 2 bits of private information.

Consider Example 8.6. Assume that $s_1 = s_2 = s_3 = 1$, the execution of $C5$ results in the trace $\langle(000)_b\rangle \longrightarrow \langle(100)_b\rangle \longrightarrow \langle(011)_b\rangle$.

At state $\langle(100)_b\rangle$, the attacker learns that the first bit of S is 1. Thus, the distribution of S is $\{4 \mapsto \frac{1}{4}, 5 \mapsto \frac{1}{4}, 6 \mapsto \frac{1}{4}, 7 \mapsto \frac{1}{4}\}$. At the final state $\langle(011)_b\rangle$, the distribution is updated to $\{7 \mapsto 1\}$, which is different from the final distribution given by $C6$. Hence, $\mathcal{L}(T) = -\log\frac{1}{8} - (-\log 1) = 3$. This result also matches the intuition that the attacker is able to derive S precisely from the execution trace of $C5$.

Notice that in our approach, instead of the *remaining uncertainty*, we use the notion of *final uncertainty*. Both *initial* and *final uncertainty* are denoted by the *same* entropy measure, i.e., Rényi's min-entropy. In addition, the notion of *remaining uncertainty* depends only on the *public outcomes* of the execution, while the *final uncertainty* is based on the distribution computed for the final state, which takes into account the *public values* in the intermediate states along the trace, and also the *program commands* (selected by the scheduler) that result in such observable values.

9.4 Leakage of a multi-threaded program

The execution of a multi-threaded program C under the control of a scheduler δ often results in a set of traces $\mathit{Trace}(\mathcal{A}_\delta)$. Therefore, the leakage of C is

[2]The quantity of uncertainty is alway non-negative, which is different from the quantity of information flow.

computed as the *expected* value of its trace leakages, i.e.,

$$\begin{aligned}\mathcal{L}(C,\pi) &= \textstyle\sum_{T\in \mathit{Trace}(\mathcal{A}_\delta)} p(T)\cdot \mathcal{L}(T)\\ &= \textstyle\sum_{T\in \mathit{Trace}(\mathcal{A}_\delta)} p(T)(\mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{i}}_T) - \mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{f}}_T)).\end{aligned}$$

Since $\mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{i}}_T)$ is the same for all $T \in \mathit{Trace}(\mathcal{A}_\delta)$, for notational convenience, we simply write it as $\mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{i}})$. Thus, we can rephrase the above expression as follows,

$$\mathcal{L}(C,\pi) = \mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{i}}) - \sum_{T\in \mathit{Trace}(\mathcal{A}_\delta)} p(T)\cdot \mathcal{H}_{R\acute{e}nyi}(S^{\mathbf{f}}_T).$$

The next section provides a case study showing that this measure is different from the traditional channel-based approaches, including the one we proposed in the previous chapter.

9.5 A case study

The following case study illustrates how the leakage of a multi-threaded program is computed. It also shows that this model of analysis is more precise for multi-threaded programs than other measures discussed in the previous chapter. Consider again Example 8.8 in Section 8.5.2,

```
$O := 0$;
{if ($O = 1$) then $O := S/4$ else $O := S \bmod 2$} || $O := 1$;
$O := S \bmod 4$;
```

where S is a 3-bit unsigned uniform integer.

The execution of this program with a uniform scheduler is modeled by a PKS $\mathcal{A}$ in Figure 9.1. The PKS consists of 20 states that are numbered from **0** (the initial state) to **19**. The contents of each state is the value of O in that state, e.g., in the initial state, the value of O is 0, which corresponds to the first command of the program: $O := 0$.

Let C_1 and C_2 denote the left and right threads. Since we consider the uniform scheduler, either thread C_1 or C_2 can be picked next with the same probability $\frac{1}{2}$. If the scheduler picks C_2 before C_1, $\mathcal{A}$ evolves from state **0** to state **1**, where $O = 1$. If C_1 is picked first, $\mathcal{A}$ might evolve from state **0** to either state **2** or state **3** with the same probability $\frac{1}{4}$. Since the value of O in state **0** is 0, the command $O := S \bmod 2$ is executed. Since the possible values of S

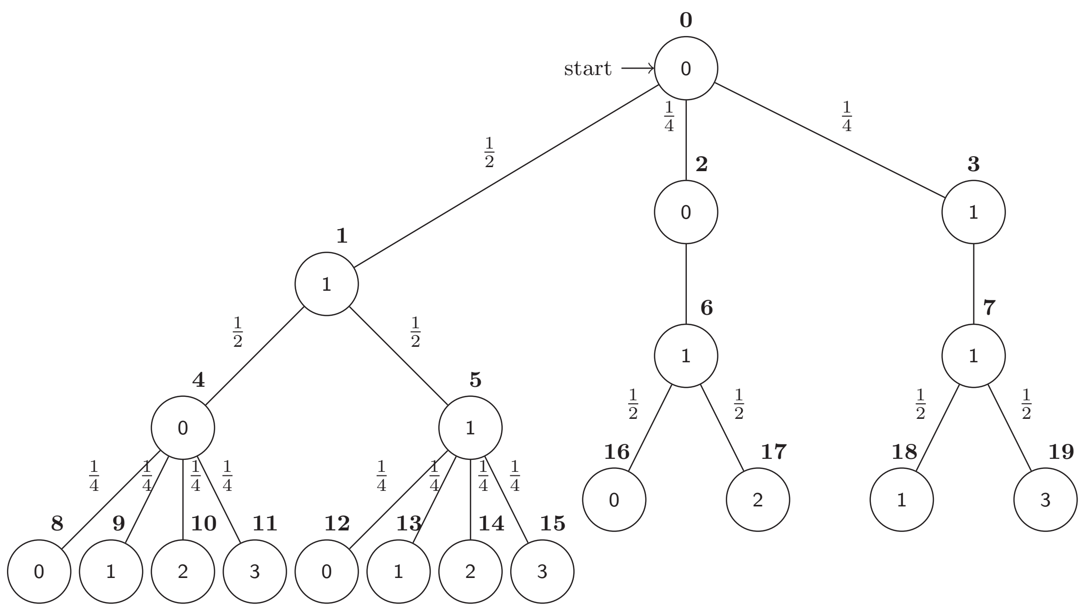


Figure 9.1: Model of the Program Execution

are $\{0, \ldots, 7\}$, the outcome O might be 0 (state **2**) if $S \in \{0, 2, 4, 6\}$, or 1 (state **3**) if $S \in \{1, 3, 5, 7\}$.

At state **1**, $\mathcal{A}$ might evolve to either state **4** or state **5** with the same probability. Since currently, O is 1, the command $O := S/4$ is executed. Thus, O might be 0 if $S \in \{0, 1, 2, 3\}$, or 1 if $S \in \{4, 5, 6, 7\}$.

The PKS $\mathcal{A}$ evolves from one state to another until the execution terminates, i.e., when the last command $O := S \mod 4$ is executed.

The attacker's initial uncertainty about S is denoted by the uniform distribution, i.e.,

$$\pi = \{0 \mapsto \frac{1}{8}, 1 \mapsto \frac{1}{8}, 2 \mapsto \frac{1}{8}, 3 \mapsto \frac{1}{8}, 4 \mapsto \frac{1}{8}, 5 \mapsto \frac{1}{8}, 6 \mapsto \frac{1}{8}, 7 \mapsto \frac{1}{8}\}.$$

At state **1**, the distribution of S is still uniform, since the attacker learns *nothing* from the command $O := 1$. At state **4**, since the execution of $O := S/4$ results in 0, the attacker learns that the true value of S must be in the set $\{0, 1, 2, 3\}$, with the same probability. Thus, the updated distribution of S at this state is:

$$\{0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\}.$$

In the next step, the outcome of $O := S \mod 4$ helps the attacker to derive S precisely, e.g., at state **8**, since $O = 0$, the distribution of S is:

$$\{0 \mapsto 1\}.$$

Similarly, the attacker is also able to derive S precisely, basing on the final distributions at states **9**, . . ., **15**.

At state **2**, the execution of $O := S \mod 2$ results in 0. Thus, the distribution of S at this state is:

$$\{0 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 4 \mapsto \frac{1}{4}, 6 \mapsto \frac{1}{4}\}.$$

This distribution remains unchanged at state **6**, since no information is gained from the execution of $O := 1$. At state **16**, the update distribution of S is:

$$\{0 \mapsto \frac{1}{2}, 4 \mapsto \frac{1}{2}\},$$

since the execution of $O := S \mod 4$ results in 0. The same form of distributions is also obtained at states **17**, **18**, **19**.

Among the 12 possible traces, 8 traces have the final uncertainty 0, i.e., $\mathcal{H}_{Rényi}(S_T^{\mathbf{f}}) = -\log 1 = 0$, and the other 4 traces have the final uncertainty 1, i.e., $\mathcal{H}_{Rényi}(S_T^{\mathbf{f}}) = -\log \frac{1}{2} = 1$. The probability of traces with the final uncertainty 0, i.e., traces end in state **8**, . . ., **15**, is equal to the probability of traces with the final uncertainty 1. Thus, according to this analysis,

$$\mathcal{L}(C, \pi) = 3 - (\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1) = 2.5 \text{ (bits)}.$$

This value coincides with the real leakage of the program. As discussed before, the last command $O := S \mod 4$ always reveals the last 2 bits of S, and the first bit of S is leaked with the probability $\frac{1}{2}$.

9.5.1 Comparison

In Section 8.5.2, we analyzed this example. According to the traditional approaches that are based on information-theoretic channels, $\mathcal{L}_{Cachin}(C, \pi) = 2.53$ (bits) (our measure), while $\mathcal{L}_{Smith}(C, \pi) = 2.58$ (bits) (traditional measure). These values do not match the real value of leakage. This proposal gives a more precise quantity of leakage, since it also takes into account the effect of the scheduler, i.e., the distribution of private data representing the attacker's knowledge depends also on the scheduler's choice. In particular, the attacker knows which program-command execution results in the public value in states.

Notice that the measure of the final uncertainty

$$\sum_{T \in \mathit{Trace}(\mathcal{A}_\delta)} p(T) \cdot \mathcal{H}_{R\acute{e}nyi}\,(S_T^{\mathbf{f}})$$

is similar to Cachin's conditional min-entropy. In fact, in the information-theoretic approach, if we also take into account the effect of schedulers, i.e., if observations in our information-theoretic channel are not only traces, but also include scheduling decisions at each step, and considering Cachin's conditional min-entropy, the measure proposed in the previous chapter coincides with this approach. However, this model of information-theoretic channel is more complicated than the program model we present here, since we have to include an extra low variable that models the choice of scheduler, and define how this variable is updated.

9.6 Technique for computing leakage

Another aspect is to develop a technique for computing the leakage of a multi-threaded program. The computation consists of two steps. First, the execution of a multi-threaded program C under the control of a given scheduler is modeled as a PKS $\mathcal{A}$ in a standard way: The states of $\mathcal{A}$ are tuples $\langle C, V \rangle$, consisting of a program fragment C and a valuation $V : \mathcal{S} \to \mathcal{D}(S)$. The state transition relation $\to$ follows the small-step semantics of C. Based on the executed command and the obtained public result, the distribution of private data at each state is derived. We apply Kozen's probabilistic semantics [56] to present the transformation of probability distributions of S during the program execution. The intuition behind state transition relation is that it transforms the *input distribution* of S to an *output distribution*, so that the execution results in traces of probability distribution of S.

The value of leakage then follows trivially as the difference between *the initial uncertainty* and *the expected value* of the final uncertainties of all traces.

9.7 Related work

As mentioned in the previous chapter, the classical measures might be counter-intuitive in some situations, and also prone to conflicts when comparing between programs. To avoid the conflict, Zhu et al. [98] propose to view the program execution as a probabilistic state transition system, where states denote probability distributions of private data. This approach is close to our model in spirit, but their approach only aims to compare between programs. Their approach constructs probability distribution functions over the residual uncertainty about private data, and then the comparison between programs is done via the *means* and the *variances* of the distributions.

The quantitative security analysis proposed by Chen et al. [25] for multi-threaded programs defines the leakage of each scheduler interleaving. The leakage of a program is then given by the expected value over all interleavings. The basis idea of this approach is that for each interleaving of the scheduler, a set of possible traces is obtained. Based on the public data of final states on these traces, the leakage of this interleaving is determined. This approach is imprecise, since it does not consider leakage in intermediate states and take into account the distribution of traces in an interleaving.

In another attempt, Chen et al. [26] define the leakage of a program trace as the sum of the products of the leakage generated by each transition step and the transition's probability. Examples in Section 8.4 show that this idea is not precise. This approach is only suitable to estimate the *coarse maximum* and *minimum leaks* of a program, i.e., the maximal and minimal values of trace-leakage. However, these values are not very helpful to judge programs, or to compare between different programs, because the *gap* between the maximum and minimum values is often large.

To define the quantitative information flow for multi-threaded programs, Malacaria et al. [62] and Andrés et al. [9] follow the classical information-theoretic approach, but the observable data in their approaches are traces of public variables, instead of only the final public outcomes. Malacaria's approach is based on Shannon entropy, while Andrés's approach uses min-entropy with Smith's version of conditional min-entropy. These measures do not give results close to the real leakage values.

Mu et al. [65] also model the execution of a probabilistic program by a probabilistic state transition system where states represent distributions of private data. However, the analysis is only for sequential programs.

9.8 Conclusions

This chapter proposed a novel approach to estimate the leakage of a multi-threaded program. Instead of using the traditional information-theoretic channel, this approach models the execution of a multi-threaded program under the control of a scheduler by a probabilistic state transition system. States of the transition system are labeled with probability distributions of private data. This distribution reflects the attacker's knowledge about the secret value. The distribution of private data changes from state to state along a trace, depending on the relation between private data and public data, and also on the command executions resulting in such public data. In comparison with the channel-based models of security analysis, we believe that this approach gives a more accurate way to study quantitatively the security of multi-threaded programs, i.e., it agrees with the intuition about the real leakage values. Notice that the idea of low input and noisy output in the previous chapter can also be applied to this program model.

Chapter 10

Conclusions

This chapter provides an overall conclusion of the thesis. We summarize each part of this thesis, together with a discussion of its contributions. We end this thesis with ideas for future work.

10.1 Thesis summary

Qualitative information flow properties. This thesis presented *scheduler-specific* definitions of observational determinism: scheduler-specific observational determinism (SSOD) and scheduler-specific probabilistic observational determinism (SSPOD) properties. We showed that these formalizations capture the intuitive idea of confidentiality for multi-threaded programs more precisely than other formalizations in the literature.

SSOD gives a formalization of a confidentiality property for *non-deterministic* multi-threaded programs. If the execution of a program under the control of a certain scheduler is accepted by SSOD, no secret information can be derived from the publicly observable data traces, and also from the relative ordering of low-variable updates.

SSPOD considers also the *probabilistic behavior* of the execution, and thus makes this property usable in a larger context. It is important to consider the probabilistic property, since this captures the realistic behavior of multi-threaded programs.

To avoid *refinement attacks*, where an attacker chooses a suitable scheduling policy to refine the set of possible program traces, the security specification should be *scheduler-independent*. Therefore, this thesis also proposed a definition of scheduler-independent observational determinism. This is derived from the scheduler-specific definitions by quantifying them over all possible scheduling policies. If a program is accepted by the scheduler-independent security property, it guarantees that no private information is leaked, regardless of any scheduling policy used to deploy the program.

Property verification and attack synthesis. To check whether secret information has been leaked, this thesis developed methods to verify automatically whether the program execution satisfies the confidentiality properties. First, we explored the approach of *logic-based verification* based on self-composition. We found that the compliance with SSOD can be verified by checking its temporal logic characterization. The characterization is developed in two steps: first we characterize *stuttering equivalence*, which serves as the basis of SSOD, and then we characterize the SSOD conditions. This results in a conjunction of an LTL and a CTL formula. This characterization is an important step towards model checking observational determinism properties.

Besides relying on existing temporal logic verification tools, this thesis also developed direct *algorithmic verification* methods for both SSOD and SSPOD. The verification uses a combination of new and existing algorithms. The new algorithms solve standard problems, i.e., checking all-trace stuttering equivalence of a Kripke structure that models the program execution and stuttering-trace equivalence between Kripke structures, which makes them applicable also in a broader context, outside the security scenarios. The advantage of algorithmic model-checking methods is that they can generate counter-examples when the verification fails. We extend our algorithms for this purpose, i.e., presenting counter-examples to synthesize information leaking attacks.

The algorithmic verification techniques together with the attack synthesis methods are implemented as a part of the *LTSmin tool set*. We provide case studies to show the feasibility of the algorithmic approaches and the practical application of the tool.

Quantitative information flow analysis. In case private data have been leaked, this thesis also discussed how to quantify this information flow. The classical *quantitative* analysis of information flow only considers simple cases where the *only input* is the secret. This thesis extended this context by considering applications that contain both *low* and *high input*, i.e., programs where an attacker can interact with the initial public values. For such programs, we adapted the traditional analysis by considering the initial low values as *parameters* of the channel that models the program execution. Via our analysis, we also discussed an important property of the information flow, i.e., the quantity of information flow might be *negative* in case the output contains noise, i.e., the system *secretly* generates noisy outcomes. In this context, the noise might mislead the attacker's belief about the private data, and thus, it increases his final uncertainty. We believe that this property would change the way people often think about how to appropriately measure the notion of uncertainty. We also discussed how to design an *efficient noisy-output policy*, which generates noisy outcomes, while still guaranteeing the system a high overall reliability.

Our quantitative analysis of information flow also proposed to consider Cachin's conditional min-entropy as an (optional) measure for the notion of the remaining uncertainty. In the literature, it has been suggested that there might be different measures of information leakage for different scenarios. We showed that our new measure agreed with the intuition about what the leakage should be in many cases. This measure has not previously been used in the theory of quantitative information flow.

In addition, this thesis also proposed a novel model of analyzing quantitatively information flow of multi-threaded programs For multi-threaded programs, it is necessary to consider the scenario that the attacker is able to select an appropriate scheduler to control the execution. For this context, we do not follow the traditional information-theoretic approaches, but model the program execution by a probabilistic transition system, where states denote the probability distributions of private data. In this approach, the notion of *program-trace leakage* is defined; and then *the leakage of a program* is given as the *expectation* of all trace-leakage values. Via a case study, we show that this approach gives a more accurate result than the traditional information-theoretic approaches. Thus, we consider this as an important contribution in the field of quantitative security analysis for multi-threaded programs.

10.2 Future work

Qualitative information flow analysis. Based on the work presented in this thesis, we see several directions for future work. It would be interesting to see if the verification algorithms for scheduler-specific confidentiality properties can be further optimized for *particular classes* of schedulers, e.g., for all round robin schedulers. Another direction for future work is to run the *attack synthesis for scheduling policies*, i.e., to find a set of schedulers that execute a given program securely, or a set of schedulers that can break the program's confidentiality.

It would also be interesting to continue the study of other security properties, i.e., anonymity, integrity, and availability. We think that the same approach of adapting existing model checking algorithms are also feasible to efficiently and precisely verify these properties.

Quantitative information flow analysis. Since there are many measures proposed for quantitative information flow analysis, and no unique measure is likely to suit all contexts, it might be interesting to evaluate each measure to determine under which circumstances, a certain measure might give the *best* answer.

Besides, it is also interesting to propose a measure for the *multiple-try* guessing model, i.e., when the current guess fails, the attacker might update his knowledge about the secret to make a new guess.

Finally, the existing quantitative information flow analysis of the *one-try* guessing model is only based on the value that the attacker believes to be the secret. Thus, the analysis ignores the *extra* useful information that might be derived from the information about the values that are not chosen by the attacker. For example, given two posteriori distributions $\pi = \{p(S = A|O = Z) = 0.5, p(S = B|O = Z) = 0.5, p(S = C|O = Z) = 0\}$ and $\pi' = \{p(S = A|O = Z) = 0.5, p(S = B|O = Z) = 0.25, p(S = C|O = Z) = 0.25\}$. The traditional analysis does not distinguish between these two distributions, since the maximum probabilities are 0.5 in both.

Following the idea of the one-try guessing model, the attacker would guess the secret to be A with the confidence 0.5 in both cases. If his guess is correct, he gets the secret. However, consider the case when he makes a wrong guess. If the posteriori probability is π, the attacker now knows for sure that the secret is B — notice that this is still the one-try guessing model, since the attacker does not make another guess. If the obtained posteriori probability is π', he only derives that B and C each has a 50% chance of being the secret. Thus, in case the guess is *wrong*, the attacker's knowledge about the secret changes differently for the two distributions, i.e., $\pi = \{p(S = A|O = Z) = 0, p(S = B|O = Z) = 1, p(S = C|O = Z) = 0\}$ and $\pi' = \{p(S = A|O = Z) = 0, p(S = B|O = Z) = 0.5, p(S = C|O = Z) = 0.5\}$. Hence, these two distributions should be considered differently in the analysis.

Therefore, we believe that the traditional analysis based on the notion of uncertainty does not give a *complete* answer to the problem of analyzing quantitatively information flow of a system. Future work may explore whether it is useful to also include the *attacker's disbelief*, i.e., the information about the values that the attacker disbelieves to be the secret, to the model of analysis. This might help to distinguish two systems that are indistinguishable by the traditional approaches, but obviously, one leaks more information than the other; and also give results that are closer to the real leakage values.

List of papers by the author

Journals/Conferences/Workshops

1. T.M. Ngo, M. Stoelinga, and M. Huisman. Effective verification of confidentiality for multi-threaded programs. In *Journal of Computer Security* (JCS), volume 22, number 2/2014, pages 269-300, IOS Press.

2. T.M. Ngo, and M. Huisman. Quantitative security analysis for programs with low input and noisy output. In *Proceedings of the 6th International Conference on Engineering Secure Software and Systems* (ESSoS'14), volume 8364 of LNCS, pages 77-94, Springer-Verlag, Munich, Germany, 2014.

3. T.M. Ngo, M. Stoelinga, and M. Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems* (ESSoS'13), volume 7781 of LNCS, pages 107-122, Springer-Verlag, Paris, France, 2013.

4. T.M. Ngo, and M. Huisman. Quantitative security analysis for multi-threaded programs. In *Proceedings of the 11th International Workshop on Quantitative Aspects of Programming Languages and Systems* (QAPL'13), volume 117 of EPTCS, pages 34-48, Rome, Italy, 2013.

5. M. Huisman, and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *Proceedings of the 2011 International Conference on Formal Verification of Object-Oriented Software (Revised Selected Papers)* (FoVeOOS'11), volume 7421 of LNCS, pages 178-195, Springer-Verlag, Turin, Italy, 2011.

6. T.M. Ngo, J.H. Weber, and K.A.S. Abdel-Ghaffar, New upper bounds on the separating redundancy of linear block codes, In *Proceedings of the 30th Symposium on Information Theory in the Benelux*, pages 209-216, Eindhoven, The Netherlands, 2009.

Technical reports

7. T.M Ngo, M. Stoelinga, and M. Huisman. (2012) Confidentiality for Probabilistic Multi-Threaded Programs and Its Verification. Technical Report TR-CTIT-13-01, Centre for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625.

8. M. Huisman, and T.M. Ngo (2011) Scheduler-specific Confidentiality for Multi-Threaded Programs and Its Logic-Based Verification. Technical Report TR-CTIT-11-22, Centre for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625.

Bibliography

[1] A.V. Aho and J.E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] R. Alur, P. Černy, and S. Chaudhuri. Model checking on trees with path equivalences. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 664–678. Springer-Verlag, 2007.

[3] M.S. Alvim, M.E. Andrés, K. Chatzikokolakis, P. Degano, and C. Palamidessi. Differential privacy: on the trade-off between utility and information leakage. *CoRR*, abs/1103.5188, 2011.

[4] M.S. Alvim, M.E. Andrés, K. Chatzikokolakis, and C. Palamidessi. Foundations of security analysis and design vi. chapter Quantitative information flow and applications to differential privacy, pages 211–230. Springer-Verlag, 2011.

[5] M.S. Alvim, M.E. Andrés, K. Chatzikokolakis, and C. Palamidessi. On the relation between differential privacy and quantitative information flow. In *Proceedings of the 38th international conference on Automata, languages and programming - Volume Part II*, ICALP'11, pages 60–76. Springer-Verlag, 2011.

[6] M.S. Alvim, M.E. Andrés, and C. Palamidessi. Information flow in interactive systems. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 102–116. Springer-Verlag, 2010.

[7] M.S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith. Measuring information leakage using generalized gain functions. In *Proceedings of*

the 2012 IEEE 25th Computer Security Foundations Symposium, CSF'12, pages 265–279. IEEE Computer Society, 2012.

[8] M.E. Andrés, P. D'Argenio, and P. Rossum. Significant diagnostic counterexamples in probabilistic model checking. In *Proceedings of the 4th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'08, pages 129–148. Springer-Verlag, 2009.

[9] M.E. Andres, C. Palamidessi, P. Rossum, and A. Sokolova. Information hiding in probabilistic concurrent systems. In *Proceedings of the 2010 Seventh International Conference on the Quantitative Evaluation of Systems*, QEST'10, pages 17–26. IEEE Computer Society, 2010.

[10] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS'08, pages 333–348. Springer-Verlag, 2008.

[11] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF'09, pages 43–59. IEEE Computer Society, 2009.

[12] T.H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS'09, pages 113–124. ACM, 2009.

[13] T.H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS'10, pages 3:1–3:12. ACM, 2010.

[14] C. Baier and M. Kwiatkowska. On the verification of qualitative properties of probabilistic processes under fairness constraints. *Information Processing Letters*, 66:71–79, 1998.

[15] G. Barthe, P.R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, CSFW'04, pages 100–114. IEEE Computer Society, 2004.

[16] G. Barthe and L.P. Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE'04, pages 13–22. ACM, 2004.

[17] S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. *International Journal on Software Tools for Technology Transfer*, 7:74–86, 2005.

[18] S. Blom, J. van de Pol, and M. Weber. LTSmin: distributed and symbolic reachability. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 354–359. Springer-Verlag, 2010.

[19] H.-C. Blondeel. Security by logic: characterizing non-interference in temporal logic. Master's thesis, KTH Sweden, 2007.

[20] C. Braun, K. Chatzikokolakis, and C. Palamidessi. Quantitative notions of leakage for one-try attacks. *Electronic Notes in Theoretical Computer Science*, 249:75–91, August 2009.

[21] C. Cachin. *Entropy Measures and Unconditional Security in Cryptography.* PhD thesis, 1997.

[22] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden. Anonymity protocols as noisy channels. In *Proceedings of the 2nd international conference on Trustworthy global computing*, TGC'06, pages 281–300. Springer-Verlag, 2007.

[23] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[24] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *International Journal on Software Tools for Technology Transfer*, 9:429–445, 2007.

[25] H. Chen and P. Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS'07, pages 31–40. ACM, 2007.

[26] H. Chen and P. Malacaria. The optimum leakage principle for analyzing multi-threaded programs. In *Proceedings of the 4th international conference on Information theoretic security*, ICITS'09, pages 177–193. Springer-Verlag, 2010.

[27] L. Christoff and I. Christoff. Efficient algorithms for verification of equivalences for probabilistic processes. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, CAV'91, pages 310–321. Springer-Verlag, 1992.

[28] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation*, 15:181–199, 2005.

[29] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC'95, pages 427–432. ACM, 1995.

[30] M.R. Clarkson, A.C. Myers, and F.B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security 2009.*

[31] M.R. Clarkson, A.C. Myers, and F.B. Schneider. Belief in information flow. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, CSFW'05, pages 31–45. IEEE Computer Society, 2005.

[32] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC'87, pages 1–6. ACM, 1987.

[33] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[34] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, pages 193–209. Springer-Verlag, 2005.

[35] L. Doyen, T.A. Henzinger, and J.F. Raskin. Equivalence of labeled Markov chains. *International Journal of Foundations of Computer Science*, 19(3):549–563, 2008.

[36] J. Engelfriet. Determinacy ? (observation equivalence = trace equivalence). *Theoretical Computer Science*, 36:21 – 25, 1985.

[37] D. Giffhorn and G. Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 2013. Submitted for publication.

[38] J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[39] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, ICALP'90, pages 626–638. Springer-Verlag, 1990.

[40] G. Le Guernic. Precise dynamic verification of confidentiality. In *Proceedings of the 5th International Verification Workshop*, 2008.

[41] A. Gurfinkel and M. Chechik. Why waste a perfectly good abstraction? In *Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 212–226. Springer-Verlag, 2006.

[42] S. Hamadou, V. Sassone, and C. Palamidessi. Reconciling belief and vulnerability in information flow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP'10, pages 79–92. IEEE Computer Society, 2010.

[43] T. Han, J.P. Katoen, and B. Damman. Counterexample generation in probabilistic model checking. *IEEE Transactions on Software Engineering*, 35:241–257, March 2009.

[44] J.E. Hopcroft and J.D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1990.

[45] M. Huisman and H.C. Blondeel. Model-checking secure information flow for multi-threaded programs. In *Proceedings of the 2011 international conference on Theory of Security and Applications*, TOSCA'11, pages 148–165. Springer-Verlag, 2012.

[46] M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. Technical Report TR-CTIT-11-22, Centre for Telematics and Information Technology University of Twente, Enschede, October 2011.

[47] M. Huisman and T.M. Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *Proceedings of the 2011 international conference on Formal Verification of Object-Oriented Software*, FoVeOOS'11, pages 178–195. Springer-Verlag, 2012.

[48] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW'06, pages 3–15. IEEE Computer Society, 2006.

[49] S.H. Hussein. A precise information flow measure from imprecise probabilities. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, SERE'12. IEEE Computer Society, 2012.

[50] S.H. Hussein. Refining a quantitative information flow metric. In *NTMS*, pages 1–7, 2012.

[51] M. Huth and M. Ryan. *Logic in Computer Science: Modeling and Reasoning about the System*. Cambridge University Press, second edition, 2004.

[52] S. Kiefer, A.S. Murawski, J. Ouaknine, B. Wachter, and J. Worrell. Language equivalence for probabilistic automata. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 526–540. Springer-Verlag, 2011.

[53] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4):291–347, 2005.

[54] B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS'07, pages 286–296. ACM, 2007.

[55] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF'09, pages 324–335. IEEE Computer Society, 2009.

[56] D. Kozen. Semantics of probabilistic programs. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, SFCS'79, pages 101–114. IEEE Computer Society, 1979.

[57] S.A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[58] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: verification of probabilistic real-time systems. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 585–591. Springer-Verlag, 2011.

[59] G. Lamperti, M. Zanella, G. Chiodi, and L. Chiodi. Incremental determinization of finite automata in model-based diagnosis of active systems. In *Proceedings of the 12th International Conference on Knowledge-Based Intelligent Information and Engineering Systems, Part I*, KES'08, pages 362–374. Springer-Verlag, 2008.

[60] G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF'07, pages 218–232. IEEE Computer Society, 2007.

[61] P. Malacaria. Risk assessment of security threats for looping constructs. *Journal of Computer Security*, 18:191–228, 2010.

[62] P. Malacaria and H. Chen. Lagrange multipliers and maximum information leakage in different observational models. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 135–146. ACM, 2008.

[63] B. Melichar and J. Skryja. On the size of deterministic finite automata. In *Revised Papers from the 6th International Conference on Implementation and Application of Automata*, CIAA'01, pages 202–213. Springer-Verlag, 2002.

[64] I.S. Moskowitz, R.E. Newman, D.P. Crepeau, and A.R. Miller. Covert channels and anonymizing networks. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, WPES'03, pages 79–88. ACM, 2003.

[65] C. Mu and D. Clark. Quantitative analysis of secure information flow via probabilistic semantics. In *Proceedings of the The Forth International Conference on Availability, Reliability and Security,* ARES'09, pages 49–57. IEEE Computer Society, 2009.

[66] T.M. Ngo and M. Huisman. Quantitative security analysis for multi-threaded programs. In Luca Bortolussi and Herbert Wiklicky, editors, *Proceedings 11th International Workshop on Quantitative Aspects of Programming Languages and Systems,* QAPL'13, volume 117 of *EPTCS*, pages 34–48, 2013.

[67] T.M. Ngo and M. Huisman. Quantitative security analysis for programs with low input and noisy output. In *Proceedings of the 6th international conference on Engineering Secure Software and Systems*, ESSoS'14, pages 77–94. Springer-Verlag, 2014.

[68] T.M. Ngo, M. Stoelinga, and M. Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. In *Proceedings of the 5th international conference on Engineering Secure Software and Systems*, ESSoS'13, pages 107–122. Springer-Verlag, 2013.

[69] T.M. Ngo, M. Stoelinga, and M. Huisman. Effective verification of confidentiality for multi-threaded programs. *Journal of Computer Security (A special issue)*, 22, number 2/2014:269–300, 2014.

[70] T.M. Ngo, M.I.A. Stoelinga, and M. Huisman. Confidentiality for probabilistic multi-threaded programs and its verification. Technical Report TR-CTIT-13-01, Centre for Telematics and Information Technology, University of Twente, Enschede, December 2012.

[71] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[72] A. Parma and R. Segala. Logical characterizations of bisimulations for discrete probabilistic systems. In *Proceedings of the 10th international conference on Foundations of software science and computational structures*, FOSSACS'07. Springer-Verlag, 2007.

[73] D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63:243–246, 1997.

[74] M.O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

[75] A.W. Roscoe. CSP and determinism in security modeling. In *IEEE Symposium on Security and Privacy*, page 114. IEEE Computer Society, 1995.

[76] A. Russo and A. Sabelfeld. Security interaction between threads and the scheduler. In *Computer Security Foundations Symposium*, pages 177–189, 2006.

[77] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF'10, pages 186–199. IEEE Computer Society, 2010.

[78] A. Sabelfeld and A. Myers. Language-based information flow security. In *IEEE Journal on Selected Areas in Communications*, volume 21, pages 5–19, 2003.

[79] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, PSI'09, pages 352–365. Springer-Verlag, 2010.

[80] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE workshop on Computer Security Foundations*, CSFW'00, pages 200–214. IEEE Computer Society, 2000.

[81] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF'07, pages 203–217. IEEE Computer Society, 2007.

[82] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings of the 16th IEEE workshop on Computer Security Foundations*, CSFW'03. IEEE Computer Society, 2000.

[83] G. Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 297–307. Springer-Verlag, 2007.

[84] G. Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'09, pages 288–302. Springer-Verlag, 2009.

[85] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'98, pages 355–364. ACM, 1998.

[86] M.I.A. Stoelinga. *Alea jacta est: verification of probabilistic, real-time and parametric systems.* PhD thesis, University of Nijmegen, the Netherlands, April 2002.

[87] T. Terauchi. A type system for observational determinism. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF'08, pages 287–300. IEEE Computer Society, 2008.

[88] M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata.* PhD thesis, 2013.

[89] W.G. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21:216–227, 1992.

[90] R. van der Meyden and C. Zhang. Algorithmic verification of noninterference properties. *Electronic Notes in Theoretical Computer Science*, 168:61–75, 2007.

[91] P. Černý and R. Alur. Automated analysis of Java methods for confidentiality. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV'09, pages 173–187. Springer-Verlag, 2009.

[92] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187.

[93] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7:231–253, 1999.

[94] H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 357–372. Springer-Verlag, 2010.

[95] S. Yoshihama, T. Yoshizawa, Y. Watanabe, M. Kudoh, and K. Oyanagi. Dynamic information flow control architecture for web applications. In *Proceedings of the 12th European conference on Research in Computer Security*, ESORICS'07, pages 267–282. Springer-Verlag, 2007.

[96] S. Zdancewic and A.C. Myers. Observational determinism for concurrent program security. In *Proceedings of 16th IEEE Computer Security Foundations Workshop*, CSFW'03, pages 29–43. IEEE Computer Society, 2003.

[97] J. Zhu and M. Srivatsa. Quantifying information leakage in finite order deterministic programs. *CoRR*, abs/1009.3951, 2010.

[98] J. Zhu and M. Srivatsa. Poster: on quantitative information flow metrics. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS'11, pages 877–880. ACM, 2011.

[99] Y. Zhu and R. Bettati. Anonymity vs. information leakage in anonymity systems. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS'05, pages 514–524. IEEE Computer Society, 2005.

Samenvatting

In de hedendaagse informatiemaatschappij speelt informatiebeveiliging een belangrijke rol in bijna alle aspecten van het dagelijks leven: communicatie tussen overheden en burgers, militaire zaken, bedrijven, financiële informatiesystemen, webgebaseerde services enz. Tegelijkertijd met de toenemende populariteit van computersystemen met meerdere processoren (CPUs), alsmede processoren met meerdere kernen (*cores*), is ook multithreaded programmeren geaccepteerd als een standaardconcept, waarbij het programma meerdere berekeningen tegelijkertijd kan uitvoeren. Er zijn echter nog veel uitdagingen bij het ontwikkelen van technieken die privé-gegevens in multithreaded programma's kunnen beschermen. Ten eerste, bij het uitvoeren van een multithreaded programma komt het vaak voor dat het gedrag van de data onvoorspelbaar is: daardoor is het lastig te voorspellen welke gegevens een aanvaller kan waarnemen. Ten tweede, met de ontwikkeling van krachtige computertechnieken, groeit ook de kracht van de aanvaller. Hoewel hier al veel onderzoek naar is gedaan, kunnen de huidige technieken niet voldoende bescherming (*confidentiality*) garanderen van gegevens in multithreaded programma's.

Het doel van dit proefschrift is om meer geschikte en efficiënte methoden te ontwikkelen die de informatiestromen in een multithreaded programma kunnen analyseren. We formaliseren twee kwalitatieve eigenschappen van vertrouwelijkheid: (1) voor nondeterministische programma's, waarbij geen rekening wordt gehouden met het probabilistische gedrag van het programma, en (2) voor probabilistische programma's, waarbij we er van uit gaan dat we kennis hebben over de kansverdeling van de schedulingstappen. We ontwikkelen ook een verificatiemethode om te kunnen verifiëren dat er geen informatie wordt gelekt. De voorgestelde technieken kunnen niet alleen de vertrouwelijkheid nauwkeurig en efficiënt verifiëren, maar ze blijken ook relevant te zijn buiten het gebied van informatiebeveiliging. Alle technieken zijn geïmplementeerd in onze tool, waarmee we vervolgens in een aantal case studies de toepasbaarheid van onze technieken laten zien.

Ten tweede kijken we verder naar programma's die privé-gegevens lekken. Voor deze programma's is het van groot belang om de juiste hoeveelheid (*kwantiteit*) van de gelekte informatie te kunnen bepalen. Daarvoor bestuderen we *kwantitatieve* beveiligingseigenschappen. Dit zijn sterkere eigenschappen dan de traditionele *kwantitatieve* eigenschappen, aangezien de hoeveelheid informatie die gelekt wordt, gebruikt kan worden om te beslissen of minder lekkage te tolereren is. We introduceren twee *nieuwe modellen* voor kwantitatieve analyse van informatiestromen: (1) een model voor een programma waarin de aanvaller de initiële waarden van publieke gegevens kan beïnvloeden, en (2) een model voor een programma waarin de aanvaller de uitvoeringsvolgorde van de threads kan bepalen.

Titles in the IPA Dissertation Series since 2008

Aspects. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded*

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

of Embedded Systems. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09

S. van der Burg. *A Reference Architecture for Distributed Software Deployment.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-10

J.J.A. Keiren. *Advanced Reduction Techniques for Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2013-11

D.H.P. Gerrits. *Pushing and Pulling: Computing push plans for disk-shaped robots, and dynamic labelings for moving points.* Faculty of Mathematics and Computer Science, TU/e. 2013-12

M. Timmer. *Efficient Modelling, Generation and Analysis of Markov Automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-13

M.J.M. Roeloffzen. *Kinetic Data Structures in the Black-Box Model.* Faculty of Mathematics and Computer Science, TU/e. 2013-14

L. Lensink. *Applying Formal Methods in Software Development.* Faculty of Science, Mathematics and Computer Science, RU. 2013-15