Thesis for the Degree of Doctor of Philosophy

Semantic Models for the Security of Sequential and Concurrent Programs

Andrei Sabelfeld

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, May 2001

Semantic Models for the Security of
Sequential and Concurrent Programs
Andrei Sabelfeld

ISBN 91-628-4852-6

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Cover: “The Trojan Theme”, by Gerda Enberg.

Printed at Chalmers, Göteborg, Sweden, May 2001

Semantic Models for the Security of Sequential and Concurrent Programs
Andrei Sabelfeld
Department of Computing Science
Chalmers University of Technology and Göteborg University

Abstract

The present thesis investigates *confidentiality*, i.e., the problem of determining whether a given program (possibly acquired from an untrusted source) has *secure information flow*. In a setting of *extensional* security, whether or not a difference in a program's behaviour is detectable by a low-level observer as the sensitive data is varied, determines whether or not the program is insecure. Such a view induces an *equivalence relation* on program behaviours in the low-level observer's (or attacker's) view.

We present a formalisation of the attacker's view: this is done via the mathematical machinery of partial equivalence relations in the case of sequential programs (covering nondeterminism and probabilistic covert channels) and bisimulations in the case of concurrent programs (arguing for scheduler-independent security and covering synchronisation, timing and probabilistic covert channels). In both cases, we arrive at compositional security specifications; this facilitates straightforward soundness proofs for compositional security analyses, such as type-based analyses. By such a formalisation of secure information flow, we contribute to a rigorous understanding of potential threats and a higher confidence in security certification based on formal definitions.

We further concentrate on integrating the formalisation of the security of computation at the level of specific programming languages (sequential and concurrent as considered in this thesis) into the security of complex systems at a more abstract level. With this objective in mind, we propose a sound and complete translation of a timing-sensitive security specification for simple multi-threaded programs into a more general security framework.

Keywords: semantics-based security, information flow control, noninterference, confidentiality, probabilistic covert channels, partial equivalence relations, powerdomains, probabilistic bisimulation, multi-threadedness.

This thesis is a collection of four articles devoted to semantic models for program security. The articles correspond to Chapters 2–5 and are published (in order) in the the following sources:

1. *A Per Model of Secure Information Flow in Sequential Programs*, written with David Sands, and published in *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001. A condensed version was published in the the *Proceedings of European Symposium on Programming*, LNCS 1576, pages 40–58, Amsterdam, March 1999. Springer-Verlag.

2. *Probabilistic Noninterference for Multi-threaded Programs*, written with David Sands, and was published in the *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 3–5, 2000. IEEE Computer Society Press.

3. *The Impact of Synchronisation on Secure Information Flow*, to appear in the *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, July 3–6, 2001, LNCS, Springer-Verlag.

4. *A Generic Approach to the Security of Multi-threaded Programs*, written with Heiko Mantel, to appear in the *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 11–13, 2001. IEEE Computer Society Press.

Contents

Acknowledgements **ix**

1 Introduction **1**

1.1 Computer Security 1

1.2 Program Confidentiality 3

1.3 Background 5

1.4 Central Contribution 6

1.5 Overview 7

2 A Per Model of Secure Information Flow in Sequential Programs **11**

2.1 Introduction 12

2.1.1 Motivation 12

2.1.2 Background 13

2.1.3 Semantic Foundations of Information Flow Analysis . 13

2.1.4 Semantics-based Models of Information Flow 14

2.1.5 Overview 15

2.2 A Per Model of Information Flow 16

2.2.1 Binding-time Analysis as Dependency Analysis 16

2.2.2 From Equivalence Relations to Pers 19

2.2.3 Observations on Strictness and Termination Properties 21

2.2.4 Cohen's Formalisation of Flow Properties 22
2.2.5 Partial Information Flow 24
2.3 Nondeterministic Information Flow 24
2.3.1 Secure Commands in a Deterministic Setting 25
2.3.2 Powerdomain Semantics for Nondeterminism 26
2.3.3 Pers on Powerdomains 28
2.3.4 The Security Condition 29
2.4 Relation to an Equational Characterisation 30
2.4.1 Equational Security and Projection Analysis 31
2.4.2 The Equational Security Condition Is Subsumed by the Per Security Condition 31
2.4.3 Abstract Variables 34
2.5 A Probabilistic Security Condition 35
2.5.1 Probabilistic Powerdomain of Distributions 35
2.5.2 Pers on Probabilistic Powerdomains 37
2.5.3 Probabilistic Security Condition for the Case of Two Variables 38
2.6 Analysis for Security Properties 39
2.6.1 A Simple Sequential Language with Probabilistic Choice 39
2.6.2 Hook-up Properties 40
2.6.3 Type-system-based Analysis 44
2.7 Conclusions 47

3 Probabilistic Noninterference for Multi-threaded Programs 55
3.1 Introduction 56
3.1.1 Motivation 56
3.1.2 Classification of Insecure Information Flows 57
3.1.3 Related Work 57
3.1.4 Overview 59

3.2 A Simple Concurrent Language . . . 59
3.3 Probabilistic Noninterference . . . 61
3.3.1 Background . . . 61
3.3.2 Relational Definitions and Notation . . . 63
3.3.3 Probabilistic Bisimulation . . . 64
3.3.4 Noninterference Based on Partial Probabilistic Bisimulation . . . 66
3.4 Scheduler-independent Noninterference . . . 67
3.4.1 Semantics with Schedulers . . . 67
3.4.2 Probabilistic Noninterference with Schedulers . . . 70
3.4.3 Scheduler-independent Noninterference . . . 71
3.4.4 Relating Low-bisimulations under Compositional Security . . . 73
3.5 Hook-up Properties . . . 75
3.6 Proving Analyses Secure . . . 78
3.6.1 Agat's Approach . . . 79
3.6.2 The Type System . . . 80
3.6.3 An Example Code Transformation . . . 82
3.6.4 Correctness of the Analysis . . . 83
3.6.5 Soundness of the Transformation . . . 84
3.6.6 Elimination of the `protect` . . . 85
3.7 Conclusions . . . 86

4 The Impact of Synchronisation on Secure Information Flow in Concurrent Programs 91
4.1 Introduction . . . 92
4.1.1 Motivation . . . 92
4.1.2 Background . . . 92
4.1.3 Insecure Flows to Eliminate . . . 93
4.1.4 Overview . . . 93

4.2 A Simple Multi-threaded Language with Synchronisation . . 93
4.2.1 Semaphores 94
4.2.2 Semaphores by Busy Waiting 94
4.2.3 Semaphores by Blocked Waiting 94
4.3 Security Specification 97
4.3.1 Noninterference 97
4.4 Type-based Security Analysis 98
4.4.1 The Type System 98
4.4.2 Correctness of the Analysis 101
4.4.3 Soundness of the Transformation 103
4.5 Secure Programming with Synchronisation 104
4.6 Conclusions and Related Work 105

5 A Generic Approach to the Security of Multi-threaded Programs 109

5.1 Introduction 110
5.1.1 Motivation 110
5.1.2 Background 110
5.1.3 Overview 112
5.2 Preliminaries 112
5.2.1 System Specifications 112
5.2.2 Security Properties 114
5.3 Generic Thread Pools 116
5.3.1 Trace-Based Formal Specification 117
5.3.2 Security of Thread Pools 120
5.4 MWL Thread Pools 121
5.4.1 The Multi-Threaded While-Language MWL 122
5.4.2 Definition of Security for MWL 123
5.4.3 Instantiating Generic Thread Pools 126

5.5 Relation between MWL Programs and MWL Thread Pools . 130
5.5.1 Adequateness of MWL Thread Pools 130
5.6 Soundness and Completeness Results 134
5.6.1 Soundness 134
5.6.2 Completeness 138
5.7 Discussion and Future Work 142

Acknowledgements

I am grateful to my supervisor David Sands, who has provided a stimulating research environment where doing research has been tremendous fun, including co-authoring papers. I admire Dave's common-sense vision of keeping Computing Science relevant to practical computational concerns. I have been lucky to learn a lot from Dave in many ways, research-related and otherwise.

Thanks for collaboration to Heiko Mantel, with whom writing a paper has been a pleasantly efficient process. Heiko: good luck with your thesis now!

This thesis work has benefited from the feedback of Peter Dybjer and John Hughes, who served on my PhD committee; together with my examiner Bengt Nordström, they have been a constant source of encouragement.

Discussions with Johan Agat have been rewarding and led to numerous improvements in this thesis. Thanks to Devdatt Dubhashi, David Modjeska and Makoto Takeyama for feedback on individual parts of this thesis.

Further thanks for useful input go to Ilya Beylin, Magnus Carlsson, Koen Claessen, Thierry Coquand, Jörgen Gustavsson, Reiner Hähnle, Patrik Jansson, Josef Sveningsson, Walid Taha and Tuomo Takkula.

Going further afield, I would like to thank Martín Abadi, Gilles Barthe, Mads Dam, Olivier Danvy, Riccardo Focardi, Roberto Gorrieri, Yuri Gurevich, Kohei Honda, Bengt Jonsson, Rustan Leino, Fabio Martinelli, Greg Nelson, Hanne Riis Nielson, Jon Riecke, Peter Ryan, Peter Schmitt, Ralf Steinbrüggen, Paul Syverson and, finally, Andrei Voronkov (who was in fact the one who referred me to Chalmers/GU) for their direct or other influence, including valuable feedback. Special thanks go to Pablo Giambiagi and Andrés Martinelli of KTH/SICS for the many activities we have enjoyed together, with a focus ranging from security to Sangiovese.

Returning to Chalmers/GU, I wish to thank everybody at the Department of Computing Science for a cooperative environment. I have never regretted accepting the PhD position here. My friends in Gothenburg deserve special thanks for the cheerful times we have had. In particular, thanks to Johan Agat and Ana Bove, with whom I had the pleasure of sharing an office.

Lastly and most pleasantly, I would like to express my love to Mum and Dad for their generous support. Спасибо, мама, за твою любовь! Finally, all my gratitude to Master Bo (БГ) for inspiration from outside.

Chapter 1

Introduction

1.1 Computer Security

Computer security is no longer a concern for only meeting military requirements. Rather, any organisation and any individual who uses computers is potentially vulnerable to computer security attacks. Not only has the number of targets for potential attacks been growing increasingly fast, but also the number of sources for such attacks. In order to perform an attack one does not need to have expert knowledge in security because one can simply exploit scripts previously created and available on the Internet.

In computer security literature, attacks are commonly classified into three categories known as the *CIA* of computer security [17, 9]. These categories are (with some examples borrowed form [10]):

Confidentiality The attacker attempts to steal confidential information, such as passwords, medical records, email logs and payroll data. The methods of attack may vary from bribing a security guard to exploiting a security hole in a system or a weakness in a cryptographic algorithm.

Integrity The attacker attempts to illegally alter parts of the system. For example, a bank employee modifies the deposit system to transfer customer money into her own account, thus compromising transaction integrity (*insider attack*). Or, a university student breaks into the university administration system to raise her examination scores, thus compromising data integrity (*outsider attack*). An attacker might also try to erase system logs in order to hide her footprint.

Availability The attacker attempts to disrupt the normal operation of a system. These are also called *denial-of-service attacks.* For example, bombarding a machine with a large number of IP packets can effectively isolate the machine from the rest of the network. A cyber terrorist might attempt to bring down the power grid or cause traffic accidents by compromising the computer-operated control systems.

These three categories of attacks are intricately related, i.e., the techniques and results of attacks in one category can often be used to assist in another. For example, by compromising confidentiality an attacker could obtain passwords and thus compromise integrity by gaining access to and then modifying system resources, which in turn could lead to successful denial-of-service attacks.

One does not have to go far for more examples of security attacks. Such examples break the news with a very high frequency. Another example of an attack that combines the three categories is the "Love Letter" virus that overwhelmed email systems worldwide and caused tens of millions of dollars in damage after it was released in the Philippines in May 2000. It infected some 15 million computers and sent servers crashing around the world. The "Love Letter" virus is an example of a macrovirus, which is distributed via email. Users usually get an email, sometimes from someone they know, asking them to check the attached "Love Letter" (see Figure 1.1). That file is a VisualBasic script, which contains the virus payload. When users of Microsoft's Outlook email software open the attachment, the virus infects their computers and sends itself to every name in the users' address book.

The author of the "Anna Kournikova" virus (a recent variation of the "Love Letter" that masquerades as an image of a sexy Russian tennis star with the same result of sending itself to every name in the users' address book once the users open the attachment, which is disguised as a photo file) claims to have *little or no programming experience*, and that a computer kit called "Visual Basic Worm Generator" by "[K]Alamar" was used to create it [20]. Indeed, one need not be a hacker in order to mount attacks, simply searching the Internet is enough to get hold of malicious software of all flavours.

Security is a complex issue and must be considered from an overall systems point of view. A system is only as secure as the system's weakest point. Although this thesis focuses on a particular problem, *program confidentiality*, one should bear in mind that a satisfactory solution to one facet of security does not guarantee overall system's security without an appropriate treatment of all aspects of security.

Figure 1.1: “Love Letter” virus

Protection mechanisms against the three types of attacks in general are related. The three basic protection categories are *prevention*, *detection* and *recovery*. Note that for confidentiality, the most important technique is prevention. Indeed, a leak of confidential data is very hard to detect, and it may be impossible to prevent the adversary from using leaked data.

1.2 Program Confidentiality

This thesis investigates *confidentiality* for programs, i.e., the problem of determining whether a given program (possibly received from an untrusted source) has *secure information flow*. Throughout the thesis, *security* will mean confidentiality.

Imagine the following possible scenario (schematically depicted in Figure 1.2). You have received a program from an untrusted source (call it company M). M promises to help you to optimise your personal financial investments, information about which you have stored in a database on your home computer. The software is free (for a limited time), under the condition that

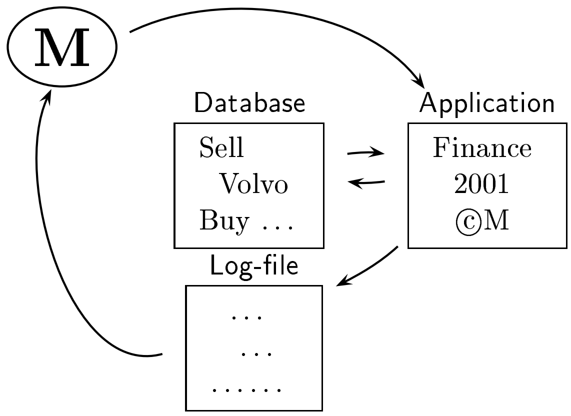


Figure 1.2: Financial adviser application

you permit a log-file containing a summary of your usage of the program to be automatically emailed back to the developers of the program (who claim they wish to determine the most commonly used features of their tool). Is such a program safe to use? The program must be allowed access to your personal investment information, and is allowed to send information, via the log-file, back to M. But how can you be sure that M is not obtaining your sensitive private financial information by cunningly encoding it in the contents of the innocent-looking log-file? This is an example of the problem of determining that the program has secure information flow. Information about your sensitive "high-security" data should not be able to propagate to the "low-security" output (the log-file). Traditional methods of access control are of limited use here since the program has legitimate access to the database. In the standard security terminology, a program with hidden side-effects that are not specified in the program documentation and are not intended by the user executing the program is called a *Trojan horse* [9]. The goal for the attachers is to make the user run the Trojan horse with the user's access rights, e.g., legitimate access to sensitive data.

The confidentiality problem is becoming increasingly important with the growth of use of mobile code. The Internet offers a powerful delivery mechanism for software (and Trojan horses). At the click of a mouse, users can download complete applications and upgrade existing packages. The financial advisor application is an example of mobile code, i.e., a program that is moved (typically over a network) before being executed. Java, ActiveX, JavaScript, VisualBasic and Lisp feature another type of mobile code, namely, a document with executable content. Users viewing a document that contains executable content by, e.g., their web-browsers or word-processors might not be aware that the document contains code that is executed. While offering useful and powerful features, the concept of executable contents introduces new security threats (as exemplified by the “Love Letter” example).

1.3 Background

The problem of secure information flow in the context of systems with multiple levels of confidentiality has been studied much since Denning’s pioneering work [5, 6] in an extension of Bell and LaPadula’s early work [2]. Denning’s solution is to apply a static analysis suitable for inclusion into a compiler. The basic idea is that security levels are represented as a lattice (for example the two point lattice $PublicDomain \leq TopSecret$). The aim of the static analysis is to ensure that information from inputs, variables or processes of a given security level only flows to outputs, variables or processes which have been assigned a higher or equal security level.

In order to verify a program analysis or a specific proof a program’s security one must have a formal specification of what constitutes secure information flow. The value of a semantics-based specification for secure information flow is that it contributes significantly to the reliability of and the confidence in such activities, and can be used in the systematic design of program analyses. Many approaches to Denning-style analyses (including the original articles) contain a fair degree of formalism but arguably are lacking a rigorous soundness proof. Volpano *et al* [21] claim to give the first satisfactory treatment of soundness of Denning’s analysis. Such a claim rests on the dissatisfaction with soundness arguments based on an instrumented operational, e.g., [18] or denotational semantics, e.g., [15], or on “axiomatic” approaches which define security in terms of a program logic [1] without any models to relate the logic to the semantics of the programming language. The problem here is that an “instrumented semantics” or a “security logic” is just a definition, not subject to any further mathematical justification. For a convincing assurance of a program’s security, one therefore needs a specification of security that is defined in terms of the original programming language semantics without any ad hoc “instrumentations”.

This motivates the notion of *noninterference* introduced in [8]. In the setting of information flow properties of programs—rather than other abstract models of computation—the earlier work of Cohen [3, 4] is the true predecessor to noninterference for programs, since it also aims to provide semantics-based characterisations of Denning-style information-flow properties. Intuitively, a program has the noninterference property whenever any behaviour observable by the attacker (e.g., low input, low output, termination, etc.) is independent of the initial high data. The problem of secure information flow, or "noninterference" is now quite mature, and very many specifications exist in the literature—see McLean's tutorial [14] for an overview. Many approaches, including those of Goguen and Meseguer and McLean, have been phrased in terms of abstract trace-based models of computation. These models are typically based on nonstandard notions of automata. One criticism of these abstractions is that they are not always well-motivated from the point of view of the kinds of systems that they are intended to abstract. For example, Goguen and Meseguer's machines described only deterministic systems; McCullough's nondeterministic version focused on defining security via communication [13, 11], but computation is modelled as a globally clocked synchronous system, which in retrospect is not well suited to modelling properties of distributed systems. Even when "standard" models are used, e.g., Moskowitz and Costich's use of classical automata [16], they have still been typically based on traces, and lack information about deadlock and branching behaviours (the way that nondeterminism is resolved).

Only more recently have attempts been made to rephrase and compare various security conditions in terms of well-known semantic models specifically designed for modelling concurrent systems, e.g., the use of labelled transition systems and bisimulation semantics by Focardi and Gorrieri [7]. We adopt the noninterference view of security and present a formalisation of noninterference based on the machinery of programming language semantics for a small sequential and a small concurrent language. We further show how this approach extends to formalise *covert channels*—implicit ways to leak sensitive information through timing, probability distribution of low output and other information the attacker might observe about the program execution. We refer to the actual chapters for further discussion of background and other, less closely related, work.

1.4 Central Contribution

The problem of confidentiality, or determining what constitutes a secure information flow in a system, is a complex one. As observed by Ryan and Schneider [19], "this central concept [confidentiality] in information security is closely related to a central concept in computer science: that of the equivalence of systems". In a setting of program confidentiality, whether or not a difference in a program's behaviour is observable by a low-level observer as the sensitive data is varied, determines whether or not the program is insecure. Indeed, such an observational view induces an equivalence relation on program behaviours in the attacker's view. The central contribution of this thesis is the formalisation of the attacker's view by the mathematical machinery of partial equivalence relations in the case of sequential programs (covering nondeterminism and probabilistic covert channels) and bisimulations in the case of concurrent programs (arguing for scheduler-independent security and covering synchronisation, timing and probabilistic covert channels). In both cases, we arrive at compositional security specifications which facilitates straightforward soundness proofs for compositional security analyses such as type-based analyses.

1.5 Overview

The rest of the thesis is organised as follows.

Chapter 2 This chapter proposes an extensional semantics-based formal specification of secure information flow properties in sequential programs based on representing degrees of security by partial equivalence relations *(pers)*. The approach is inspired by (and in the deterministic case equivalent to) the use of partial equivalence relations in specifying binding-time analysis, and is thus able to specify security properties of higher-order functions and "partially confidential data". The approach is extended to handle nondeterminism by using powerdomain semantics and *probabilistic covert channels* by using probabilistic powerdomain semantics.

Chapter 3 This chapter concerns a probability-sensitive confidentiality specification—a form of *probabilistic noninterference*—for a small multi-threaded programming language with dynamic thread creation. In principle, the denotational semantics machinery should be sufficient to handle concurrent languages, but the mathematical overheads are high. Thus, in Chapter 3 we turn to operationally-based use of pers as probabilistic-bisimulation-like relations. Bisimulation is the key to capturing *timing* channels of information flow. Notice that in a concurrent setting timing information about a computation can be observed internally—by other threads. Thus, a security specification must take into account timing channels. Probabilistic covert channels arise from a scheduler which is probabilistic. Since scheduling policy is typically outside the language specification for multi-threaded languages, we describe how to generalise the security condition in order to define robust security with respect to a wide class of schedulers and program-controlled thread priorities.

Chapter 4 Synchronisation is fundamental to concurrent programs. Chapter 4 investigates the security of information flow in multi-threaded programs in the presence of synchronisation. We give a small-step operational semantics for a simple shared-memory multi-threaded language with synchronisation and present a compositional bisimulation-based confidentiality specification that captures timing leaks. Inspired by [12], we propose a type-based analysis improving on previous approaches to reject insecure programs.

In both denotational and operational approaches, the security conditions satisfy compositionality properties, which facilitates straightforward proofs of correctness, e.g., for security type systems. The usefulness of the compositional nature of the security specifications is illustrated in Chapters 2–4 by straightforward correctness proofs for simple type-based security analyses.

Chapter 5 The security of computation at the level of specific programming languages (sequential and concurrent as considered in this thesis) and the security of complex systems at a more abstract level are two major areas of current security research. With the objective to integrate the two, Chapter 5 proposes a translation of a timing-sensitive security specification for simple multi-threaded programs into a more general security framework. Interestingly, our notion of security for programs is bisimulation-based while the security framework is trace-based. Nevertheless, the translation is *sound* and *complete* in the sense that the trace-based specification which results from the translation of a multi-threaded program is secure if and only if the original program is secure. The translation is presented as a two-step process where the first step is independent of the concrete programming language.

References

[1] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[2] D.E. Bell and L.J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. MTR-2997, Rev. 1, The MITRE Corporation, Bedford, Mass., 1976.

[3] Ellis S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[4] Ellis S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[5] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[6] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[7] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994/1995.

[8] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, April 1982.

[9] D. Gollmann. *Computer Security.* Wiley, 1999.

[10] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, 1999.

[11] J.W. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990. IEEE Computer Society Press.

[12] J. Hughes. Sampling and proof: A half-case study. In G. Jones and M. Sheeran, editors, *Proceedings of the Workshop on Designing Correct Circuits*, Workshops in Computing, pages 92–98, Oxford, England, September 1990. Springer-Verlag.

[13] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[14] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[15] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[16] I. S. Moskowitz and O. L. Costich. A classical automata approach to noninterference type problems. In *The Computer Security Foundations Workshop V proceedings: June 16–18, 1992, the Franconia Inn, Franconia, New Hampshire*, pages 2–8. IEEE Computer Society Press, 1992.

[17] Commission of the European Communities. *Information Technology Security Evaluation Criteria*. June 1991. Version 1.2.

[18] P. Ørbæk. Can you Trust your Data? In M. I. Schwartzbach P. D. Mosses and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag.

[19] P. Y. A. Ryan and S. A. Schneider. Process algebra and noninterference. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.

[20] Daniel Sieber. Kournikova virus slams U.S., Europe, misses Asia, *CNN.com*, February 13, 2001.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Chapter 2

A Per Model of Secure Information Flow in Sequential Programs

This chapter proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations (pers). The specification clarifies and unifies a number of specific correctness arguments in the literature and connections to other forms of program analysis. The approach is inspired by (and in the deterministic case equivalent to) the use of partial equivalence relations in specifying binding-time analysis, and is thus able to specify security properties of higher-order functions and "partially confidential data". We also show how the per approach can handle nondeterminism for a first-order language, by using powerdomain semantics and show how probabilistic security properties *can be formalised by using probabilistic powerdomain semantics. We illustrate the usefulness of the compositional nature of the security specifications by presenting a straightforward correctness proof for a simple type-based security analysis.*

This chapter was written with David Sands, and published in *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001. An earlier version of the chapter was published in the the *Proceedings of European Symposium on Programming*, LNCS 1576, pages 40–58, Amsterdam, March 1999. Springer-Verlag.

2.1 Introduction

2.1.1 Motivation

You have received a program from an untrusted source. Let us call it company M. M promises to help you to optimise your personal financial investments, information about which you have stored in a database on your home computer. The software is free (for a limited time), under the condition that you permit a log-file containing a summary of your usage of the program to be automatically e-mailed back to the developers of the program (who claim they wish to determine the most commonly used features of their tool). Is such a program safe to use? The program must be allowed access to your personal investment information, and is allowed to send information, via the log-file, back to M. But how can you be sure that M is not obtaining your sensitive private financial information by cunningly encoding it in the contents of the innocent-looking log-file? This is an example of the problem of determining that the program has *secure information flow*. Information about your sensitive "high-security" data should not be able to propagate to the "low-security" output (the log-file). Traditional methods of access control are of limited use here since the program has legitimate access to the database.

This paper proposes an extensional semantics-based formal specification of secure information-flow properties in sequential programs based on representing degrees of security by partial equivalence relations (pers[1]). The specification clarifies and unifies a number of specific correctness arguments in the literature, and connections to other forms of program analysis. The approach is inspired by and, in the deterministic case, equivalent to the use of partial equivalence relations in specifying binding-time analysis [19], and is thus able to specify security properties of higher-order functions and "partially confidential data" (e.g. one's financial database could be deemed to be partially confidential if the number of entries is not deemed to be confidential even though the entries themselves are). We show how the approach can also be adapted to handle nondeterminism in a first-order language, and illustrate how the various choices of powerdomain semantics affect the kinds of security properties that can be expressed, ranging from termination-insensitive properties (corresponding to the use of the Hoare (partial correctness) powerdomain) to *probabilistic security properties*, obtained when one uses a probabilistic powerdomain.

[1] A partial equivalence relation is symmetric and transitive but not necessarily reflexive.

2.1.2 Background

The study of information flow in the context of systems with multiple levels of confidentiality was pioneered by Denning [9, 10] in an extension of Bell and LaPadula's early work [5]. Denning's approach is to apply a static analysis suitable for inclusion into a compiler. The basic idea is that security levels are represented as a lattice (for example the two point lattice $\mathit{PublicDomain} \leq \mathit{TopSecret}$). The aim of the static analysis is to ensure that information from inputs, variables or processes of a given security level only flows to outputs, variables or processes which have been assigned a higher or equal security level.

2.1.3 Semantic Foundations of Information Flow Analysis

In order to verify a program analysis or a specific proof of a program's security one must have a formal specification of what constitutes secure information flow. The value of a semantics-based specification for secure information flow is that it contributes significantly to the reliability of and the confidence in such activities, and can be used in the systematic design of program analyses. Many approaches to Denning-style analyses (including the original articles) contain a fair degree of formalism but arguably are lacking a rigorous soundness proof. Volpano et al. [47] claim to give the first satisfactory treatment of soundness of Denning's analysis. Such a claim rests on the dissatisfaction with soundness arguments based on an instrumented operational semantics [35] or denotational semantics [32], or on "axiomatic" approaches which define security in terms of a program logic [3] without any models to relate the logic to the semantics of the programming language. The problem here is that an "instrumented semantics" or a "security logic" is just a definition, not subject to any further mathematical justification. McLean points out [31] in a related discussion about the (non language-specific) Bell and LaPadula model:

> One problem is that ... they *[the Bell LaPadula security properties]* constitute a possible implementation of security, ... , rather than an abstract specification of what all secure systems must satisfy. By concerning themselves with particular controls over files inside the computer, rather than limiting themselves to the relation between input and output, they make it harder to reason about the requirements, ...

This criticism points to more abstract, *extensional* notions of soundness, based on, for example, the idea of *noninterference* introduced by Goguen and Meseguer [12]. In the setting of information flow properties of programs—rather than other abstract models of computation—the earlier work of Cohen [6, 7] is the true predecessor to the present article, since it also aims to provide semantics-based characterisations of Denning-style information-flow properties. Cohen's formalisation of information flow properties of sequential deterministic programs is discussed in Section 2.2.4.

2.1.4 Semantics-based Models of Information Flow

The problem of secure information flow, or "noninterference" is now quite mature, and very many specifications exist in the literature—see McLean's tutorial [29] for an overview. Many approaches – including those of Goguen and Meseguer and McLean have been phrased in terms of abstract trace-based models of computation. These models are typically based on nonstandard notions of automata. One criticism of these abstractions is that they are not always well-motivated from the point of view of the kinds of systems that they are intended to abstract. For example, Goguen and Meseguer's machines described only deterministic systems; McCullough's nondeterministic version focused on defining security via communication [28, 13], but computation is modelled as a globally clocked synchronous system, which in retrospect is not well suited to modelling properties of distributed systems. Even when "standard" models are used e.g., Moskowitz and Costich's use of classical automata [33], they have still been typically based on traces, and lack information about deadlock and branching behaviours (the way that nondeterminism is resolved). Only more recently have attempts been made to rephrase and compare various security conditions in terms of well-known semantic models specifically designed for modelling concurrent systems e.g. the use of labelled transition systems and bisimulation semantics by Focardi and Gorrieri [11].

In this paper we consider the problem of information-flow properties of sequential systems. The systems are those concretely realised by programs, and we use the framework of *denotational semantics* as our formal model of computation. Our approach is based on standard semantic models rather than ad hoc nonstandard extensions specific to the problem of information flow – c.f., the addition of special "security variables" as found in the work of Banâtre et al. [4], or the "trust" tags in the nonstandard dependence models used by Ørbæk [36].

Along the way we consider some relations to specific static analyses, such as the ***Secure Lambda** Calculus* [15] and an alternative semantic condition for secure information flow proposed by Leino and Joshi [25].

2.1.5 Overview

The rest of the paper is organised as follows.

Section 2.2 shows how the per-based condition for soundness of binding-time analysis is also a model of secure information flow. We show how this provides insight into the treatment of higher-order functions and structured data.

Section 2.3 shows how the approach can be adapted to the setting of a nondeterministic imperative language by appropriate use of a powerdomain-based semantics. We show how the choice of powerdomain (upper, lower or convex) affects the nature of the security condition.

Section 2.4 focuses on an alternative semantic specification due to Leino and Joshi. Modulo some technicalities we show that Leino's condition—and a family of similar conditions—are in agreement with, and can be represented using our form of specification.

Section 2.5 considers the problem of preventing unwanted *probabilistic* information flows in programs. We show how this can be solved in the same framework by utilising a probabilistic semantics based on the probabilistic powerdomain [21].

Section 2.6 shows how the probabilistic security specification satisfies compositionality properties which facilitates straightforward proofs of correctness for compositional analyses. We illustrate the usefulness of the compositional nature of the security specifications by presenting a straightforward correctness proof for a type-based security analysis.

Section 2.7 concludes.

2.2 A Per Model of Information Flow

In this section we introduce the way that *partial equivalence relations* (pers) can be used to model dependencies in programs. The basic idea comes from Hunt's use of pers to model and construct abstract interpretations for strictness properties in higher-order functional programs [18, 17], and in particular its use to model dependencies in *binding-time* analysis [19]. Related ideas already occur in the denotational formulation of live-variable analysis [34].

2.2.1 Binding-time Analysis as Dependency Analysis

Given a description of the parameters in a program that will be known at partial evaluation time (called the *static* arguments), a binding-time analysis (BTA) must determine which parts of the program are dependent solely on these known parts (and therefore also known at partial evaluation time). The safety condition for binding-time analysis must ensure that there is no dependency between the *dynamic* (i.e., non-static) arguments and the parts of the program that are deemed to be static. Viewed in this way, binding-time analysis is purely an analysis of dependencies.[2]

Dependencies in Security In the security field, the property of absence of unwanted dependencies is often called *noninterference*, after Goguen and Meseguer [12]. Many problems in security come down to forms of dependency analysis. For example, in the case of *confidentiality*, the aim is to show that the outputs of a program which are deemed to be of low confidentiality do not have any dependence on inputs of a higher degree of confidentiality. In the case of *integrity* (*trust*), one must ensure that the value of some trusted data does not depend on some untrusted source.

But what is the precise meaning of the phrase "does not depend"? This is formalised by asking whether *variation* in high input can ever cause variation in the low output [7]. If no variation can be conveyed, then there is no information flow from high inputs to low outputs. Put more simply, if we change high security inputs, the low security output is not affected. In the

[2]Unfortunately, from the perspective of a partial evaluator, BTA is not *purely* a matter of dependencies; it was shown [16] that the pure dependency models [24, 19] are not quite adequate to ensure the safety of partial evaluation.

remainder of this section we show how these ideas can be conveniently expressed using *partial equivalence relations* to model the concept of variation, and we compare with Cohen's early formalisation of the problem [7].

Some intuitions about information flow Let us consider a program modelled as a function from some input domain to an output domain. Now consider the following simple functions mapping inputs to outputs: $\mathsf{snd} : D \times E \to E$ for some sets (or domains) D and E, and shift and test, functions in $\mathbf{N} \times \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ and $\mathbf{N} \times \mathbf{N} \to \mathbf{N}$, defined by:

$$\begin{array}{rcl} \mathsf{snd}(x, y) & = & y \\ \mathsf{shift}(x, y) & = & (x + y, y) \\ \mathsf{test}(x, y) & = & \mathsf{if}\ x > 0\ \mathsf{then}\ y\ \mathsf{else}\ y + 1 \end{array}$$

Now suppose that (h, l) is a pair where h is some high security information, and l is low, "public domain", information. Without knowing about what the actual values h and l might be, we know about the result of applying function snd to (h, l) will be a low value, and, in the case that we have a pair of numbers, the result of applying shift will be a pair with a high first component and a low second component.

Note that the function test does not enjoy the same security property that snd does, since although it produces a value which is constructed from purely low-security components, the actual value is dependent on the first component of the input. This is what is known as an indirect information flow [9].

It is rather natural to think of these properties as "security types":

$$\begin{array}{c} \mathsf{snd} : \mathit{high} \times \mathit{low} \to \mathit{low} \\ \mathsf{shift} : \mathit{high} \times \mathit{low} \to \mathit{high} \times \mathit{low} \\ \mathsf{test} : \mathit{high} \times \mathit{low} \to \mathit{high} \end{array}$$

But what notion of "type", and what interpretation of "high" and "low" can formalise these more intuitive type statements? Interpreting types as sets of values is not adequate to model "high" and "low". To track degrees of dependence between inputs and outputs we need a more dynamic view of a type as a degree of variation. We must vary (parts of) the input and observe which (parts of) the output vary. For the application to confidentiality we want to determine if there is possible information leakage from a high level input to the parts of an output which are intended to be visible to a low security observer. We can detect this by observing whether the "low" parts of the output vary in any way as we vary the high input.

The simple properties of the functions snd and shift described above can be captured formally by the following formulae:

$$\forall x, x', y.\, \mathsf{snd}(x, y) = \mathsf{snd}(x', y) \tag{2.1}$$

$$\forall x, x', y.\, \mathsf{snd}(\mathsf{shift}(x, y)) = \mathsf{snd}(\mathsf{shift}(x', y)) \tag{2.2}$$

Indeed, this kind of formula forms the core of the correctness arguments for the security analyses proposed by e.g., Volpano and Smith et al. [47, 42], and also for the extensional correctness proofs in the core of the *SLam calculus* [15, 1].

High and Low as Equivalence Relations We show how we can interpret "security types" in general as partial equivalence relations. We will interpret *high* (for values in D) as the equivalence relation All_D, and *low* as the relation Id_D where for all $x, x' \in D$:

$$\begin{gathered} x\ All_D\ x' \\ x\ Id_D\ x' \iff x = x' \end{gathered}$$

For a function $f : D \to E$ and binary relations $P \in Rel(D)$ and $Q \in Rel(E)$, we write $f : P \twoheadrightarrow Q$ iff

$$\forall x, x' \in D.\, x\ P\ x' \implies (f\ x)\ Q\ (f\ x')$$

For binary relations P, Q we define the relation $P \times Q$ by:

$$(x, y)\ P \times Q\ (x', y') \iff x\ P\ x'\ \ \&\ \ y\ Q\ y'$$

Now the security property of snd described by (2.1) can be captured by:

$$\mathsf{snd} : All_D \times Id_E \twoheadrightarrow Id_E$$

and (2.2) is given by:

$$\mathsf{shift} : All_{\mathbf{N}} \times Id_{\mathbf{N}} \twoheadrightarrow All_{\mathbf{N}} \times Id_{\mathbf{N}}$$

An intuition for the relations *All* and *Id* is that they represent the perspective of the "low" user (the user who does not have access to the high information). The low user can see the difference between distinct low data elements, but one high datum is (or rather *should be*) indistinguishable from any other. This perspective is important when considering more complex lattices of security properties than just the simple two point lattice of low and high security levels. In the multi-level case one would model the perspective of a user at some security level k by using the relation Id for all data of security class less than or equal to k, and by All for all other classes—representing the indistiguishability of data items which are not classified as being visible at level k.

2.2.2 From Equivalence Relations to Pers

We have seen how the equivalence relations All and Id may be used to describe security "properties" *high* and *low*. It turns out that these are exactly the same as the interpretations given to the notions "dynamic" and "static" by the per-based model of BTA [19]. This means that the binding-time analysis for a higher-order functional language can also be read as a security information-flow analysis. This connection between security and binding-time analysis is already folklore (see e.g. a comparison [44] of a particular security type system and a particular binding-time analysis, and an investigation [8] which shows how the incorporation of indirect information flows from Denning's security analysis can improve binding-time analyses).

It is worth highlighting a few of the pertinent ideas from the per-based treatment of BTA [19]. Beginning with the equivalence relations All and Id to describe *high* and *low* respectively, there are two important extensions to the basic idea in order to handle *structured data types* and *higher-order functions*. Both of these ideas are handled by the analysis [19] which rather straightforwardly extends Launchbury's projection-based binding-time analysis [24] to higher types. To some extent the per model [19] anticipates the treatment of partially-secure data types in the SLam calculus [15], and the use of logical relations in their proof of noninterference.

For structured data it is useful to have more refined notions of security than just *high* and *low*; we would like to be able to model various *degrees* of security. For example, we may have a list of records containing name-password pairs. Assuming passwords are considered *high*, we might like to express the fact that although the whole list cannot be considered *low*, it can be considered as a $(low \times high)list$. Constructing equivalence relations which represent such properties is straightforward—see reference [19] for examples (which are adapted directly from Launchbury's work), and reference [17] for a more general treatment of finite lattices of "binding times" for recursive types.

To represent security properties of higher-order functions we use a less restricted class of relations than the equivalence relations. A *partial equivalence relation* (per) on a set D is a binary relation on D which is *symmetric* and *transitive*. If P is such a per let $|P|$ denote the domain of P, given by:

$$|P| = \{x \in D \mid x \; P \; x\}$$

Note that the domain and range of a per P are both equal to $|P|$ (so for any $x, y \in D$, if $x \; P \; y$ then $x \; P \; x$ and $y \; P \; y$), and that the restriction of P to $|P|$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $|P| = D$). Partial equivalence relations over various applicative structures have been used to construct models of the polymorphic lambda calculus (see, for example, Abadi and Plotkin's paper [2]). As far as we are aware, the first use of pers in static program analysis is that presented in [18].

For a given set D let $Per(D)$ denote the partial equivalence relations over D. $Per(D)$ is a meet semi-lattice, with meets given by set-intersection, and top element All.

Given pers $P \in Per(D)$ and $Q \in Per(E)$, we may construct a new per $(P \twoheadrightarrow Q) \in Per(D \to E)$ defined by:

$$\begin{gathered} f \; (P \twoheadrightarrow Q) \; g \\ \Longleftrightarrow \\ \forall x, x' \in D. \, x \; P \; x' \Longrightarrow (f \; x) \; Q \; (g \; x') \end{gathered}$$

If P is a per, we will write $x : P$ to mean $x \in |P|$. This notation and the above definition of $P \twoheadrightarrow Q$ are consistent with the notation used previously, since now

$$\begin{aligned} & f : P \twoheadrightarrow Q \\ \Longleftrightarrow \quad & f \; (P \twoheadrightarrow Q) \; f \\ \Longleftrightarrow \quad & \forall x, x' \in D. \, x \; P \; x' \Longrightarrow (f \; x) \; Q \; (f \; x') \end{aligned}$$

Note that even if P and Q are both total (i.e., equivalence relations), $P \twoheadrightarrow Q$ may be partial. A simple example is $All \twoheadrightarrow Id$. If $f : All \twoheadrightarrow Id$ then we know that given a *high* input, f returns a *low* output. A constant function $\lambda x.42$ has this property, but clearly not all functions satisfy this.

2.2.3 Observations on Strictness and Termination Properties

We are interested in the security properties of functions which are the denotations of programs (in a Scott-style denotational semantics), and so there are some termination issues which should be addressed. The formulation of security properties given above is sensitive to termination. Consider, for example, the following function $f : \mathbf{N}_\perp \to \mathbf{N}_\perp$

$$f = \lambda x.\mathsf{if}\ x > 0\ \mathsf{then}\ x\ \mathsf{else}\ f x$$

Clearly, if the argument is high then the result must be high. Now consider the security properties of the function $g \circ f$ where g is the constant function $g = \lambda x.2$. We might like to consider that g has type $high \to low$. However, if function application is considered to be strict (as in ML) then g is not in $|All_{\mathbf{N}_\perp} \twoheadrightarrow Id_{\mathbf{N}_\perp}|$ since $\perp\ All_{\mathbf{N}_\perp}\ 1$ but $g(\perp) \neq g(1)$. Hence the function $g \circ f$ does *not* have security type $high \to low$ (in our semantic interpretation). This is correct, since on termination of an application of this function, the low observer will have learned that the value of the high argument was positive.

The specific security analysis of e.g. the first calculus of Smith and Volpano [42] is termination sensitive—and this is enforced by a rather sweeping measure: all "while"-loop conditions must be low and all "while"-loop bodies must be low commands.

On the other hand, the type system of the SLam calculus [15] is not termination sensitive in general. This is due to the fact that it is based on a call-by-value semantics, and indeed the composition $g \circ f$ could be considered to have a security type corresponding to "$high \to low$". The correctness proof for noninterference carefully avoids saying anything about nonterminating executions. What is perhaps worth noting here is that had they chosen a non-strict semantics for application then the *same* type-system would yield termination sensitive security properties! So we might say that lazy programs are intrinsically more secure than strict ones. This phenomenon is closely related to properties of parametrically polymorphic functions [40][3]. From the type of a polymorphic function one can predict certain properties about its behaviour—the so-called "free theorems" of the type [49]. However, in a strict language one must add an additional condition in order that the theorems hold: the functions must be *bottom-reflecting* ($f(a) = \perp \implies a = \perp$).

[3]Not forgetting that the use of pers in static analysis was inspired, in part, by Abadi and Plotkin's per model of polymorphic types [2].

The same side condition can be added to make e.g. the type system of the SLam calculus termination-sensitive.

To make this observation precise we introduce one further constructor for pers. If $R \in Per(D)$ then we will also let R denote the corresponding per on $D_\perp$ without explicit injection of elements from D into elements in $D_\perp$. We will write $R_\perp$ to denote the relation in $Per(D_\perp)$ which naturally extends R by $\perp R \perp$.

Now we can be more precise about the properties of g under a strict (call-by-value) interpretation: $g : (All_{\mathbf{N}})_\perp \twoheadrightarrow Id_{\mathbf{N}_\perp}$, which expresses that g is a constant function, modulo strictness. More informatively we can say that $g : All_{\mathbf{N}} \twoheadrightarrow Id_{\mathbf{N}}$ which expresses that g is a non-bottom constant function.

It is straightforward to express per properties in a subtype system of compositional rules (although we don't claim that such a system would be in any sense complete). Pleasantly, all the expected subtyping rules are sound when types are interpreted as pers and the subtyping relation is interpreted as subset inclusion of relations. For the abstract interpretation presented in [19] this has already been undertaken by e.g. Jensen [20] and Hankin and Le Métayer [14]. This natural subtyping relation also suggests a straightforward generalisation to arbitrary lattices of security levels.

2.2.4 Cohen's Formalisation of Flow Properties

In the case of information flow properties, the first programming-language-based formalisation of (in)security was due to Ellis Cohen [6, 7]. This was based on the view of programs as state transformers, as given by a denotational semantics (although there is no explicit discussion of nontermination issues). In [7], Cohen also considers "proof rules" for reasoning about flow properties of programs, and these are claimed to correspond to Denning's informal flow analysis conditions [10]. Here we recount the key definition of *strong dependency* from Cohen, and the related notion of *selective dependency*, and show how these notions can be expressed easily in the per setting.

For the sake of simplicity, we adapt Cohen's definitions slightly so as only to consider information flow between the initial value of a variable h, and the final value of some variable l. Assume that the computation state s is simply a record containing values $s.h$ and $s.l$ for the variables h and l respectively.

The notion of *strong dependency* between h and l, for a program C, written

$h \rhd^C l$, is defined by:

$$h \rhd^C l \iff \exists s_1, s_2\, (s_1.l = s_2.l \;\;\&\;\; ([\![C]\!]s_1).l \neq ([\![C]\!]s_2).l)$$

This states that C conveys definite information flow from the initial value of h to the final value of l. Negating this condition we see that

$$\begin{aligned}
\neg(h \rhd^C l) &\iff \neg\exists s_1, s_2\, (s_1.l = s_2.l \;\;\&\;\; ([\![C]\!]s_1).l \neq ([\![C]\!]s_2).l)\\
&\iff \forall s_1, s_2\, \neg(s_1.l = s_2.l \;\;\&\;\; ([\![C]\!]s_1).l \neq ([\![C]\!]s_2).l)\\
&\iff \forall s_1, s_2\, (s_1.l = s_2.l \Longrightarrow ([\![C]\!]s_1).l = ([\![C]\!]s_2).l)
\end{aligned}$$

This corresponds to the expected noninterference condition. We have argued that this condition can be expressed using pers. But what is more interesting is Cohen's notion of *selective dependency*—since this can be used to model partial information flow.

Let us define the following strengthened form of non-dependency (corresponding to the negation of the predicate defined in [7], Definition 3-4). Let ϕ be a predicate on the value of h. The *selective independence* property, written $h \not\rhd^C_\phi l$, is defined by: $h \not\rhd^C_\phi l$ iff

$$\forall s_1, s_2\, (\phi(s_1.h) \;\;\&\;\; \phi(s_2.h) \;\;\&\;\; s_1.l = s_2.l \Longrightarrow ([\![C]\!]s_1).l = ([\![C]\!]s_2).l)$$

The role of ϕ is to place a restriction on the initial value of h, so as to ask whether, despite this constraint, there is information flow to l. Consider the program $C = \textsf{if } h > 0 \textsf{ then } l := h$. If we know that, for example, the value of h is negative, then we know that C cannot leak information, since the assignment $l := h$ can never be reached. This fact corresponds to the statement

$$h \not\rhd^C_{h<0} l$$

It is straightforward to express properties of this form using pers. The basic idea is to define the relation All^ϕ to be the relation All restricted to values satisfying ϕ. Define $u\; All^\phi\; v \iff \phi(u) \;\;\&\;\; \phi(v)$.

If we assume that a state s is simply a pair of the values of h and l respectively, then the information flow property $h \not\rhd^C_\phi l$ corresponds to

$$[\![C]\!] : (All^\phi \times Id) \twoheadrightarrow (All \times Id)_\perp$$

2.2.5 Partial Information Flow

The selective independence notation is used to make statements about partial information flow. We conclude with such an example (from [7]), and show how these statements can also be expressed (more) concisely with pers.

Let D be the program $l := h \;\mathsf{mod}\; 4$. Clearly there is information leakage, but the information leaked is only the two least significant bits of h. To express this using selective independence, Cohen suggests the following. For each $i \in \{0,1,2,3\}$, define the predicate

$$\phi_i(v_h) \iff i = v_h \pmod 4$$

Now the collection of predicates $\{\phi_i\}_{i=0,1,2,3}$ can be used to define the flow property of the program:

$$\forall i \in \{0,1,2,3\} \,.\, h \not\triangleright^{D}_{\phi_i} l$$

Clearly we can make a similar "for all" statement using pers. In the per setting this can be expressed as a single statement. Let $\hat{\phi}$ denote the equivalence relation formed by taking the set of states satisfying each ϕ_i to be the equivalence classes of the relation. This corresponds to the relation given by $\cup_i All^{\phi_i}$. Then we can express the selective independence property as:

$$[\![D]\!] : (\hat{\phi} \times Id) \twoheadrightarrow (All \times Id)_{\perp}$$

The equivalence of this statement to Cohen's formulation follows from the general fact that for any disjoint pers P and Q we have that $P \cup Q$ is a per, and that

$$f : P \twoheadrightarrow S \;\;\&\;\; f : Q \twoheadrightarrow S \iff f : (P \cup Q) \twoheadrightarrow S$$

2.3 Nondeterministic Information Flow

In this section we show how the per model of security can be extended to describe nondeterministic computations. We see nondeterminism as an important feature as it arises naturally when considering the semantics of a concurrent language (although the treatment of a concurrent language remains outside the scope of the present paper.)

In order to focus on the essence of the problem we consider a very simplified setting—the analysis of commands in some simple imperative language containing a nondeterministic choice operator. We assume that there is some discrete (i.e., unordered) domain $\mathbf{St}$ of states (which might be viewed as finite maps from variables to discrete values, or simply just a tuple of values).

2.3.1 Secure Commands in a Deterministic Setting

In the deterministic setting we can take the denotation of a command C, written $[\![C]\!]$, to be a function in $[\mathbf{St}_\perp \to \mathbf{St}_\perp]$, where by $[D_\perp \to E_\perp]$ we mean the set of strict and continuous maps between domains $D_\perp$ and $E_\perp$. Note that we could equally well take the set of all (trivially continuous) functions in $\mathbf{St} \to \mathbf{St}_\perp$, which is isomorphic. The fact that $\mathbf{St}_\perp$ is no more than a flat domain (i.e., a domain of height two) will be important in the technical development.

Now suppose that the state is just a simple partition into a high-security half and a low-security half, so the set of states is the product $\mathbf{St}_{high} \times \mathbf{St}_{low}$. Then we might define a command C to be secure if no information from the high part of the state can leak into the low part:

$$\begin{gathered} C \text{ is secure } \iff \\ [\![C]\!] : (All \times Id)_\perp \twoheadrightarrow (All \times Id)_\perp \end{gathered}$$

This is equivalent to saying that $[\![C]\!] : (All \times Id) \twoheadrightarrow (All \times Id)_\perp$ since we only consider strict functions. Note that this does not imply that $[\![C]\!]$ terminates, but that the termination behaviour is not influenced by the values of the high part of the state. It is easy to see that the sequential composition of secure commands is a secure command, since firstly, the denotation of the sequential composition of commands is just the function-composition of denotations, and secondly, in general for functions $g : D \to E$ and $f : E \to F$, and pers $P \in Per(D)$, $Q \in Per(E)$ and $R \in Per(F)$ it is easy to verify the soundness of the inference rule:

$$\frac{g : P \twoheadrightarrow Q \qquad f : Q \twoheadrightarrow R}{f \circ g : P \twoheadrightarrow R}$$

2.3.2 Powerdomain Semantics for Nondeterminism

A standard approach to giving meaning to a nondeterministic language—for example Dijkstra's guarded command language—is to interpret a command as a mapping which yields a *set* of results. However, when defining an ordering on the results in order to obtain a domain, there is a tension between the internal order of $\mathbf{St}_\perp$ and the subset order of the powerset. This is resolved by considering a suitable *powerdomain* structure [39, 43]. The idea is to define a preorder on the *finitely generated* (f.g.) subsets (those non-empty subsets which are either finite, or contain $\perp$) of $\mathbf{St}_\perp$ in terms of the order on their elements. By quotienting "equivalent sets" one obtains a partial ordering, each depending on a different view of what sets of values should be considered equivalent. Consider the following three programs (an example from Plotkin's notes on Domain Theory [38]):

$$(1)\ x := 1 \qquad\qquad (2)\ x := 1 \mathbin{[]} \mathsf{loop} \qquad\qquad (3)\ \mathsf{loop}$$

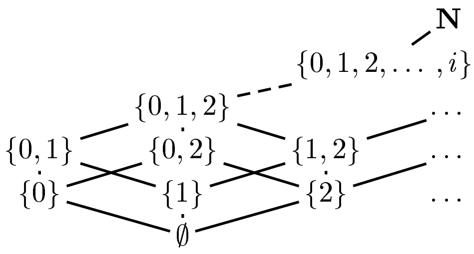
In the "Hoare" or *partial correctness* interpretation the first two programs are considered to be equal since, ignoring nontermination, they yield the same sets of outcomes. This view motivates the definition of the *Hoare* or *lower* powerdomain, $\mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]$.

In the "Smyth" or *total correctness* interpretation, programs (2) and (3) are considered equal (equally bad!) because neither of them can guarantee an outcome. In the general case this view motivates the *Smyth* or *upper* powerdomain, $\mathcal{P}_{\mathsf{U}}[\mathbf{St}_\perp]$ [43].

In the "Egli-Milner" interpretation (leading to the *convex* or *Plotkin* powerdomain in the general case) all three programs are considered to have distinct denotations.

A domain D is *flat* whenever it has a trivial partial ordering ($\forall x, y \in D.\, x \sqsubseteq y \iff x = y$ or $x = \perp$). Examples of flat domains are $\mathbf{N}_\perp$ and $\mathbf{St}_\perp$. A *discrete powerdomain* is the powerdomain of a flat domain. We give a formal definition of each powerdomain construction in turn, and give an idea about the corresponding discrete powerdomain $\mathcal{P}[\mathbf{St}_\perp]$.

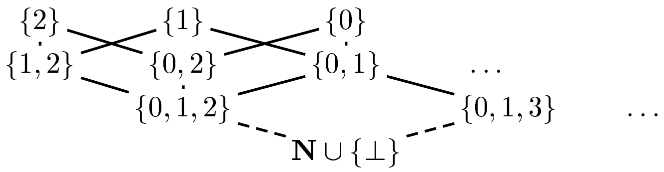
Lower powerdomain Let $A \preceq_{\mathsf{L}} B$ iff $\forall x \in A.\, \exists y \in B.\, x \sqsubseteq y$. In this case the induced discrete powerdomain $\mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]$ is isomorphic to the powerset of $\mathbf{St}$ ordered by subset inclusion. This means that the domain $[\mathbf{St}_\perp \to \mathcal{P}_{\mathsf{L}}[\mathbf{St}_\perp]]$ is isomorphic to all subsets of $\mathbf{St} \times \mathbf{St}$—i.e. the *relational semantics*. The Hasse diagram for the construction isomorphic to $\mathcal{P}_{\mathsf{L}}[\mathbf{N}_\perp]$ is:



Upper powerdomain The upper ordering on f.g. sets A, B, is given by:

$$A \preceq_{\mathsf{U}} B \iff \forall y \in B.\, \exists x \in A.\, x \sqsubseteq y$$

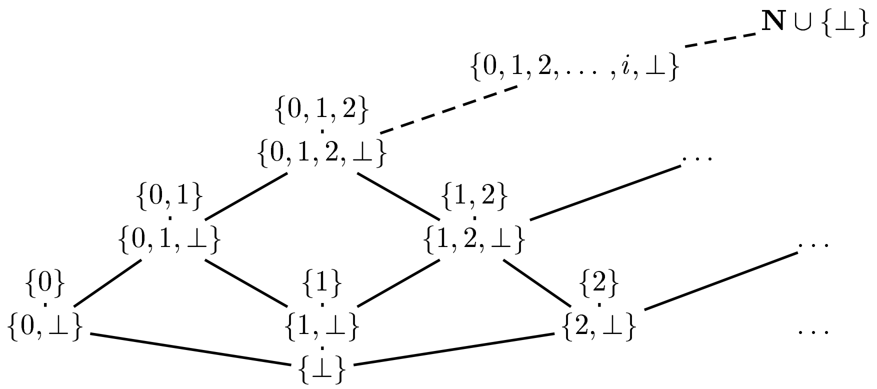
Here the induced discrete powerdomain $\mathcal{P}_{\mathsf{U}}[\mathbf{St}_\perp]$ is isomorphic to the set of finite non-empty subsets of $\mathbf{St}$ together with $\mathbf{St}_\perp$ itself, ordered by superset inclusion. The Hasse diagram for the construction isomorphic to $\mathcal{P}_{\mathsf{U}}[\mathbf{N}_\perp]$ is:



Convex powerdomain Let $A \preceq_{\mathsf{C}} B$ iff $A \preceq_{\mathsf{U}} B$ and $A \preceq_{\mathsf{L}} B$. This is also known as the Egli-Milner ordering. The resulting powerdomain $\mathcal{P}_{\mathsf{C}}[\mathbf{St}_\perp]$ is isomorphic to the f.g. subsets of $\mathbf{St}_\perp$, ordered by:

$$A \sqsubseteq_{\mathsf{C}} B \iff \begin{array}{l} \text{either } \perp \notin A \ \ \& \ \ A = B \\ \text{or } \perp \in A \ \ \& \ \ A \setminus \{\perp\} \subseteq B \end{array}$$

The Hasse diagram for $\mathcal{P}_{\mathsf{C}}[\mathbf{N}_\perp]$ is:



Note 1 *The isomorphisms mentioned and depicted in the diagrams above will be used implicitly in a number of constructions in this section in order to give pointwise definitions and proofs about powerdomains. (This is also the reason that the following development does not directly generalise from discrete to arbitrary powerdomains.) We will use the subsets from the isomorphic constructions to represent elements of the powerdomains $\mathcal{P}[\mathbf{St}_\perp]$ (the equivalence classes of f.g. subsets). In the case of $\mathcal{P}_\mathsf{C}[\mathbf{St}_\perp]$ (or $\mathcal{P}_\mathsf{C}[\mathbf{N}_\perp]$) it is straightforward—the equivalence classes are singleton sets each containing an f.g. set (see the above diagram for $\mathcal{P}_\mathsf{C}[\mathbf{N}_\perp]$).*

A few basic properties and definitions on powerdomains will be needed. For each powerdomain constructor $\mathcal{P}[-]$ define the order-preserving "unit" map $\eta_D : D \to \mathcal{P}[D]$ which takes each element $a \in D$ into (the powerdomain equivalence class of) the singleton set $\{a\}$. For each function $f \in [D \to \mathcal{P}[E]]$ we define the *extension* of f, denoted f^*, where $f^* \in [\mathcal{P}[D] \to \mathcal{P}[E]]$ by:

$$f^*(A) = \cup_{x \in A} f(x)$$

For each powerdomain constructor $\mathcal{P}[-]$ the structure $(\mathcal{P}[D], \eta_D(x), *)$ is a *Kleisli triple* or, alternatively, a *monad*, and thus satisfies the Kleisli triple laws [27]:

$$\begin{aligned} \eta_D^* &= id_{\mathcal{P}[D]} \\ f &= f^* \circ \eta_D \\ g^* \circ f^* &= (g^* \circ f)^* \end{aligned}$$

This provides a canonical way of composing the semantics of programs. In the particular setting of the denotations of commands, it is worth noting that $[\![C_1; C_2]\!]$ would be given by:

$$[\![C_1; C_2]\!] = [\![C_2]\!]^* \circ [\![C_1]\!]$$

2.3.3 Pers on Powerdomains

Given one of the discrete powerdomains, $\mathcal{P}[\mathbf{St}_\perp]$, we will need a "logical" way to lift a per $P \in Per(\mathbf{St}_\perp)$ to a per in $Per(\mathcal{P}[\mathbf{St}_\perp])$.

Definition 1 *For each $R \in Per(D_\perp)$ and each choice of power domain $\mathcal{P}[-]$, let $\mathcal{P}[R]$ denote the relation on $\mathcal{P}[D_\perp]$ given by:*

$$\begin{aligned} A\ \mathcal{P}[R]\ B \iff & \forall a \in A.\, \exists b \in B.\, a\ R\ b \\ \& \quad & \forall b \in B.\, \exists a \in A.\, a\ R\ b \end{aligned}$$

where we implicitly adopt the set-based representation of the elements of the powerdomain (Note 1).

It is easy to check that $\mathcal{P}[R]$ is a per, and in particular that $\mathcal{P}[Id_{D_\perp}] = Id_{\mathcal{P}[D_\perp]}$.

Henceforth we shall restrict our attention to the semantics of simple commands, and hence the three discrete powerdomains $\mathcal{P}[\mathbf{St}_\perp]$.

Proposition 1 *For any $f \in [\mathbf{St}_\perp \to \mathcal{P}[\mathbf{St}_\perp]]$ and any $R, S \in Per(\mathbf{St}_\perp)$,*

$$f : R \twoheadrightarrow \mathcal{P}[S] \iff f^* : \mathcal{P}[R] \twoheadrightarrow \mathcal{P}[S]$$

From this it easily follows that the following inference rule is sound:

$$\frac{[\![C_1]\!] : P \twoheadrightarrow \mathcal{P}[Q] \qquad [\![C_2]\!] : Q \twoheadrightarrow \mathcal{P}[R]}{[\![C_1; C_2]\!] : P \twoheadrightarrow \mathcal{P}[R]}$$

2.3.4 The Security Condition

We will investigate the implications of the security condition under each of the powerdomain interpretations. Let us suppose that, as before the state is partitioned into a high part and a low part: $\mathbf{St} = \mathbf{St}_{high} \times \mathbf{St}_{low}$. With respect to a particular choice of powerdomain let the security "type" $C : high \times low \to high \times low$ denote the property

$$[\![C]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{P}[(All \times Id)_\perp]$$

In this case we say that C is secure. Now we explore the implications of this definition on each of the possible choices of powerdomain:

1. In the lower powerdomain, the security condition describes in a weak sense termination-insensitive information flow. For example, the program

   $$\textsf{if } h = 0 \textsf{ then skip } [\!] \textsf{ loop else skip}$$

   (h is the high part of the state) is considered secure under this interpretation but the termination behaviour is influenced by h (it can fail to terminate only when $h = 0$).

2. In the upper powerdomain nontermination is considered catastrophic. This interpretation seems completely unsuitable for security unless one only considers programs which are "totally correct"—i.e. which must terminate on their intended domain. Otherwise, a possible nonterminating computation path will mask any other insecure behaviours a term might exhibit. This means that for *any* program C, the program $C \mathbin{[]} \mathsf{loop}$ is secure!

3. The convex powerdomain gives the appropriate generalisation of the deterministic case in the sense that it is termination sensitive, and does not have the shortcomings of the upper powerdomain interpretation.

2.4 Relation to an Equational Characterisation

In this section we relate the per-based security condition to a proposal by Leino and Joshi [25]. As before, assume for simplicity we have programs with just two variables: h and l of high and low secrecy respectively. Assume that the state is simply a pair, where h refers to the first projection and l is the second projection.

Leino and Joshi [25] define the security condition for a program C by:

$$HH; C; HH = C; HH$$

where "=" stands for semantic equality (the style of semantic specification is left unfixed), and HH is the program that "assigns to h arbitrary values"—aka "Havoc on H". We will refer to this equation as the equational security condition. Intuitively, the equation says that we cannot learn anything about the initial values of the high variables by variation of the low security variables. The postfix occurrences of HH on each side mean that we are only interested in the final value of l. The prefix HH on the left-hand side means that the two programs are equal if the final value of l does not depend on the initial value of h.

In relating the equational security condition to pers we must first decide upon the denotation of HH. Here [25] we run into some potential problems since it is necessary that HH always terminates, but nevertheless exhibits unbounded nondeterminism. Although this appears to pose no problems in [25] (in fact it goes without mention), to handle this we would need to work with non-ω-continuous semantics, and powerdomains for unbounded nondeterminism. Instead, we side-step the issue by assuming that the domain of h, $\mathbf{St}_{high}$, is finite. Suppose that $\mathbf{St}_{high} = \{v_1, \ldots, v_n\}$. Under this assumption, the definition of HH can be given by the command:

$$h := v_1 \,[\!]\, \cdots \,[\!]\, h := v_n$$

and its denotation can be written:

$$[\![HH]\!](\bot) = \{\bot\} \qquad [\![HH]\!](h, l) = \left\{(h', l) \mid h' \in \mathbf{St}_{high}\right\}$$

2.4.1 Equational Security and Projection Analysis

A first observation is that the the equational security condition is strikingly similar to the well-known form of static analysis for functional programs known as projection analysis [48]. Given a function f, a projection analysis aims to find projections (continuous lower closure operators on the domain) α and β such that

$$\beta \circ f \circ \alpha = \beta \circ f$$

For (generalised) strictness analysis and dead-variable analysis, one is given β, and α is to be determined; for binding time analysis [24] it is a forwards analysis problem: given α one must determine some β.

For strict functions (e.g., the denotations of commands) projection analysis is not so readily applicable. However, in the convex powerdomain HH is rather projection-like, since it effectively hides all information about the high variable; in fact it is an embedding (an upper closure operator) so the connection is rather close.

2.4.2 The Equational Security Condition Is Subsumed by the Per Security Condition

Hunt [18] showed that projection properties of the form $\beta \circ f \circ \alpha = \beta \circ f$ could be expressed naturally as a per property of the form $f : R_\alpha \twoheadrightarrow R_\beta$ for equivalence relations derived from α and β by relating elements which get mapped to the same point by the corresponding projection.

Using the same idea we can show that the per-based security condition subsumes the equation specification in a similar manner.

We will establish the following:

Theorem 1 *For any command C*

$$[\![HH;C;HH]\!] = [\![C;HH]\!]$$
$$\textit{iff}$$
$$C : high \times low \rightarrow high \times low$$

The idea will be to associate an equivalence relation to the function HH. More generally, for any command C let $ker(C)$, the *kernel* of C, denote the relation on $\mathbf{St}_\perp$ satisfying

$$s_1 \; ker(C) \; s_2 \iff [\![C]\!]s_1 = [\![C]\!]s_2$$

Extend $ker(C)$ to be the relation $ker^*(C)$ on $\mathcal{P}[\mathbf{St}_\perp]$ given by:

$$A \; ker^*(C) \; B \iff [\![C]\!]^*A = [\![C]\!]^*B$$

Recall the per interpretation of the type signature of C.

$$C : high \times low \rightarrow high \times low$$
$$\iff$$
$$[\![C]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{P}[(All \times Id)_\perp]$$

Observe that $(All \times Id)_\perp = ker(HH)$ since for any h, l, h', l' it holds $[\![HH]\!](h,l) = [\![HH]\!](h',l')$ iff $l = l'$ iff $(h,l)(All \times Id)_\perp(h',l')$.

The proof of the theorem is based on this observation and on the following two facts:

- $\mathcal{P}[(All \times Id)_\perp] = ker^*(HH)$ and
- $[\![HH;C;HH]\!] = [\![C;HH]\!] \iff [\![C]\!] : ker(HH) \twoheadrightarrow ker^*(HH)$.

Let us first prove the latter fact by proving a more general statement similar to Proposition 3.1.5 from Hunt's thesis [17] (the correspondence between projections and per-analysis). Note that we do not use the specifics of the convex powerdomain semantics here, so the proof is valid for any of the three choices of powerdomain.

Theorem 2 *Let us say that a command B is* idempotent *iff $[\![B;B]\!] = [\![B]\!]$. For any commands C and D, and any idempotent command B*

$$[\![B;C;D]\!] = [\![C;D]\!] \iff [\![C]\!] : ker(B) \twoheadrightarrow ker^*(D)$$

Proof. ($\Rightarrow$) : Assume $[\![B;C;D]\!] = [\![C;D]\!]$. Let s_1, s_2 denote arbitrary states such that s_1 $ker(B)$ s_2. We are required to show that $[\![C]\!]s_1$ $ker^*(D)$ $[\![C]\!]s_2$. Now

$$\begin{array}{ll}
[\![C;D]\!]s_1 & \\
= [\![B;C;D]\!]s_1 & \text{assn.} \\
= [\![C;D]\!]^* \circ [\![B]\!]s_1 & \\
= [\![C;D]\!]^* \circ [\![B]\!]s_2 & ker(B) \text{ defn.} \\
= [\![B;C;D]\!]s_2 & \\
= [\![C;D]\!]s_2. &
\end{array}$$

Now $[\![C;D]\!]s_1 = [\![C;D]\!]s_2$ implies

$$[\![D]\!]^*([\![C]\!]s_1) = [\![D]\!]^*([\![C]\!]s_2)$$

i.e.,

$$[\![C]\!]s_1 \; ker^*(D) \; [\![C]\!]s_2$$

as required.

($\Leftarrow$:) Assume $[\![C]\!] : ker(B) \twoheadrightarrow ker^*(D)$. Now $[\![B]\!]^*([\![B]\!]^*A) = [\![B]\!]^*A$ by idempotence, which implies

$$\begin{array}{lll}
\multicolumn{2}{l}{[\![B]\!]^*A \; ker^*(B) \; A} & \\
\Longrightarrow & [\![C]\!]^*([\![B]\!]^*A) \; ker^*(D) \; [\![C]\!]^*A & \text{assn.} \\
\Longleftrightarrow & [\![D]\!]^*([\![C]\!]^*([\![B]\!]^*A)) = [\![D]\!]^*([\![C]\!]^*A) & \text{defn. } ker^*(D) \\
\Longrightarrow & [\![B;C;D]\!]^*A = [\![C;D]\!]^*A & \square
\end{array}$$

Corollary. Since $[\![HH]\!]$ is idempotent we can conclude that

$$[\![HH;C;HH]\!] = [\![C;HH]\!] \iff [\![C]\!] : ker(HH) \twoheadrightarrow ker^*(HH)$$

It remains to establish the first fact.

Theorem 3 $\mathcal{P}[(All \times Id)_\perp] = ker^*(HH)$

Proof. We reason "pointwise" using the implicit representation of A and B as sets of states (cf. Note 1). ($\subseteq$): Suppose $A \; \mathcal{P}[(All \times Id)_\perp] \; B$, we need to show $[\![HH]\!]^*A = [\![HH]\!]^*B$. Cases

$$
\begin{array}{lll}
\perp \in [\![HH]\!]^*A & & \\
\iff & \perp \in A & HH\ \perp\text{-reflecting} \\
\implies & \perp \in B & \\
\iff & \perp \in [\![HH]\!]^*B & \\
(h,l) \in [\![HH]\!]^*A & & \\
\iff & \exists h_0\,.\,(h_0,l) \in A & \text{defn. } (\cdot)^* \\
\implies & \exists h_1\,.\,(h_1,l) \in B & \\
\iff & \forall h'\,.\,(h',l) \in [\![HH]\!]^*B &
\end{array}
$$

($\supseteq$): For the other direction assume $[\![HH]\!]^*A = [\![HH]\!]^*B$.

$$
\begin{array}{lll}
\perp \in A & \implies & \perp \in B \\
(h,l) \in A & \iff & \exists h_0\,.\,(h_0,l) \in [\![HH]\!]^*A \\
 & \implies & \exists h_0\,.\,(h_0,l) \in [\![HH]\!]^*B \\
 & \iff & \exists h_1\,.\,(h_1,l) \in B \qquad \square
\end{array}
$$

Thus, the equational and per security conditions in this simple case are equivalent.

2.4.3 Abstract Variables

In a more recent extension of the paper, [26], Leino and Joshi update their relational semantics to handle termination-sensitive leakages and introduce abstract variables—a way to support partially confidential data. Abstract variables h and l are defined as functions of the concrete variables in a program. For example, for a list of low length and high elements, l would be the length of the list and h would be the list itself. In the general case the choice of h and l could be dependent, so an *independence condition* must be verified.

Abstract variables are easily represented in our setting. Suppose that some function $g \in \mathbf{St} \to D$ yields the value (in some domain D) of the abstract low variable from any given state, then we can represent the security condition on abstract variables by: $[\![C]\!] : R_g \twoheadrightarrow \mathcal{P}[(All \times Id)_\perp]$ where $s_1 R_g s_2 \iff g\,s_1 = g\,s_2$.

2.5 A Probabilistic Security Condition

There are still some weaknesses in the security condition when interpreted in the convex powerdomain when it comes to the consideration of nondeterministic programs. In the usual terminology of information flow, we have considered *possibilistic information flows.* The probabilistic nature of an implementation may allow *probabilistic information flows* for "secure" programs. Consider the program

$$h := h \ \mathtt{mod} \ 100; (l := h \ [\!] \ l := \mathtt{rand}(99))$$

This program is secure in the convex powerdomain interpretation since regardless of the initial value of h, the final value of l can be any value in the range $\{0 \ldots 99\}$. But with a reasonably fair implementation of the nondeterministic choice and of the randomised assignment, it is clear that a few runs of the program, for a fixed input value of h, could yield a rather clear indication of its value by observing only the possible final values of l, e.g.:

$$17, 2, 45, 2, 2, 33, 2, 97, 2, 8, 57, 2, 2, 66, \ldots$$

—from which we might reasonably conclude that the value of h was 2. This observation is not new to this paper. It was first addressed in an abstract setting by Gray [13]. The above example is adapted from McLean [30]. In the setting of a concrete imperative language, Volpano and Smith recently devised a probabilistic security type-system [46] with a soundness proof based on an adaptation of Kozen's probabilistic operational semantics [22, 23]. They do not provide a *definition* of what it means for a program to be secure, and the security condition implicit in their correctness argument is not directly comparable—due to the fact that they consider parallel deterministic threads and a non-compositional semantics.

To counter the problem indicated by the example we consider *probabilistic powerdomains* [21] which allow the probabilistic nature of choice to be reflected in the semantics of programs, and hence enable us to capture the fact that varying the value of h causes *a change in the probability distribution* of values of l.

2.5.1 Probabilistic Powerdomain of Distributions

In the "possibilistic" setting we had the denotation of a command C to be a continuous function in $[\mathbf{St}_\perp \rightarrow \mathcal{P}_\mathsf{C}[\mathbf{St}_\perp]]$. In the probabilistic case, given an input to C not only do we keep track of possible outputs, but also of probabilities at which they appear. Thus, we consider a domain $\mathcal{E}[\mathbf{St}_\perp]$ of distributions over $\mathbf{St}_\perp$. The denotation of C is going to be a function in $[\mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]]$.

The general probabilistic powerdomain construction [21] on an inductive partial order $\mathcal{E}[D]$ is taken to be the domain of *evaluations*, which are certain continuous functions on $\Omega(D) \to [0,1]$, where $\Omega(D)$ is the lattice of open subsets of D. We will omit a description of the general probabilistic powerdomain of evaluations since for the present paper it is sufficient and more intuitive to work with discrete powerdomains, and hence a simplified notion of probabilistic powerdomain in terms of distributions. Evaluations rather than distributions would be needed to give a probabilistic semantics to a concurrent language based on the domain of *resumptions* [21]—but this is beyond the scope of the present article.

If S is a set (e.g., the domain of states for a simple sequential language) then we define the probabilistic powerdomain of $S_\perp$, written $\mathcal{E}[S_\perp]$ to be the domain of *distributions* on $S_\perp$, where a distribution μ is a function from $S_\perp$ to $[0,1]$ such that $\sum_{d \in S_\perp} \mu d = 1$. The ordering on $\mathcal{E}[S_\perp]$ is defined pointwise by $\mu \leq \nu$ iff $\forall d \neq \perp.\, \mu d \leq \nu d$. This structure is isomorphic to Jones and Plotkin's probabilistic powerdomain of evaluations for this special case.

As a simple instance of the probabilistic powerdomain construction [21], one can easily see that $\mathcal{E}[S_\perp]$ is an inductively complete partial order with directed lubs defined pointwise, and with a least element $\eta_{S_\perp}(\perp)$, where $\eta_{S_\perp}$ is the *point-mass distribution* defined for an $x \in S_\perp$ by:

$$\eta_{S_\perp}(x)d = \begin{cases} 1, & \text{if } d = x \\ 0, & \text{otherwise} \end{cases}$$

To lift a function $f : D_1 \to \mathcal{E}[D_2]$ to type $\mathcal{E}[D_1] \to \mathcal{E}[D_2]$ we define the *extension* of f by:

$$f^*(\mu)(y) = \sum_{x \in D_1} f(x)(y) \cdot \mu(x)$$

As in the case of the powerdomains for nondeterminism, $(\mathcal{E}[D], \eta_D(x), *)$ is a Kleisli triple, and thus, similarly to Section 2.3.2, we have a mechanism for composing the probabilistic semantics of any two given programs. Suppose $f : D_1 \to \mathcal{E}[D_2]$ and $g : D_2 \to \mathcal{E}[D_3]$ are such. Then the lifted composition $(g^* \circ f)^*$ can be computed by one of the Kleisli triple laws as $g^* \circ f^*$.

2.5.2 Pers on Probabilistic Powerdomains

The next step towards the security condition is to define how pers work on discrete probabilistic powerdomains. To lift pers to $\mathcal{E}[D]$ we need to consider a definition which takes into consideration the whole of each R-equivalence class in one go. The intuition is that an equivalence class of a per is a set of points that are indistinguishable by a low-level observer. For a given evaluation, the probability of a given observation by a low level user is thus the sum of probabilities over all elements of the equivalence class.

For a per $R \in \mathit{Per}(D)$ we define D/R to be the partition of $|D|$ onto R-equivalence classes. Given $e_R \in D/R$ and a distribution μ on D, let $\mu(e_R)$ be short-hand for $\sum_{x \in e_R} \mu(x)$. Define the per relation $\mathcal{E}[R]$ on $\mathcal{E}[D]$ for $\mu, \nu \in \mathcal{E}[D]$ by:

$$\mu \; \mathcal{E}[R] \; \nu \iff \forall e_R \in D/R.\, \mu(e_R) = \nu(e_R)$$

Naturally, $\mu \; \mathcal{E}[\mathit{Id}] \; \nu \iff \mu = \nu$ and $\forall \mu, \nu \in \mathcal{E}[D].\, \mu \; \mathcal{E}[\mathit{All}] \; \nu$.

As an example, consider $\mathcal{E}[(\mathit{All} \times \mathit{Id})_\perp]$. For two distributions μ and ν on $\mathbf{St}_\perp$ we have $\mu \; \mathcal{E}[(\mathit{All} \times \mathit{Id})_\perp] \; \nu$ iff the probability of any given low value l in the left-hand distribution, given by $\sum_h \mu(h, l)$, is equal to the probability in the right-hand distribution, namely $\sum_h \nu(h, l)$.

The probabilistic security condition is indeed a strengthening of the possibilistic one—when we consider programs whose possibilistic and probabilistic semantics are in agreement.

Theorem 4 *Suppose we have a possibilistic (convex) semantics $[\![\cdot]\!]_{\mathsf{C}}$ and a probabilistic semantics $[\![\cdot]\!]_{\mathcal{E}}$, which satisfy a basic consistency property that for any command C, if $[\![C]\!]_{\mathcal{E}} i\, o > 0$ then $o \in [\![C]\!]_{\mathsf{C}} i$.*

Now suppose that R and S are equivalence relations on D. Suppose further that C is any command such that possibilistic behaviour agrees with its probabilistic behaviour, i.e., $o \in [\![C]\!]_{\mathsf{C}} i \implies [\![C]\!]_{\mathcal{E}} i\, o > 0$. Then we have that $[\![C]\!]_{\mathcal{E}} : R \twoheadrightarrow \mathcal{E}[S]$ implies $[\![C]\!]_{\mathsf{C}} : R \twoheadrightarrow \mathcal{P}_{\mathsf{C}}[S]$.

Proof. For some $a, b \in D$ let $\mu = [\![C]\!]_{\mathcal{E}} a$ and $\nu = [\![C]\!]_{\mathcal{E}} b$. By $[\![C]\!]_{\mathcal{E}} : R \twoheadrightarrow \mathcal{E}[S]$ deduce $a \; R \; b \implies \forall d \in D.\, \mu([d]_S) = \nu([d]_S)$. What we need to prove is $a \; R \; b \implies [\![C]\!]_{\mathsf{C}} a \; \mathcal{P}_{\mathsf{C}}[S] \; [\![C]\!]_{\mathsf{C}} b$. So, assume $a \; R \; b$ and let us show that

$$\forall x \in [\![C]\!]_{\mathsf{C}} a.\, \exists y \in [\![C]\!]_{\mathsf{C}} b.\, x \; S \; y$$

Take any $x \in [\![C]\!]_{\mathsf{C}} a$. By the agreement hypothesis $\mu x > 0$, since if x is a possible output of program C run on data a, then the probability of getting this output must be greater then 0. So $\mu([x]_S) > 0$ and therefore $\nu([x]_S) > 0$. There must exist a $y \in [x]_S$ such that $\nu y > 0$. Thus, by the consistency property of the theorem's assumption, y is a possible output of program C run on data b, and $y \in [\![C]\!]_{\mathsf{C}} b$. The fact that $y \in [x]_S$ implies $x \; S \; y$. □

2.5.3 Probabilistic Security Condition for the Case of Two Variables

Let us derive the probabilistic powerdomain security condition for the case of two variables h and l and domain $D = \mathbf{St}_\perp$. C is secure iff

$$
\begin{aligned}
&[\![C]\!]_{\mathcal{E}} : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp] \\
&\iff (i_h, i_l) \; (All \times Id)_\perp \; (i'_h, i'_l) \implies [\![C]\!]_{\mathcal{E}}(i_h, i_l) \; \mathcal{E}[(All \times Id)_\perp] \; [\![C]\!]_{\mathcal{E}}(i'_h, i'_l) \\
&\iff \forall i_h, i'_h, i_l. \, [\![C]\!]_{\mathcal{E}}(i_h, i_l) \; \mathcal{E}[(All \times Id)_\perp] \; [\![C]\!]_{\mathcal{E}}(i'_h, i_l) \\
&\text{Let } \mu = [\![C]\!]_{\mathcal{E}}(i_h, i_l) \text{ and } \nu = [\![C]\!]_{\mathcal{E}}(i'_h, i_l) \\
&\iff \forall (o_h, o_l) \in \mathbf{St}_\perp. \, \mu([(o_h, o_l)]_{(All \times Id)_\perp}) = \nu([(o_h, o_l)]_{(All \times Id)_\perp}) \\
&\iff \forall o_l \in \mathbf{St}_{low}. \mu\perp = \nu\perp \;\; \& \;\; \textstyle\sum_{o_h \in \mathbf{St}_{high}} \mu(o_h, o_l) = \sum_{o_h \in \mathbf{St}_{high}} \nu(o_h, o_l)
\end{aligned}
$$

So, a command C is secure iff

$$[\![C]\!]_{\mathcal{E}}(i_h, i_l) \; \perp = [\![C]\!]_{\mathcal{E}}(i'_h, i_l) \; \perp \;\; \&$$

$$\sum_{o_h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i_h, i_l)(o_h, o_l) = \sum_{o_h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i'_h, i_l)(o_h, o_l)$$

for any i_l, i_h, i'_h and o_l. Intuitively the equation means that if you vary i_h the distribution of low variables does not change. The sums provide projecting out the high variable.

Let us introduce probabilistic powerdomain semantics definitions for some language constructs. Here we omit the $\mathcal{E}$-subscripts to mean the probabilistic semantics. Given two programs C_1, C_2 such that $[\![C_1]\!] : \mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]$ and $[\![C_2]\!] : \mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]$ the composition of two program semantics is defined by:

$$[\![C_1; C_2]\!] \; i \; o = \sum_{s \in \mathbf{St}_\perp} ([\![C_1]\!] \; i \; s) \cdot ([\![C_2]\!] \; s \; o)$$

The semantics of the uniformly distributed choice $C_1 \,[]\, C_1$ is defined by:

$$[\![C_1 \,[]\, C_2]\!]\ i\ o = 0.5[\![C_1]\!]\ i\ o + 0.5[\![C_2]\!]\ i\ o$$

In Section 2.6 we give the semantics of other language constructs.

Example. Recall the program

$$h := h \ \mathsf{mod}\ 100; (l := h \,[]\, l := \mathsf{rand}(99))$$

Now we investigate the security condition by varying the initial value of h from 0 to 1. Take $i_l = 0, i_h = 0, i'_h = 1$ and $o_l = 0$. The left-hand side is:

$$\sum_{o_h \in [0,\ldots,99]} [\![C]\!]_{\mathcal{E}}(0,0)(o_h,0) = 0.5 \cdot 1 + 0.5 \cdot 0.01 = 0.505$$

whereas the right-hand side is:

$$\sum_{o_h \in [0,\ldots,99]} [\![C]\!]_{\mathcal{E}}(1,0)(o_h,0) = 0.5 \cdot 0 + 0.5 \cdot 0.01 = 0.005$$

So, the security condition does not hold and the program must be rejected.

2.6 Analysis for Security Properties

The compositionality of the security condition can be fruitfully exploited when proving correctness of various compositional analyses. Instances of such are type-system-based analyses. In this section we investigate the compositionality properties of the security condition and show how an example type system [45], compositionally extended to handle probabilistic choice, can be proved correct with respect to the probabilistic security condition of Section 2.5.

2.6.1 A Simple Sequential Language with Probabilistic Choice

Consider a very simple sequential language with probabilistic choice ($[]_p$), described by the grammar in Figure 2.1. The semantics of the language is defined in Figure 2.2. The definition is an adaptation of the probabilistic semantics of Jones and Plotkin [21]. The $eval(Exp, s)$ function evaluates a boolean or arithmetic expression Exp in a state s. A state is a tuple

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid C_1 \,[\!]_p\, C_2 \\ & \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ B\ \mathsf{do}\ C \end{aligned}$$

Figure 2.1: Command syntax

$$\begin{aligned} [\![\mathsf{skip}]\!] &= \lambda s.\eta_s \\ [\![Id := Exp]\!] &= \lambda s.\eta_{s'},\ \text{where } s' = [eval(Exp, s)/Id]s \\ [\![C_1; C_2]\!] &= [\![C_2]\!]^* \circ [\![C_1]\!] \\ [\![C_1 \,[\!]_p\, C_2]\!] &= p \cdot [\![C_1]\!] + (1-p) \cdot [\![C_2]\!] \\ [\![\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!] &= \lambda s.\mathbf{if}\ eval(B, s)\ \mathbf{then}\ [\![C_1]\!]\ s\ \mathbf{else}\ [\![C_2]\!]\ s \\ [\![\mathsf{while}\ B\ \mathsf{do}\ C]\!] &= \mathsf{fix}\, f.\lambda s.\mathbf{if}\ eval(B, s)\ \mathbf{then}\ (f^* \circ [\![C]\!])s\ \mathbf{else}\ \eta_s \end{aligned}$$

Figure 2.2: Probabilistic semantics of commands

of variable values. Given an input state, executing commands skip and $Id := Exp$ results in the point-mass distribution for the input state (updated in the case of assignment $Id := Exp$).

The semantics of the sequential composition is defined using the extension operator introduced in Section 2.5.1.

When running $C_1 [\!]_p C_2$, command C_1 is chosen for execution with probability p and C_2 is chosen with probability $1-p$. The existence of the least fixpoint fix in the definition of the semantics for while is guaranteed by the inductive partial order construction of Jones and Plotkin [21].

Let us first establish the compositional nature of the probabilistic security condition and then go on with a straightforward proof of correctness of the type system.

2.6.2 Hook-up Properties

We start with a proposition that will be useful when reasoning about sequential composition of security types.

Proposition 2 *For any* $f \in [\mathbf{St}_\perp \to \mathcal{E}[\mathbf{St}_\perp]]$ *and any* $R, S \in Per(\mathbf{St}_\perp)$,

$$f : R \twoheadrightarrow \mathcal{E}[S] \iff f^* : \mathcal{E}[R] \twoheadrightarrow \mathcal{E}[S]$$

Proof. ($\Rightarrow$): Assume $f : R \twoheadrightarrow \mathcal{E}[S]$. It means $s_1 \; R \; s_2 \Longrightarrow f(s_1) \; \mathcal{E}[S] \; f(s_2)$ or, using the law $f = f^* \circ \eta$:

$$s_1 \; R \; s_2 \Longrightarrow \forall e_S \in \mathbf{St}_\perp/S.\, (f^* \eta_{s_1}) e_S = (f^* \eta_{s_2}) e_S.$$

Suppose $\mu \; \mathcal{E}[R] \; \nu$. We need to show $(f^*\mu) \; \mathcal{E}[S] \; (f^*\nu)$ or, unwinding the definition of the per $\mathcal{E}[S]$: $\forall e_S \in \mathbf{St}_\perp/S.\, (f^*\mu)e_S = (f^*\nu)e_S$. For any $e_S \in \mathbf{St}_\perp/S$:

$$\begin{array}{ll}
\multicolumn{2}{l}{(f^*\mu)e_S} \\
 & \text{rewrite } \mu(x) = \sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r) \cdot \eta_r(x) \\
= & (f^*(\sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r) \cdot \eta_r))e_S \\
 & \text{distributivity of } \cdot \text{ and } + \text{ over } f^* \\
= & (\sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r) \cdot (f^*\eta_r))e_S \\
 & \text{distributivity of } \cdot \text{ and } + \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r) \cdot ((f^*\eta_r)e_S) \\
 & \text{lhs of proposition} \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} \sum_{r \in e_R} \mu(r) \cdot ((f^*\eta_{r_0})e_S) \\
 & \text{for some } r_0 \text{ such that } [r_0]_R = e_R \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} ((f^*\eta_{r_0})e_S) \cdot \sum_{r \in e_R} \mu(r) \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} ((f^*\eta_{r_0})e_S) \cdot \mu(e_R) \\
 & \text{use } \forall e_R.\, \mu(e_R) = \nu(e_R) \\
= & \sum_{e_R \in \mathbf{St}_\perp/R} ((f^*\eta_{r_0})e_S) \cdot \nu(e_R) \\
= & (f^*\nu)e_S
\end{array}$$

($\Leftarrow$): Suppose $r_1 \; R \; r_2$. Need to show $f(r_1) \; \mathcal{E}[S] \; f(r_2)$.

$$\begin{array}{lll}
\multicolumn{3}{l}{r_1 \; R \; r_2} \\
\Longrightarrow & \eta_{r_1} \; \mathcal{E}[R] \; \eta_{r_2} & \\
\Longrightarrow & f^*(\eta_{r_1}) \; \mathcal{E}[S] \; f^*(\eta_{r_2}) & \text{rhs of proposition} \\
\Longrightarrow & f(r_1) \; \mathcal{E}[S] \; f(r_2) & f = f^* \circ \eta \quad \Box
\end{array}$$

Corollary. From this proposition it easily follows that the following inference rule is sound:

$$\frac{[\![C_1]\!] : P \twoheadrightarrow \mathcal{E}[Q] \qquad [\![C_2]\!] : Q \twoheadrightarrow \mathcal{E}[R]}{[\![C_1; C_2]\!] : P \twoheadrightarrow \mathcal{E}[R]}$$

In the standard security terminology, the *hook-up* property [28] says that if two programs are secure then their composition (sequential or some other kind) is secure as well. We will investigate what kinds of composition are allowed. These properties are important since they are the key to the utility of the specification for the purpose of proving the correctness of syntax-directed program analyses.

Recall the probabilistic powerdomain security condition for the case of two variables h and l and domain $D = \mathbf{St}_\perp$. A command C is secure iff

$$[\![C]\!]_{\mathcal{E}}(i_h, i_l) \perp = [\![C]\!]_{\mathcal{E}}(i'_h, i_l) \perp \;\; \&$$
$$\sum_{o_h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i_h, i_l)(o_h, o_l) = \sum_{o_h \in \mathbf{St}_{high}} [\![C]\!]_{\mathcal{E}}(i'_h, i_l)(o_h, o_l)$$

for any i_l, i_h, i'_h and o_l.

We will need some definitions to study various kinds of composition. A (boolean or arithmetic) expression Exp is *low* iff $\lambda s.eval(Exp, s) : All \times Id \twoheadrightarrow Id$. Otherwise, the expression is *high*.

We cannot plug a secure program into an arbitrary context to obtain a secure program. (Take, for example, any ground context that is insecure.) The security property is preserved by *secure contexts*, i.e. contexts built with secure components. Let $[\bullet]$ be a hole for a command. A context $\mathbb{C}[\bullet_1, \bullet_2]$ is secure iff it has one of the following forms:

$$\begin{aligned}\mathbb{C}[\bullet_1, \bullet_2] ::= \;& \mathsf{skip} \mid h := Exp \mid l := Exp \;\; (Exp \text{ is low})\\ & \mid [\bullet_1]; [\bullet_2] \mid [\bullet_1] \;[\!]_p\; [\bullet_2]\\ & \mid \mathsf{if}\; B \;\mathsf{then}\; [\bullet_1] \;\mathsf{else}\; [\bullet_2] \;\; (B \text{ is low })\\ & \mid \mathsf{while}\; B \;\mathsf{do}\; [\bullet_1] \;\; (B \text{ is low })\end{aligned}$$

Suppose $f : D \to \mathcal{E}[D]$. Define f's i'th *iteration* $f^i : D \to \mathcal{E}[D]$ by:

$$f^0 = \eta_D \qquad\qquad f^{n+1} = f^* \circ f^n$$

Theorem 5 (Hook-up) • *If C_1, C_2 are secure and $\mathbb{C}[\bullet_1, \bullet_2]$ is a secure context, then $\mathbb{C}[C_1, C_2]$ is secure.*

- *If $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided*

$$[\![C_1]\!]\ ((All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp])\ [\![C_2]\!]$$

- *If $\mathbb{C}[\bullet_1, \bullet_2] = \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided*

$$\begin{array}{ll} (i) & \forall i, j \in \mathbf{N}.\, [\![C_1]\!]^i\ ((All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp])\ [\![C_1]\!]^j \\ (ii) & \forall s.\, [\![\mathbb{C}[C_1, C_2]]\!]\ s \perp = 0 \end{array}$$

Proof. Cases on $\mathbb{C}[C_1, C_2] = C$. In each case we inspect the behaviour of the program fed with two *low-equal* input states s_1 and s_2 (the low components of the states are the same but the high components are possibly different).

skip

Start with low-equal s_1 and s_2. With either state, the computation of skip terminates in one step with the same states. So, the final states are low-equal as well.

$l := Exp$ (Exp is low)

Since Exp is a low expression, the computation of the command with low-equal initial states s_1 and s_2 terminates with the same states that are still low-equal.

$h := Exp$

Exp can be arbitrary. Computing the command will not change the low component of the states.

$C_1; C_2$

We know that $[\![C_i]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp], i = \{1, 2\}$. Applying the inference rule of the corollary of Proposition 2 yields $[\![C_2]\!]^* \circ [\![C_1]\!] : (All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp]$. Thus, C is secure.

$C_1 \,[\!]_p\, C_2$

Trivial, as computation does not depend on h.

if B then C_1 else C_2 (B is low)

Start with low-equal s_1 and s_2. The low expression B has the same value in either state. Thus, one step of computation will lead to a configuration with the same command (C_1 or C_2 in both cases) and low-equal states. C_1 and C_2 are secure and, therefore so is C.

if B **then** C_1 **else** C_2 **(B is high)**
Start with low-equal s_1 and s_2. Here the value of B might differ for s_1 and s_2, but, due to the assumption of the theorem, we have $[\![C_i]\!]\ (\mathcal{E}[(All \times Id)_\perp])\ [\![C_j]\!]$ for $i, j \in \{1, 2\}$. This guarantees the security of C.

while B **do** C_1 **(B is low)**
Start with low-equal s_1 and s_2. The low expression B has the same value in either state. Thus, the fixpoints in the semantics of the while under both initial states will be the same.

while B **do** C_1 **(B is high)**
Start with low-equal s_1 and s_2. The value of B might differ for s_1 and s_2. However, from the assumption for this case we know that (ii) the while-loop terminates and (i) any number of C_1-iterations will result in the same distribution of the final values of l. □

2.6.3 Type-system-based Analysis

Let us exemplify how the compositional nature of the security condition can be used for straightforward proofs of correctness of compositional analyses. In this subsection we consider a nontermination-covert-flow-sensitive type system by Volpano and Smith's [45], adapted to our language and extended to handle probabilistic choice. Note that this is an instance of a particular analysis that will reject some secure programs. The compositionality of the security condition can be used for justification of any compositional analysis of arbitrary precision. The point of this subsection is not to develop a powerful analysis, but rather to illustrate how simple the proof technique is using the compositional security condition.

Let us now describe the type system. The basic idea is to give security types (note the difference between these types and semantic security types used in security specification) to expressions (*high* or *low*) and commands (*high cmd* or *low cmd*) and make sure that these types do not mismatch when composing the commands. The main job of the types is to make sure that (i) an expression of type *low* does not contain an occurrence of the high variable h, (ii) a command of type *high cmd* does not have an occurrence of

$$
\begin{array}{ll}
[\mathrm{Var}] & h : high \qquad l : low \\[1em]
[\mathrm{Exp}] & n : \tau \qquad Exp : high \\[1em]
[\mathrm{Arithm}_{low}] & \dfrac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low} \\[1em]
[\mathrm{Skip}] & \mathsf{skip} : \tau\ cmd \\[1em]
[\mathrm{Assign}_{low}] & \dfrac{Exp : low}{l := Exp : low\ cmd} \\[1em]
[\mathrm{Assign}_{high}] & h := Exp : \tau\ cmd \\[1em]
[\mathrm{Seq}] & \dfrac{C_1 : \tau\ cmd \quad C_2 : \tau\ cmd}{C_1; C_2 : \tau\ cmd} \\[1em]
[\mathrm{Choice}] & \dfrac{C_1 : \tau\ cmd \quad C_2 : \tau\ cmd}{C_1 \mathbin{[]_p} C_2 : \tau\ cmd} \\[1em]
[\mathrm{If}] & \dfrac{B : \tau \quad C_1 : \tau\ cmd \quad C_2 : \tau\ cmd}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \tau\ cmd} \\[1em]
[\mathrm{While}] & \dfrac{B : low \quad C : \tau\ cmd}{\mathsf{while}\ B\ \mathsf{do}\ C : low\ cmd}
\end{array}
$$

Figure 2.3: Type system for security

assignment to the low variable l and (iii) while-loops have the type $low\ cmd$. These properties can be easily proved by induction.

The type system is presented in Figure 2.3. The rule Assign$_{low}$ prevents direct insecure information flows—for example, the assignment $l := h$ is not typable. The rule If prevents indirect insecure flows—the command $\mathsf{if}\ h = 0\ \mathsf{then}\ l := 0\ \mathsf{else}\ l := 1$ cannot be typed since the guard has type $high$ and the branches have type $low\ cmd$. In this particular system, the guard of the while-loop has to be low to prevent nontermination leaks. The while-loop has to have type $low\ cmd$. This guarantees that a while cannot be placed in a branch of a high conditional, with could also introduce a potential nontermination leak.

Let us now prove the security of the type system. We start off with a simple lemma. The lemma intuitively says that the behaviour of two commands with no loops or assignments to l can only differ in the high output as one varies the high input.

Lemma 1 $C_1 : high\ cmd, C_2 : high\ cmd \Longrightarrow$
$[\![C_1]\!]\ ((All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp])\ [\![C_2]\!]$

Proof. We need to show

$$[\![C_1]\!]_{\mathcal{E}}(i_h, i_l)\ \perp = [\![C_2]\!]_{\mathcal{E}}(i'_h, i_l)\ \perp\ \ \&$$
$$\sum_{o_h \in \mathbf{St}_{high}} [\![C_1]\!]_{\mathcal{E}}(i_h, i_l)(o_h, o_l) = \sum_{o_h \in \mathbf{St}_{high}} [\![C_2]\!]_{\mathcal{E}}(i'_h, i_l)(o_h, o_l)$$

for any i_l, i_h, i'_h and o_l. Clearly, $[\![C_1]\!]_{\mathcal{E}}(i_h, i_l)\ \perp = 0 = [\![C_2]\!]_{\mathcal{E}}(i'_h, i_l)\ \perp$ since nontermination is only possible in the presence of a while-loop. By property (iii) of the type system, no while-loop can occur in a high command. From the fact that there are no assignments to l ((ii)-property), deduce

$$\sum_{o_h \in \mathbf{St}_{high}} [\![C_i]\!]_{\mathcal{E}}(i_h, i_l)(o_h, o_l) = \begin{cases} 1, & \text{if } i_l = o_l \\ 0, & \text{otherwise} \end{cases}$$

for both $i = 1, 2$. This implies the second equality. □

The following theorem is a straightforward utilisation of the hook-up properties shown in Theorem 5.

Theorem 6 (Security of the Analysis)

$C : \tau\ cmd \implies C$ is secure.

Proof. Induction on C and direct application of the hook-up theorem. Let us consider the cases by which $C : \tau\ cmd$. The cases $\mathsf{skip} : \tau\ cmd$, $l := Exp : low\ cmd\ (Exp : low)$ and $h := Exp : \tau\ cmd$ are immediate since C is a ground secure context. (The case $l := Exp : low\ cmd\ (Exp : low)$ relies on the (i)-property of the type system which guarantees $Exp : low \Longrightarrow Exp$ is low.)

The cases $C_1; C_2 : \tau\ cmd$, $C_1 [\!]_p C_2 : \tau\ cmd$, $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : low\ cmd\ (B : low)$ and $\mathsf{while}\ B\ \mathsf{do}\ C : low\ cmd\ (B : low)$ provide secure contexts $[\bullet_1]; [\bullet_2]$, $[\bullet_1]\ [\!]_p\ [\bullet_2]$, $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ and $\mathsf{while}\ B\ \mathsf{do}\ [\bullet]$ (B is low by in the two latter by the (i)-property) respectively. By the induction hypothesis the sub-commands are secure, so the conclusion follows by the hook-up theorem.

The last case is $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : high\ cmd$. Both C_1 and C_2 have type $high\ cmd$. To apply the if-on-high case of Theorem 5 note that by Lemma 1 we have $[\![C_1]\!]\ ((All \times Id)_\perp \twoheadrightarrow \mathcal{E}[(All \times Id)_\perp])\ [\![C_2]\!]$. □

2.7 Conclusions

We have developed an extensional semantics-based specification of secure information flow in sequential programs, by embracing and extending earlier work on the use of partial equivalence relations to model binding times [19]. We have shown how this idea can be extended to handle nondeterminism and also probabilistic information flow.

A final remark on some recent related work: Abadi, Banerjee, Heintze and Riecke [1] show that a single calculus (DCC), based on Moggi's computational lambda calculus, can capture a number of specific static analyses for security, binding-time analysis, program slicing and call-tracking. Although their calculus does not handle nondeterministic language features, it is notable that the semantic model given to DCC is per-based, and the logical presentations of the abstract interpretation for per-based BTA [19, 20, 14] readily fit this framework (although this specific analysis is not one of those considered for DCC). They also show that what we have called "termination insensitive" analyses (Section 2.2.3) can be modelled by extending the semantic relations to relate bottom (nontermination) to every other domain point (without insisting on transitivity). It is encouraging to note that—at least in the deterministic setting—the per-based approach can be weakened to a termination-insensitive condition without significant technical difficulties. We do not, however, see any obvious way to make the probabilistic security condition insensitive to termination in a similar manner.

We conclude by considering a few possible extensions and limitations:

Multi-level security There is no problem with handling lattices of security levels rather than the simple *high-low* distinction. But one cannot expect to assign any intrinsic semantic meaning to such lattices of security levels, since they represent a "social phenomenon" which is external to the programming language semantics. In the presence of multiple security levels one must simply formulate conditions for security by considering information flows between levels in a pairwise fashion (although of course a specific static analysis is able to do something much more efficient).

Downgrading and Trusting There are operations which are natural to consider but which cannot be modelled in an obvious way in an extensional framework. One such operation is the downgrading of information from high to low without losing information—for example representing the secure encryption of high level information. This seems impossible since an encryption operation does not lose information about a value and yet should have type $high \to low$—but the only functions of type $high \to low$ are the constant functions. An analogous problem arises with Ørbæk and Palsberg's trust primitive if we try to use pers to model their *integrity analysis* [37].

Operational Semantics We are not particularly married to the denotational perspective on programming language semantics. We have reported an examination of operational formulations of pers on a multi-threaded language [41], based on partial bisimulations.

Constructing Program Analyses Although the model seems useful to compare other formalisations, further work is needed to show that it can assist in the systematic design of program analyses.

Concurrency Handling nondeterminism can be viewed as the main stepping stone to formulating a language-based security condition for concurrent languages. We have made an investigation of per-based security conditions for a multi-threaded language [41], using an operational rather than denotational semantic model of security.

Acknowledgements

We would like to thank the anonymous referees for valuable feedback.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.

[2] M. Abadi and G. Plotkin. A per model of polymorphism and recursive types. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 355–365. IEEE, 1990.

[3] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[4] J.-P. Banâtre, C. Bryce, and D. Le Métayer. Compile-time detection of information flow in sequential programs. In D. Gollmann, editor, *Computer Security—ESORICS '94, 3rd European Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–73. Springer Verlag, 1994.

[5] D.E. Bell and L.J. LaPadula. Secure Computer Systems: Unified Exposition and Multics Interpretation. MTR-2997, Rev. 1, The MITRE Corporation, Bedford, Mass., 1976.

[6] Ellis S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[7] Ellis S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[8] M. Das, T. Reps, and P. Van Hentenryck. Semantic foundations of binding-time analysis for imperative programs. In *Partial Evaluation*

and Semantics-Based Program Manipulation, pages 100–110, La Jolla, California, June 1995. ACM.

[9] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[10] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[11] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994.

[12] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, April 1982.

[13] J.W. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990. IEEE Computer Society Press.

[14] C. L. Hankin and D. Le Métayer. A type-based framework for program analysis. In *Proceedings of the First Static Analysis Symposium*, volume 864 of *LNCS*. Springer-Verlag, 1994.

[15] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, January 19–21, 1998.

[16] F. Henglein and D. Sands. A semantic model of binding times for safe partial evaluation. In Manuel Hermenegildo and S.D̃oaitse Swierstra, editors, *Proc. Programming Languages: Implementations, Logics and Programs (PLILP), Utrecht, The Netherlands*, volume 982 of *Lecture Notes in Computer Science*, pages 299–320. Springer-Verlag, September 1995.

[17] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1991.

[18] S. Hunt. PERs generalise projections for strictness analysis. In *Draft Proceedings of the Third Glasgow Functional Programming Workshop*, Ullapool, 1990.

[19] S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'91)*, pages 154–164, September 1991. ACM SIGPLAN Notices 26(9).

[20] T. P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, University of London, November 1992. Available as DIKU Report 93/11 from DIKU, University of Copenhagen.

[21] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[22] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22:328–350, 1981.

[23] D. Kozen. A probabilistic pdl. *Journal of Computer and System Sciences*, 30:162–178, 1985.

[24] J. Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, 1989.

[25] K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. In *MPC'98*, Springer-Verlag LNCS, 1998.

[26] K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1), 2000.

[27] E. Manes. *Graduate Texts in Mathematics*, volume 26. Springer-Verlag, 1976.

[28] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[29] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering.* Wiley & Sons, 1994.

[30] John McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, California, 1990. IEEE Computer Society Press.

[31] John McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, January 1990.

[32] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[33] I. S. Moskowitz and O. L. Costich. A classical automata approach to noninterference type problems. In *The Computer Security Foundations Workshop V proceedings: June 16–18, 1992, the Franconia Inn, Franconia, New Hampshire*, pages 2–8. IEEE Computer Society Press, 1992.

[34] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science—Fundamental Studies*, 69:117–242, 1989.

[35] Peter Ørbæk. Can you Trust your Data? In P. D. Mosses, M. I. Schwartzbach, and M. Nielsen, editors, *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag.

[36] Peter Ørbæk. *Trust and Dependence Analysis*. PhD thesis, Dept. of Computer Science, Univ. of Aarhus, 1997. BRICS report DS-97-2.

[37] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(4), 1997.

[38] G. Plotkin. Post-graduate lecture notes in advanced domain theory (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburgh, 1981.

[39] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976.

[40] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Proceedings 9th IFIP World Computer Congress, Information Processing '83, Paris, France, 19–23 Sept 1983*, pages 513–523. North-Holland, Amsterdam, 1983.

[41] A. Sabelfeld and D. Sands. Probabilistic noninterference for multithreaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, Cambridge, England, July 2000. IEEE Computer Society Press.

[42] Geoffrey Smith and Dennis Volpano. Secure information flow in a multithreaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, 1998.

[43] Michael B. Smyth. Powerdomains. *Journal of Computer and Systems Sciences*, 16(1):23–36, February 1978.

[44] P. Thiemann and H. Klaeren. Binding-time analysis by security analysis. Universitt Tübingen, November 1997.

[45] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[46] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[47] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

[48] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In *1987 Conference on Functional Programming and Computer Architecture*, pages 385–407, Portland, Oregon, September 1987.

[49] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

Chapter 3

Probabilistic Noninterference for Multi-threaded Programs

We present a probability-sensitive confidentiality specification – a form of probabilistic noninterference *– for a small multi-threaded programming language with dynamic thread creation. Probabilistic covert channels arise from a scheduler which is probabilistic. Since scheduling policy is typically outside the language specification for multi-threaded languages, we describe how to generalise the security condition in order to define robust security with respect to a wide class of schedulers, not excluding the possibility of deterministic (e.g., round-robin) schedulers and program-controlled thread priorities. The formulation is based on an adaptation of Larsen and Skou's notion of* probabilistic bisimulation. *We show how the security condition satisfies compositionality properties which facilitate straightforward proofs of correctness for, e.g., security type systems. We illustrate this by defining a security type system which improves on previous multi-threaded systems, and by proving it correct with respect to our stronger scheduler-independent security condition.*

This chapter was written with David Sands, and a condensed version was published in the *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 3–5, 2000. IEEE Computer Society Press.

3.1 Introduction

3.1.1 Motivation

When is an untrusted program safe to use? One aspect of safety is confidentiality. Given you have some confidential (high) data and some public (low) data in your computer, you want to make sure the attacker – the supplier of the untrusted code – will not learn anything about your personal data, despite the fact that the *application* (e.g., a spreadsheet) may require legitimate access to the confidential data in order to perform its task, and legitimate communication with the supplier of the code (e.g., a registration process for all users).

In this study we assume that the attacker is external to the (trusted) system upon which the program is run. Our aim is to specify when a program is safe to run – from the point of view of its confidentiality properties – with an aim to providing automatic methods for certifying programs prior to execution.

This paper describes some security specifications for multi-threaded programs executing on a single processor. Programs which execute several concurrent processes not only introduce nondeterminism, but also the potential for an attacker to observe the internal timing behaviours of programs, via the effect that the running time of commands might have on the scheduler's choice of when various threads are executed. Since a scheduler's behaviour might be probabilistic, this opens up an opportunity for the attacker to set up a probabilistic covert channel whereby information is obtained by statistical inferences drawn from the relative frequency of outcomes of a repeated computation.

We describe how *confidentiality* (the security property of not having insecure information flows) can be specified operationally for a simple imperative language with dynamic thread creation, and how the compositional nature of the security condition facilitates a straightforward proof of correctness of type-based program analyses.

An important contribution is to abstract over the scheduler's behaviour, so as to obtain a notion of security which holds for a broad class of schedulers, possibly including both probabilistic and deterministic behaviours. We illustrate this by defining a security type system which improves on a previous system of Volpano and Smith [20] for a simpler language, and by proving it correct with respect to our stronger explicitly scheduler-independent security condition.

3.1.2 Classification of Insecure Information Flows

It will be useful to informally classify a number of insecure information flows that might arise in programs. Let us restrict ourselves to a simplified setting: two variables h and l of high and low confidentiality respectively. Consider the examples of insecure information flows that security specifications and specific program analyses might attempt to rule out:

Direct: $l := h$
Direct information flow from the initial value of h to the final value of l.

Indirect: $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$
Secret information *indirectly* leaks from the initial value of h to the final value of l.

Termination behaviour: $\mathsf{if}\ h = 1\ \mathsf{then\ loop}$
If the program terminates, the attacker knows that h was not 1.

Probabilistic:
$h := h\ \mathsf{mod}\ 2; (l := h\ [\!]_{\frac{1}{2}}\ (l := 0\ [\!]_{\frac{1}{2}}\ l := 1))$
where $[\!]_{\frac{1}{2}}$ is a coin-flip choice operator. Here there is no information flow if we only consider the *possible* behaviours of the system, but the final value of l will reveal information about h with a probability of $\frac{3}{4}$.

Externally observable timing: $\mathsf{if}\ h = 1\ \mathsf{then\ sleep}\ 100$
The attacker with a stop-watch can learn information about h. This paper does not consider externally observable timing.

Internally observable timing:
$((\mathsf{if}\ h = 1\ \mathsf{then\ sleep}\ 100); l := 1) \mid l := 0$
where $\mid$ denotes the concurrent execution of the two commands. With many schedulers, the timing leaks will translate into an effect on the (distribution of) final value(s) of l. The information flow in this case is a combination of the timing and potentially probabilistic flows in the case that the scheduler exhibits stochastic behaviour. The ability to dynamically create threads also leads to similar possibilities.

3.1.3 Related Work

There is a substantial body of work in the security community in studying definitions and reasoning principles relating to information flow and confidentiality. For an overview see, e.g., [15]. The basic approach we take is based on an extentional characterisation usually known as *noninterference*. The particular flavour of noninterference studied here is most closely related to Gray's notion of *P-Restrictiveness* [10], since it aims to eliminate dependence between classified data and the probability distribution of unclassified outputs.

The use of program analysis as a means to eliminate insecure flows was pioneered by Denning [6, 7]. A modern incarnation of this work which recasts Denning's analysis as a type system and proves its correctness is [21].

Semantically, our security condition is founded on a probabilistic notion of bisimulation. Focardi and Gorrieri have promoted the use of (non-probabilistic) bisimulation in formalising and analysing security conditions in process calculus setting (see, e.g., [8]).

Focusing on concurrent imperative programs, there have been some recent considerations of concurrent programming languages. Banâtre, Bryce and Le Métayer [4] consider a programming language with static process structure and CSP-like communication primitives (see also [2]), although there is no extensional specification of correctness in these studies, and probabilistic information flows are not considered. Heintze and Riecke [11] present an analysis for a security-oriented lambda calculus, extended with threads and imperative features. Although they prove a noninterference property for the functional core of the language, they (quote) *"have not proved a noninterference theorem for the concurrent setting because ... the notion is unclear in the concurrent setting"*. Smith and Volpano [18] extend their previous work ([21]) with static threads. They prove a possibilistic correctness result for their analysis, and note that the analysis does not eliminate probabilistic covert channels (other than by making it overly restrictive). A later work [20] shows how to lift this restriction and proves some correctness properties based on a probabilistic semantics. We will return to consider this last work in more detail. In [16] we showed how partial equivalence relations (pers) and denotational semantics could capture security properties in sequential languages, including probabilistic information flows (noninterference-style) with the help of probabilistic powerdomains. In principle this machinery should be sufficient to handle concurrent languages, but the mathematical overheads are high. In this paper we also use partial equivalence relations, but this time operationally-based using probabilistic-bisimulation-like relations.

3.1.4 Overview

The rest of the paper is organised as follows. **Section 3.2** introduces syntax and probabilistic semantics of a multi-threaded language. **Section 3.3** defines the security condition based on probabilistic bisimulation. In **Section 3.4**, we introduce a scheduler-independent security condition. We go on to strengthen it by defining a strong security and illustrate its properties in **Section 3.5**. **Section 3.6** considers the applicability of the conditions to proving the soundness of compositional security analyses. **Section 3.7** concludes.

3.2 A Simple Concurrent Language

In order to be concrete – and in order to explore the compositionality properties of some of our security conditions – we consider a very simple shared-variable language described by the grammar in Figure 3.1. Let $C, D, E, \ldots$ range over commands Com, and let $\vec{C}$ denote a vector of commands of the form $\langle C_0 \ldots C_{n-1} \rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \ldots$ range over $\vec{Com} = \cup_{n \in \mathbb{N}} Com^n$, the set of thread pools (or programs).

We consider a small-step semantics for the language. A *configuration* c $(d, e, \ldots)$ is a pair of a command and a state (memory). In this paper, a *state* $s \in \mathbf{St}$ is a finite mapping from variables to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables respectively. In concrete examples we will often take just variables h and l, and in this case we will write the state as a pair. Define low-equivalence $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same. Let *Config* denote $\vec{Com} \times \mathbf{St}$. The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 3.2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\rightharpoonup$ transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle$, which means termination with the final state s', or $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| C'\vec{D}, s' |\!\rangle$. Here, one step of computation starting with command C in a state s gives a new main thread C', (possibly empty) vector of spawned processes $\vec{D}$ and a new state s'. Command $\mathsf{fork}(C\vec{D})$ dynamically creates a new vector of processes $\vec{D}$ which run in parallel with the main thread C. This has the effect of adding the vector of processes to the configuration.

The concurrent part of the semantics is presented in Figure 3.3. The $\rightarrow_p$

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

Figure 3.1: Command syntax

$$\langle\!|\mathsf{skip}, s|\!\rangle \twoheadrightarrow \langle\!|\langle\rangle, s|\!\rangle$$

$$\frac{\langle\!|Exp, s|\!\rangle \downarrow n}{\langle\!|Id := Exp, s|\!\rangle \twoheadrightarrow \langle\!|\langle\rangle, [Id = n]s|\!\rangle}$$

$$\frac{\langle\!|C_1, s|\!\rangle \twoheadrightarrow \langle\!|\langle\rangle, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \twoheadrightarrow \langle\!|C_2, s'|\!\rangle} \qquad \frac{\langle\!|C_1, s|\!\rangle \twoheadrightarrow \langle\!|C_1'\vec{D}, s'|\!\rangle}{\langle\!|C_1; C_2, s|\!\rangle \twoheadrightarrow \langle\!|(C_1'; C_2)\vec{D}, s'|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{True}}{\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s|\!\rangle \twoheadrightarrow \langle\!|C_1, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{False}}{\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s|\!\rangle \twoheadrightarrow \langle\!|C_2, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{True}}{\langle\!|\mathsf{while}\ B\ \mathsf{do}\ C, s|\!\rangle \twoheadrightarrow \langle\!|C; \mathsf{while}\ B\ \mathsf{do}\ C, s|\!\rangle}$$

$$\frac{\langle\!|B, s|\!\rangle \downarrow \mathsf{False}}{\langle\!|\mathsf{while}\ B\ \mathsf{do}\ C, s|\!\rangle \twoheadrightarrow \langle\!|\langle\rangle, s|\!\rangle}$$

$$\langle\!|\mathsf{fork}(C\vec{D}), s|\!\rangle \twoheadrightarrow \langle\!|C\vec{D}, s|\!\rangle$$

Figure 3.2: Small-step deterministic semantics of commands

$$\frac{\langle\!|C_i, s|\!\rangle \twoheadrightarrow \langle\!|\vec{C}, s'|\!\rangle}{\langle\!|\langle C_0 \dots C_{n-1}\rangle, s|\!\rangle \rightarrow^i_{\frac{1}{n}} \langle\!|\langle C_0 \dots C_{i-1}\vec{C}C_{i+1} \dots C_{n-1}\rangle, s'|\!\rangle}$$

$$(\Sigma)\frac{p = \sum \{\!|q \mid \langle\!|\vec{C}, s|\!\rangle \rightarrow^i_q \langle\!|\vec{D}, s'|\!\rangle|\!\}}{\langle\!|\vec{C}, s|\!\rangle \rightarrow_p \langle\!|\vec{D}, s'|\!\rangle}$$

Figure 3.3: Uniform probabilistic semantics of thread pools

transitions are probabilistic. The scheduler modelled here is purely probabilistic; it chooses to execute one of the threads of $\vec{C}$ with equal probability. We call this a *uniform* scheduler. In Section 3.4 we will parameterise over the choice of scheduler. Write $\langle\!|\vec{C}, s|\!\rangle \to \langle\!|\vec{D}, s'|\!\rangle$ whenever $\exists p > 0.\, \langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle$.

As is standard, in this paper we use multiset comprehensions $\{\!|p| \dots |\!\}$ when summing probabilities.

Labelling the transitions in the rules for parallel composition is performed in order to ensure that the sum of probabilities of transitions from a given nonterminal configuration is one. If there were no labels or rule Σ, then there would be $\langle\!|\langle \mathsf{skip} \mid \mathsf{skip}\rangle, s|\!\rangle \to_{1/2} \langle\!|\langle \mathsf{skip}\rangle, s|\!\rangle$ rather than $\langle\!|\langle \mathsf{skip} \mid \mathsf{skip}\rangle, s|\!\rangle \to_1 \langle\!|\langle \mathsf{skip}\rangle, s|\!\rangle$, where $\mid$ is used for separating threads in a thread pool. That is due to the fact that transition $\langle\!|\langle \mathsf{skip}, \mathsf{skip}\rangle, s|\!\rangle \to_{1/2} \langle\!|\langle \mathsf{skip}\rangle, s|\!\rangle$ could be performed by two different rules. That cannot be observed without the labelling or by making similar distinctions between different ways of obtaining the same transition. Similar constructions can be found in other probabilistic semantics.

The probabilistic semantics satisfies the following "soundness" properties for all nonterminal programs $\vec{C}$:

$$(i)\ \forall s.\ \sum \{\!|p|\, \langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle |\!\} = 1,$$
$$(ii)\ \langle\!|\vec{C}, s|\!\rangle \to_p \langle\!|\vec{D}, s'|\!\rangle, \langle\!|\vec{C}, s|\!\rangle \to_q \langle\!|\vec{D}, s'|\!\rangle \Longrightarrow p = q.$$

Observe that rule Σ guarantees property (ii) while property (i) must be independently enforced (and is dependent on expressions being total). Given these soundness conditions, the semantics can be viewed as an *unlabelled probabilistic transition system* [13], or a simple probabilistic automaton.

3.3 Probabilistic Noninterference

3.3.1 Background

Now that we have defined the syntax and semantics of the multi-threaded language, we are ready to specify what it means for a program to be secure. We are taking the view of *noninterference* to express the confidentiality property of a program. The central idea of noninterference [9] is that a program is secure whenever varying the initial values of high variables cannot change the low-observable (observable by the attacker) behaviour of the program.

A natural covert channel generalising the nontermination channel is the timing channel. A realistic version of noninterference should include timing as a possibly low-observable action. The first step in the direction of incorporating timing into noninterference was made by Volpano and Smith [20]. Volpano and Smith's noninterference is based on a lock-step execution of *probabilistic states*. A distribution μ of a set X (write $\mu \in \mathcal{D}(X)$) is a function $\mu \in X \to [0,1]$ such that $\sum_{x \in X} \mu x = 1$. A probabilistic state μ is a distribution of configurations $\mu \in \mathcal{D}(\vec{C} \times \mathbf{St})$. The computation is modelled by a Markov chain of probabilistic states. The initial probabilistic state is $\langle\!|\vec{C}, s|\!\rangle \mapsto 1$ where s is the initial memory and $\vec{C}$ is a static pool of threads (they do not handle new thread generation). Every following probabilistic state μ' is computed by multiplying the row vector of probabilities from μ by the stochastic transition matrix T. The rows and columns of T correspond to possible configurations reachable from the initial one. There might be a countably infinite number of such states, so the matrix might be countably infinite. The elements $T(i,j)$ of T are the probabilities of a transition from configuration i to configuration j.

From the point of view of the low observer two probabilistic states are indistinguishable (write $\sim_l$) iff projecting out high variables (which involves collapsing several probabilistic states into one in case the commands and low parts of the memories are the same; see [20] for the formal definition) makes the two states the same.

From the main correctness theorem we can extract the following probabilistic noninterference condition for a program $\vec{C}$ with stochastic transition matrix T:

$$\mu_1 \sim_l \mu_2 \text{ implies } \mu_1 T \sim_l \mu_2 T. \qquad (*)$$

While intuitive and adequate to prove the security of Volpano and Smith's type-system based analysis as an analysis-independent *definition* of security it suffers from the three drawbacks:

Noncompositional The probabilistic states are distributions of configurations (elements of $\mathcal{D}(\vec{C} \times \mathbf{St})$). This makes compositional reasoning at the command level impossible – commands are parts of configurations together with memories, i.e., the meaning of a term contains syntax.

Imprecise Since syntactic equality for commands is involved in the definition of low-equivalence, there is also a loss of precision. Consider the following example (or indeed any program with conditionals and no low variables):

$$\mathsf{if}\ h < 0\ \mathsf{then}\ h := h + 1\ \mathsf{else}\ h := h - 1.$$

Intuitively, for the low observer this program has the same behaviour regardless of the initial value of h – since the program does not modify l. However, it is considered insecure if one takes $(*)$ as a definition. Indeed, take $\mu_1 = \{\langle\!|\mathsf{if}\ h < 0\ \mathsf{then}\ h := h+1\ \mathsf{else}\ h := h-1, (7,0)|\!\rangle \mapsto 1\}$ and $\mu_2 = \{\langle\!|\mathsf{if}\ h < 0\ \mathsf{then}\ h := h + 1\ \mathsf{else}\ h := h - 1, (-7,0)|\!\rangle \mapsto 1\}$. Note $\mu_1 \sim_l \mu_2$. Now, $\mu_1 T = \{\langle\!|h := h - 1, (7,0)|\!\rangle \mapsto 1\}$ and $\mu_2 T = \{\langle\!|h := h + 1, (-7,0)|\!\rangle \mapsto 1\}$. After projecting out the high variable we are still left with syntactically different commands which implies $\mu_1 T \not\sim_l \mu_2 T$. To make the above example secure according to $(*)$, Volpano and Smith require that any such conditional with a high guard be executed atomically.

Scheduler Specific The security condition is specific to a particular scheduler, namely the scheduler with uniform probability of picking every thread. If the scheduler is changed then secure programs may become insecure. [1]

In the remainder of this section we begin by addressing the question of *precision* in the definition of security. The central problem of $(*)$ is that it does not abstract sufficiently from the syntactic program state. To abstract from the state we should factor in this (probabilistic) equivalence, in the manner of Gray's P-Restrictiveness condition [10]. We will develop probabilistic noninterference criteria based on probabilistic bisimulation. The idea is based on the definition of bisimulation on probabilistic transition systems [13, 3].

3.3.2 Relational Definitions and Notation

Let us first introduce some standard definitions and notation for relations. Let $Rel(A)$ denote the set of binary relations on A. If $R \in Rel(A)$ is an

[1] Note that this is not a problem for the actual security type system presented by Volpano and Smith; the proofs of correctness of their type system are easily seen to go through with other choices of probabilities in the transition matrix.

equivalence relation then A/R is the set of $R-$equivalence classes, $A/R \subseteq \wp(A)$. Id_A denotes the identity relation on some set A; we write just Id when A is clear from the context. As usual, R^+ denotes the transitive closure of R and R^* denotes the transitive and reflexive closure of R. For binary relations $P \in Rel(A)$, $Q \in Rel(B)$ we define the relation $P \times Q$ on $A \times B$ by

$$(x, y)\ P \times Q\ (x', y') \iff x\ P\ x' \ \&\ y\ Q\ y'.$$

A *partial equivalence relation* (per) on a set A is a binary relation on A which is *symmetric* and *transitive*. If P is such a per let $dom(P)$ denote the domain of P, given by

$$dom(P) = \{x \in A \mid x\ P\ x\}.$$

Note that the domain and range of a per P are both equal to $dom(P)$ (so for any $x, y \in A$, if $x\ P\ y$ then $x\ P\ x$ and $y\ P\ y$), and that the restriction of P to $dom(P)$ is an equivalence relation. Clearly, an equivalence relation is just a per which is reflexive (so $dom(P) = A$).

3.3.3 Probabilistic Bisimulation

In this subsection we recall the definition of probabilistic bisimulation, specialising it to the case of unlabelled systems.

An *unlabelled probabilistic transition system* (following [13]) is a set of states $\mathbb{S}$ and a set of transitions of the form $\mathbb{s} \to \mu$ where $\mathbb{s} \in \mathbb{S}$ and $\mu \in \mathcal{D}(\mathbb{S})$. The probabilistic semantics corresponds to a transition system where probabilistic transition states are identified with configurations. A small-step transition from a configuration to a distribution on configurations $\langle\!| \vec{C}, s |\!\rangle \to \mu$ where $\mu \in \mathcal{D}(\mathit{Config})$ can be viewed as a transition in the unlabelled probabilistic transition system such that $\mu(\langle\!| \vec{D}, s' |\!\rangle) = p$ iff $\langle\!| \vec{C}, s |\!\rangle \to_p \langle\!| \vec{D}, s' |\!\rangle$.

Define small-step semantics transitions from a configuration c to a *set* of configurations S by

$$c \to_p S \iff p = \sum \{\!| q | c \to_q d, d \in S |\!\}.$$

Inspired by the definition of Larsen and Skou, let us define probabilistic bisimulation on program configurations.

Definition 1 ([13]) *An equivalence relation $R \in Rel(\mathit{Config})$ is a* probabilistic bisimulation *if whenever $c\ R\ d$ then*

$$\forall S \in \mathit{Config}/R.\ c \to_p S \Longrightarrow d \to_p S.$$

Two configurations are probabilistically bisimilar if there is a *probabilistic bisimulation* which relates them. In the remainder of this article we will use a weakening of the definition of probabilistic bisimulation by relaxing the reflexivity condition.

Definition 2 *A per* $R \in \mathit{Rel}(\mathit{Config})$ *is a* partial probabilistic bisimulation *if whenever* $c\ R\ d$ *then*

$$(i)\ \forall S \in \mathit{Config}/R.\ c \rightarrow_p S \Longrightarrow d \rightarrow_p S,$$
$$(ii)\ c \rightarrow_0 \mathit{Config} \setminus \mathit{dom}(R).$$

The partiality introduced here can be thought of as simply removing "unreachable" parts of the relation. The modification will be useful when we specify security, but otherwise does not significantly change the definition. Note that any probabilistic bisimulation is also a partial probabilistic bisimulation. In the other direction we have:

Proposition 1 *R is a partial probabilistic bisimulation implies R^* is a probabilistic bisimulation.*

From this it follows that any two configurations which are contained in some partial probabilistic bisimulation R are also probabilistically bisimilar, and vice-versa.

Since we are interested in studying programs rather than configurations, we recover an equivalence relation on programs (vectors of commands) as follows.

Definition 3 *A per* $R \in \mathit{Rel}(\vec{\mathit{Com}})$ *is a* partial probabilistic bisimulation on thread pools *iff* $R \times \mathit{Id}_{\mathbf{St}}$ *is a partial probabilistic bisimulation on configurations.*

We write $c \sim d$ (resp. $C \sim D$) iff there exists a partial probabilistic bisimulation on configurations (commands) relating c and d (C and D).

It is possible to show that both $\sim$ relations are themselves (partial) probabilistic bisimulations (c.f. [3]). We sketch a proof of this fact in Appendix 3.A.

The partial probabilistic bisimulation on thread pools could form the basis of a compositional semantics – although one must at least restrict it to relations between equal-length vectors to obtain compositionality. We will not pursue this investigation further in this paper since we are not particularly interested in the uniform scheduler.

3.3.4 Noninterference Based on Partial Probabilistic Bisimulation

Our aim at this point is to construct a modification of probabilistic bisimulation that reflects the "equivalence" on program behaviour visible for the attacker. The intuition behind the construction is that a program is secure iff for any two states which differ only in the values of high variables, two configurations containing the program and each of the states, execute in such a way that their behaviour is indistinguishable (or "low-equivalent") from the attacker's observation of the low parts of the state and the probability with which they occur. Let us formally define such a "low-equivalence" as a partial probabilistic bisimulation. The development parallels Section 3.3.3, with the modification that the states of equivalent configurations can be different in the high components.

Definition 4 *A per $R \in Rel(\vec{Com})$ is a* partial probabilistic low-bisimulation *iff $R \times (=_L)$ is a partial probabilistic bisimulation on configurations.*

Write $C \sim_L D$ (C and D are *low-bisimilar*) iff there exists a partial probabilistic low-bisimulation relating programs C and D.

There is an alternative and equivalent way to define this partial probabilistic low-bisimulation directly in terms of the operational semantics:

Proposition 2 *A per R is a partial probabilistic low-bisimulation on commands iff whenever $\vec{C}\ R\ \vec{D}$ then*

$$\begin{array}{l}
\forall s_1 =_L s_2 . \langle\!|\vec{C}, s_1|\!\rangle \to \langle\!|\vec{C'}, s_1'|\!\rangle \Longrightarrow \\
\quad \exists \vec{D'}, s_2' .\ \langle\!|\vec{D}, s_2|\!\rangle \to \langle\!|\vec{D'}, s_2'|\!\rangle \\
\qquad (i) \sum \{\!|p|\langle\!|\vec{C}, s_1|\!\rangle \to_p \langle\!|\vec{S}, s|\!\rangle, \vec{S} \in [\vec{C'}]_R, s =_L s_1'|\!\} = \\
\qquad\quad \sum \{\!|p|\langle\!|\vec{D}, s_2|\!\rangle \to_p \langle\!|\vec{S}, s|\!\rangle, \vec{S} \in [\vec{D'}]_R, s =_L s_2'|\!\} \\
\qquad (ii)\ \ \vec{C'}\ R\ \vec{D'}, s_1' =_L s_2',
\end{array}$$

where $[\vec{E}]_R$ stands for the $R-$equivalence class which contains $\vec{E}$.

With this formulation the (informal) connection to Gray's P-restrictiveness condition becomes clearer.

Analogously to Section 3.3.3, we obtain the result that $\sim_L$ is itself a partial probabilistic low-bisimulation and it is the greatest fixed point of the corresponding functor. We are now ready for the culmination of this section, the security specification for a thread pool:

$$\vec{C} \text{ is } \textit{secure} \iff \vec{C} \sim_L \vec{C}.$$

The observations about proof techniques for $\sim$ from Appendix 3.A hold for the case of $\sim_L$ as well.

3.4 Scheduler-independent Noninterference

The security specification above captures the probabilistic and internal timing covert channels. We have so far presumed the uniform scheduler is used for choosing the next thread. However, the scheduler is typically not specified by the language definition, and may vary from implementation to implementation. In different implementations, different probabilistic policies of a scheduler can be used, and these specific policies can be exploited by the attacker. Under the worst case assumption the security condition should be scheduler-independent – in other words we should assume that the attacker chooses the scheduler. In this section, we generalise the security condition to be robust with respect to any particular scheduler used – for a reasonable class of schedulers. The schedulers in question will be defined as a function from the history of the computation (and the low variables) to a probability distribution on the set of live threads.

Let us start with an example that illustrates that the uniform scheduler assumption does not imply security for other schedulers. Suppose that h is a boolean. Consider the following program:

$$\mathsf{if}\ h > 0\ \mathsf{then}\ \mathsf{fork}(l := 0 l := 1)\ \mathsf{else}\ \mathsf{fork}(l := 1 l := 0)$$

While this is a secure program for the uniform scheduler that picks threads with equal probability, it is insecure for any other scheduler. For example, a round-robin scheduler might execute threads in a deterministic order. Although the attacker may not know this in advance, he may easily be able to determine it with reasonable certainty (using a program which does not touch high data), and thus deduce the value of h. This motivates the introduction of a *scheduler-independent low-bisimulation* ($\approx_L$).

3.4.1 Semantics with Schedulers

In order to obtain a security condition which is robust with respect to schedulers we extend the semantics of the language to explicitly schedule threads.

But what is a scheduler? Abstractly we will take a scheduler to be a mechanism for selecting threads which itself satisfies some noninterference property, i.e., its behaviour is independent of high data. Why should a scheduler have access to the state at all? By allowing the scheduler to depend on the low variable, it becomes possible to model threads with *dynamic priorities*, i.e., where the program can control the scheduler's behaviour using the values of certain distinguished low program variables.

The scheduler needs more than the low part of the state however. A minimum requirement of any reasonable scheduler is that it also knows the number of threads from which it can choose. This information will be part of the *history* of the computation so far. A scheduler will be defined to be a function with arguments (i) the *history* of the computation so far, and (ii) the low part of a state, and yielding as its result a probability distribution on the number of threads currently in the configuration.

Let us first inductively define the set of possible schedule histories $Hist$. A *history* H is a sequence of pairs $H \in (\mathbb{N} \times \mathbb{N})^*$, representing information about the computation steps so far. In each pair of the sequence, the first component is the index of the thread last chosen for computation, and the second component is the total number of threads that remained in the configuration after that thread's computation step. Not all sequences describe valid histories. In any two adjacent pairs in a history, $\ldots (i, m)(j, n) \ldots$, the tread j that is selected must be in the range $0 \ldots m-1$, and the number of threads remaining after j's step (n) cannot be smaller than $m-1$, since at most one thread can die in any computation step (although many may be created). Assuming without loss of generality that a computation always starts with a single thread and the empty history, we have the following inductive rules that define $Hist$:

$$\epsilon \in Hist \qquad (0, n) \in Hist$$

$$\frac{H(i, m) \in Hist}{H(i, m)(j, n) \in Hist}(j \leq m - 1 \leq n)$$

Define the number of live processes for a given history by

$$live(\epsilon) = 1, \qquad live(H(i, n)) = n.$$

A scheduler σ is any function that given a history H and the low part l of a state returns a probability distribution on live processes. The dependent type of $\sigma(H,l)$ is

$$\sigma(H,l) \in \mathcal{D}(\{0 \ldots live(H) - 1\}).$$

Given a scheduler σ not all histories are feasible under σ. Define the set of *σ-feasible* histories $Hist_\sigma \subseteq Hist$ by

$$\epsilon \in Hist_\sigma$$

$$\frac{H \in Hist_\sigma \quad H(i,m) \in Hist \quad \exists l.\, \sigma(H,l)i > 0}{H(i,m) \in Hist_\sigma}$$

At each step of a computation with low-state l, only those threads can be picked for which $\sigma(H,l) > 0$. $Hist_\sigma$ will be used in the definition of σ-specific low-bisimulation – although it should be noted that $Hist_\sigma$ is still, in general, a superset of the histories which are *actually* feasible since not all l are necessarily feasible.

To introduce the scheduler to the semantics, we need to update the computation history with each successive computation step. We extend configurations to contain a history along with a command vector and a state. An initial configuration has the form $\langle\!|\epsilon, C, s|\!\rangle$. Now we replace the semantics of thread pools in Figure 3.3 by the rule

$$\frac{\langle\!| C_i, s |\!\rangle \rightharpoonup \langle\!| \vec{C}, s' |\!\rangle}{\langle\!| H, \langle C_0 \ldots C_i \ldots C_{n-1}\rangle, s |\!\rangle \rightarrow_p \langle\!| H', \langle C_0 \ldots \vec{C} \ldots C_{n-1}\rangle, s' |\!\rangle}$$

where $H' = H(i, n + |\vec{C}| - 1), p = \sigma(H,l)\, i$ and $s = (h,l)$ for some h. The history is updated with the number of the thread being picked and the new thread-pool size. In a context where we are discussing several different schedulers, we will explicitly label the transition arrow thus $\rightarrow_p^\sigma$.

This model of the semantics with schedulers is general enough to describe any scheduler that uses the full history of thread creation/deletion to generate probabilities for picking the threads. Note however that it cannot model a scheduler which uses real-time. Consider some examples of common schedulers. Any scheduler must pick the first (and the only) thread given the empty history, so we can safely omit this case in the following examples.

For any history H, the deterministic *round robin* scheduler is defined by

$$roundr(H(i,n),l)\, x = \begin{cases} 1, & \text{if } x = (i+1) \ mod\ n, \\ 0, & \text{otherwise.} \end{cases}$$

A round-robin scheduler which chooses the same thread for several steps before switching would need to look further back in the history.

The *uniform scheduler* presented in Section 3.2 is simply

$$uni(H, l)x = 1/live(H),$$

for any $x \in \{0 \ldots live(H) - 1\}$. As in the example above, the previous history and the value of the low variable are disregarded. Note that the history contains sufficient information to calculate the individual lifetimes of threads, or the creation hierarchy of the threads in order that these may be used to influence – either deterministically or stochastically – the policy of the scheduler.

3.4.2 Probabilistic Noninterference with Schedulers

Given a scheduler σ, let us define the scheduler-specific partial probabilistic low-bisimulation. This bisimulation is a kind of lock-step execution as we have already seen for $\sim_L$. For two bisimilar configurations, if one makes a transition to an equivalence class then the other configuration should also be able to make a transition with the same probability to the same class. However, the history in the second configuration can vary from the one in the first configuration. Let us capture the degree of how the history can vary by defining a relation $=_\sigma$ on σ-feasible histories. It relates σ-feasible histories which are indistinguishable for σ. Define $=_\sigma \in Rel(Hist_\sigma)$ to be the largest relation satisfying

$$\frac{H =_\sigma H'}{\forall l.\, \sigma(H, l) = \sigma(H', l)} \qquad \frac{H =_\sigma H'}{H(i, m) =_\sigma H'(i, m)}$$

For example, $=_{uni}$ relates any two histories such that the number of live threads is the same.

Using the method of Section 3.3 define the scheduler-specific partial probabilistic bisimulation on configurations who now include histories. We have now everything we need to define a scheduler-specific partial probabilistic low-bisimulation on thread pools.

Definition 5 *Given a scheduler σ, a per $R \in Rel(\vec{Com})$ is a* σ-specific partial probabilistic low-bisimulation *iff $(=_\sigma) \times R \times (=_L)$ is a σ-specific partial probabilistic bisimulation on configurations.*

Write $\vec{C} \sim_L^\sigma \vec{D}$ ($\vec{C}$ and $\vec{D}$ are *low-bisimilar under* σ) iff there exists a partial probabilistic low-bisimulation relating programs $\vec{C}$ and $\vec{D}$.

Note that for the uniform scheduler uni, we have $\sim_L^{uni} = \sim_L$, where $\sim_L$ is the low-bisimulation from Section 3.3. Given a scheduler σ and a program $\vec{C}$,

$$\vec{C} \text{ is } \sigma\text{-}secure \iff \vec{C} \sim_L^\sigma \vec{C}.$$

3.4.3 Scheduler-independent Noninterference

As we have argued, a realistic security condition should be scheduler-independent. We are now ready to define the *scheduler-independent* low-bisimulation $\approx_L$ and the scheduler-independent security (SI-security). Define

$$\vec{C} \approx_L \vec{D} \iff \forall \sigma. \vec{C} \sim_L^\sigma \vec{D}, \text{ and}$$

$$\vec{C} \text{ is } \mathit{SI\text{-}secure} \iff \vec{C} \approx_L \vec{C}.$$

Trivially, $\approx_L \subseteq \sim_L$. The example in the beginning of this section shows that the reverse inclusion does not hold. As a corollary, we have $\vec{C}$ is SI-secure implies $\vec{C}$ is secure.

On a side note, the quantification over schedulers is somewhat reminiscent of the "strategies" in the adversary model used by Syverson and Gray [19], which are also functions from computation histories to distributions. Note however that we are not using them to model attacker behaviour (e.g., I/O) – but rather an internal component of the system.

We define yet another low-bisimulation along with a security condition as an aid to prove various analyses secure. This condition is potentially the strongest of all considered in the paper; and proofs of the soundness of particular analyses go through easiest just with this one. Call the new low-bisimulation *strong*.

In order to achieve a compositional bisimulation we want to restrict the strong low-bisimulation so that any two strongly low-bisimilar thread pools must be of equal size and must create/kill exactly the same number of processes at each step under any scheduler. This means that we do not need the machinery for explicit scheduling. In fact, the new low-bisimulation is purely nondeterministic.

Definition 6 *Define the* strong *low-bisimulation* $\approxeq_L$ *to be the union of all symmetric relations* R *on thread pools of equal size, such that whenever* $\langle C_0 \dots C_{n-1}\rangle \; R \; \langle D_0 \dots D_{n-1}\rangle$ *then*

$$\begin{aligned} &\forall s_1 =_L s_2 \forall i. \langle\!| C_i, s_1 |\!\rangle \rightharpoonup \langle\!| \vec{C}', s_1' |\!\rangle \implies \\ &\qquad \exists \vec{D}', s_2'. \langle\!| D_i, s_2 |\!\rangle \rightharpoonup \langle\!| \vec{D}', s_2' |\!\rangle, \vec{C}' \; R \; \vec{D}', s_1' =_L s_2'. \end{aligned}$$

The strong security specification goes:

$$\vec{C} \text{ is } \textit{strongly secure} \iff \vec{C} \approxeq_L \vec{C}.$$

We are going to use this specification of security in the proofs of analyses correctness. Let us first show that the strong low-bisimulation is indeed stronger than the scheduler-independent one.

Proposition 3 $\vec{C}$ *is strongly secure* $\implies \vec{C}$ *is SI-secure.*

Proof. In order to show $\approxeq_L \subseteq \approx_L$, let us check that $\approxeq_L$ is a σ-specific partial probabilistic low-bisimulation for all σ. Assuming $\vec{C} \approxeq_L \vec{D}$, we need to show that for any σ:

$$\begin{aligned} &\forall H_1 =_\sigma H_2, s_1 =_L s_2. \langle\!| H_1, \vec{C}, s_1 |\!\rangle \to \langle\!| H_1', \vec{C}', s_1' |\!\rangle \implies \\ &\exists H_2', \vec{D}', s_2'. \; \langle\!| H_2, \vec{D}, s_2 |\!\rangle \to \langle\!| H_2', \vec{D}', s_2' |\!\rangle \\ &\qquad (i) \sum \{\!| p \,|\, \langle\!| H_1, \vec{C}, s_1 |\!\rangle \to_p \langle\!| H, \vec{S}, s |\!\rangle, \vec{S} \in [\vec{C}']_{\approxeq_L}, \\ &\qquad\qquad H =_\sigma H_1', s =_L s_1' |\!\} = \\ &\qquad\quad \sum \{\!| p \,|\, \langle\!| H_2, \vec{D}, s_2 |\!\rangle \to_p \langle\!| H, \vec{S}, s |\!\rangle, \vec{S} \in [\vec{D}']_{\approxeq_L}, \\ &\qquad\qquad H =_\sigma H_2', s =_L s_2' |\!\}, \\ &\qquad (ii) \;\; H_1' =_\sigma H_2', \vec{C}' \approxeq_L \vec{D}', s_1' =_L s_2'. \end{aligned}$$

Given arbitrary $H_1 =_\sigma H_2$ and $s_1 =_L s_2$, suppose $\vec{C} = \langle C_0 \dots C_{n-1}\rangle$ and $\langle\!| H_1, \vec{C}, s_1 |\!\rangle \to \langle\!| H_1', \vec{C}', s_1' |\!\rangle$. Then there exists i such that the probabilistic transition was triggered by a deterministic transition within C_i using the rule

$$\frac{\langle\!| C_i, s_1 |\!\rangle \rightharpoonup \langle\!| \vec{I}, s_1' |\!\rangle}{\langle\!| H_1, \vec{C}, s_1 |\!\rangle \to_p \langle\!| H_1', \langle C_0 \dots \vec{I} \dots C_{n-1}\rangle, s_1' |\!\rangle}$$

where $H_1' = H_1(i, n+|\vec{I}|-1)$, $p = \sigma(H_1, l)\, i$ and $s_1 = (h_1, l)$ for some h_1 and $p > 0$. From the definition of $\cong_L$ deduce that $\vec{D}$ has the form $\langle D_0 \ldots D_{n-1}\rangle$ such that $\exists \vec{J}, s_2'.\langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{J}, s_2' |\!\rangle$ and $\vec{I} \cong_L \vec{J}, s_1' =_L s_2'$. Hence,

$$\frac{\langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{J}, s_2' |\!\rangle}{\langle\!| H_2, \vec{D}, s_2 |\!\rangle \rightarrow_q \langle\!| H_2', \langle D_0 \ldots \vec{J} \ldots D_{n-1}\rangle, s_2' |\!\rangle}$$

where $H_2' = H_2(i, n+|\vec{J}|-1)$, $q = \sigma(H_2, l)\, i$ and $s_2 = (h_2, l)$ for some h_2 and $q > 0$. Observe that $H_1 =_\sigma H_2$ immediately yields $p = q$. Note that $|\vec{I}| = |\vec{J}|$ implies $H_1' = H_2'$ which, in turn, gives $H_1' =_\sigma H_2'$. Since $C_i \cong_L D_i$ for all $i = 0, \ldots, n-1$ and $\vec{I} \cong_L \vec{J}$, looking ahead in Section 3.5, apply the secure congruence theorem (Theorem 1) to obtain $\vec{C'} = \langle C_0 \ldots \vec{I} \ldots C_{n-1}\rangle \cong_L \langle D_0 \ldots \vec{J} \ldots D_{n-1}\rangle = \vec{D'}$. This completes the proof of the condition (*ii*). The sum equality of (*i*) holds since there is a one-to-one correspondence between the two multisets that are summed over. □

3.4.4 Relating Low-bisimulations under Compositional Security

Let us investigate the relation between low-bisimulations in case the security conditions are required to be compositional. Notice that the quantification over all schedulers in the SI-security condition forces the restriction on the number of threads of two bisimilar programs, i.e., if $\vec{C} \approx_L \vec{D}$ then $|\vec{C}| = |\vec{D}|$. Indeed, otherwise there is a scheduler that can use the difference in the number of threads to modify the low variable which can be used to leak information. For example, it holds that $\mathsf{skip} \mid \mathsf{skip} \approx_L \mathsf{skip}; \mathsf{skip}$ because for any scheduler the two programs execute deterministically in lock-step. However, the parallel composition of the each of the two with the thread $l := 5$ will not preserve the bisimulation $l := 5 \mid \mathsf{skip} \mid \mathsf{skip} \not\approx_L l := 5 \mid (\mathsf{skip}; \mathsf{skip})$ since the probability of l to become 5 depends on the number of threads and many schedulers will reflect it in the semantics. This applies, for example, to the uniform scheduler: $l := 5 \mid \mathsf{skip} \mid \mathsf{skip} \not\sim_L l := 5 \mid (\mathsf{skip}; \mathsf{skip})$.

Under the assumption that the number of threads of low-bisimilar programs must be the same, the definition of scheduler-specific low-bisimulation under a history-independent probabilistic scheduler is this (cf. Proposition 2):

Definition 7 *A per R is a partial probabilistic low-bisimulation on com-*

mands iff whenever $\vec{C}\ R\ \vec{D}$ *then* $|\vec{C}| = |\vec{D}|$ *and*

$$\begin{aligned}
&\forall s_1 =_L s_2. \langle\!| \vec{C}, s_1 |\!\rangle \to \langle\!| \vec{C}', s_1' |\!\rangle \Longrightarrow \\
&\quad \exists \vec{D}', s_2'. \langle\!| \vec{D}, s_2 |\!\rangle \to \langle\!| \vec{D}', s_2' |\!\rangle \\
&\qquad (i) \sum \{\!| p | \langle\!| \vec{C}, s_1 |\!\rangle \to_p \langle\!| \vec{S}, s |\!\rangle, \vec{S} \in [\vec{C}']_R, s =_L s_1' |\!\} = \\
&\qquad\quad \sum \{\!| p | \langle\!| \vec{D}, s_2 |\!\rangle \to_p \langle\!| \vec{S}, s |\!\rangle, \vec{S} \in [\vec{D}']_R, s =_L s_2' |\!\} \\
&\qquad (ii)\ \ \vec{C}'\ R\ \vec{D}', s_1' =_L s_2'
\end{aligned}$$

We define a scheduler under which scheduler-specific bisimulation coincides with strong (and scheduler-independent) bisimulation. The scheduler *gen* is defined as follows:

$$gen(H, l)i = \frac{2^i}{2^n - 1}$$

where $n = live(H)$ and $i = 0, \ldots, n-1$. Note $\forall H, l.\ \sum_{i=0,\ldots,n-1} gen(H, l)i = \sum_{i=0,\ldots,n-1} \frac{2^i}{2^n-1} = 1$. The key property of the scheduler is that for any $X, Y \subseteq \{0, \ldots, n-1\}$ the equality $\sum_{k\in X} \frac{2^{i_k}}{2^n-1} = \sum_{k\in Y} \frac{2^{i_k}}{2^n-1}$ holds iff $X = Y$. That is true because the numerators in the fractions are essentially decimal encodings for binary sequences that correspond to the selection of numbers from $\{0, \ldots, n-1\}$ to X and Y. The encodings are equal iff the sequences are equal, i.e, $X = Y$.

Define $\vec{C}\lfloor_X$ for a thread pool $\vec{C}$ and a subset X of the set $\left\{0, \ldots, |\vec{C}| - 1\right\}$ to be the thread pool obtained from $\vec{C}$ by erasing the threads that are not indexed in X.

Proposition 4 $\sim_L^{gen} \subseteq \approxeq_L$

Proof. In order to show $\sim_L^{gen} \subseteq \approxeq_L$, let us construct a relation R such that $\sim_L^{gen} \subseteq R$ and R has the property from Definition 6 that guarantees that $R \subseteq \approxeq_L$. Define

$$R = \left\{ (\vec{C}\lfloor_X, \vec{D}\lfloor_X) \mid X \subseteq \left\{0, \ldots, |\vec{C}| - 1\right\}, \vec{C} \sim_L^{gen} \vec{D} \right\}$$

First, $\sim_L^{gen} \subseteq R$ (setting $X = \left\{0, \ldots, |\vec{C}| - 1\right\}$). Let us check whether R satisfies the condition from Definition 6. Indeed, R is a symmetric relation on thread pools of equal size. Suppose $\vec{C}\lfloor_X R\ \vec{D}\lfloor_X$ for some

$X \subseteq \left\{0, \ldots, |\vec{C}|-1\right\}$. Suppose that for some s_1, s_2 ($s_1 =_L s_2$) and some $i \in X$ we have $\langle\!| C_i, s_1 |\!\rangle \twoheadrightarrow \langle\!| \vec{E}, s_1' |\!\rangle$. We need to find $\vec{F}, s_2'$ such that $\langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{F}, s_2' |\!\rangle, \vec{E}\ R\ \vec{F}, s_1' =_L s_2'$. Let $\vec{C'} = \langle C_0 \ldots C_{i-1} \vec{E} C_{i+1} \ldots C_{n-1}\rangle$. Then $\langle\!| \vec{C}, s_1 |\!\rangle \to \langle\!| \vec{C'}, s_1' |\!\rangle$. Because $\vec{C} \sim_L^{gen} \vec{D}$ we have:

$$
\begin{aligned}
&\exists \vec{D'}, s_2'. \langle\!| \vec{D}, s_2 |\!\rangle \to \langle\!| \vec{D'}, s_2' |\!\rangle \\
&\quad (i) \sum \{\!| p \mid \langle\!| \vec{C}, s_1 |\!\rangle \to_p \langle\!| \vec{S}, s |\!\rangle, \vec{S} \in [\vec{C'}]_R, s =_L s_1' |\!\} = \\
&\qquad \sum \{\!| p \mid \langle\!| \vec{D}, s_2 |\!\rangle \to_p \langle\!| \vec{S}, s |\!\rangle, \vec{S} \in [\vec{D'}]_R, s =_L s_2' |\!\} \\
&\quad (ii)\ \ \vec{C'} \sim_L^{gen} \vec{D'}, s_1' =_L s_2',
\end{aligned}
$$

The probability equality from (i) has the form $\sum_{k\in X} \frac{2^{i_k}}{2^n-1} = \sum_{k\in Y} \frac{2^{i_k}}{2^n-1}$ for some $X, Y \subseteq \{0, \ldots, n-1\}$ and $n = |\vec{C}|$. The sets X and Y correspond to different choices of the scheduler that takes the two configurations to the same equivalence classes. As we have observed, the probability equality holds iff $X = Y$. Thus, $\exists \vec{F}.\langle\!| D_i, s_2 |\!\rangle \twoheadrightarrow \langle\!| \vec{F}, s_2' |\!\rangle$ and $\vec{D'}$ can be written as $\vec{D'} = \langle D_0 \ldots D_{i-1} \vec{F} D_{i+1} \ldots D_{n-1}\rangle$. Due to $\vec{C'} \sim_L^{gen} \vec{D'}$ we have $\vec{E}\ R\ \vec{F}$ (i.e., $\vec{C'}\lfloor_{\{i\}} R\ \vec{D'}\lfloor_{\{i\}}$) which together with $s_1' =_L s_2'$ finishes the proof. □

Corollary 1 *Under the assumption that low-bisimilar programs are required to have the same number of threads, we have* $\sim_L^{gen} = \approx_L = \approxeq_L$.

Proof. We have the inclusions $\approxeq_L \subseteq \approx_L \subseteq \sim_L^{gen}$ (cf. Proposition 3). By Proposition 4, $\sim_L^{gen} \subseteq \approxeq_L$. □

3.5 Hook-up Properties

In this section, we investigate the hook-up properties of strong security. Strong bisimulation is constructed to be suitable for compositional reasoning and yet to be general enough to be able to prove security of known analyses. The type-based program analyses that we have considered all satisfy the strongest notion of security. For the rest of the paper "secure" will mean strongly secure.

In the standard security terminology, the *hook-up* property [14] says that if two programs are secure then their composition (parallel, sequential or some other kind) is secure is well. We will show what kinds of composition are allowed. These properties are important since they are the key to the utility of the specification for the purpose of proving the correctness of syntax-directed program analyses.

The low-bisimulation is not a congruence. (Take, for example, any ground context that is insecure.) The low-bisimulation is preserved by *secure contexts*, i.e., contexts built with secure components. Let $[\vec{\bullet}]$ be a hole for a command vector and $[\bullet]$ be a hole for a singleton command. A context $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is secure iff it has one of these forms:

$$\begin{aligned}\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \\ & \mid [\bullet_1];[\bullet_2] \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low }) \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low }) \mid \mathsf{fork}([\bullet_1][\vec{\bullet_2}]) \mid \langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle\end{aligned}$$

where a (boolean or arithmetic) expression Exp is defined to be *low* iff $\forall s_1 =_L s_2. \exists n. \langle Exp, s_1 \rangle \downarrow n \ \&\ \langle Exp, s_2 \rangle \downarrow n$. Otherwise, the expression is *high*.

Theorem 1 (Secure congruence) *If $\vec{C_1} \approxeq_L \vec{C_1'}$, $\vec{C_2} \approxeq_L \vec{C_2'}$ and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then it is the case that $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] = \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$ provided $\vec{C_1} \approxeq_L \vec{C_2}$.*

Proof. Cases on $\mathbb{C}$. Consider first the single-threaded case $\mathbb{C}[\vec{C_1}, \vec{C_2}] = C$ and $\mathbb{C}[\vec{C_1'}, \vec{C_2'}] = C'$. Let us start off with the cases $\mathsf{skip}, x := e, \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B : low), \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B : high)$ and $\mathsf{fork}([\bullet_1][\vec{\bullet_2}])$.

In the proof below, let us make use of the so-called "up-to" technique for proving two programs $\vec{I}$ and $\vec{J}$ bisimilar. In such a technique, one constructs a relation $R \in Rel(\vec{Com})$ such that $\vec{I}\ R\ \vec{J}$ and $R \subseteq F_{\approxeq_L}(R \cup \approxeq_L)$ where $F_{\approxeq_L}(\cdot)$ is the corresponding function on relations such that R is a strong low-bisimulation whenever R is a subset of $F(R)$ (c.f. Appendix 3.A).

The standard nondeterministic bisimulation theory guarantees us then that $R \subseteq \approxeq_L$ and thereby $\vec{I} \approxeq_L \vec{J}$. It is indeed enough to have the singleton relation $R = \{(C, C')\}$ for a proof of the first cases in question. Let us inspect these cases of $\mathbb{C}$:

skip

Start with s_1 and s_2 ($s_1 =_L s_2$). With either state, the computation of skip terminates in one step with the same states. So, the final states are low-equal as well.

$l := Exp$ (*Exp* is low)

Since *Exp* is a low expression, the computation of the command with different initial states s_1 and s_2 ($s_1 =_L s_2$) terminates in one step with the same states that are still low-equal.

$h := Exp$

Exp can be arbitrary. Computing the command will not change the low component of the states.

if B **then** $[\bullet_1]$ **else** $[\bullet_2]$ $(B : low)$

Start with $\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s_1|\!\rangle$ and $\langle\!|\mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2', s_2|\!\rangle$ ($s_1 =_L s_2$). B has the same value in either state. Thus, one step of computation will lead to the configuration with bisimilar commands (C_1 and C_1', or C_2 and C_2') and low-equal states. Further, we know $C_1 \approxeq_L C_1'$ and $C_2 \approxeq_L C_2'$.

if B **then** $[\bullet_1]$ **else** $[\bullet_2]$ $(B : high)$

Start with s_1 and s_2 ($s_1 =_L s_2$). Here the value of B might differ for s_1 and s_2, but since $C_i \approxeq_L C_j'$ for $i, j = 1, 2$ one step of computation ends up in bisimilar commands and and low-equal states in this case as well.

fork$([\bullet_1][\vec{\bullet_2}])$

Trivial, as computation does not depend on h.

We have three cases left:

$[\bullet_1]; [\bullet_2]$

C_1 (and C_1') might spawn new processes, so we need to have all possible bisimilar postfix vectors after C_2 (C_2') in R. $R = \{((E; C_2)\vec{G}, (E'; C_2')\vec{G'}) \mid E \approxeq_L E', \vec{G} \approxeq_L \vec{G'}\}$. Then $R \subseteq F_{\approxeq_L}(R \cup \approxeq_L)$.

while B **do** $[\bullet_1]$ (B is low)

Choose R to be the relation $\{((E; \mathsf{while}\ B\ \mathsf{do}\ C_1)\vec{G}, (E'; \mathsf{while}\ B\ \mathsf{do}\ C_1')\vec{G'}) \mid E \approxeq_L E', \vec{G} \approxeq_L \vec{G'}\}$. Start with s_1 and s_2 ($s_1 =_L s_2$). B has the same value in either state. Thus, one step of computation will lead both configurations either to the termination of the loop (with low-equal states) or to configurations with commands $C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1$ and $C_1'; \mathsf{while}\ B\ \mathsf{do}\ C_1'$ and low-equal states. C_1 and C_1' might also spawn new processes.

$\langle[\vec{\bullet_1}][\vec{\bullet_2}]\rangle$

Finally, the case of multiple threads. Take $R = \{(\vec{E_1}\vec{E_2}, \vec{G_1}\vec{G_2}) | \vec{E_i} \approxeq_L \vec{G_i}, i = 1, 2\}$. Suppose $\vec{E_i} = \langle E_i^0 \dots E_i^{n_i - 1}\rangle$ and $\vec{G_i}' = \langle G_i^0 \dots G_i^{m_i - 1}\rangle$ for $i = 1, 2$. Let us prove $R \subseteq F_{\approxeq_L}(\approxeq_L)$ by showing:

$$
\begin{aligned}
&(i) |\vec{E_1}\vec{E_2}| = |\vec{G_1}\vec{G_2}| \\
&(ii)\, \forall s_1 =_L s_2 \forall i, j. \langle\!| E_i^j, s_1 |\!\rangle \rightharpoonup \langle\!| \vec{E}, s_1' |\!\rangle \Longrightarrow \\
&\qquad\qquad \exists \vec{G}, s_2'. \langle\!| G_i^j, s_2 |\!\rangle \rightharpoonup \langle\!| \vec{G}, s_2' |\!\rangle, \\
&\qquad\qquad\qquad \vec{E} \approxeq_L \vec{G}, s_1' =_L s_2'.
\end{aligned}
$$

The (i)-item is straightforward, as $|\vec{E_i}| = |\vec{G_i}|$ for $i = 1, 2$. The (ii)-item follows from the two separate (ii)-items (when i is fixed to be either 1 or 2) which come from the unwinding of the definition of the strong bisimulation for each of the $\vec{E_i} \approxeq_L \vec{G_i}$. □

An immediate corollary is the hook-up property result.

Corollary 2 (Hook-up) *If $\vec{C_1}, \vec{C_2}$ are secure and $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is a secure context, then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. If $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] =$ if B then $[\bullet_1]$ else $[\bullet_2]$ (B is high), then $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure provided $\vec{C_1} \approxeq_L \vec{C_2}$.*

Proof. Observe that $\vec{C_1}, \vec{C_2}$ are secure implies $\vec{C_i} \approxeq_L \vec{C_i}$ for $i = 1, 2$. Thus by the security congruence theorem $\mathbb{C}[\vec{C_1}, \vec{C_2}] \approxeq_L \mathbb{C}[\vec{C_1}, \vec{C_2}]$, i.e., $\mathbb{C}[\vec{C_1}, \vec{C_2}]$ is secure. □

3.6 Proving Analyses Secure

The compositionality of the security condition can be fruitfully exploited when proving the correctness of various compositional program analyses. In this section we develop one such analysis.

A popular form for presenting program analysis is as a nonstandard type system – as exemplified in the present context by Volpano, Smith and Irvine's type system [21] expressing Denning's flow analysis [6]. The type system was extended to a statically-threaded language with a uniform scheduler in [20]. The style of such type systems is to give security types to expressions (*high* or *low*) and commands (*high cmd* or *low cmd*) and make sure that these types do not mismatch when composing the commands.

The hook-up properties of Corollary 2 are sufficient to make the correctness proof of Volpano and Smith's analysis extremely simple – and the proof thereby includes correctness with respect to, e.g., round-robin schedulers and schedulers using dynamic priorities (via dependence on low "priority" variables in the program).

As an example we will consider a useful improvement of Volpano and Smith's system (for our slightly richer language) based on ideas from a recent analysis developed by Johan Agat [1] which shows how timing channels can be eliminated from sequential programs by a combination of typing and code transformation.

The improvement in question concerns the treatment of the security of conditional expressions in the case when the boolean depends on high variables. Consider a command of the form $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where $B : high$. Volpano and Smith's analysis requires that (i) no low assignments are made within C_1 or C_2, (ii) no while loops occur within the branches, and (iii) the whole conditional is executed atomically (i.e., without interruption by another thread.) The last of these points is enforced in the Volpano-Smith system by an additional language construct called protect. A disadvantage of using protect is that it is not a typical synchronisation primitive of modern concurrent languages. The protect statement also appears prohibitively expensive to implement using, e.g., locks, since in order to block all threads during a protect, *every* atomic command C in the program must be transformed into $\mathsf{getlock}; C; \mathsf{releaselock}$. Smith [17] has suggested implementing the protect by using thread priorities. The idea is to give a thread that executes a protected section the highest priority such that all other threads are blocked in the meantime. However, neither implementation can be generalised to close external timing leaks.

We will instead use a technique due to Agat to avoid the need for protect statements, and obtain an analysis that is more permissive than Volpano and Smith's analysis.

3.6.1 Agat's Approach

As Agat observed in the sequential case, each of the restrictions on a high conditional can be lifted to some degree by observing that it is sufficient for C_1 and C_2 to be low-bisimilar. The hook-up result shows that this is true in the present setting also.

Agat's approach is to *transform* the branches of the conditional so that they become bisimilar in a manner that is easily checked by a syntax directed set of rules. The essence of the transformation is to pad the branches of the conditionals. In our case this padding will add skip instructions and possibly dummy forks. Agat's aim was the elimination of *external* timing attacks; we can also argue this here, although we simply focus on the internal timing leaks.

In the rest of the section we present the combined analysis and transformation system, prove its correctness with respect to strong security and show how this approach eliminates the need for protect's.

3.6.2 The Type System

The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct and indirect insecure flows then it might be accepted by the system and transformed into a program free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$, where $\vec{C}$ is a program, C' is the result of its transformation and $\vec{Sl}$ is the "type" of $\vec{C'}$. The types of programs are their *low slices*. A low slice is essentially a copy of a secure program in which assignments to high variables have been replaced by skip's. The slice Sl does not have any occurrences of h, but has the same structure as C' and models the timing behaviour of C', as observable by other threads.

The typing and transformation rules are presented in Figure 3.4. The variables h and l have the types *high* and *low* respectively. Integer literals n may be considered as either *high* or *low*. An arbitrary expression Exp may be considered as *high*. In all but If_{high} rules, the transformed program is constructed compositionally using the same constructs as the original program. The information about the low slice of the new program is recorded in the typing. Command skip is its own low slice and therefore its own type. The rule Assign_{low} prevents direct insecure information flows – the assignment $l := h$ is not typable. The rule Assign_{high} types an assignment to the high variable with the low slice skip. The rules Arithm_{low}, Seq, If_{low}, Par and Fork propagate types compositionally. The guard of the while-loop in the rule While has to be low in order to prevent the timing (and nontermination) flow from the loop's guard.

The rule If_{high} prevents indirect insecure flows and timing flows. Let $al(C)$ be a predicate returning True whenever there is a syntactic occurrence of an

$$[\mathrm{Var}]\quad h : high \qquad l : low$$

$$[\mathrm{Exp}]\quad n : \tau \qquad Exp : high$$

$$[\mathrm{Arithm}_{low}]\quad \frac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$$

$$[\mathrm{Skip}]\quad \mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}$$

$$[\mathrm{Assign}_{low}]\quad \frac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp}$$

$$[\mathrm{Assign}_{high}]\quad h := Exp \hookrightarrow h := Exp : \mathsf{skip}$$

$$[\mathrm{Seq}]\quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$$

$$[\mathrm{While}]\quad \frac{B : low \quad C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$$

$$[\mathrm{Par}]\quad \frac{C_0 \hookrightarrow C_0' : Sl_0 \ \ldots\ C_{n-1} \hookrightarrow C_{n-1}' : Sl_{n-1}}{\langle C_0 \ldots C_{n-1}\rangle \hookrightarrow \langle C_0' \ldots C_{n-1}'\rangle : \langle Sl_0 \ldots Sl_{n-1}\rangle}$$

$$[\mathrm{Fork}]\quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad \vec{C_2} \hookrightarrow \vec{C_2'} : \vec{Sl_2}}{\mathsf{fork}(C_1\vec{C_2}) \hookrightarrow \mathsf{fork}(C_1'\vec{C_2'}) : \mathsf{fork}(Sl_1\vec{Sl_2})}$$

$$[\mathrm{If}_{low}]\quad \frac{B : low \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2' : \mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2}$$

$$[\mathrm{If}_{high}]\quad \frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$$

Figure 3.4: Transformation to eliminate timing leaks

assignment to low variable l in the command C and returning False otherwise. The condition $al(Sl_1) = al(Sl_2) =$ False prevents the indirect leaks.

The interesting rule is the case when the guard of an if command is high. In this case both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure it is also necessary that the two branches be low-bisimilar. This is achieved by cross-copying the low slice of one branch into the other. The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip corresponding to the time tick for the guard inspection.

3.6.3 An Example Code Transformation

Let us give a simple example of the application of the transformation system. The example is based on one in [1], and is used to illustrate the effect of the transformation. The heart of RSA encryption is the computation of $a^b \ mod\ n$, where a is an integer representing the plaintext and b is the integer encryption key. To efficiently compute $a^b \ mod\ n$, the modular exponentiation algorithm can be used, but, as shown by Kocher [12], a careless implementation will leak b through timing. On the left in Figure 3.5, we give an example of such an implementation. (Assume we have an if without an else with the obvious semantics.) All variables except for i and k in the program are high. The result of computation is stored in d.

We represent b as a $k+1$ element long array of secret booleans, with the most significant bit of b at $b[k]$. The initial program is not secure since the two branches of the if have different timing behaviours (they are not low-bisimilar). For some schedulers, a (secure) thread running in parallel with this code may compute different low outputs depending on the values of the b_i (we omit a concrete example for reasons of space). The transformation algorithm presented in Figure 3.4 (with the minor extension necessary to handle arrays with a low index) will close these timing leaks by transforming the program into the one given on the right in Figure 3.5. The security of this padded program relies on atomic execution of the assignment operators. Finally, the low slice of the padded program is shown in the bottom of Figure 3.5.

Original program $(i, k : low)$

$c := 0;\ d := 1;\ i := k;$
while $i \geq 0$
  do $i := i - 1;$
    $c := 2 \cdot c;$
    $d := (d \cdot d)\ mod\ n;$
    if $b_i = 1$
      then $c := c + 1;$
        $d := (d \cdot a)\ mod\ n$

$\hookrightarrow$

Transformed program $(i, k : low)$

$c := 0;\ d := 1;\ i := k;$
while $i \geq 0$
  do $i := i - 1;$
    $c := 2 \cdot c;$
    $d := (d \cdot d)\ mod\ n;$
    if $b_i = 1$
      then $c := c + 1;$
        $d := (d \cdot a)\ mod\ n$
      else skip;
        skip

:

Low slice

skip; skip; $i := k;$
while $i \geq 0$
  do $i := i - 1;$
    skip;
    skip;
    skip;
    skip;
    skip

Figure 3.5: Producing a secure modular exponentiation thread

3.6.4 Correctness of the Analysis

Let us prove the security of Agat's analysis extended to the multi-threaded language. The following theorem is a straightforward use of the secure congruence properties of the security condition proved in Theorem 1.

Theorem 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \approxeq_L \vec{Sl}$

Proof. Proceed by structural induction on the thread pool $\vec{C'}$. Let us consider the cases by which $\vec{C'} : \vec{Sl}$. In the cases skip : skip, $l := Exp : l := Exp$ $(Exp : low)$ and $h := Exp$: skip the program $\vec{C'}$ is low-bisimilar to it's low slice. The cases $C_1; C_2$, $\mathsf{while}\ B\ \mathsf{do}\ C_1$: $\mathsf{while}\ B\ \mathsf{do}\ Sl'$ $(B : low)$, $\langle C_0 \ldots C_{n-1}\rangle$: $\langle Sl_0 \ldots Sl_{n-1}\rangle$, $\mathsf{fork}(C_1\vec{C_2})$ and $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$: $\mathsf{if}\ B\ \mathsf{then}\ Sl_1\ \mathsf{else}\ Sl_2$ $(B : low)$ provide secure contexts $[\bullet_1]; [\bullet_2]$, $\mathsf{while}\ B\ \mathsf{do}\ [\bullet]$, $\langle[\bullet_0] \ldots [\bullet_{n-1}]\rangle$, $\mathsf{fork}([\bullet_1][\vec{\bullet_2}])$ and $\mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]$ $(B : low)$ respectively. By the the induction hypothesis the sub-commands have low-bisimilar low slices, so the conclusion follows by the hook-up property.

The last case is $\mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2'$: $\mathsf{skip}; Sl_1; Sl_2$. In order to apply the special if-on-high case of the hook-up property we need to show $C_1'; Sl_2 \approxeq_L Sl_1; C_2'$. By the induction hypothesis, $C_i' \approxeq_L Sl_i$ $(i = 1, 2)$. Now since $[\bullet_1]; [\bullet_2]$ is a secure context, we have that $C_1'; Sl_2 \approxeq_L Sl_1; C_2'$. Finally, this is sufficient to argue that $\{(\mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2', \mathsf{skip}; Sl_1; Sl_2)\}$ is a strong bisimulation up to $\approxeq_L$ (which requires just a case analysis on the outcome of B). □

Corollary 3 (Security of the Analysis) *$\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'}$ is secure.*

Proof. Theorem 2 gives $\vec{C'} \approxeq_L \vec{Sl}$. The symmetry and transitivity of $\approxeq_L$ entails $\vec{C'} \approxeq_L \vec{C'}$, i.e., $\vec{C'}$ is secure. □

3.6.5 Soundness of the Transformation

We have shown that the result of the transformation is secure, but what of its relation to the original program? We argue below that the transformed program is a possibilistic refinement of the original program, and that under certain additional assumptions, that it is equivalent.

Clearly, the padding introduced by the transformation can change the timing and thereby the program's probabilistic behaviour, but otherwise it is just additional "stuttering". A more serious problem is that it might introduce extra nontermination, in the case that a nonterminating loop is cross-copied into a branch that would otherwise have terminated.

To make the first point precise, let us define a *possibilistic (bi)simulation* on programs. To obtain a possibilistic transition relation we can simply erase the probability from the uniform scheduler semantics defined in Section 3.2. Let $cfg \to^\star cfg'$ hold iff $cfg = cfg'$ or $cfg \to cfg'$.

Definition 8 *Define the possibilistic simulation $\preceq$ (resp., bisimulation $\simeq$) to be the the union of all (resp. symmetric) relations R on thread pools such that whenever $\vec{C}\ R\ \vec{D}$ then for all s, s', C', there exists a D' such that*

$$\langle\!|\vec{C}, s|\!\rangle \to \langle\!|\vec{C'}, s'|\!\rangle \Longrightarrow \langle\!|\vec{D}, s|\!\rangle \to^\star \langle\!|\vec{D'}, s'|\!\rangle \text{ and } \vec{C'}\ R\ \vec{D'}.$$

Intuitively, $\vec{C} \preceq \vec{D}$ holds whenever $\vec{C}$ performs a (maybe infinitely) slowed down version of the computation of $\vec{D}$.

Proposition 5 *$\preceq$ is reflexive, transitive and preserved by all contexts (not necessarily secure), i.e., $\preceq$ is a precongruence.*

Proposition 6 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \preceq \vec{C}$.

Proof. Induction on the height of the transformation derivation. The cases Skip, Assign$_{low}$, Assign$_{high}$ follow from the reflexivity of $\preceq$. The cases Seq, While, Par, Fork and If$_{low}$ are implied by the induction hypotheses and the precongruence of $\preceq$.

The remaining case is If$_{high}$, i.e., $\mathsf{if}\ B\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2$. By induction we have $C_1' \preceq C_1$ and $C_2' \preceq C_2$. The side condition guarantees that there are no assignments in the slices Sl_1 and Sl_2 of the branches C_1 and C_2 of the if. Thus Sl_1 and Sl_2 contain only dummy computation without changing the state. The insertion of such a computation might only slow down the computation. Thus, $C_1'; Sl_2 \preceq C_1$ and $Sl_1; C_2' \preceq C_2$. The precongruence of $\preceq$ yields $C' \preceq C$. □

3.6.6 Elimination of the protect

Let us define the notion of a protected program from [20]. A program $\vec{C}$ is *protected* iff any conditional on the high variable is wrapped in a protect term. A protect cannot have other protect's or while's within its scope of protection. We are now ready to sketch the protect elimination result.

Theorem 3 (Protect elimination) *If $\vec{C_p}$ is well-typed and protected in Volpano-Smith's type system then $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$ where $\vec{C}$ is the result of erasing the protect's from $\vec{C_p}$, for some secure $\vec{C'}$ and some $\vec{Sl}$ such that $\vec{C} \simeq \vec{C'}$.*

Proof. (Sketch) Although there are only two types of commands in Volpano-Smith's system (*low cmd* and *high cmd*), the type inference is still performed compositionally and similarly to the transformation in Figure 3.4. Hence we are able to type all programs that are typable in Volpano-Smith's system. Whenever $\vec{C_p}$ is well-typed then $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl}$. By Proposition 6 we immediately have $\vec{C'} \preceq \vec{C}$.

The program $\vec{C}$ has no occurrences of fork (no dynamic thread creation in Volpano-Smith's language). Therefore, the padding transformation cannot introduce new fork's. While-loops cannot be introduced since the only place where the transformation generates new code is the if-on-high case, and since such terms are protected they contain no occurrences of while. What the transformation might introduce in the branches of a high if is a sequential composition of low if's and skip's (these are the only possibilities for the low slices of the branches). The insertion of these commands does not affect the possibilistic semantics or the termination properties of the initial program (we omit the routine proofs of these laws). Using this intuition, the rest of the proof of $\vec{C'} \simeq \vec{C}$ follows similarly to Proposition 6. □

3.7 Conclusions

We have developed extensional semantics-based specifications of secure information flow for multi-threaded programs. The specifications capture probabilistic covert channels that arise from the scheduling of concurrent threads. We have argued that under the worst case assumption the security condition should be scheduler-independent.

By defining security specifications as noninterference based on a probabilistic bisimulation, we have achieved compositional reasoning (build secure programs from secure components) and precision over approaches proposed in the literature, without the mathematical overheads of a denotational approach using, e.g., probabilistic powerdomains. Proving type-system based analyses is simple since such analyses are usually compositional as well.

In this paper, we have been considering unsynchronised multi-threaded concurrency with dynamic thread creation. Future work must include the study of synchronisation primitives.

Another simplification is in our treatment of I/O. Implicitly, we permit the attacker (the external supplier of the program) to observer intermediate low outputs, and even provide the initial low input, but not to selectively provide inputs to the program as it runs. Blocking input gives the attacker(s) the ability to externally control the timing behaviour of threads. To study the essence of these problems, it might be appropriate to extend Focardi and Gorrieri's process-calculus-based study [8]) to the probabilistic case. This should also enable a more rigorous connection to be made to Gray's P-Restrictiveness.

3.A Observations on Partial Probabilistic Bisimulation

This appendix presents some observations on properties of partial probabilistic bisimulations. The union of all partial probabilistic bisimulations can itself be shown to be a partial probabilistic bisimulation. This follows from a standard Tarski fixed-point argument (although to our knowledge this argument has not been used elsewhere for probabilistic bisimulation). The key is to note that pers on a given set form a complete lattice (with meet given by set-intersection, and join given by the transitive closure of the set-union), and that the definition can be phrased in terms of a continuous functional.

Recall that an unlabelled probabilistic transition system is a set of states $\mathbb{S}$ and a set of transitions of the form $\mathbb{s} \to \mu$ where $\mathbb{s} \in \mathbb{S}$ and $\mu \in \mathcal{D}(\mathbb{S})$. Let us restrict our attention to probabilistic transition systems such that $\mathbb{s} \to \mu$ and $\mathbb{s} \to \mu'$ implies $\mu = \mu'$ (which is clearly the case for the transition system that corresponds to the probabilistic semantics of the multi-threaded language).

Fix such a probabilistic transition system. Let F denote the function from pers to pers (over $\mathbb{S}$) given by:

$$\mathbb{s}\ F(R)\ \mathbb{s}' \iff \begin{array}{l} (i)\ \forall S \in \mathbb{S}/R.\, \mathbb{s} \to_p S \Longrightarrow \mathbb{s}' \to_p S \\ (ii)\ \mathbb{s} \to_0 \mathbb{S} \setminus dom(R) \end{array}$$

It is straightforward to check that $F(R)$ is indeed transitive and symmetric. The following two propositions state the monotonicity and co-continuity of F.

Proposition 7 *Suppose $P, Q \in Rel(\mathbb{S})$ are pers. Then $P \subseteq Q \Longrightarrow F(P) \subseteq F(Q)$.*

Proof. Given $\mathbb{s}\ F(P)\ \mathbb{s}'$, we need to show $\mathbb{s}\ F(Q)\ \mathbb{s}'$. Suppose $\mathbb{s} \to_q S_Q$ for some $S_Q \in \mathbb{S}/Q$. Since $P \subseteq Q$, we can partition S_Q into $\cup_i S_P^i \cup S$, where the S_P^i are equivalence classes of P, and S is a subset of $\mathbb{S} \setminus dom(P)$. Suppose $\mathbb{s} \to_{p_i} S_P^i$. Note $\mathbb{s} \to_0 S$. By $\mathbb{s}\ F(P)\ \mathbb{s}'$ we have $\mathbb{s}' \to_{p_i} S_P^i$ and $\mathbb{s}' \to_0 S$. This implies $\mathbb{s}' \to_q S_Q$ by simple summing. Conclude $\mathbb{s}\ F(Q)\ \mathbb{s}'$. □

Proposition 8 *For a nonincreasing ω-chain of pers $R_0 \supseteq \ldots \supseteq R_i \supseteq \ldots$, F preserves co-limits, i.e.,*

$$F(\cap_{i<\omega} R_i) = \cap_{i<\omega} F(R_i).$$

Proof. Since $\cap_{i<\omega} R_i \subseteq R_j$ for any j, apply the monotonicity of F to get, for all j, $F(\cap_{i<\omega} R_i) \subseteq F(R_j)$ which implies the $\subseteq$ part of the set equality. For the reverse inclusion, we need to show that if for some $\mathbb{s}$ and $\mathbb{s}'$ we have $\mathbb{s}\ (\cap_{i<\omega} F(R_i))\ \mathbb{s}'$ then $\mathbb{s}\ F(\cap_{i<\omega} R_i)\ \mathbb{s}'$. The assumption that $\mathbb{s}\ (\cap_{i<\omega} F(R_i))\ \mathbb{s}'$ entails $\mathbb{s}\ F(R_i)\ \mathbb{s}'$ for all i, which can be rewritten as

$$(i)\ \forall S \in \mathbb{S}/R_i.\, \mathbb{s} \to_p S \Longrightarrow \mathbb{s}' \to_p S$$
$$(ii)\ \mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(R_i)$$

for all i. Suppose now for some $S \in \mathbb{S}/\cap_{i<\omega} R_i$ we have $\mathbb{s} \to_p S$. There exists a sequence of pers $\{S_i\}_{i<\omega}$ such that $S_i = \mathbb{S}/R_i$ and $S = \cap_{i<\omega} S_i$. Note that $S_0 \supseteq \ldots \supseteq S_i \supseteq \ldots \supseteq S$. Consider the sequence of nonincreasing probabilities $\{p_i\}_{i<\omega}$ such that $\mathbb{s} \to_{p_i} S_i$. The fact that the summing operation over countable sets is continuous guarantees $p = \inf\{p_i\}_{i<\omega}$. Thus, $\mathbb{s}' \to_{p_i} S_i$ for all i, implying $\mathbb{s}' \to_p S$. Analogously, $\forall i.\, \mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(R_i)$ implies $\mathbb{s} \to_0 \mathbb{S} \setminus \mathit{dom}(\cap_{i<\omega} R_i)$ which concludes the proof of $\mathbb{s}\ F(\cap_{i<\omega} R_i)\ \mathbb{s}'$. □

Let us define $\sim\, = \cup\{R \mid R \text{ is a per}, R \subseteq F(R)\}$. We are now in a position to establish the fixed-point result.

Proposition 9 (Fixed point) *The relation $\sim$ is the greatest fixed point of F in the lattice of pers. It can be alternatively represented by $\sim\, = \cap_{i<\omega} F^i(\mathbb{S} \times \mathbb{S})$.*

Proof. The proof is a standard argument, by appeal to the Knaster-Tarski fixed-point theorem (see, e.g., [5]). The alternative representation statement holds due to the co-continuity of F. Observe that since, by induction, $F^i(\mathbb{S} \times \mathbb{S})$ is a per for any i then so is $\sim$ itself. □

When proving a per R to be a partial probabilistic bisimulation (or $R \subseteq \sim$) one can use the the following analogue of the standard "up-to" technique.

Proposition 10 *Given a per $R \in \mathit{Rel}(\mathbb{S})$,*

$$R \subseteq F((R \cup \sim)^+) \Longrightarrow R \subseteq \sim.$$

Proof. Assume $R \subseteq F((R \cup \sim)^+)$. Note that $\sim\, \subseteq F(\sim) \subseteq F((R \cup \sim)^+)$ due to the fixed-point proposition and F's monotonicity. Thus, $R \cup \sim\, \subseteq F((R \cup \sim)^+)$. Since the relation on the right-hand side is a per, it must be true that $(R \cup \sim)^+ \subseteq F((R \cup \sim)^+)$. By the definition of $\sim$, we have $(R \cup \sim)^+ \subseteq \sim$ which, in particular, implies $R \subseteq \sim$. □

References

[1] J. Agat. Transforming out timing leaks. In *POPL'00: The 27:th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 40–53. ACM, January 2000.

[2] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[3] C. Baier and M. Z. Kwiatkowska. Domain equations for probabilistic processes. *Mathematical Structures in Computer Science*, June 1999.

[4] J.-P. Banatre, C. Bryce, and D. Le Metayer. An approach to information security in distributed systems. In *Proceedings of the 5th IEEE International Workshop on Future Trends in Distributed Computing Systems*, pages 384–394, 1995.

[5] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[6] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[8] R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *J. Computer Security*, 3(1):5–33, 1994/1995.

[9] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, April 1982.

[10] J. Gray III. Probabilistic interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, California, May 1990.

[11] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 1998.

[12] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer-Verlag, 1996.

[13] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[14] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. 1987.

[15] J. McLean. The specification and modeling of computer security. *Computer*, 23(1), January 1990.

[16] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1), 2001.

[17] G. Smith. Personal communication. 2000.

[18] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 January 1998.

[19] P. Syverson and J. Gray III. The epistemic representation of information flow security in probabilistic systems. In *Proc. 8th IEEE Computer Security Foundations Workshop*, 1995.

[20] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Chapter 4

The Impact of Synchronisation on Secure Information Flow in Concurrent Programs

Synchronisation is fundamental to concurrent programs. This chapter investigates the security of information flow in multi-threaded programs in the presence of synchronisation. We give a small-step operational semantics for a simple shared-memory multi-threaded language with synchronisation, and present a compositional timing-sensitive bisimulation-based confidentiality specification. We propose a type-based analysis improving on previous approaches to reject potentially insecure programs.

This chapter is to appear in the *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, Akademgorodok, Novosibirsk, Russia, July 3–6, 2001, LNCS, Springer-Verlag.

4.1 Introduction

4.1.1 Motivation

This paper focuses on the problem of program *confidentiality*, i.e., determining whether a given shared-memory multi-threaded program has secure information flow. The program runs on data partitioned on *high* (private) and *low* (public) security data, although a more general lattice of security levels can be considered. The program is not trusted, and the program's low output is publicly available as well as, possibly, timing information about the program's execution. Such a program might be received over the Internet and legitimately send its low output over the Internet such that times when the program makes Internet accesses are observable.

4.1.2 Background

The problem of confidentiality for various programming languages has been investigated by many researchers including [7, 8, 6, 3, 13, 5, 14, 19, 10, 17, 1, 18, 11, 15, 2, 16]. The issue of secure information flow has become especially important with the growing popularity of mobile code and networked information systems. For distributed programming, the use of multi-threaded programming languages has become extremely popular [4]. However, in the setting of an imperative shared-memory multi-threaded language, the majority of investigations in the area of secure information flow, e.g., [17, 18, 15] do not have *synchronisation* in the language. Although the security logic of [3] does treat synchronisation primitives, there is neither a soundness proof nor a decision algorithm given for the logic. Because synchronisation is fundamental to concurrent programs, it is highly desirable to have a robust security specification and tools that aid in the design of secure programs with synchronisation.

To bridge the gap, this paper presents a compositional bisimulation-based confidentiality specification for multi-threaded programs with synchronisation and proposes a type-based analysis improving on previous approaches to reject potentially insecure programs.

4.1.3 Insecure Flows to Eliminate

Let us exemplify the types of insecure information flow that are in the focus of this paper. Suppose h is a high security variable and l is a low security one. There are several ways to leak information from h to the attacker. An example of a *direct* flow is the simple program $l := h$. An instance of an *indirect* flow through branching on a high condition is $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0$. From the timing behaviour of the program the attacker may deduce secret information. The program $l := 0; \mathsf{if}\ h = 1\ \mathsf{then}\ (\mathsf{while}\ l < \mathit{MaxInteger}\ \mathsf{do}\ l := l + 1)\ \mathsf{else}\ \mathsf{skip}$ is an instance of a *timing* leak. The program $\mathsf{if}\ h = 1\ \mathsf{then}\ (\mathsf{while}\ \mathsf{True}\ \mathsf{do}\ \mathsf{skip})\ \mathsf{else}\ \mathsf{skip}$ is a variation of the timing leak called a *termination* leak. Observing the termination of the program reveals that h was not 1. Blocking a thread can change the observable behaviour of a computation, e.g., its termination behaviour. If blocking depends on high data, then the attacker might learn secrets through the observable behaviour. Concrete examples of *synchronisation* leaks are postponed until Section 4.4 where concrete synchronisation primitives will be available.

4.1.4 Overview

The rest of the paper is organised as follows. Section 4.2 introduces the syntax and semantics of a multi-threaded language. Section 4.3 motivates and specifies a bisimulation-based security definition. Section 4.4 gives a type system for detecting secure programs and shows its correctness with respect to the security definition. Section 4.5 gives an example of secure programming with synchronisation. Section 4.6 concludes by discussing related work.

4.2 A Simple Multi-threaded Language with Synchronisation

To set up the scene, let us define the syntax and semantics of a simple multi-threaded language with dynamic thread creation and synchronisation. Semaphores are widely used as a synchronisation primitive in shared-memory languages. As pointed out by Andrews ([4], p.viii): *"Semaphores were the first high-level concurrent programming mechanism and remain one of the most important."* Semaphores are general enough to implement both *mutual exclusion* and *condition synchronisation.* In this section we introduce semaphores in the language and give a small-step operational semantics.

4.2.1 Semaphores

A *semaphore* [9] is a special variable (call it sem) that can only be manipulated by two commands: $\mathsf{wait}(sem)$ and $\mathsf{signal}(sem)$. The value of sem ranges over nonnegative integers. Initially, the value is 0 for every semaphore. The $\mathsf{wait}(sem)$ command blocks until sem is positive. Once sem is positive, it gets decremented by 1. The $\mathsf{signal}(sem)$ command increments sem by 1.

4.2.2 Semaphores by Busy Waiting

One approach to introducing semaphores in the language is to use macrodefinitions and define the synchronisation primitives by while-loop-based *busy waiting* (as in, e.g., [3]):

$$\begin{aligned} \mathsf{wait}(sem) &= (\mathsf{while}\ sem = 0\ \mathsf{do}\ \mathsf{skip}); sem := sem - 1 \\ \mathsf{signal}(sem) &= sem := sem + 1 \end{aligned}$$

Although these definitions are intuitive, there are two major problems with them. First, a waiting process *occupies the CPU with idle spinning.* Also, *delay and decrement must be a single atomic action.* Otherwise two $\mathsf{wait}(sem)$ threads might succeed when the initial value of sem is 1! Atomicity may be hard to implement in a distributed setting (not to mention that timing behaviour will spin out of control since one atomic action no longer takes one time unit). *Blocked waiting*, which commonly underlies semaphore implementations, does not have the disadvantages above. It is important that we adapt blocked waiting, since the dynamics of thread (un)blocking in a program's execution needs to be explicitly modelled due to its potential to affect the program's security.

4.2.3 Semaphores by Blocked Waiting

In order to define blocked waiting, we will use a special pool of waiting processes. The idea is that blocked processes should be sleeping (as opposed to spinning in busy waiting) until the respective signal is sent.

The syntax of the language is given in Figure 4.1. Let $C, D, E, \ldots$ range over commands Com, and let $\vec{C}$ denote a vector of commands of the form

$\langle C_1 \dots C_n \rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \dots$ range over $\vec{Com} = \cup_{n \in \mathbb{N}} Com^n$, the set of thread pools (or programs). A *state* $s \in \mathbf{St}$ is a finite mapping from variables (including special semaphore variables) to values. The set of variables is partitioned into high and low security classes. h and l will denote typical high and low variables, respectively. Define *low-equivalence* by $s_1 =_L s_2$ iff the low components of s_1 and s_2 are the same.

The small-step semantics is given by transitions between *configurations*, i.e., between pairs each containing a program and a state. The deterministic part of the semantics is defined by the transition rules in Figure 4.2. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\rightharpoonup$ transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| \langle\rangle, s' |\!\rangle$, which means termination with the final state s', or $\langle\!| C, s |\!\rangle \rightharpoonup \langle\!| C'\vec{D}, s' |\!\rangle$. Here, one step of computation that starts with a command C in a state s gives a new main thread C', a (possibly empty) vector of spawned processes $\vec{D}$ and a new state s'.

Let sem be a variable from the special set Sem of semaphore variables. The $\mathsf{wait}(sem)$ command emits the "block" label $\otimes sem$ in case the value of sem is 0. Otherwise it decreases the value of sem. The $\mathsf{signal}(sem)$ emits the "unblock" label $\odot sem$. The labels are propagated through the sequential composition to the top level.

The concurrent part of the semantics is presented in Figure 4.3. The $\rightarrow$ transitions are nondeterministic. The nondeterminism is resolved by the scheduler in a particular implementation. The scheduler might be probabilistic and history-dependent, but, aiming at scheduler-independent security, we make no assumptions about peculiarities of a particular scheduler. The $\rightarrow$ transitions have the form $\langle\!| \vec{C}, w, s |\!\rangle \rightarrow \langle\!| \vec{C'}, w', s' |\!\rangle$ where configurations are equipped with queues of waiting processes $w, w' : Sem \rightarrow \vec{Com}$. Whenever the top level receives a $\otimes sem$ signal, the blocked thread is put in the end of the FIFO queue associated with sem. If the top level receives an $\odot sem$ signal, the first thread in the FIFO gets awakened or, in case, the FIFO is empty, the value of sem gets incremented.

We can extract a simple model of the timing behaviour of multi-threaded programs from the small-step semantics. This is done by assuming that each $\rightharpoonup$ transition takes a single unit of time to execute. This approach gives only a rough approximation of the actual timing behaviour, but simple extensions are possible in order to make it sensitive to the timing behaviour of particular commands (cf. [2]).

$$C ::= \ \mathsf{skip} \mid Id := Exp \mid C_1; C_2 \mid \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$$
$$\mid \mathsf{while}\ B\ \mathsf{do}\ C \mid \mathsf{fork}(C, \vec{D}) \mid \mathsf{wait}(Sem) \mid \mathsf{signal}(Sem)$$

Figure 4.1: Command syntax

$$\langle\!| \mathsf{skip}, s |\!\rangle \rightharpoondown \langle\!| \langle\rangle, s |\!\rangle \qquad \frac{\langle\!| exp, s |\!\rangle \downarrow n}{\langle\!| x := exp, s |\!\rangle \rightharpoondown \langle\!| \langle\rangle, [x := n]s |\!\rangle}$$

$$\frac{\langle\!| C_1, s |\!\rangle \rightharpoondown \langle\!| \langle\rangle, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \rightharpoondown \langle\!| C_2, s' |\!\rangle} \qquad \frac{\langle\!| C_1, s |\!\rangle \rightharpoondown \langle\!| C_1' \vec{D}, s' |\!\rangle}{\langle\!| C_1; C_2, s |\!\rangle \rightharpoondown \langle\!| (C_1'; C_2)\vec{D}, s' |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \rightharpoondown \langle\!| C_1, s |\!\rangle} \qquad \frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, s |\!\rangle \rightharpoondown \langle\!| C_2, s |\!\rangle}$$

$$\frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{True}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \rightharpoondown \langle\!| C; \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle} \qquad \frac{\langle\!| B, s |\!\rangle \downarrow \mathsf{False}}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, s |\!\rangle \rightharpoondown \langle\!| \langle\rangle, s |\!\rangle}$$

$$\frac{\langle\!| sem, s |\!\rangle \downarrow n \quad n > 0}{\langle\!| \mathsf{wait}(sem), s |\!\rangle \rightharpoondown \langle\!| \langle\rangle, [sem := sem - 1]s |\!\rangle} \qquad \frac{\langle\!| sem, s |\!\rangle \downarrow 0}{\langle\!| \mathsf{wait}(sem), s |\!\rangle \overset{\otimes sem}{\rightharpoondown} \langle\!| \langle\rangle, s |\!\rangle}$$

$$\langle\!| \mathsf{fork}(C, \vec{D}), s |\!\rangle \rightharpoondown \langle\!| C\vec{D}, s |\!\rangle \qquad \frac{\langle\!| C_1, s |\!\rangle \overset{\alpha}{\rightharpoondown} \langle\!| C_1', s' |\!\rangle \quad \alpha \in \{\otimes sem, \odot sem\}}{\langle\!| C_1; C_2, s |\!\rangle \overset{\alpha}{\rightharpoondown} \langle\!| C_1'; C_2, s' |\!\rangle}$$

$$\langle\!| \mathsf{signal}(sem), s |\!\rangle \overset{\odot sem}{\rightharpoondown} \langle\!| \langle\rangle, s |\!\rangle$$

Figure 4.2: Small-step deterministic semantics of commands

$$\frac{\langle\!| C_i, s |\!\rangle \rightharpoondown \langle\!| \vec{C}, s' |\!\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \to \langle\!| \langle C_1 \dots C_{i-1} \vec{C} C_{i+1} \dots C_n \rangle, w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\otimes sem}{\rightharpoondown} \langle\!| C', s' |\!\rangle \qquad w_{sem} = \vec{D}}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \to \langle\!| \langle C_1 \dots C_{i-1} C_{i+1} \dots C_n \rangle, [w_{sem} := \vec{D}C']w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot sem}{\rightharpoondown} \langle\!| C', s' |\!\rangle \qquad w_{sem} = C\vec{D}}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \to \langle\!| \langle C_1 \dots C_{i-1} C' C_{i+1} \dots C_n C \rangle, [w_{sem} := \vec{D}]w, s' |\!\rangle}$$

$$\frac{\langle\!| C_i, s |\!\rangle \overset{\odot sem}{\rightharpoondown} \langle\!| C', s' |\!\rangle \qquad w_{sem} = \langle\rangle}{\langle\!| \langle C_1 \dots C_n \rangle, w, s |\!\rangle \to \langle\!| \langle C_1 \dots C_{i-1} C' C_{i+1} \dots C_n \rangle, w, [sem := sem + 1]s' |\!\rangle}$$

Figure 4.3: Concurrent semantics of thread pools

4.3 Security Specification

What is a secure program in the language we have just defined? This section focuses on the definition of confidentiality and motivates the choice of the definition.

4.3.1 Noninterference

The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, the absence of dependencies). If direct, indirect, and timing flows are considered then, intuitively, a program has the extensional *noninterference* property if *varying the high input will not change the possible low-level observations*, i.e., low inputs, low outputs and timing. This differs from intensional security which relies on particular security primitives that are only motivated by intuition rather than a mathematical justification. Many investigations have successfully pursued the extensional view, including [6, 19, 10, 17, 1, 18, 11, 15, 2, 16] for the justification of security analyses and verification techniques for different languages. We follow the extensional approach and focus on the extensional security for our language.

The main idea behind the bisimulation-based approach promoted by [15] is to define a *low-bisimulation* on commands such that the indistinguishability of the behaviours of two programs C and D for the attacker is formalised by $C \sim_L D$, where $\sim_L$ is a low-bisimulation. Such an approach is flexible in the choice of an appropriate low-bisimulation (different low-bisimulations are available for different degrees of security). For a given low-bisimulation $\sim_L$, the definition of security is simply:

$$C \text{ is secure iff } C \sim_L C$$

For the purpose of this paper we adapt a variation of the *strong low-bisimulation* [15].

Definition 1 *Define the* strong *low-bisimulation* $\cong_L$ *to be the union of all symmetric relations* R *on thread pools of equal size, such that if* $\langle C_1 \dots C_n \rangle$ R

$\langle D_1 \dots D_n \rangle$ *then*

$$\forall s_1 =_L s_2 \forall i \forall \alpha . \langle\!| C_i, s_1 |\!\rangle \overset{\alpha}{\rightharpoonup} \langle\!| \vec{C}', s_1' |\!\rangle \Longrightarrow$$
$$\exists \vec{D}', s_2' . \langle\!| D_i, s_2 |\!\rangle \overset{\alpha}{\rightharpoonup} \langle\!| \vec{D}', s_2' |\!\rangle, \vec{C}' \; R \; \vec{D}', s_1' =_L s_2'$$

where $\alpha \in \{\epsilon, \otimes sem, \odot sem\}$. The security specification then is:

Definition 2 $\vec{C}$ *is* secure $\Longleftrightarrow \vec{C} \approxeq_L \vec{C}$

One can show that the relation $\approxeq_L$ is transitive, but not reflexive. E.g., $l := h \not\approxeq_L l := h$ which reflects the fact that the program behaves differently for the attacker, depending on the initial value of h. The choice of this bisimulation allows for robust security. As argued in [15], *strong low-bisimulation captures timing flows.* If two commands may have a different timing behaviour depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. Also, *strong low-bisimulation is scheduler-independent.* Thus, our notion of security is robust with respect to any choice of a particular scheduler. Thanks to the clear separation between deterministic and nondeterministic semantics, the underlying theory of [15] is applicable to the new semantics, which, e.g, suggests a generalisation to probabilistic schedulers (although it is outside the scope of this paper). Despite the fact that these features impose restrictions on what can be considered low-bisimilar, the choice of *strong low-bisimulation is adequate* (not too restrictive) for the type-based analysis proposed in Section 4.4. We will prove that whenever a command is typable in our system, it is secure.

4.4 Type-based Security Analysis

This section presents an automatic compositional analysis for certifying secure programs, extending and improving on previous approaches [3, 18, 2, 15]. The section concludes by stating soundness results.

4.4.1 The Type System

The analysis is based on a type system that transforms a given program into a new program. If the initial program is free of direct, indirect and synchronisation insecure information flows then it might be accepted by the system and transformed into a program that is also free of timing leaks. Otherwise the initial program is rejected. The transformation rules have the form $\vec{C} \hookrightarrow \vec{C}' : \vec{Sl}$, where $\vec{C}$ is a program, $\vec{C}'$ is the result of its transformation and $\vec{Sl}$ is the type of $\vec{C}'$. The type $\vec{Sl}$ is $\vec{C}'$'s *low slice*, i.e., essentially a copy of $\vec{C}'$ in which assignments to (and conditionals on) high variables have been replaced by skip's. The low slice $\vec{Sl}$ has no occurrences of h and models the timing behaviour of C', as observable by other threads.

The typing and transformation rules are presented in Figure 4.4. The variables h and l have the types *high* and *low* respectively. Value literals n may be considered as either *high* or *low*. An arbitrary expression Exp is considered *high*. In all but If_{high} and If'_{high} rules, the transformed program is constructed compositionally using the same constructs as the original program. The information about the low slice of the new program is recorded in the typing. Command skip is its own low slice and therefore its own type. The rule Set_{low} prevents direct insecure information flows – the assignment $l := h$ is not typable. The rule Set_{high} types an assignment to the high variable with the low slice skip. The rules Exp_{low}, Seq, If_{low}, Par and Fork propagate types compositionally. The guard of the while-loop in the rule While has to be low in order to prevent the timing (and nontermination) flow from the loop's guard. The rules Wait and Signal guarantee that only low semaphores are used for synchronisation.

In addition to the insecure flows exemplified in Section 4.1, there are further ways to leak information through blocking. Synchronising on a high semaphore variable leads to (un)blocking of a thread and, clearly, may affect termination behaviour of the program. As a consequence, neither wait(sem) nor signal(sem) on a high semaphore is allowed by the type system. The rule If_{high} prevents indirect insecure flows and timing flows. An indirect flow might be performed through synchronisation on a low semaphore depending on a high guard: if $h = 1$ then signal(sem) else skip. Thus, the type system prevents from synchronisation in the branches of an if-on-high. Let $al(C)$ be a boolean function returning True whenever there is a syntactic occurrence of either an assignment to low variable l or a synchronisation primitive (wait or signal) in the command C and returning False otherwise. The condition $al(Sl_1) = al(Sl_2) =$ False prevents indirect leaks. Both branches must be typable (i.e., they must have a low slice). For the transformed program to be secure (preventing timing leaks) it is also necessary that the two branches be low-bisimilar. This is achieved by cross-copying the low slice of one branch into the other (cf. Figure 4.5). The slice of the overall command is the sequential composition of the slices of the branches prefixed with a skip

$$[\text{Exp}] \quad h : high \qquad l : low \qquad n : \tau \qquad Exp : high$$

$$[\text{Exp}_{low}] \quad \frac{Exp_1 : low \quad Exp_2 : low}{op(Exp_1, Exp_2) : low}$$

$$[\text{Skip}] \quad \mathsf{skip} \hookrightarrow \mathsf{skip} : \mathsf{skip}$$

$$[\text{Set}_{low}] \quad \frac{Exp : low}{l := Exp \hookrightarrow l := Exp : l := Exp}$$

$$[\text{Set}_{high}] \quad h := Exp \hookrightarrow h := Exp : \mathsf{skip}$$

$$[\text{Seq}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{C_1; C_2 \hookrightarrow C_1'; C_2' : Sl_1; Sl_2}$$

$$[\text{While}] \quad \frac{B : low \quad C \hookrightarrow C' : Sl}{\mathsf{while}\ B\ \mathsf{do}\ C \hookrightarrow \mathsf{while}\ B\ \mathsf{do}\ C' : \mathsf{while}\ B\ \mathsf{do}\ Sl}$$

$$[\text{Par}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \ \ldots\ C_n \hookrightarrow C_n' : Sl_n}{\langle C_1 \ldots C_n \rangle \hookrightarrow \langle C_1' \ldots C_n' \rangle : \langle Sl_1 \ldots Sl_n \rangle}$$

$$[\text{Fork}] \quad \frac{C_1 \hookrightarrow C_1' : Sl_1 \quad \vec{C_2} \hookrightarrow \vec{C_2'} : \vec{Sl_2}}{\mathsf{fork}(C_1, \vec{C_2}) \hookrightarrow \mathsf{fork}(C_1', \vec{C_2'}) : \mathsf{fork}(Sl_1, \vec{Sl_2})}$$

$$[\text{Wait}] \quad \frac{sem : low}{\mathsf{wait}(sem) \hookrightarrow \mathsf{wait}(sem) : \mathsf{wait}(sem)}$$

$$[\text{Signal}] \quad \frac{sem : low}{\mathsf{signal}(sem) \hookrightarrow \mathsf{signal}(sem) : \mathsf{signal}(sem)}$$

$$[\text{If}_{low}] \quad \frac{B : low \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2}{\mathsf{if}\, B\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \hookrightarrow \mathsf{if}\, B\, \mathsf{then}\, C_1'\, \mathsf{else}\, C_2' : \mathsf{if}\, B\, \mathsf{then}\, Sl_1\, \mathsf{else}\, Sl_2}$$

$$[\text{If}_{high}] \quad \frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\mathsf{if}\, B\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \hookrightarrow \mathsf{if}\, B\, \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{skip}; Sl_1; Sl_2}$$

$$[\text{If}'_{high}] \quad \frac{B : high \quad C_1 \hookrightarrow C_1' : Sl_1 \quad C_2 \hookrightarrow C_2' : Sl_2 \quad al(Sl_1) = al(Sl_2) = \mathsf{False}}{\begin{array}{r} \mathsf{if}\, B\, \mathsf{then}\, C_1\, \mathsf{else}\, C_2 \hookrightarrow \mathsf{if}\, B\, \mathsf{then}\ \mathsf{fork}(C_1'; \mathsf{wait}(sem), Sl_2; \mathsf{signal}(sem)) \\ \mathsf{else}\ \mathsf{fork}(Sl_1; \mathsf{wait}(sem), C_2'; \mathsf{signal}(sem)) \\ : \mathsf{skip}; \mathsf{fork}(Sl_1; \mathsf{wait}(sem), Sl_2; \mathsf{signal}(sem)) \end{array}}$$

Figure 4.4: Security type system

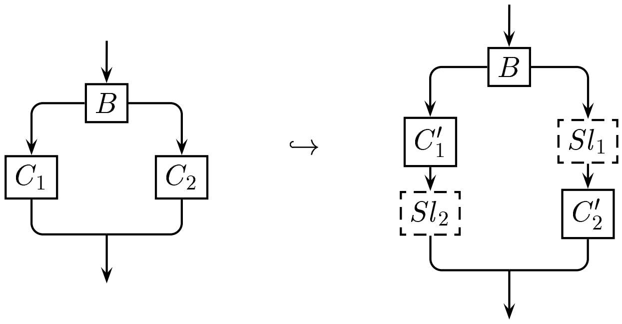


Figure 4.5: Rule If_{high}

corresponding to the time tick for the guard inspection.

A modification of the rule If_{high} might be used to guarantee that dummy computation is inserted "evenly" and it does not block useful computation within the branches of the resulting program. Such a rule makes use of the parallel composition instead of the sequential one when cross-copying (cf. Figure 4.6). The alternative rule is If'_{high}, where *sem* is a fresh semaphore, unused anywhere in the program. The use of the semaphore in the rule ensures that both main threads in each branch of the resulting code have terminated before passing the control outside the if. This is necessary to prevent potential changes in the order of execution of the if and the following commands.

An example of secure programming with synchronisation that is accepted by the type system is given in Section 4.5.

4.4.2 Correctness of the Analysis

The key to straightforward correctness proofs is the *compositionality of the security property* (Definition 2). In the standard security terminology, this is called the *hook-up* property [12]. Having the hook-up property facilitates modular development of secure code.

Clearly, an arbitrary context is not necessarily secure, i.e., the low-bisimulation is not a congruence. The program $l := h$ is an example of an insecure ground

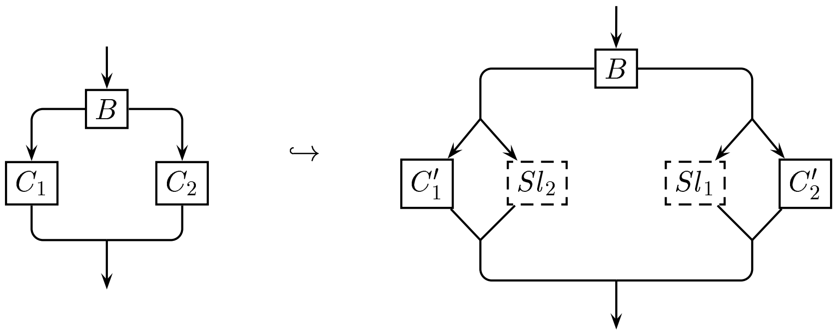


Figure 4.6: Rule If'_{high}

context. Let us go through the program constructs of the language and investigate what constraints they impose for building *secure contexts*. We identify secure contexts and give a formal proof.

Let $[\vec{\bullet}]$ be a hole for a command vector and $[\bullet]$ be a hole for a singleton command. A context $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ is secure iff it has one of these forms:

$$\begin{aligned}\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}] ::= {} & \mathsf{skip} \mid h := Exp \mid l := Exp \ (Exp \text{ is low}) \\ & \mid [\bullet_1]; [\bullet_2] \mid \mathsf{if}\ B\ \mathsf{then}\ [\bullet_1]\ \mathsf{else}\ [\bullet_2]\ (B \text{ is low}) \\ & \mid \mathsf{while}\ B\ \mathsf{do}\ [\bullet_1]\ (B \text{ is low}) \mid \mathsf{fork}([\bullet_1], [\vec{\bullet_2}]) \mid \langle [\vec{\bullet_1}][\vec{\bullet_2}] \rangle \\ & \mid \mathsf{wait}(sem)\ (sem \text{ is low}) \mid \mathsf{signal}(sem)\ (sem \text{ is low})\end{aligned}$$

where a (boolean or arithmetic) expression Exp is defined to be *low* iff $\forall s_1 =_L s_2.\, \exists n.\, \langle\!|Exp, s_1|\!\rangle \downarrow n \ \&\ \langle\!|Exp, s_2|\!\rangle \downarrow n$. Otherwise, the expression is *high*.

Theorem 1 (Secure congruence) *Suppose* $\vec{C_1} \cong_L \vec{C_1'}$, $\vec{C_2} \cong_L \vec{C_2'}$. *If* $\mathbb{C}[\vec{\bullet_1}, \vec{\bullet_2}]$ *is a secure context, then it is the case that* $\mathbb{C}[\vec{C_1}, \vec{C_2}] \cong_L \mathbb{C}[\vec{C_1'}, \vec{C_2'}]$. *If* $\mathbb{C}[\bullet_1, \bullet_2] = \mathit{if}\ B\ \mathit{then}\ [\bullet_1]\ \mathit{else}\ [\bullet_2]$ *(B is high), then* $\mathbb{C}[C_1, C_2] \cong_L \mathbb{C}[C_1', C_2']$ *provided* $C_1 \cong_L C_2$.

Due to the separation of the semantics into deterministic and nondeterministic parts, we are able to reuse the proof of secure congruence for the language without synchronisation from [15] since it operates only on the deterministic part of the semantics. Adding the secure contexts $\mathsf{wait}(sem)$ and $\mathsf{signal}(sem)$ (given sem is low) poses no substantial problems. An immediate corollary is the hook-up result.

Corollary 1 (Hook-up) *If $\vec{C}_1, \vec{C}_2$ are secure and $\mathbb{C}[\vec{\bullet}_1, \vec{\bullet}_2]$ is a secure context, then $\mathbb{C}[\vec{C}_1, \vec{C}_2]$ is secure. If $\mathbb{C}[\bullet_1, \bullet_2] = \textsf{if } B \textsf{ then } [\bullet_1] \textsf{ else } [\bullet_2]$ (B is high), then $\mathbb{C}[C_1, C_2]$ is secure provided $C_1 \cong_L C_2$.*

Proof. Observe that $\vec{C}_1, \vec{C}_2$ are secure implies $\vec{C}_i \cong_L \vec{C}_i$ for $i = 1, 2$. Thus by the security congruence theorem $\mathbb{C}[\vec{C}_1, \vec{C}_2] \cong_L \mathbb{C}[\vec{C}_1, \vec{C}_2]$, i.e., $\mathbb{C}[\vec{C}_1, \vec{C}_2]$ is secure. □

Since both the security property and the analysis are compositional, the correctness proof is a simple structural induction.

Theorem 2 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \cong_L \vec{Sl}$.

Corollary 2 (Correctness of the Analysis) $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'}$ *is secure.*

To see why the corollary holds, note that Theorem 2 gives $\vec{C'} \cong_L \vec{Sl}$. The symmetry and transitivity of $\cong_L$ entails $\vec{C'} \cong_L \vec{C'}$, i.e., $\vec{C'}$ is secure.

4.4.3 Soundness of the Transformation

We have shown that the result of the transformation is secure, but what of its relation to the original program? Clearly, the padding introduced by the transformation can change the timing, but otherwise it is just additional "stuttering". To make this point precise, let us define a *weak (bi)simulation* on configurations. Let $\langle\!| C, s |\!\rangle \to^\star \langle\!| C', s' |\!\rangle$ hold iff $\langle\!| C, s |\!\rangle = \langle\!| C', s' |\!\rangle$ or $\langle\!| C, s |\!\rangle \to \langle\!| C', s' |\!\rangle$.

Definition 3 *Define the weak simulation $\preceq$ (resp., bisimulation $\simeq$) to be the union of all (resp., symmetric) relations R on thread pools such that if $\vec{C}\ R\ \vec{D}$ and $\forall sem.\, w_{sem}\ R\ v_{sem}$ then for all s, s', w', C', there exists D' and v' such that*

$$\langle\!| \vec{C}, w, s |\!\rangle \to \langle\!| \vec{C'}, w', s' |\!\rangle \Longrightarrow \langle\!| \vec{D}, v, s |\!\rangle \to^\star \langle\!| \vec{D'}, v', s' |\!\rangle, \\ \vec{C'}\ R\ \vec{D'}, \forall sem.\, w'_{sem}\ R\ v'_{sem}$$

Lemma 1 *$\preceq$ and $\simeq$ are reflexive, transitive and respected by all contexts (not necessarily secure), i.e., $\preceq$ is a precongruence and $\simeq$ is a congruence.*

Theorem 3 $\vec{C} \hookrightarrow \vec{C'} : \vec{Sl} \Longrightarrow \vec{C'} \preceq \vec{C}$.

Proof. Induction on the height of the transformation derivation. The cases Skip, Set_{low}, Set_{high}, Wait and Signal follow from the reflexivity of $\preceq$. The cases Seq, While, Par, Fork and If_{low} are implied by the induction hypotheses and the congruence of $\preceq$.

The remaining case is If_{high} (reasoning for If'_{high} is analogous). We have $\mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ B\ \mathsf{then}\ C'_1; Sl_2\ \mathsf{else}\ Sl_1; C'_2 : \mathsf{skip}; Sl_1; Sl_2$. By induction, $C'_1 \preceq C_1$ and $C'_2 \preceq C_2$. The side condition guarantees that there are no assignments or synchronisation in the slices Sl_1 and Sl_2 of the branches C_1 and C_2. Thus, Sl_1 and Sl_2 contain only (potentially nonterminating) dummy computation without changing the state. The insertion of such a computation might (infinitely) slow down the computation. By the definition of $\preceq$, we have $C'_1; Sl_2 \preceq C_1$ and $Sl_1; C'_2 \preceq C_2$. The congruence of $\preceq$ yields $C' \preceq C$. $\square$

4.5 Secure Programming with Synchronisation

Despite the restrictions imposed by the security requirements, one can still write useful programs with synchronisation. This section gives a simple example of a secure program that uses synchronisation.

Let us use *split binary semaphores* [4] in an example implementation of simple web-form processing. Suppose web forms are used for registering users of a security-sensitive website. As an entry gets processed, the low and high outputs are computed. In particular, the low variable *counter* is incremented. The high variable *milcounter* is incremented whenever the affiliation of the user is military. The program fragment is given in Figure 4.7.

The high variables are *record*, *buf*, *result* and *milcounter*. The low variables are *counter* and the semaphore variables *empty* and *full*. We assume a language with enriched data structures (records) for the purpose of this example, but this does not affect the security argument.

The program is succesfully accepted by the type system. This means that the program is secure and, in particular, free of timing leaks.

```
Main : C; fork(D1, D2)
   C : empty := 1; full := 0; counter := 0; milcounter := 0
  D1 : while True
         do ... /* produce a new record when a new form is submitted */
            wait(empty);
            buf := record; /* the record is now in the buffer */
            signal(full)
  D2 : while True
         do wait(full);
            result := buf; /* fetch the record */
            signal(empty);
            if result.org = military
              then milcounter := milcounter + 1
               else skip
            counter := counter + 1;
```

Figure 4.7: Example of secure programming with synchronisation

4.6 Conclusions and Related Work

We have investigated the security of information flow in multi-threaded programs in the presence of synchronisation. The main result is that allowing neither synchronisation on high nor any synchronisation (not even on low) in the branches of an if-on-high is sufficient for building up a compositional timing-sensitive security specification from previous definitions that did not consider synchronisation. We have also proposed a type-based analysis that certifies programs according to the security specification. Let us conclude by discussing some improvements of this analysis compared to previous certification systems for security.

Figure 4.8 gives a comparative overview of some of the most related approaches to analysing confidentiality of multi-threaded programs. The first column **Ref** gives references to the related systems.

NI means whether or not the system has been proved to certify programs to be secure under an extensional noninterference-like security property. The second and the fourth systems have been proved to guarantee *probabilistic noninterference*. Probabilistic noninterference seems to hold for the presented system. In fact, it has been designed with probabilistic noninterfer-

Ref	**NI**	**Aut**	**Sync**	**Time**
Andrews&Reitman [3]	No	No	Yes	No
Volpano&Smith [18]	Yes	Yes	No	protect
Agat [2]	Yes	Yes	No	Yes
Sabelfeld&Sands [15]	Yes	Yes	No	Yes
This paper	Yes	Yes	Yes	Yes

Figure 4.8: Approaches to security analysis of multi-threaded programs

ence in mind—future work includes a formal development of the soundness proof with respect to probabilistic noninterference.

Aut stands for automatic. All but the first system are automatic analyses. The first system is formulated as a logic without any decision algorithms or soundness proofs.

Sync indicates whether or not the underlying language has synchronisation primitives. Only the first and the present investigations consider synchronisation. Furthermore, Agat's study [2] only considers sequential programs.

Time says whether or not the system captures timing covert channels. Andrews and Reitman's paper [3] sketches the possibility of taking into account timing leaks in their security logic. However, the proposed mechanism rejects all programs that branch on high variables. Volpano and Smith do consider timing flows in [18]. They allow branching on high by requiring all if-on-high commands to be embraced by special protect commands. The protect executes atomically by definition, making the timing difference invisible for the attacker. Such a command seems difficult to implement without locking the execution of every atomic command in the language or using additional mechanisms like thread priorities. Even that will not close external timing leaks, i.e., a time-consuming computation will not be hidden by a protect from the attacker with a stop-watch.

Acknowledgements

Thanks are due to David Sands for fruitful discussions and to Johan Agat and Makoto Takeyama for many valuable comments.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages (January 1999)*, 1999.

[2] J. Agat. Transforming out timing leaks. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 40–53, January 2000.

[3] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM TOPLAS*, 2(1):56–75, January 1980.

[4] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.

[5] J.-P. Banatre, C. Bryce, and D. Le Metayer. Compile-time detection of information flow in sequential programs. *LNCS*, 875:55–73, 1994.

[6] E. S. Cohen. Information transmission in sequential programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[7] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[9] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[10] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[11] K. R. M. Leino and Rajeev Joshi. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1), 2000.

[12] D. McCullough. Specifications for multi-level security and hook-up property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, 1987.

[13] M. Mizuno and D. Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6A):727–754, 1992.

[14] P. Ørbæk. Can you Trust your Data? In *Proceedings of the TAPSOFT/FASE'95 Conference*, LNCS 915, pages 575–590, May 1995.

[15] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, England, July 2000.

[16] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[17] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 355–364, 19–21 January 1998.

[18] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[19] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):1–21, 1996.

Chapter 5

A Generic Approach to the Security of Multi-threaded Programs

The security of computation at the level of a specific programming language and the security of complex systems at a more abstract level are two major areas of current security research. With the objective to integrate the two, this chapter proposes a translation of a timing-sensitive security property for simple multi-threaded programs into a more general security framework. Interestingly, our notion of security for programs is bisimulation-based while the security framework is trace-based. Nevertheless, we show that the translation is sound *and* complete *in the sense that the trace-based specification which results from the translation of a multi-threaded program is secure if and only if the original program is secure. The translation is presented as a two-step process where the first step is independent from the concrete programming language.*

This chapter was written with Heiko Mantel, and a condensed version is to appear in the *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, June 11-13, 2001. IEEE Computer Society Press.

5.1 Introduction

5.1.1 Motivation

An important step in the specification of secure information flow in a complex distributed system where local parts are written in a particular programming language is to combine two types of security. Namely, the first type is the security of communication between local computations and the second type is the security of the local computations themselves. The former is often defined as security of an event-based system (as in the underlying model of [18]) whereas the latter relies on the security specification of the programming language (as in the underlying model of [27] for a simple imperative multi-threaded language). Embracing the two kinds of security into a single security framework is the motivation of this paper.

5.1.2 Background

There is a large body of research on information flow control aiming at specifying, verifying, and analyzing security. In the traditional abstract view, security is often defined for an abstract *trace-based* model of computation. In particular, a system can be represented as a set of its traces and, thus, security is a property that can be true or false for a given set of traces. In a distributed setting, these traces can be viewed as sequences of events like, e.g., communication of local processes in a distributed network. Many different approaches to this type of general information flow control have been proposed (e.g., [13, 30, 12, 22, 16, 14, 33, 25, 26]), which increased the need to unify and to compare. This has led to uniform frameworks and detailed comparisons [23, 11, 34, 18].

Another line of research that is becoming increasingly popular is information flow control in a setting of a concrete programming language. The efforts in this area are focused on determining whether a given program written in a particular programming language has secure information flow. More concrete assumptions are usually made about local computations. For example, one might assume that the program runs on a partition of data on high (private) and low (public) security data (although a more general lattice of security levels can be considered). The program is not trusted (possibly received over the Internet). The program's low output is publicly available (e.g., sent over the Internet) as well as, perhaps, timing information about the program's execution (e.g., times when the program makes

Internet accesses are observable).

Originating from early work of Denning [8, 9] and Cohen [5, 6], secure information flow in programming languages received its recent reincarnation in work of Volpano et al. [32] with the main contribution being soundness proofs for a Denning-style security analysis. Many other researchers have investigated the problem of secure information flow including Joshi and Leino's equational specification [17], a single calculus for security, binding-time analysis, program slicing and call-tracking (DCC) by Abadi et al. [1], Heintze and Riecke's Secure Lambda Calculus (Slam) [15], Volpano and Smith's investigations on security of concurrent programs [29, 31], and Sabelfeld and Sands's security formalization based on partial equivalence relations [28] and a scheduler-independent probability-sensitive security specification for multi-threaded programs [27].

The security formalization in the studies mentioned founds on the extensional approach to security, namely *non-interference* [13]. The idea behind non-interference is that a system is considered secure if high inputs do not interfere with low-observable behavior of the system (low outputs, timing, etc.).

It has often been claimed that extensional programming-language-based security can be viewed as a form of non-interference (e.g., in [32]), especially since the revival of the interest in language-based security. Nevertheless, for the language-based extensional security models that have been proposed since the mid-nineties a rigorous connection to non-interference-like properties has not so far been established to the best of our knowledge. This paper is a step in this direction.

Our choice for the abstract event-based framework is Mantel's assembly kit [18]. Adapting the assembly kit allows picking the appropriate security property from the assembly kit rather than inventing a new one. This also allows for combining the security of programs with the security of other components in a (potentially distributed system) using the assembly kit as an interface. This means integrating programming-language-based security at a higher level of abstraction, opening the opportunity for plugging the security of sub-systems written in a particular programming language to the global security of the system defined in a general event-based framework.

Finally, the assembly kit enjoys a number of useful extensions including local verification conditions [19], intransitive security policies [20], and refinement operators [21], which potentially enables us to use these verification techniques, to apply intransitive security policies, and to do stepwise development in the setting of secure information flow in programs (although these issues are outside the scope of the present article).

The focus of this paper is on a simple multi-threaded language (MWL) and a timing-sensitive security specification (*strong security* [27]) that implies robust security independently of a particular scheduler. We translate MWL programs into state-event systems, pick an appropriate definition of security from the assembly kit, and establish a precise correspondence between the security of MWL programs and their translations. Namely, that the translation is *sound* in the sense that the translation of any secure MWL program is secure as a state-event system; and *complete* in the sense that if the translation of an MWL program is secure as a state-event system then the original program is secure.

5.1.3 Overview

After recalling some preliminaries in Section 5.2, we introduce the concept of thread pools in Section 5.3. In Section 5.4, we specialize this generic model according to the syntax and semantics of the MWL programming language. That this specialization indeed reflects the semantics of MWL, is ensured by a collection of theorems in Section 5.5. The key contribution of our translation is that it preserves the specification of secure information flow. Section 5.6 shows that a thread pool is considered to be secure in the MWL programming language if and only if the corresponding state-event system is also considered to be secure in the assembly kit. We conclude by a discussion in Section 5.7.

5.2 Preliminaries

5.2.1 System Specifications

The behavior of systems can often be adequately specified by the set of its possible execution sequences. We follow this trace-based approach throughout this article (with the exception of parts where we use a concrete programming language). A *trace* is a sequence of events that models a possible execution sequence of the system. An *event* is an atomic action like, e.g., the sending or receiving of a message on some channel. We distinguish between input and output events. The underlying intuition is that input events are controlled by the environment of a system while output events are controlled by the system. The distinction between input and output events is somewhat fuzzy. When a system is capable to prevent the occurrence of input events, then this can be interpreted as a signal to the environment. To avoid this kind of communication, *input totality* is often assumed, i.e., that a system cannot prevent the occurrence of input events. Since input totality is quite restrictive, we refrain from making this assumption in this article. In complex systems, communication between components is done by synchronization on the occurrence of shared events (usually output events of the one component that are input events of others).

For specifying systems, we do not define the set of traces directly but rather use states as an auxiliary concept. This allows us to define the possible traces inductively by a transition relation. The system model, we use for specification, are state-event systems. This system model allows for the specification of non-deterministic systems where the non-determinism is reflected by the choice between different events that are enabled. For simplicity, any non-determinism in the effects of events is ruled out.

Definition 1 *Let S be a set of states, E be a set of events, and $T \subseteq S \times E \times S$ be a transition relation. A* state-event system *SES is a tuple (S, S_I, E, I, O, T) where $S_I \subseteq S$ are the initial states and $I, O \subseteq E$ respectively are the input and output events. Throughout this paper we assume that S_I is a singleton set and that for a given state s and event e there is at most one state s' with $(s, e, s') \in T$.*

Let $s_1, s_2, s' \in S$, $e \in E$, and $\gamma \in E^*$. Instead of $(s_1, e, s_2) \in T$ we sometimes use the notation $s_1 \xrightarrow{e}_T s_2$. For multi-event transitions, we use the notation $s_1 \overset{\gamma}{\Longrightarrow}_T s'$. If T is obvious from the context then we omit the index and write $s_1 \xrightarrow{e} s_2$ or $s_1 \overset{\gamma}{\Longrightarrow} s'$. The relation $\overset{\gamma}{\Longrightarrow}_T$ is formally defined as follows:

$$
\begin{array}{ll}
s_1 \overset{\langle\rangle}{\Longrightarrow}_T s' & \text{, if } s_1 = s' \\
s_1 \overset{e.\gamma}{\Longrightarrow}_T s' & \text{, if } \exists s_2 \in S. s_1 \xrightarrow{e}_T s_2 \wedge s_2 \overset{\gamma}{\Longrightarrow}_T s'
\end{array}
$$

A sequence $\tau \in E^*$ of events is a *trace* of a state-event system $SES = (S, \{s_0\}, E, I, O, T)$ if it is accepted in the initial state, i.e., $\exists s' \in S. s_0 \overset{\tau}{\Longrightarrow}_T s'$. The set of all traces for SES is denoted by Tr_{SES}. We omit the index and simply write Tr if the state-event system is obvious from the context. The tuple (E, I, O, Tr_{SES}) is referred to as the *event system* corresponding to SES. A state $s \in S$ is *reachable*, denoted by *reachable*(s), if there exists a trace $\tau \in E^*$ such that $s_0 \stackrel{\tau}{\Longrightarrow} s$. The *projection* $\alpha|_{E'}$ of a sequence $\alpha \in E^*$ to the events in $E' \subseteq E$ results from α by deleting all events not in E'.

5.2.2 Security Properties

Security requirements can be expressed as restrictions on the information flow within a system. To express confidentiality or integrity by such restrictions is the key idea of information flow control. A *security property* $SecProp_{TP}$ consists of three elements: a flow policy *FP*, a domain assignment *dom*, and a security predicate *SP*.

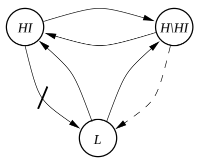
A *flow policy* specifies restrictions on the information flow within a system. For this purpose, firstly, a set of security domains is chosen. Typical domains are, e.g., groups of users, collections of files, or memory sections. Secondly, relations $\not\rightsquigarrow, \rightsquigarrow_V, \rightsquigarrow_N \subseteq \mathcal{D} \times \mathcal{D}$ are defined. The *non-interference relation* $\not\rightsquigarrow$ specifies where information flow between domains is forbidden. E.g., $D_1 \not\rightsquigarrow D_2$ expresses that there must be *no information flow* from D_1 to D_2. The *interference relation* $\rightsquigarrow_V$ specifies that certain domains are *visible* for others. $D_1 \rightsquigarrow_V D_2$ expresses that D_1 is visible for D_2. Finally, the relation $\rightsquigarrow_N$, specifies between which domains information flow is *not* restricted. $D_1 \rightsquigarrow_N D_2$ expresses that D_1 is not visible for D_2 but that information about D_1 may be deducible for D_2.

Definition 2 *A* flow policy *FP is a tuple* $(\mathcal{D}, \rightsquigarrow_V, \rightsquigarrow_N, \not\rightsquigarrow)$ *where* $\rightsquigarrow_V, \rightsquigarrow_N, \not\rightsquigarrow \subseteq \mathcal{D} \times \mathcal{D}$ *form a disjoint partition of* $\mathcal{D} \times \mathcal{D}$ *and* $\rightsquigarrow_V$ *is reflexive. FP is called* transitive *if* $\rightsquigarrow_V$ *is transitive and, otherwise,* intransitive.

A *domain assignment* links a flow policy to a system specification by associating domains to events. We often denote the set of all events with a given domain D also by D, the name of the security domain, and use that name in lower case, possibly with indices or primes, e.g., $d, d_1, \ldots$, to denote events with that domain.

Definition 3 *A* domain assignment $dom : E \to \mathcal{D}$ *is a function that assigns domains to events.*

We depict flow policies as graphs where each node corresponds to a security domain. The relations $\rightsquigarrow_V$, $\rightsquigarrow_N$, and $\not\rightsquigarrow$ are respectively depicted as solid, dashed, and crossed arrows. For the sake of readability, the reflexive subrelation of $\rightsquigarrow_V$ is usually omitted. This graphical representation is shown on



	$(V,$	$N,$	$C\ )$
$\mathcal{V}_{HI}$	$(E,$	$\emptyset,$	$\emptyset\ )$
$\mathcal{V}_{L}$	$(L,$	$H\backslash HI,$	$HI)$
$\mathcal{V}_{H\backslash HI}$	$(E,$	$\emptyset,$	$\emptyset\ )$

Figure 5.1: The flow policy FP_{TP} and the views of all domains

the left hand side of Figure 5.1 for the flow policy FP_{TP}, which consists of three domains HI (high-level input events), L (low-level events), and $H\backslash HI$ (high-level internal and output events). According to FP_{TP}, occurrences of low-level events are visible for both high-level domains. High-level inputs must not be deducible for the low-level ($HI \not\rightsquigarrow L$). Other high-level events may be deducible ($H\backslash HI \rightsquigarrow_N L$), if this does not reveal information about high-level inputs.

Traditionally, FP_{TP} would be defined as a policy with two domains L, H and the relations $H \not\rightsquigarrow L$, $L \rightsquigarrow H$. This leaves it implicit that occurrences of events in $H\backslash HI$ may be deducible for L. Our distinction between $\not\rightsquigarrow$ and $\rightsquigarrow_N$ allows us to make such assumptions explicit in the flow policy.

A *security predicate* specifies under which conditions a system specification satisfies a flow policy for some domain assignment. It can also be understood as a *definition of what information flow means.* SP must be satisfied for the view of each domain, whereas the *view* $\mathcal{V}_D = (V, N, C)$ *for a domain* $D \in \mathcal{D}$ in FP is defined by $V = \bigcup\{D' \in \mathcal{D} \mid D' \rightsquigarrow_V D\}$, $N = \bigcup\{D' \in \mathcal{D} \mid D' \rightsquigarrow_N D\}$, and $C = \bigcup\{D' \in \mathcal{D} \mid D' \not\rightsquigarrow D\}$. Basically, V contains all events that are *visible* for D, C contains all events that are *confidential* for D, and N contains all events that are *neither* visible nor confidential. The views for all domains of FP_{TP} are depicted on the right-hand side of Figure 5.1. Among these, the view of domain L is the only interesting one because it gives rise to a non-trivial proof obligation. The precise proof obligation, of course, depends on the security predicate.

An assembly kit that allows for the uniform and modular representation of security predicates, has been previously proposed by one of the authors [18]. It simplifies the comparison among the existing security predicates and a goal-directed construction of new ones. In the assembly kit, security predicates are composed by conjunction from one or more basic security predicates (abbreviated by BSP).

For the purposes of the current paper, a simple security predicate suffices which consists only of a single *BSP*, *backwards strict insertion of admissible confidential events* (abbreviated by *BSIA*). $BSIA_{\mathcal{V}}$ requires that the occurrence of an event from C does *not remove* possible low-level observations. Considering the system after a trace β has occurred, any observation $\overline{\alpha} \in V^*$ that is possible must also be possible after an arbitrary confidential event $c \in C$ has occurred. If the observation $\overline{\alpha}$ results from $\alpha \in (V \cup N)^*$, i.e., $\alpha|_V = \overline{\alpha}$, then some $\alpha' \in (V \cup N)^*$ must be possible after c has occurred where α' may differ from α only in events from N. The premise $\beta.c \in Tr$ ensures that the event c is admissible after β. For a given view $\mathcal{V} = (V, N, C)$, $BSIA_{\mathcal{V}}$ is formally defined as follows:

$$\begin{aligned} BSIA_{V,N,C}(Tr) \;&\equiv \forall \alpha, \beta \in E^*.\, \forall c \in C.\, ((\beta.\alpha \in Tr \wedge \alpha|_C = \langle\rangle \wedge \beta.c \in Tr) \\ &\Longrightarrow \exists \alpha' \in E^*.\, (\alpha'|_V = \alpha|_V \wedge \alpha'|_C = \langle\rangle \wedge \beta.c.\alpha' \in Tr)) \end{aligned}$$

The security guarantee provided by *BSIA* is: if an adversary observes $\overline{\alpha}$ starting in some state then he or she cannot deduce that a confidential event c *has not* occurred. Clearly, it could also be important to prevent an adversary from deducing that a confidential event *has* occurred. For *BSPs* which provide this type of guarantee (and others), we refer to [18, 19, 20].

5.3 Generic Thread Pools

For distributed programming, the use of multi-threaded programming languages has become extremely popular [4]. The use of concurrent threads that operate in the same address space appears to be the adequate approach for applications that are, e.g., based on the client-server paradigm. For example, this allows one to program a file server that creates, for every incoming request, a new thread that handles this request and terminates afterwards. Compared to parallelism at the level of processes, an important advantage is that context switching is far less expensive for threads.

To model the behavior of multi-threaded processes in state-event systems is technically somewhat difficult.[1] The main difficulty is that threads communicate with each other asynchronously via shared memory, while state-event systems are based on a synchronous, message-passing-like communication paradigm (cf. Section 5.2.1). However, to specify processes with these formalisms is very natural because inter-process communication is usually synchronous.

[1] Similar problems occur when using process algebras like CSP or CCS.

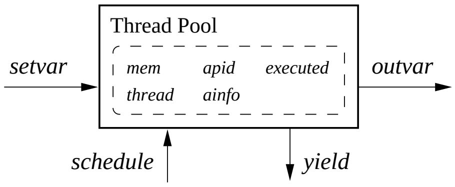


Figure 5.2: Generic thread pool with interface events and state objects

In this section, we demonstrate how the behavior of multi-threaded processes can be modeled using state-event systems. The proposed specification is highly generic because it is not only parametric in the particular program but also in the programming language. How to instantiate this specification for the concrete programming language MWL will be demonstrated in Section 5.4.

5.3.1 Trace-Based Formal Specification

In our specification, a multi-threaded process is modeled as a collection of threads that shares a global memory. We refer to such a collection as a *thread pool*. As depicted in Figure 5.2, a thread pool has five state objects (*mem*, *thread*, *apid*, *ainfo*, *executed*) and can communicate with the environment by four kinds of interface events (*setvar*-, *outvar*-, *schedule*- and *yield*-events).

The *shared memory* of a thread pool is modeled by the function $mem : \mathtt{VAR} \to \mathtt{VAL}$ that assigns values (from VAL) to variables (from VAR). The shared memory can be updated at the interface of a thread pool by *setvar*-events. If an event $setvar(var, val)$ occurs then variable *var* is assigned value *val*. *outvar*-events output the value of variables to the environment. An event $outvar(var, val)$ is only enabled if *var* currently has value *val*. For simplicity, we assume that *outvar*-events have no other preconditions and that *setvar*-events are always enabled.

The *local state of threads* is modeled by the function $thread : \mathtt{PID} \to (\mathtt{THREAD} \cup \{\bot, \top, \langle\rangle\})$. $thread(pid)$ returns a local state (from THREAD) for the identifier $pid \in \mathtt{PID}$. The results $\bot$, $\top$, and $\langle\rangle$ do not denote a proper local state but have a special meaning. If a thread with identifier *pid* has never existed then $thread(pid) = \bot$ holds. After a thread has spawned child processes, the identifier of the parent thread is modified and *thread* returns $\top$ for the old identifier. If $thread(pid) = \langle\rangle$ then a thread with identifier *pid* has existed but has already terminated.

The remaining state objects are used for controlling the execution of threads. The value of $apid \in \mathtt{PID} \cup \{\bot\}$ denotes the identifier of the thread that is currently active in the thread pool. $apid = \bot$ indicates that no thread is active. For simplicity, we assume that there is at most one active thread at any point of time. *ainfo* is a buffer in which information is collected that shall be send to the scheduler. Note that the scheduler is external to a thread pool. The flag $executed : \mathtt{BOOL}$ is used for managing context switching. Thread execution proceeds as follows.

- If no thread is active (indicated by $apid = \bot$) then *schedule*-events are enabled. After an occurrence of $schedule(pid)$, *apid* is set to *pid*, and the thread with local state $thread(pid)$ becomes active. $schedule(pid)$ is only enabled if the thread is alive ($thread(pid) \notin \{\bot, \top, \langle\rangle\}$).

- If there is an active thread (indicated by $apid \neq \bot \wedge executed = ff$) then this thread can run. Thread execution is formally modeled by the occurrence of events that are internal to the thread pool. Since these internal events depend closely on the particular instantiation of a generic thread pool, especially on the programming language, they are intentionally not modeled at the generic level. During execution, a thread can affect the state objects *mem* and *thread*. Additionally, information for the scheduler is stored in *ainfo*. Eventually, the active thread stops executing (indicated by $executed = tt$).

- After the active thread has stopped ($executed = tt$), the scheduler can be informed about this by a *yield*-event. $yield(info)$ is only enabled if *info* corresponds to the actual scheduler information ($info = ainfo$). A *yield*-event resets the *executed*-flag, *apid*, and *ainfo*.

For the initial state, we assume that all variables are initialized with the same value *initval*. Moreover, we assume that there is exactly one thread. This thread has *initpid* as identifier and *initthread* as local state. In the initial state, the *executed*-flag, *apid*, and *ainfo* are reset.

Generic thread pools are formalized as state-event systems in the following definition.

Definition 4 *Let* VAR, VAL, PID, THREAD, *and* INFO *be types. Let* S, s_0, E_{pool}, I_{pool}, O_{pool}, *and* T_{pool} *be defined as depicted in Figure 5.3. Let* $initval \in \mathtt{VAL}$, $initpid \in \mathtt{PID}$, $initthread \in \mathtt{THREAD}$, E_{local} *be a set of events that is disjoint from* E_{pool}, *and* $T_{local} \subseteq S \times E_{local} \times S$ *be a transition relation.*

$$
\begin{aligned}
S \quad &= \quad \{mem, thread, apid, ainfo, executed \mid \\
&\qquad mem : \mathtt{VAR} \to \mathtt{VAL}, thread : \mathtt{PID} \to \mathtt{THREAD} \cup \{\bot, \top, \langle\rangle\}, \\
&\qquad apid : \mathtt{PID} \cup \{\bot\}, ainfo : \mathtt{INFO} \cup \{\bot\}, executed : \mathtt{BOOL}\}
\end{aligned}
$$

$$
\begin{aligned}
s_0 \quad &= \quad \{\forall var \in \mathtt{VAR}.\, mem(var) = initval, thread(initpid) = initthread, \\
&\qquad \forall pid \in \mathtt{PID}.\, pid \neq initpid \implies thread(pid) = \bot, \\
&\qquad apid = \bot, ainfo = \bot, executed = ff\}
\end{aligned}
$$

$$E_{pool} \quad = \quad I_{pool} \cup O_{pool}$$

$$
\begin{aligned}
I_{pool} \quad &= \quad \{setvar(var, val) \mid var \in \mathtt{VAR} \wedge val \in \mathtt{VAL}\} \\
&\quad \cup \{schedule(pid) \mid pid \in \mathtt{PID}\}
\end{aligned}
$$

$$
\begin{aligned}
O_{pool} \quad &= \quad \{outvar(var, val) \mid var \in \mathtt{VAR} \wedge val \in \mathtt{VAL}\} \\
&\quad \cup \{yield(info) \mid info \in \mathtt{INFO}\}
\end{aligned}
$$

T_{pool} is defined by

- $setvar(var, val)$ affects $mem(var)$
  $Pre:$ $true$
  $Post:$ $mem'(var) = val$
- $schedule(pid)$ affects $apid$
  $Pre:$ $apid = \bot \wedge thread(pid) \notin \{\bot, \top, \langle\rangle\}$
  $Post:$ $apid' = pid$
- $yield(info)$ affects $executed$, $apid$, $ainfo$
  $Pre:$ $executed = tt \wedge ainfo = info$
  $Post:$ $executed' = ff \wedge apid' = \bot \wedge ainfo' = \bot$
- $outvar(var, val)$ affects —
  $Pre:$ $mem(var) = val$
  $Post:$ $true$

Figure 5.3: Definition of fixed components of a generic thread pool

The generic thread pool *which is parametric in* VAR*,* VAL*,* PID*,* THREAD*,* INFO*, initval, initpid, initthread, E_{local}, and T_{local}, is defined by the following state-event system:*

$$GenPool(\mathtt{VAR}, \mathtt{VAL}, \mathtt{PID}, \mathtt{THREAD}, \mathtt{INFO}, initval, initthread, initpid, E_{local}, T_{local})$$
$$= (S, \{s_0\}, E_{pool} \cup E_{local}, I_{pool}, O_{pool}, T_{pool} \cup T_{local})$$

5.3.2 Security of Thread Pools

The problem of information flow control in multi-threaded programming languages is to prevent information flow from high to low variables. For this purpose, a security level (*low* or *high*) is assigned to each variable by a function $dom_{var} : var \to \{low, high\}$. This differs from the event-based approach, in which information flow control prevents that occurrences or non-occurrences of confidential events affect the possibility of observable behaviors. Although both approaches share the same intuitive motivation, i.e., that there should be no information flow from high to low, this technical difference complicates an integration of the two approaches. However, an integration is very desirable because it allows for a uniform investigation of information flow at the level of processes as well as at the level of threads.

The key observation, which will allow us to integrate the two approaches, is that high-level data can only be introduced into a thread pool by occurrences of *setvar*-events that change the value of high-level variables. All other events can change the state of the thread pool but cannot increase the confidentiality of data. Thus, we can express the security requirement by demanding that the occurrences of these *setvar*-events must not influence the possibility of low-level observations.

The flow policy FP_{TP} (cf. left-hand side of Figure 5.1) expresses the necessary restrictions on information flow. We assume that a (malicious) low-level user has complete knowledge about the definition of thread pools (as usual), can observe the occurrence of *schedule*- and *yield*-events, and can observe the occurrences of *outvar*- and *setvar*-events that involve only low-level variables. Consequently, all these events are assigned domain L (cf. Figure 5.4). *setvar*-events that involve high-level variables are assigned domain HI because the occurrence of these events must not be deducible by a low-level user. Occurrences of all other events must not be observable by the low-level

e	$dom_{TP}(e)$	
$schedule(pid)$ $yield(info)$	L	
$setvar(var, val)$ $outvar(var, val)$	L	, *if* $dom_{var}(var) = low$
$setvar(var, val)$	HI	, *if* $dom_{var}(var) = high$
$outvar(var, val)$	$H \backslash HI$	, *if* $dom_{var}(var) = high$
e	$H \backslash HI$	, *if* $e \in E_{local}$

Figure 5.4: Domain assignment dom_{TP}

user. They may be deducible. However, such deductions must not reveal any information about occurrences of events in *HI*.

Definition 5 *The security property* $SecProp_{TP}$ *for generic thread pools is* $(FP_{TP}, dom_{TP}, BSIA)$.

According to FP_{TP}, proof obligations arise only for the view of domain L. Thus, a thread pool *satisfies* $SecProp_{TP}$ if $BSIA_{\mathcal{V}}$ holds for the view $\mathcal{V}_L = (L, H \backslash HI, HI)$. Note that *BSIA* (cf. Section 5.2.2) is indeed an appropriate definition of information flow for this application. The argument is as follows: if changing the value of high-level variables does not eliminate the possibility of low-level behaviors, then there is no information flow from high to low because high-level variables could have any value at any given point of time. Technically, a similar effect could be achieved by demanding a *BSP* that deletes confidential events, like, e.g., *BSD* (cf. [19]). However, this possibility is not important for the purposes of this paper.

In general, choosing a definition of information flow closely depends on the particular application under consideration and there appears not to be a single "right" definition (as, e.g., also observed in [26]). The assembly kit offers a (still growing) collection of very primitive definitions of information flow (*BSPs*) and allows one to assemble these to more complex definitions (security predicates). This fine-grained view has proved to be very helpful for determining $SecProp_{TP}$.

5.4 MWL Thread Pools

In this section, we revisit the simple multi-threaded while-language (abbreviated by MWL) along with the timing-sensitive definition of security for

MWL from [27]. Further, we demonstrate how our generic specification of thread pools from Section 5.3 can be instantiated for MWL.

5.4.1 The Multi-Threaded While-Language MWL

MWL is a shared-variable multi-threaded while-loop-language with dynamic thread creation. The syntax of MWL commands is given by the grammar in Figure 5.5. As usual, boolean expressions B range over $\mathtt{BOOL}$ and arithmetic expressions Exp range over $\mathtt{EXP}$. Let $C, D, E, \ldots$ range over commands (MWL threads) $\mathtt{CMD}$, and let $\vec{C}$ denote a vector of commands of the form $\langle C_1 \ldots C_n \rangle$. Vectors $\vec{C}, \vec{D}, \vec{E}, \ldots$ range over $\vec{\mathtt{CMD}} = \cup_{n \in \mathbb{N}} \mathtt{CMD}^n$, the set of multi-threaded programs.

MWL programs execute under a shared memory on a single processor (or in a single process) such that at most one thread can be active at any given point of time. A *configuration* $\langle\!| C, mem |\!\rangle$ (or $\langle\!| \vec{C}, mem |\!\rangle$) is a pair, consisting of a command $C \in \mathtt{CMD}$ (or a vector of commands $\vec{C} \in \vec{\mathtt{CMD}}$) and a memory $mem \in \mathtt{VAR} \rightarrow \mathtt{VAL}$. mem is a finite mapping from variables to values, as in Section 5.3. The set of variables is partitioned into high and low security classes. For simplicity (but without loss of generality), we will assume that there is only one variable for each security class, h and l, respectively. We will often write the memory simply as a pair (val_h, val_l) with the values val_h for h and val_l for l. Further, we define *low-equivalence* on memories by "$mem_1 =_L mem_2$ if and only if the values of l for mem_1 and mem_2 are the same". The small-step semantics is given by transitions between configurations. The deterministic part of the semantics is defined by the transition rules in Figure 5.6. Arithmetic and boolean expressions are executed atomically by $\downarrow$ transitions. The $\twoheadrightarrow$-transitions are deterministic. The general form of a deterministic transition is either $\langle\!| C, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem' |\!\rangle$, which means termination with the final memory mem', or $\langle\!| C, mem |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, mem' |\!\rangle$. Here, one step of computation starting with command C in a memory mem gives a new main thread C', a vector $\vec{D}$ of spawned threads (possibly empty), and a new memory mem'. The command $\mathsf{fork}(C\vec{D})$, where $\vec{D}$ is required to be non-empty, dynamically creates a new vector $\vec{D}$ of threads that, afterwards, run in parallel with the main thread C. This has the effect of adding the vector $\vec{D}$ to the configuration. The rule [Pick] in Figure 5.7 defines the concurrent semantics of MWL. Whenever the scheduler picks a thread C_i for execution, then a $\rightarrow$-transition takes place updating the command pool and the shared memory according to a (small) computation step of C_i. Let $\rightarrow^*$ denote the reflexive and transitive closure of $\rightarrow$.

We can extract a simple model of the timing behavior of multi-threaded programs from the small-step semantics. This is done by the assumption that each $\twoheadrightarrow$-transition takes a single unit of time to execute. This approach gives only a rough approximation of real timing behavior, but simple extensions are possible in order to make it sensitive to the timing behavior of particular commands (cf. [2]).

5.4.2 Definition of Security for MWL

Now, we define the security of MWL programs and motivate the choice of this definition. The central idea of *extensional* security, as opposed to *intensional* security, is that confidentiality should not be specified by a special-purpose security formalism, but, rather, should be defined in terms of a standard semantics as a dependency property (more precisely, absence of dependence). If direct, indirect, and timing flows are considered, then, intuitively, a program has the extensional *noninterference* property, if varying the high input will not change the possible low-level observations, i.e., low inputs/outputs and timing. This differs from intensional security which relies on particular security primitives that are only motivated by intuition rather than a mathematical justification. Many investigations have successfully followed the extensional view including [6, 32, 15, 29, 1, 31, 17, 27, 2, 28] for justification of security analysis and verification techniques for different languages. We follow the extensional approach and focus on extensional security for MWL.

A previous investigation [27] gives an account on choosing an adequate definition of extensional security for multi-threaded programs. Which definition is appropriate depends on, for instance, whether a particular scheduler is assumed, or a robust scheduler-independent security is wanted. The central idea of the bisimulation-based approach is to define a *low-bisimulation* on commands such that the indistinguishability of the behaviors of two programs C and D for the attacker is formalized by $C \sim_L D$, where $\sim_L$ is a *low-bisimulation*. Such an approach is flexible in the choice of an appropriate low-bisimulation (different low-bisimulations are available for different degrees of security). For a given low-bisimulation $\sim_L$, the definition of security is simply: "C is secure iff $C \sim_L C$". For the purpose of this paper we adapt the *strong low-bisimulation* [27].

Definition 6 *Define* strong low-bisimulation $\approxeq_L$ *to be the union of all symmetric relations R on MWL command pools (programs) of equal size for*

$$\begin{aligned}\texttt{CMD} ::= \ & \mathsf{skip} \mid \texttt{VAR} := \texttt{EXP} \mid \texttt{CMD}_1;\texttt{CMD}_2 \mid \mathsf{if}\ \texttt{BOOL}\ \mathsf{then}\ \texttt{CMD}_1\ \mathsf{else}\ \texttt{CMD}_2\\ & \mid \mathsf{while}\ \texttt{BOOL}\ \mathsf{do}\ \texttt{CMD} \mid \mathsf{fork}(\texttt{CMD}\ \vec{\texttt{CMD}})\end{aligned}$$

Figure 5.5: Command syntax

$$\text{[Skip]} \qquad \langle\!| \mathsf{skip}, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem |\!\rangle$$

$$\text{[Assign]} \qquad \frac{Exp \downarrow^{mem} = n}{\langle\!| Id := Exp, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, [Id = n]mem |\!\rangle}$$

$$\text{[Seq}_1\text{]} \qquad \frac{\langle\!| C_1, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem' |\!\rangle}{\langle\!| C_1; C_2, mem |\!\rangle \twoheadrightarrow \langle\!| C_2, mem' |\!\rangle}$$

$$\text{[Seq}_2\text{]} \qquad \frac{\langle\!| C_1, mem |\!\rangle \twoheadrightarrow \langle\!| C_1'\vec{D}, mem' |\!\rangle}{\langle\!| C_1; C_2, mem |\!\rangle \twoheadrightarrow \langle\!| (C_1'; C_2)\vec{D}, mem' |\!\rangle}$$

$$\text{[If}_{tt}\text{]} \qquad \frac{B \downarrow^{mem} = tt}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, mem |\!\rangle \twoheadrightarrow \langle\!| C_1, mem |\!\rangle}$$

$$\text{[If}_{ff}\text{]} \qquad \frac{B \downarrow^{mem} = ff}{\langle\!| \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, mem |\!\rangle \twoheadrightarrow \langle\!| C_2, mem |\!\rangle}$$

$$\text{[While}_{tt}\text{]} \qquad \frac{B \downarrow^{mem} = tt}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, mem |\!\rangle \twoheadrightarrow \langle\!| C; \mathsf{while}\ B\ \mathsf{do}\ C, mem |\!\rangle}$$

$$\text{[While}_{ff}\text{]} \qquad \frac{B \downarrow^{mem} = ff}{\langle\!| \mathsf{while}\ B\ \mathsf{do}\ C, mem |\!\rangle \twoheadrightarrow \langle\!| \langle\rangle, mem |\!\rangle}$$

$$\text{[Fork]} \qquad \langle\!| \mathsf{fork}(C\vec{D}), mem |\!\rangle \twoheadrightarrow \langle\!| C\vec{D}, mem |\!\rangle$$

Figure 5.6: Small-step deterministic semantics of commands

$$\text{[Pick]} \qquad \frac{\langle\!| C_i, mem |\!\rangle \twoheadrightarrow \langle\!| \vec{C}, mem' |\!\rangle}{\langle\!| \langle C_1 \dots C_n\rangle, mem |\!\rangle \rightarrow \langle\!| \langle C_1 \dots C_{i-1}\vec{C}C_{i+1} \dots C_n\rangle, mem' |\!\rangle}$$

Figure 5.7: Concurrent semantics of programs

which whenever $\langle C_1 \dots C_n \rangle \; R \; \langle D_1 \dots D_n \rangle$ *then*

$$\forall mem_1, mem_2, i. \langle\!| C_i, mem_1 |\!\rangle \rightarrowtriangle \langle\!| \vec{C}', mem_1' |\!\rangle \wedge \; mem_1 =_L mem_2 \Longrightarrow$$
$$\exists \vec{D}', mem_2'. \langle\!| D_i, mem_2 |\!\rangle \rightarrowtriangle \langle\!| \vec{D}', mem_2' |\!\rangle \wedge \; mem_1' =_L mem_2' \wedge \vec{C}' \; R \; \vec{D}'$$

The choice of this particular bisimulation results in a definition of security that is timing-sensitive and scheduler-independent. *Strong bisimulation captures timing flows.* If two commands might have a different timing behavior depending on high data (which would result in information flow from high to low) then they are not low-bisimilar. *Strong bisimulation is scheduler-independent.* Thus, our notion of security is robust with the respect to any choice of a particular scheduler (including probabilistic schedulers as shown in [27]). Although these features impose restrictions on what can be considered low-bisimilar, the choice of *strong low-bisimulation is adequate* (not too restrictive), e.g., for the type-based analysis that is proposed in [27]. This analysis is sound with respect to the security definition, i.e., if a program passes the analysis, then it must be secure. For more details on the power of this type of security definition to capture insecure programs and examples of secure programming with common algorithms, such as sorting and searching, we refer to [27, 3].

Definition 7 *An MWL program* $\vec{C}$ *is* secure *if and only if* $\vec{C} \approxeq_L \vec{C}$.

In order to illustrate Definitions 6 and 7 we give some examples of secure and insecure information flow which may occur in MWL programs.

$l := h$ This program is an example of a *direct* flow. To see that this program is insecure according to Definition 7, choose $mem_1 = (0,0)$ and $mem_2 = (1,0)$. Since $\langle\!| l := h, (0,0) |\!\rangle \rightarrowtriangle \langle\!| \langle\rangle, (0,0) |\!\rangle$ and $\langle\!| l := h, (1,0) |\!\rangle \rightarrowtriangle \langle\!| \langle\rangle, (1,1) |\!\rangle$ holds, the resulting memories are not low-equivalent $(0,0) \neq_L (1,1)$. Thus, there cannot be a relation with the properties necessary for strong low-bisimilarity.

if $h = 1$ **then** $l := 1$ **else** $l := 0$ This program exemplifies an *indirect* flow through branching on a high condition. If the computation starts with low-equivalent memories $(0,0)$ and $(1,0)$, then, after one step of the computation (the test of the condition), the memories are still low-equivalent. However, after another computation step they become different depending on the initial value of h. There cannot be a relation with the properties necessary for strong low-bisimilarity.

$l := 0$; **if** $h = 1$ **then** (**while** $l < MaxInt$ **do** $l := l + 1$) **else skip** Here, the attacker may deduce secret information from the timing behavior of the program. This is an instance of a *timing* leak. Clearly, the timing behavior of the branches is different. This is captured by Definition 7. Indeed, in case the then-branch of the if is chosen, there will be no transition in the other branch to match the transitions of the while-loop.

if $h = 1$ **then** (**while** *true* **do skip**) **else skip** is a variation of the timing leak called a *termination* leak.

All examples above are insecure according to our definition. Here is an instance of a secure program:

if $h = 1$ **then** $h := h + 1$ **else skip** Indeed, the timing behavior is independent of the value of h, as well as the low variable l. A suitable symmetric relation that makes this program low-bisimilar to itself is, e.g., the relation $\{(\mathsf{if}\ h = 1\ \mathsf{then}\ h := h + 1\ \mathsf{else\ skip}, \mathsf{if}\ h = 1\ \mathsf{then}\ h := h + 1\ \mathsf{else\ skip}), (h := h + 1, \mathsf{skip}), (\mathsf{skip}, h := h + 1), (h := h + 1, h := h + 1), (\mathsf{skip}, \mathsf{skip}), (\langle\rangle, \langle\rangle)\}$.

5.4.3 Instantiating Generic Thread Pools

We now instantiate our generic model for thread pools from Section 5.3 in order to model the behavior of the multi-threaded programs of MWL. Recall, that, according to Definition 4, the following parameters must be actualized:

- types: `VAR`, `VAL`, `PID`, `THREAD`, `INFO`
- initial values: *initval*, *initthread*, *initpid*,
- internal events: E_{local}; and their behavior: T_{local}.

Consistently with the simplification of Section 5.4.1, the set `VAR` of variables consists of only two variables h and l (having in mind that h is a high-level and l a low-level variable). We do not further specify the set `VAL` of values. However, we assume that there is a set `EXP` of expressions. $Exp \downarrow^{mem} = val$ denotes that $Exp \in$ `EXP` evaluates to val where the memory mem in the index is only important if Exp contains variables. Moreover, assume a set `BOOL` of boolean expressions. $B \downarrow^{mem} = tt$ and $B \downarrow^{mem} = ff$ denote, respectively, that

$$\langle \mathit{first}(C), \mathit{rest}(C)\rangle = \begin{cases} \langle C, \langle\rangle\rangle & \text{, if } C \in \{\mathsf{skip}, \mathit{var} := \mathit{Exp}, \\ & \qquad \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2, \\ & \qquad \mathsf{while}\ B\ \mathsf{do}\ C, \mathsf{fork}(C\vec{D})\} \\ \langle C_1, C_2\rangle & \text{, if } C = (C_1; C_2) \end{cases}$$

Figure 5.8: Definition of *first* and *rest*

$$\begin{aligned} E^{\text{MWL}}_{local} \quad = \quad & \{\mathit{skip}\} \\ & \cup\{\mathit{assign}(\mathit{var}, \mathit{val}) \mid \mathit{var} \in \mathtt{VAR} \wedge \mathit{val} \in \mathtt{VAL}\} \\ & \cup\{\mathit{ite}^{tt}(B, C_1, C_2), \mathit{ite}^{\mathit{ff}}(B, C_1, C_2) \mid B \in \mathtt{BOOL} \wedge C_1, C_2 \in \mathtt{CMD}\} \\ & \cup\{\mathit{while}^{tt}(B, C_1), \mathit{while}^{\mathit{ff}}(B, C_1) \mid B \in \mathtt{BOOL} \wedge C_1 \in \mathtt{CMD}\} \\ & \cup\{\mathit{fork}(C, \vec{D}) \mid C \in \mathtt{CMD} \wedge \vec{D} \in \vec{\mathtt{CMD}}\} \end{aligned}$$

Figure 5.9: Definition of local events E^{MWL}_{local} of an MWL thread pool

$B \in \mathtt{BOOL}$ evaluates to *true* or *false*. `PID` is specialized to the set of sequences of natural numbers ($\mathtt{PID} = \mathbb{N}^*$). The set `THREAD` is specialized to `CMD`, i.e., the local state of a thread is simply an MWL command. `INFO` is specialized to $\mathtt{VAL} \times \mathtt{INT}$ where `VAL` is the value of the priority variable (which is adapted to be l for simplicity) and the `INT` part says whether the process has been killed (value -1), continues running (value 0) or has spawned $n > 0$ new processes (value n).

We do not further specify *initval*, the initial value of all variables. The identifier of the (unique) initial thread is zero, i.e., $\mathit{initpid} = 0$. MWL thread pools shall be parametric in the initial thread (parameter *initthread*).

We now introduce two auxiliary functions $\mathit{first} : \mathtt{CMD} \to \mathtt{CMD}$ and $\mathit{rest} : \mathtt{CMD} \to \mathtt{CMD} \cup \{\langle\rangle\}$. The purpose of *first* and *rest* is to decompose sequential compositions. In the definition given in Figure 5.8 we assume that C_1 does *not* have the form $D_1; D_2$, i.e., C_1 is not a sequential composition on the top level. The set E^{MWL}_{local} of internal events of an MWL thread pool is defined in Figure 5.9. Note that for each of these events there is a corresponding rule of the small-step semantics (cf. Figure 5.6). E.g., the *assign*-events correspond to rule Assign and the events ite^{tt} and $\mathit{ite}^{\mathit{ff}}$ respectively correspond to If_{tt} and $\mathrm{If}_{\mathit{ff}}$. With the exception of the rules Seq_1 and Seq_2, there are corresponding events in E^{MWL}_{local} for each rule in Figure 5.6. The reason for this correspondence is that, on the one hand side, events model atomic actions and, on the other hand, rules of a small-step semantics model atomic

T^{MWL}_{local} is defined by (assuming that $terminates(thread(apid))$ equals -1 if $rest(thread(apid)) = \langle\rangle$ and 0 otherwise):

- $skip$ affects $thread(apid)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge first(thread(apid)) = \mathsf{skip}$
  $Post:\ thread'(apid) = rest(thread(apid)) \wedge executed' = tt$
  $\wedge\ ainfo' = (mem(l), terminates(thread(apid)))$
- $assign(var, val)$ affects $mem(var)$, $thread(apid)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge Exp \downarrow^{mem}= val$
  $\wedge\ first(thread(apid)) = var := Exp$
  $Post:\ mem'(var) = val \wedge thread'(apid) = rest(thread(apid))$
  $\wedge\ executed' = tt \wedge ainfo' = (mem(l), terminates(thread(apid)))$
- $ite^{tt}(B, C_1, C_2)$ affects $thread(apid)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge B \downarrow^{mem}= tt$
  $\wedge\ first(thread(apid)) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$
  $Post:\ thread'(apid) = C_1; rest(thread(apid)) \wedge executed' = tt$
  $\wedge\ ainfo' = (mem(l), 0)$
- $ite^{f\!f}(B, C_1, C_2)$ affects $thread(apid)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge B \downarrow^{mem}= f\!f$
  $\wedge\ first(thread(apid)) = \mathsf{if}\ B\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$
  $Post:\ thread'(apid) = C_2; rest(thread(apid)) \wedge executed' = tt$
  $\wedge\ ainfo' = (mem(l), 0)$
- $while^{tt}(B, C_1)$ affects $thread(apid)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge B \downarrow^{mem}= tt$
  $\wedge\ first(thread(apid)) = \mathsf{while}\ B\ \mathsf{do}\ C_1$
  $Post:\ thread'(apid) = C_1; \mathsf{while}\ B\ \mathsf{do}\ C_1; rest(thread(apid))$
  $\wedge\ executed' = tt \wedge ainfo' = (mem(l), 0)$
- $while^{f\!f}(B, C_1)$ affects $thread(apid)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge B \downarrow^{mem}= f\!f$
  $\wedge\ first(thread(apid)) = \mathsf{while}\ B\ \mathsf{do}\ C_1$
  $Post:\ thread'(apid) = rest(thread(apid)) \wedge executed' = tt$
  $\wedge ainfo' = (mem(l), terminates(thread(apid)))$
- $fork(C, D_1 \ldots D_n)$ affects $thread(apid)$, $thread(apid.0) \ldots thread(apid.n)$, $executed$, $ainfo$
  $Pre:\ executed = f\!f \wedge apid \neq \bot \wedge first(thread(apid)) = \mathsf{fork}(CD_1 \ldots D_n)$
  $Post:\ thread'(apid) = \top \wedge thread'(apid.0) = C; rest(thread(apid))$
  $\wedge\ \forall i \in \{1, \ldots, n\} : thread'(apid.i) = D_i \wedge executed' = tt$
  $\wedge\ ainfo' = (mem(l), n)$

Figure 5.10: Definition of transition relation T^{MWL}_{local} of an MWL thread pool

$$\mathtt{cseq_{aux}}(pid, thread) = \begin{cases} \langle\rangle & \text{, if } thread(pid) \in \{\bot, \langle\rangle\} \\ thread(pid) & \text{, if } thread(pid) \in \mathtt{CMD} \\ \mathtt{cseq_{aux}}(pid.0, thread) \ldots & \\ \ldots \mathtt{cseq_{aux}}(pid.n, thread) & \text{, if } thread(pid) = \top \\ & \ \ n \in \mathbb{N} \text{ is maximal so} \\ & \ \ \text{that } thread(pid.n) \neq \bot \end{cases}$$

Figure 5.11: Definition of $\mathtt{cseq_{aux}}$

transitions between states (or configurations – in the case of MWL). The atomic actions that can occur during the execution of an MWL thread pool include, taking up time (caused by skip), assignments to variables, branching in the control flow depending on boolean tests (if then else or while do), or spawning of threads (fork). Note that, we do not consider the decomposition of sequentially composed commands as a separate action. Thus, there are no corresponding events.

The behavior of internal events is defined by the transition relation T^{MWL}_{local} (cf. Figure 5.10). Clearly, T^{MWL}_{local} should reflect the semantics of MWL. The pre- and postcondition of each event shall capture the corresponding rule of the small-step semantics. E.g., the precondition of $assign(var, val)$ requires that there is an active thread ($apid \neq \bot$) that has not already executed a command ($executed = f\!f$), the current command must be an assignment ($first(thread(apid)) = var := Exp$), and the expression Exp must evaluate to val under the current memory ($Exp \downarrow^{mem} = val$). Note that, when new threads are spawned, then the generation of identifiers is managed in such a way that no pid is used for two different processes (cf. postcondition of $fork$). That T^{MWL}_{local} indeed reflects the semantics of MWL will be proved in Section 5.5.

The instantiation of generic thread pools for MWL is summarized in the following definition.

Definition 8 *Let initthread* $\in$ CMD. *The* MWL thread pool *for initthread results from the following instantiation of generic thread pools:*

$$MWLPool(initthread) = \\ GenPool(\{l,h\}, \mathtt{VAL}, \mathbb{N}^*, \mathtt{CMD}, \mathtt{VAL} \times \mathtt{INT}, initval, initthread, 0, E^{MWL}_{local}, T^{MWL}_{local})$$

5.5 Relation between MWL Programs and MWL Thread Pools

The objective of our specification of MWL thread pools was to provide an adequate model of MWL programs and their behavior. Firstly, any behavior of an MWL thread pool should comply with the MWL semantics. Secondly, any behavior that complies with the MWL semantics should be possible for an MWL thread pool. That our specification, indeed, is adequate is ensured by the results presented in the current section.

Recall that the system models that, respectively, underly MWL programs and MWL thread pools are somewhat different. The model underlying MWL programs is based on *trees of states* (to be precise, configurations). It is possible to enrich these trees with events but from the perspective of the underlying paradigm these events would be mere decorations. Since the model of computation is state-based, the natural communication paradigm is via shared memory. The system model underlying MWL thread pools is based on *sequences of events*. It is possible to enrich these sequences with states but from the perspective of the underlying model these states would be mere decorations. Since the system model is event-based, the natural communication paradigm is via message passing. These differences between the system models on which MWL programs and MWL thread pools are based, somewhat complicate the proofs of the following theorems.

5.5.1 Adequateness of MWL Thread Pools

In Theorem 1 we will show that every trace of an MWL thread pool models a behavior that complies with the semantics of MWL. We define the function $\texttt{cseq}$, which translates a function $\mathit{thread} : \texttt{PID} \to (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\})$ into a corresponding vector of MWL commands. Note that this definition exploits that identifiers are chosen incrementally by *fork*-events and that $\mathit{thread}(\mathit{pid}) = \langle\rangle$ holds after termination of a thread with identifier *pid*.

Definition 9 $\texttt{cseq} : (\texttt{PID} \to (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\}) \to \vec{\texttt{CMD}}$ *returns a corresponding vector of MWL commands for each function thread* $: \texttt{PID} \to (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\})$*.* $\texttt{cseq}$ *is defined by* $\texttt{cseq}(\mathit{thread}) = \texttt{cseq}_{\texttt{aux}}(0, \mathit{thread})$ *where* $\texttt{cseq}_{\texttt{aux}} : \texttt{PID} \to (\texttt{PID} \to (\texttt{CMD} \cup \{\bot, \top, \langle\rangle\}) \to \vec{\texttt{CMD}}$ *is defined in Figure 5.11.*

We now present two lemmas that are helpful for proving Theorems 1,3 and 4. Throughout this section, we assume that $SES = (S, S_I, E, I, O, T)$ models an MWL thread pool, i.e., that $SES = MWLPool(initthread)$ holds for some command $initthread \in$ `CMD`.

Lemma 1 *If $reachable(s)$ holds in a state s of SES then*

- $executed_s = f\!f \wedge apid_s \neq \bot \wedge thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$,
- $executed_s = tt \wedge apid_s \neq \bot$, *or*
- $executed_s = f\!f \wedge apid_s = \bot$ *holds.*

Proof. There exists an event sequence $\gamma \in E^*$ such that $s_0 \overset{\gamma}{\Longrightarrow} s$ because $reachable(s)$ holds. The proof can be done by a straightforward induction on the length of γ where in the step case a distinction on the three cases for states which are mentioned in the lemma is necessary. □

Lemma 2 *Let s, s' be states of SES such that $executed_s = f\!f$, $apid_s \neq \bot$, $thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$, and $e \in E^{MWL}_{local}$ be an event such that $s \xrightarrow{e} s'$ holds.*

- *If $e \neq fork(C, D_1 \ldots D_n)$ then*
  $\langle\!|thread_s(apid_s), mem_s|\!\rangle \twoheadrightarrow \langle\!|thread_{s'}(apid_{s'}), mem_{s'}|\!\rangle$.
- *If $e = fork(C, D_1 \ldots D_n)$ then*
  $\langle\!|thread_s(apid_s), mem_s|\!\rangle \twoheadrightarrow$
  $\langle\!|thread_{s'}(apid_s.0) \ldots thread_{s'}(apid_s.n), mem_{s'}|\!\rangle$.

Proof. The proof can be done by a straightforward (but somewhat tedious) case distinction on the event $e \in E^{\mathrm{MWL}}_{local}$. □

Theorem 1 *Let $s, s' \in S$ be states of SES, $\gamma \in E^*$ be a sequence of events, and $\vec{D}_s, \vec{D}_{s'}$ be vector of MWL commands with $\vec{D}_s = \texttt{cseq}(thread_s)$ and $\vec{D}_{s'} = \texttt{cseq}(thread_{s'})$. If $reachable(s)$, $s \overset{\gamma}{\Longrightarrow} s'$, and γ contains no setvar-events then $\langle\!|\vec{D}_s, mem_s|\!\rangle \rightarrow^* \langle\!|\vec{D}_{s'}, mem_{s'}|\!\rangle$.*

Proof. The proof proceeds by induction on the length of γ.

Base case ($\gamma = \langle\rangle$): According to the definition of $\overset{\gamma}{\Longrightarrow}$ holds $s' = s$. $\rightarrow^*$ is reflexive.

Step case ($\gamma = e.\delta$): There exists a state $s_i \in S$ such that $s \xrightarrow{e} s_i$ and $s_i \xRightarrow{\delta} s'$ hold. If e is an *outvar*-event then $s_i = s$ and the proposition follows from the induction hypothesis. In the sequel, we assume that e is no *outvar*-event. We make a case distinction on s according to Lemma 1:

1. $executed_s = ff \wedge apid_s \neq \bot \wedge thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$,
2. $executed_s = tt \wedge apid_s \neq \bot$, or
3. $executed_s = ff \wedge apid_s = \bot$.

Case 1: Follows from $e \in E^{\mathrm{MWL}}_{local}$, Lemma 2, the frame-axioms for *thread* for events in E^{MWL}_{local}, Definition 9, rule Pick, and the induction hypothesis.

Case 2: $e = yield(ainfo_s)$ holds. $mem_{s_i} = mem_s$ and $\mathtt{cseq}(thread_{s_i}) = \vec{D}_s$ are implied by the postcondition of *yield*. The proposition follows from the induction hypothesis.

Case 3: $e = schedule(pid)$ for some $pid \in \mathtt{PID}$. The equalities $mem_{s_i} = mem_s$ and $\mathtt{cseq}(thread_{s_i}) = \vec{D}_s$ are implied by the postcondition of *schedule*. The proposition follows from the induction hypothesis. □

In Theorem 2, we will show that for every behavior that complies with the semantics of MWL, there is a corresponding trace of an MWL thread pool which models that behavior. We now present a lemma that is helpful for proving that theorem and also Theorems 3 and 4.

Lemma 3 *Let $C' \in \mathtt{CMD}$, $D_1 \ldots D_n \in \vec{\mathtt{CMD}}$, and $mem' : \mathtt{VAR} \to \mathtt{VAL}$. Moreover, let s be a state of SES such that $executed_s = ff$, $apid_s \neq \bot$, and $thread_s(apid_s) \notin \{\bot, \top, \langle\rangle\}$.*

1. *If $\langle\!| thread_s(apid_s), mem_s |\!\rangle \twoheadrightarrow \langle\!| C', mem' |\!\rangle$ then there exists an event $e \in E^{MWL}_{local}$ and a state s' of SES with $s \xrightarrow{e} s'$, $mem_{s'} = mem'$, $thread_{s'}(apid_s) = C'$, $apid_{s'} = apid_s$, and $executed_{s'} = tt$. Moreover, for all $pid \in \mathtt{PID}$ with $pid \neq apid_s$ holds $thread_{s'}(pid) = thread_s(pid)$.*
2. *If $\langle\!| thread_s(apid_s), mem_s |\!\rangle \twoheadrightarrow \langle\!| C' D_1 \ldots D_n, mem' |\!\rangle$ (with $n \geq 1$) then there exists an event $e \in E^{MWL}_{local}$ and a state s' of SES with $s \xrightarrow{e} s'$, $mem_{s'} = mem'$, $thread_{s'}(apid_s) = \top$, $apid_{s'} = apid_s$, and $executed_{s'} = tt$. Moreover, $thread_{s'}(apid_s.0) = C'$, $thread_{s'}(apid_s.i) = D_i$ holds for all $i \in \{1, \ldots, n\}$, and $thread_{s'}(pid) = thread_s(pid)$ holds for all $pid \in \mathtt{PID}$ with $pid \notin \{apid_s, apid_s.0, \ldots, apid_s.n\}$.*

Proof. Assume $\mathcal{D}$ is a derivation of $\langle\!| thread_s(apid_s), mem_s |\!\rangle \twoheadrightarrow \langle\!| C', mem' |\!\rangle$ (or a derivation of $\langle\!| thread_s(apid_s), mem_s |\!\rangle \twoheadrightarrow \langle\!| C' D_1 \ldots D_n, mem' |\!\rangle$). According to the operational semantics of MWL (cf. Figure 5.6) there must be exactly one application of one of the rules Skip, Assign, If_{tt}, If_{ff}, While_{tt}, While_{ff}, or Fork in $\mathcal{D}$. The proof can be done by a straightforward (but somewhat tedious) case distinction depending on which of these rules occurs in $\mathcal{D}$. □

Theorem 2 *Let s be a state of SES such that $apid_s = \bot$, $executed_s = ff$, and $reachable(s)$. Let $\vec{D}_s = \texttt{cseq}(thread_s)$, $\vec{D'} \in \vec{\texttt{CMD}}$, and $mem' : var \to val$. If $\langle\!| \vec{D}_s, mem_s |\!\rangle \to^* \langle\!| \vec{D'}, mem' |\!\rangle$ then there exists a sequence $\gamma : E^*$ that contains no setvar-events and a state $s' \in S$ such that $s \stackrel{\gamma}{\Longrightarrow} s'$, $mem_{s'} = mem'$, and $\vec{D'} = \texttt{cseq}(thread_{s'})$.*

Proof. Due to our assumption, there is a derivation $\mathcal{D}$ for $\langle\!| \vec{D}_s, mem_s |\!\rangle \to^* \langle\!| \vec{D'}, mem' |\!\rangle$. The proof proceeds by induction on n, the number of applications of the rule Pick in $\mathcal{D}$.

Base case ($n = 0$): If $\mathcal{D}$ contains no applications of Pick then it contains no rule applications at all. Thus, $\vec{D'} = \vec{D}_s$ and $mem' = mem_s$. Consequently, the proposition holds for $\gamma = \langle\rangle$ and $s' = s$.

Step case ($n = n' + 1$): There exists a configuration $\langle\!| \vec{D}_i, mem_i |\!\rangle$ such that $\langle\!| \vec{D}_s, mem_s |\!\rangle \to \langle\!| \vec{D}_i, mem_i |\!\rangle$ and $\langle\!| \vec{D}_i, mem_i |\!\rangle \to^* \langle\!| \vec{D'}, mem' |\!\rangle$. Let $\mathcal{D}_i$ be a derivation of $\langle \vec{D}_s, mem_s \rangle \to \langle\!| \vec{D}_i, mem_i |\!\rangle$ and $\mathcal{D}'$ be a derivation of $\langle\!| \vec{D}_i, mem_i |\!\rangle \to^* \langle\!| \vec{D'}, mem' |\!\rangle$ with n' application of rule Pick. If we can show that there is a sequence $\gamma_i : E^*$ which contains no *setvar*-events and a state s_i such that $s \stackrel{\gamma_i}{\Longrightarrow} s_i$, $mem_{s_i} = mem_i$, $\texttt{cseq}(thread_{s_i}) = \vec{D}_i$, $apid_{s_i} = \bot$, and $executed_{s_i} = ff$ then the proposition follows from the induction hypothesis because $\mathcal{D}'$ contains only $n' = n - 1$ applications of the rule Pick.

There occurs exactly one application of Pick in $\mathcal{D}_i$. Let pid be the identifier of the thread which is selected in this application. $e_1 = schedule(pid)$ is enabled in s. For $s_1 \in S$ with $s \stackrel{e_1}{\longrightarrow} s_1$ holds $mem_{s_1} = mem_s$, $thread_{s_1} = thread_s$, $apid_{s_1} \neq \bot$, and $executed_{s_1} = ff$. According to Lemma 3 there exist $e_2 \in E^{\text{MWL}}_{local}$, $s_2 \in S$ with $s_1 \stackrel{e_2}{\longrightarrow} s_2$, $mem_{s_2} = mem_i$, $\texttt{cseq}(thread_{s_2}) = \vec{D}_i$, $apid_{s_2} \neq \bot$, and $executed_{s_2} = tt$. $e_3 = yield(ainfo_{s_2})$ is enabled in s_2. There exists $s_i \in S$ with $s_2 \stackrel{e_3}{\longrightarrow} s_i$. Thus, for $\gamma_i = e_1.e_2.e_3$ holds $s \stackrel{\gamma_i}{\Longrightarrow} s_i$, $mem_{s_i} = mem_i$, $\texttt{cseq}(thread_{s_i}) = \vec{D}_i$, $apid_{s_i} = \bot$, and $executed_{s_i} = ff$. □

Theorem 1 and 2 ensure that MWL thread pools are an adequate specification of MWL programs and their behavior. All behaviors of an MWL thread pool comply with the semantics of MWL and all behaviors that comply with the semantics of MWL are possible for an MWL thread pool.

5.6 Soundness and Completeness Results

The aim of this section is to establish the soundness and completeness results. First, we recall the definition of the translation of a program in MWL into a state-event system and then proceed by proving soundness (if C is secure as an MWL program then its translation is secure as a state-event system) and completeness (if C's translation is secure as a state-event system then C is secure).

According to Definition 8 from Section 5.4, the translation $\mathit{MWLPool}(C)$ of an MWL program C is the thread pool with $\mathit{initthread} = C$. The following two sub-sections present the soundness and completeness results respectively.

5.6.1 Soundness

Before we present the soundness theorem we state a security invariant lemma. Intuitively, the lemma says that if computation starts with a secure program then all the threads in the thread pool are secure at all times. Define an auxiliary boolean function $\mathit{live}(s, \mathit{pid}) = \mathit{thread}_s(\mathit{pid}) \notin \{\bot, \top, \langle\rangle\}$ that takes the value tt whenever thread at pid in the state s is alive (exists and has not terminated). Note that $\mathit{live}(s, \mathit{pid}) = \mathit{tt}$ is the precondition for scheduling the thread at pid in s.

Lemma 4 *Assume an MWL program C is secure and $\beta \in \mathit{Tr}$ is a trace for $\mathit{MWLPool}(C)$ such that $s_0 \stackrel{\beta}{\Longrightarrow} s$. Then $\forall \mathit{pid}.\, \mathit{live}(s, \mathit{pid}) \Longrightarrow \mathit{thread}_s(\mathit{pid}) \cong_L \mathit{thread}_s(\mathit{pid})$.*

Proof. The proof is by induction on the length of β. In the base case $\beta = \langle\rangle$ we have $\forall \mathit{pid} \neq 0.\, \mathit{live}(s, \mathit{pid}) = \mathit{ff}$ and $\mathit{thread}_s(0) = \mathit{thread}_s(0) = C$. We have $C \cong_L C$ by the security of C.

The inductive step assumes $s_0 \stackrel{\delta}{\Longrightarrow} t \stackrel{e}{\longrightarrow} s$ such that $\forall \mathit{pid}.\, \mathit{live}(t, \mathit{pid}) \Longrightarrow \mathit{thread}_t(\mathit{pid}) \cong_L \mathit{thread}_t(\mathit{pid})$. If e is not local, then no threads may be created or updated. Thus, e cannot affect thread_t so that $\mathit{thread}_t = \mathit{thread}_s$ which completes the inductive step for this case. Now suppose e is local. We need to show $\forall pid.\, live(s, pid) \implies thread_s(pid) \approxeq_L thread_s(pid)$. We will appeal to Lemma 2 in order to match the transition in the semantics of MWL and unwind Definition 6 of strong low-bisimulation. We have two cases on e:

$first(thread_t(apid_t)) \neq$ **fork** We have $executed_t = \mathit{ff}$, $apid_t \neq \bot$, $live(t, apid_t)$, and $t \xrightarrow{e} s$ hold. By Lemma 2 we obtain $\langle\!|thread_t(apid_t), mem_t|\!\rangle \rightarrowtriangle \langle\!|thread_s(apid_s), mem_s|\!\rangle$. By the induction hypothesis, $thread_t(apid_t) \approxeq_L thread_t(apid_t)$. Unwinding Definition 6 of strong low-bisimulation, there exist C' and mem' such that $\langle\!|thread_t(apid_t), mem_t|\!\rangle \rightarrowtriangle \langle\!|C', mem'|\!\rangle$ where $thread_s(apid_s) \approxeq_L C'$ and $mem_s =_L mem'$. Since $\rightarrowtriangle$-transitions are deterministic, it must be the case that $C' = thread_s(apid_s)$ and $mem' = mem_s$. This gives $thread_s(apid_s) \approxeq_L thread_s(apid_s)$ and because no other pid's are affected, the proof for this case is completed.

$first(thread_t(apid_t)) =$ **fork** We have $executed_t = \mathit{ff}$, $apid_t \neq \bot$, $live(t, apid_t)$, and $t \xrightarrow{e} s$ hold. By Lemma 2 we obtain $\langle\!|thread_t(apid_t), mem_t|\!\rangle \rightarrowtriangle \langle\!|thread_s(apid_t.0) \ldots thread_s(apid_t.n),\ mem_s|\!\rangle$ where $n \in \mathbb{N}$ is the maximal natural number for which $thread_s(apid_t.n) \neq \bot$ holds. According to the induction hypothesis, $thread_t(apid_t) \approxeq_L thread_t(apid_t)$. By unwinding the definition of strong low-bisimulation (Definition 6), there exists a command C', a command sequence $\vec{D} = \langle D_1 \ldots D_n \rangle$, and mem' such that $\langle\!|thread_{t'}(apid_{t'}), mem_{t'}|\!\rangle \rightarrowtriangle \langle\!|C'\vec{D}, mem'|\!\rangle$ where $thread_s(apid_t.0) \approxeq_L C'$, $thread_s(apid_t.1) \approxeq_L D_1, \ldots, thread_s(apid_t.n) \approxeq_L D_n$ and $mem_s =_L mem'$. Since $\rightarrowtriangle$-transitions are deterministic, it must be the case that $C' = thread_s(apid_t.0), D_1 = thread_s(apid_t.1), \ldots D_n = thread_s(apid_t.n)$ and $mem' = mem_s$. We have that $thread_s(apid_t.0) \approxeq_L thread_s(apid_t.0)$, $thread_s(apid_t.1) \approxeq_L thread_s(apid_t.1), \ldots, thread_s(apid_t.n) \approxeq_L thread_s(apid_t.n)$. Noting that $live(s, apid_t) = \mathit{ff}$ and that no other pid's have been affected completes the proof.

$\square$

Theorem 3 (Soundness) *If an MWL program C is secure then the MWL thread pool $MWLPool(C)$ satisfies the security property $SecProp_{TP}$.*

Proof. Assume that C is secure. We need to show that $MWLPool(C)$ satisfies $SecProp_{TP}$. In Section 5.3.2 we have argued that $SecProp_{TP}$ is equivalent to $BSIA_{\{L\},\{H\setminus HI\},\{HI\}}$ under the flow policy FP_{TP}. By definition

$BSIA_{\{L\},\{H\setminus HI\},\{HI\}}$ holds iff

$$\forall \alpha, \beta \in E^*. \forall c \in HI.((\beta.\alpha \in Tr \wedge \alpha|_{HI} = \langle\rangle \wedge \beta.c \in Tr)$$
$$\Longrightarrow \exists \alpha' \in E^*.(\alpha'|_L = \alpha|_L \wedge \alpha'|_{HI} = \langle\rangle \wedge \beta.c.\alpha' \in Tr))$$

Let C be secure. Given $\alpha, \beta \in E^*$ and $c \in HI$ satisfying the conditions above, our aim is to construct an appropriate α' by modifying α.

The proof idea is to construct α' by induction on the length of α making use of Lemmas 2, 3. The only kind of events classified as HI is $setvar(h, \cdot)$-events. Hence, $c = setvar(h, val)$ for some val. In the inductive construction, we will prove the following invariant for sequences α,α' of the same length. The invariant $Inv(\alpha, \alpha', s, s', i)$ holds iff

$$1.\, length(\alpha) = length(\alpha) = i \qquad 2.\, \alpha'|_L = \alpha|_L \qquad 3.\, \alpha'|_{HI} = \langle\rangle$$
$$4.\, s_0 \overset{\beta.\alpha}{\Longrightarrow} s \qquad 5.\, s_0 \overset{\beta.c.\alpha'}{\Longrightarrow} s' \text{ (and thereby } \beta.c.\alpha' \in Tr)$$
$$6.\, Dom(thread_s) = Dom(thread_{s'})$$
$$7.\, \forall\, pid.thread_s(pid) = thread_{s'}(pid) = \bot \vee\ thread_s(pid) = thread_{s'}(pid) = \top$$
$$\vee\ thread_s(pid) = thread_{s'}(pid) = \langle\rangle \vee\ thread_s(pid) \approxeq_L thread_{s'}(pid)$$
$$8.\, mem_s =_L mem_{s'} \qquad 9.\, apid_s = apid_{s'}$$
$$10.\, ainfo_s = ainfo_{s'} \qquad 11.\, executed_s = executed_{s'}$$

Initially, we set $\alpha = \alpha' = \langle\rangle$. Suppose $s_0 \overset{\beta.c}{\Longrightarrow} s'$ for some s'. Clearly, we have $Inv(\alpha, \alpha', s, s', 0)$ since adding $setvar(h, val)$ can be done at any time in the computation (the precondition for a $setvar(h, val)$-event is $true$). Such an event can only change the value of h in $mem_{s'}$. Lemma 4 guarantees that part 7 of the invariant holds (traversing β preserves the security of the respective threads in the thread pool).

In the inductive step we have to construct α', s' such that $Inv(\alpha, \alpha', s, s', i{+}1)$ assuming that $\alpha = \delta.e$ and, by the induction hypothesis, $Inv(\delta, \delta', t, t', i)$ for some δ', t'. The proof is by considering cases on e such that $s_0 \overset{\beta.\delta}{\Longrightarrow} t \overset{e}{\longrightarrow} s$ for some s. In all cases we will aim at preserving the invariant. We have $s_0 \overset{\beta.\delta}{\Longrightarrow} t$ and $s_0 \overset{\beta.c.\delta'}{\Longrightarrow} t'$. Clearly, e cannot be a $setvar(h, \cdot)$ by the $\alpha|_{HI} = \langle\rangle$ condition on α. The rest of the cases on e are:

$setvar(l, val)$ Set $\alpha' = \delta'.e$. Then $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \overset{e}{\longrightarrow} s'$ for some s'. A $setvar(l, \cdot)$-event is always enabled. The event makes the same update in the low part of the memory in both mem_t and $mem_{t'}$. By induction hypothesis,

$mem_t =_L mem_{t'}$ which yields $mem_s =_L mem_{s'}$. The event does not affect any other part of either t or t' (in the transition to s or s' respectively) which gives $Inv(\alpha, \alpha', s, s', i+1)$.

outvar(l, val) Set $\alpha' = \delta'.e$. The precondition for this event in $t \xrightarrow{e} s$ is $mem_t = val$. Due to the induction hypothesis $mem_t =_L mem_{t'}$ we have $mem_{t'} = val$ which gives the precondition for $t' \xrightarrow{e} s'$ for some s'. Thus, $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e} s'$ for some s'. The event does not change any part of either t or t' (in the transition to s or s' respectively) which gives $Inv(\alpha, \alpha', s, s', i+1)$.

outvar(h, val) Set $\alpha' = \delta'.e'$ where $e' = outvar(h, val')$ and $val' = mem_{t'}(h)$. Then $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e'} s'$ for some s'. We have $Inv(\alpha, \alpha', s, s', i+1)$ since the potential difference in val for α' and α does not affect $\alpha'|_L = \alpha|_L$ because $outvar(h, \cdot)$ events are classified as $H \setminus HI$.

schedule(pid) Set $\alpha' = \delta'.e$. Then $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e} s'$ for some s' since the preconditions for $t' \xrightarrow{e} s'$ are guaranteed by the induction hypothesis $apid_t = apid_{t'} = \bot$, $live(t', pid) = tt$. The postconditions are updated in the same way, namely, $apid_s = apid_{s'} = pid$ ensuring $Inv(\alpha, \alpha', s, s', i+1)$.

yield$(info)$ Set $\alpha' = \delta'.e$. Yielding $info$ in $t \xrightarrow{e} s$ has the precondition $executed_t = tt \wedge ainfo_t = info$. According to the induction hypothesis, $executed_t = executed_{t'} = tt \wedge ainfo_t = ainfo_{t'} = info$ which gives $s_0 \overset{\beta.\delta'}{\Longrightarrow} t' \xrightarrow{e} s'$ for some s' The postconditions are updated in the same way: $executed_s = executed_{s'} = ff \wedge ainfo_s = ainfo_{s'} = \bot$ ensuring $Inv(\alpha, \alpha', s, s', i+1)$.

Let us turn to local events. We are able to treat all events at once by appealing to Lemmas 2, 3 and unwinding strong low-bisimulation according to Definition 6. We only have two cases on e:

$first(thread_t(apid_t)) \neq \textsf{fork}$ We have $executed_t = ff$, $apid_t \neq \bot$, $live(t, apid_t)$, and $t \xrightarrow{e} s$ hold. By applying the first case of Lemma 2 we receive that $\langle\!| thread_t(apid_t), mem_t |\!\rangle \twoheadrightarrow \langle\!| thread_s(apid_s), mem_s |\!\rangle$. By the induction hypothesis holds $mem_t =_L mem_{t'}$ and $thread_t(apid_t) \approxeq_L thread_{t'}(apid_{t'})$ (note that $apid_t = apid_{t'}$). Unwinding the definition of strong low-bisimulation (Definition 6) yields that there exist C' and mem' such that $\langle\!| thread_{t'}(apid_{t'}), mem_{t'} |\!\rangle \twoheadrightarrow \langle\!| C', mem' |\!\rangle$,

$thread_s(apid_s) \approxeq_L C'$, and $mem_s =_L mem'$. According to the induction hypothesis, we have $executed_{t'} = \mathit{ff}$, $apid_{t'} \neq \bot$, and $live(t', apid_{t'})$. At this point we can apply Lemma 3. By this lemma there exists an event $e' \in E^{\text{MWL}}_{local}$ and an SES-state s' with $t' \xrightarrow{e'} s'$, $apid_{s'} = apid_{t'}$, $thread_{s'}(apid_{s'}) = C'$ and $mem_{s'} = mem'$, $executed_{s'} = tt$. We have $thread_s(apid_s) \approxeq_L thread_{s'}(apid_{s'})$. Moreover, for all pid with $pid \neq apid_{t'}$ holds $thread_{s'}(pid) = thread_{t'}(pid)$. Finally, $executed$ and $ainfo$ are updated in the same way for both s and s'. Therefore, setting $\alpha' = \delta'.e'$ gives $s_0 \xRightarrow{\beta.\delta'} t' \xrightarrow{e'} s'$ such that $Inv(\alpha, \alpha', s, s', i+1)$.

$first(thread_t(apid_t)) =$ **fork** We have $executed_t = \mathit{ff}$, $apid_t \neq \bot$, $live(t, apid_t)$, and $t \xrightarrow{e} s$ hold. By Lemma 2 we obtain $\langle\!| thread_t(apid_t), mem_t |\!\rangle \twoheadrightarrow \langle\!| thread_s(apid_t.0) \ldots thread_s(apid_t.n), mem_s |\!\rangle$ where $n \in \mathbb{N}$ is the maximal natural number for which $thread_s(apid_t.n) \neq \bot$ holds. By the induction hypothesis, we have $mem_t =_L mem_{t'}$ and $thread_t(apid_t) \approxeq_L thread_{t'}(apid_{t'})$ (note that $apid_t = apid_{t'}$). Unwinding the definition of strong low-bisimulation (Definition 6) yields that there exists a command C', a command sequence $\vec{D} = \langle D_1 \ldots D_n \rangle$, and a memory mem' such that $\langle\!| thread_{t'}(apid_{t'}), mem_{t'} |\!\rangle \twoheadrightarrow \langle\!| C'\vec{D}, mem' |\!\rangle$ where $thread_s(apid_t.0) \approxeq_L C'$, $thread_s(apid_t.1) \approxeq_L D_1, \ldots, thread_s(apid_t.n) \approxeq_L D_n$ and $mem_s =_L mem'$. According to the induction hypothesis, we have $executed_{t'} = \mathit{ff}$, $apid_{t'} \neq \bot$, and $live(t', apid_{t'})$. Thus, we can apply Lemma 3. By Lemma 3 there is an event $e' \in E^{\text{MWL}}_{local}$ and an SES-state s' with $t' \xrightarrow{e'} s'$, $apid_{s'} = apid_{t'}.0$, $thread_{s'}(apid_{t'}) = \top$, $mem_{s'} = mem'$ and $executed_{s'} = tt$. Moreover, $thread_{s'}(apid_{t'}.0) = C'$ and $thread_{s'}(apid_{t'}.1) = D_1, \ldots, thread_{s'}(apid_{t'}.n) = D_n$. Further, for all pid with $pid \notin \{apid_{t'}, apid_{t'}.0, \ldots, apid_{t'}.n\}$ holds $thread_{s'}(pid) = thread_{t'}(pid)$. Finally, $executed$ and $ainfo$ are updated in the same way for both s and s'. Therefore, setting $\alpha' = \delta'.e'$ gives $s_0 \xRightarrow{\beta.\delta'} t' \xrightarrow{e'} s'$ such that $Inv(\alpha, \alpha', s, s', i+1)$.

□

5.6.2 Completeness

Let us first recall some facts from standard bisimulation theory before we turn to proving completeness. Restating Definition 6, two thread pools $\vec{C} = \langle C_1 \ldots C_n \rangle$ and $\vec{D} = \langle D_1 \ldots D_n \rangle$ are strongly low-bisimilar $\vec{C} \approxeq_L \vec{D}$

iff $\exists R.\, R \subseteq F(R)$ where function F from pers to pers (partial equivalence relations over $\vec{\mathtt{CMD}}$) is given by: $\vec{C}\ F(R)\ \vec{D}$ iff

$$\begin{aligned}&\forall mem_1, mem_2, i.\langle\!|C_i, mem_1|\!\rangle \twoheadrightarrow \langle\!|\vec{C}', mem_1'|\!\rangle \wedge\ mem_1 =_L mem_2 \Longrightarrow \\ &\qquad \exists \vec{D}', mem_2'.\langle\!|D_i, mem_2|\!\rangle \twoheadrightarrow \langle\!|\vec{D}', mem_2'|\!\rangle \wedge\ mem_1' =_L mem_2' \wedge \vec{C}'\ R\ \vec{D}'\end{aligned}$$

Let us state two lemmas that give an alternative representation for the strong low-bisimulation. The proof of the lemmas is a standard argument, by appeal to the Knaster-Tarski fixed-point theorem (see, e.g., [7]).

Lemma 5 *Function F is ω-cocontinuous, i.e., for a non-increasing ω-chain of pers $R_0 \supseteq \ldots \supseteq R_i \supseteq \ldots$, F preserves colimits:*

$$F(\cap_{i<\omega} R_i) = \cap_{i<\omega} F(R_i).$$

Lemma 6 (Fixed point) *The relation $\cong_L$ is the greatest fixed point of F in the lattice of pers. It can be alternatively represented by $\cong_L = \cap_{i<\omega} \cong_L^i$ where $\cong_L^{i+1} = F(\cong_L^i)$ and $\cong_L^0$ is the total relation $\vec{\mathtt{CMD}} \times \vec{\mathtt{CMD}}$.*

We are now ready to present the completeness result.

Theorem 4 (Completeness) *An MWL program C is secure provided the MWL-thread pool $MWLPool(C)$ satisfies the security predicate $SecProp_{TP}$.*

Proof. [Sketch] Assuming that $MWLPool(C)$ satisfies $BSIA_{\{L\},\{H\setminus HI\},\{HI\}}$ we need to show that C is secure, i.e., $C \cong_L C$ (by Definition 7). Let us prove this statement by contraposition. In other words, assuming $C \not\cong_L C \wedge SecProp_{TP}(MWLPool(C))$ we aim to arrive at a contradiction.

By Lemma 6 $C \cong_L C \iff C(\cap_{i<\omega} \cong_L^i)C$. Assuming $C \not\cong_L C$ implies $\exists i. C \not\cong_L^i C$. Take $k = \min\{i \mid C \cong_L^i C \wedge C \not\cong_L^{i+1} C\}$. Note that $k \geq 0$ since, obviously, $C \cong_L^0 C$. Assume for simplicity that no fork-command occurs in C, i.e., C never spawns new threads. Along the way, we discuss how the proof can be modified to go through without the assumption. We consider two sequences of transitions of the form given in Figure 5.12. Note that each element of the sequences inherits the command in the configuration from the previous element. Observe that the low parts of the memory progress in both sequences in the same way. The sequences continue as shown in Figure 5.13. These sequences must exist due to $\forall i \in \{0 \ldots k\}. C \cong_L^i C$ and $C \not\cong_L^{k+1} C$. Matching the first k steps in both sequences and the low-equivalence of the memories during the first k steps

$$\begin{array}{ll}
\text{Seq1: } \langle C,(h_0,l_0)\rangle \twoheadrightarrow \langle C_1,(\hat{h}_1,\hat{l}_1)\rangle & \langle C_1,(h_1,l_1)\rangle \twoheadrightarrow \langle C_2,(\hat{h}_2,\hat{l}_2)\rangle \quad \langle C_2,(h_2,l_2)\rangle \twoheadrightarrow \langle C_3,(\hat{h}_3,\hat{l}_3)\rangle \\
\text{Seq2: } \langle C,(h'_0,l_0)\rangle \twoheadrightarrow \langle D_1,(\hat{h}'_1,\hat{l}_1)\rangle & \langle D_1,(h'_1,l_1)\rangle \twoheadrightarrow \langle D_2,(\hat{h}'_2,\hat{l}_2)\rangle \quad \langle D_2,(h'_2,l_2)\rangle \twoheadrightarrow \langle D_3,(\hat{h}'_3,\hat{l}_3)\rangle
\end{array}$$

Figure 5.12: Sequences Seq1 and Seq2

$$\begin{array}{ll}
\text{Seq1: } \ldots\langle C_{k-1},(h_{k-1},l_{k-1})\rangle \twoheadrightarrow \langle C_k,(\hat{h}_k,\hat{l}_k)\rangle & \langle C_k,(h_k,l_k)\rangle \twoheadrightarrow \langle C_{k+1},(\hat{h}_{k+1},\hat{l}_{k+1})\rangle \\
\text{Seq2: } \ldots\langle D_{k-1},(h'_{k-1},l_{k-1})\rangle \twoheadrightarrow \langle D_k,(\hat{h}'_k,\hat{l}_k)\rangle & \langle D_k,(h'_k,l_k)\rangle \not\twoheadrightarrow \langle D_{k+1},(\hat{h}'_{k+1},\hat{l}_{k+1})\rangle
\end{array}$$

Figure 5.13: The continuation of Seq1 and Seq2

$$\begin{aligned}
\gamma =& schedule(pid_0).setvar(l,l_0).setvar(h,h_0).e_1.yield(info_1).outvar(l,\hat{l}_1).\\
& schedule(pid_1).setvar(l,l_1).setvar(h,h_1).e_2.yield(info_2).outvar(l,\hat{l}_2).\ldots\\
& \ldots schedule(pid_{k-1}).setvar(l,l_{k-1}).setvar(h,h_{k-1}).e_k.yield(info_k).outvar(l,\hat{l}_k).\\
& schedule(pid_k).setvar(l,l_k).setvar(h,h_k).e_{k+1}.yield(info_{k+1}).outvar(l,\hat{l}_{k+1})
\end{aligned}$$

Figure 5.14: Sequence γ

$$\begin{aligned}
\gamma' =& schedule(pid_0).setvar(l,l_0).e'_1.yield(info_1).outvar(l,\hat{l}_1).\\
& schedule(pid_1).setvar(l,l_1).e'_2.yield(info_2).outvar(l,\hat{l}_2).\ldots\\
& \ldots schedule(pid_{k-1}).setvar(l,l_{k-1}).e'_k.yield(info_k).outvar(l,\hat{l}_k).\\
& schedule(pid_k).setvar(l,l_k).e'_{k+1}.yield(info_{k+1}).outvar(l,\hat{l}_{k+1})
\end{aligned}$$

Figure 5.15: Sequence γ'

$$\begin{aligned}
\gamma'' =& schedule(pid_0).setvar(l,l_0).setvar(h,h'_0).e''_1.yield(info_1).outvar(l,\hat{l}_1).\\
& schedule(pid_1).setvar(l,l_1).setvar(h,h'_1).e''_2.yield(info_2).outvar(l,\hat{l}_2).\ldots\\
& \ldots schedule(pid_{k-1}).setvar(l,l_{k-1}).setvar(h,h'_{k-1}).e''_k.yield(info_k).outvar(l,\hat{l}_k).\\
& schedule(pid_k).setvar(l,l_k).setvar(h,h'_k).e''_{k+1}.yield(info_{k+1}).outvar(l,\hat{l}_{k+1})
\end{aligned}$$

Figure 5.16: Sequence γ''

$$\begin{array}{l}
\text{Seq2: } \langle C,(h'_0,l_0)\rangle \twoheadrightarrow \langle D_1,(\hat{h}'_1,\hat{l}_1)\rangle \quad \langle D_1,(h'_1,l_1)\rangle \twoheadrightarrow \langle D_2,(\hat{h}'_2,\hat{l}_2)\rangle \quad \langle D_2,(h'_2,l_2)\rangle \twoheadrightarrow \langle D_3,(\hat{h}'_3,\hat{l}_3)\rangle \\
\quad \ldots\langle D_{k-1},(h'_{k-1},l_{k-1})\rangle \twoheadrightarrow \langle D_k,(\hat{h}'_k,\hat{l}_k)\rangle \quad \langle D_k,(h'_k,l_k)\rangle \twoheadrightarrow \langle D_{k+1},(\hat{h}_{k+1},\hat{l}_{k+1})\rangle
\end{array}$$

Figure 5.17: New form of Seq2

are guaranteed by $\forall i \in \{0 \ldots k\}. C \cong_L^i C$. However, at step $k+1$ we have $\forall D_{k+1}, \hat{l}'_{k+1}. \langle\!| D_k, (h'_k, l_k) |\!\rangle \twoheadrightarrow \langle\!| D_{k+1}, (\hat{h}'_{k+1}, \hat{l}'_{k+1}) |\!\rangle \implies \hat{l}_{k+1} \neq \hat{l}'_{k+1}$. In case C may spawn new threads, the difference is that instead of inheriting the commands from the previous element in the sequences Seq1 and Seq2, the next command is chosen from the command in the previous configuration by selecting the thread that is the counterexample for the low-bisimulation of thread pools obtained at the previous step. Importantly, the sequences of *pid*'s chosen in both Seq1 and Seq2 are then identical. We will use this observation later.

We proceed by constructing two traces of $MWLPool(C)$ that correspond to the two sequences. We will transform one trace into the other using $SecProp_{TP}$ such that the low-equivalences and step matching is preserved. This will take us to a contradiction at step $k+1$. Start off by constructing a trace of $MWLPool(C)$ that corresponds to Seq1. Similarly to the proof of Theorem 3, we appeal to Lemma 3 to obtain step-by-step construction of a trace γ of the form given in Figure 5.14 for some $pid_0, \ldots, pid_k, info_1, \ldots, info_{k+1}$ where each e_i $(i = 1, \ldots, k+1)$ is the internal event that corresponds to the $\twoheadrightarrow$-transition in Seq1 according to Lemma 3. In case no threads are spawned $pid_i = 0$ for all $i = 0, \ldots, k$. As we noted, in case C may spawn new threads the sequences of *pid*'s chosen in both Seq1 and Seq2 are identical. By a similar argument the information contained in *info* sequences must also be identical for Seq1 and Seq2 up to $info_k$.

Due to $SecProp_{TP}$ we can insert high events into right tails of γ that do not contain any high events. We get a legitimate trace after the insertion. Let us insert the $setvar(h, \hat{h}_k)$ event between $setvar(h, h_k)$ and e_{k+1} in β. Define $c = setvar(h, \hat{h}_k)$ $\alpha = e_{k+1}.yield(info_{k+1}).outvar(l, \hat{l}_{k+1})$ and $\beta = \delta.setvar(h, h_k)$ for some δ such that $\gamma = \beta.\alpha$. We have $\alpha|_{HI} = \langle\rangle$. By $SecProp_{TP}$ we have $\exists \alpha'. \alpha'|_L = \alpha|_L \wedge \alpha'|_{HI} = \langle\rangle \wedge \delta.setvar(h, h_k).setvar(h, \hat{h}_k).\alpha' \in Tr$. Observe that setting h to $\hat{h}_k$ means restoring the value of h from the result of the previous transition in Seq1. We can just as well omit both updates of h implying $\delta.\alpha' \in Tr$.

Carrying on with the elimination of the rightmost event $setvar(h, h_i)$ for $i = k, \ldots, 0$ we get a trace $\gamma' \in Tr$ that (after removing all occurrences of $outvar(h, \cdot)$-events without loss of generality) has the form depicted in Figure 5.15. Due to the low events embracing each local event e_i for $i = 1, \ldots, k+1$ it must be the case that there is a one-to-one correspondence between e_i and e'_i for $i = 1, \ldots, k+1$ (although they are not necessarily identical).

The next pass is the insertion of $setvar(h, \cdot)$ following the sequence Seq2. Let us construct new versions of c, α, β in order to apply $SecProp_{TP}$. Let $c = setvar(h, h'_0)$, $\beta = schedule(pid_0).setvar(l, l_0)$ and α is such that $\gamma' = \beta.\alpha$. By $SecProp_{TP}$ we have $\exists \alpha'. \alpha'|_L = \alpha|_L \wedge \alpha'|_{HI} = \langle\rangle \wedge \beta.setvar(h, h'_0).\alpha' \in Tr$. Continuing rightmost $setvar(h, h'_i)$ $(i = 0, \ldots, k+1)$ insertion we get a trace $\gamma'' \in Tr$ that (again, after removing all occurrences of $outvar(h, \cdot)$-events without loss of generality) has the form depicted on Figure 5.16. Due to the low *schedule*- and *yield*-events embracing each local event e'_i $(i = 1, \ldots, k+1)$ it must be the case that there is a one-to-one correspondence between e'_i and e''_i for $i = 1, \ldots, k+1$ (although they are not necessarily identical).

According to Lemma 2 we can now convert the trace γ'' into a sequence of $\twoheadrightarrow$-transitions. The crucial property is that these transitions are deterministic, i.e., if $\langle E, mem \rangle \twoheadrightarrow \langle E', mem' \rangle$ then $\forall E'', mem''. \langle E, mem \rangle \twoheadrightarrow \langle E'', mem'' \rangle \Longrightarrow E' = E'' \wedge mem' = mem''$. In case C may spawn processes, we also need the observation we made about the same sequences of *pid*'s and *info*'s that are used in the construction of Seq1 and Seq2. This is important to restore the branching behavior of traces as it was in Seq1 and Seq2. The fact that only programs with the same branching structure can be low-bisimilar is reflected in the traces, because the branching behavior is recorded in the low *schedule*-events. Thus, by induction, we can restore the sequence Seq2, as depicted in Figure 5.17 for some command D_{k+1}, which contradicts our original assumption about Seq2. □

5.7 Discussion and Future Work

Contributions. We have established a one-to-one correspondence between a time-sensitive definition of security for the multi-threaded programs of MWL (from [27]) and a security property based on traces of events that was originally developed in the context of a general security framework – the assembly kit (from [18, 19]). As a prerequisite for this, we had to model the semantics of MWL using state-event systems, which resulted in the specification of MWL thread pools. The development of this specification has been straightforward (although technically subtle). To us, it is appealing that generic thread pools, which served as an intermediate step in this process, are independent of MWL. We expect that this will allow the adaptation of other multi-threaded programming languages, e.g., Slam [15].

The main motivation of our work has been the objective to integrate the two kinds of security: the security of local computations and the security of their communications. Event-based security aims at protecting occurrences of events and programming-language-based security aims at protecting secret values. Our work is a step to aid in the systematic security analysis of complex (potentially distributed) system where some of the components are (or shall be) implemented in a specific programming language. To the best of our knowledge this article is the first attempt to establish a rigorous connection between these two notions of security. The connection suggests directions for mutual benefits where the two areas can borrow from each other (cf. Future Work).

As a side effect, we have demonstrated how to use the assembly kit [18] at the concrete example of the multi-threaded programming language MWL. Using the assembly kit has turned out to be very helpful in the identification of an appropriate security property. This application is also interesting because it shows how time-sensitive security can be specified in the assembly kit. For a different technique to address timing channels by explicit *tick*-events we refer to [10].

Bisimulation vs Trace-based Equivalence. The reader familiar with transition-system-based semantics might be surprised by the fact that the article relates a bisimulation-based property of programs with a trace-based one. It is well-known that small-step bisimulation makes more distinctions than trace-based equivalence. It is also well-known that trace-based properties are usually not compositional whereas bisimulation-based ones often are. Nevertheless, we have been able to prove correctness and completeness results for our translation of the security property.

What made these developments possible in the presence of the two major differences between the bisimulation-based and trace-based models? The crucial property is the deterministic nature of strong low-bisimulation. Indeed, bisimulation is defined on deterministic transitions ensuring that two bisimilar thread pools have the same branching behavior.[2] This property is necessary for guaranteeing *scheduler-independent security.* Two programs have to have identical branching behavior in order to be indistinguishable for the attacker under a scheduler-independent low-bisimulation. Otherwise, the two programs in the then and else branches, respectively, of an if

[2]Technically one can view the identifier *pid* of a thread that is chosen by the scheduler for the next transition as a label that is distinguished by the strong low-bisimulation.

statement with a secret condition could be used to leak the secret condition through observing the branching behavior ([27] shows how to implement this attack using the properties of a particular scheduler).

Although the determinism of bisimulation is the key feature to relating bisimulation-based and trace-based models, it is not crucial for the actual security definition of MWL (certain security properties can be defined through low determinism, as in [25, 24]). For example, if MWL had a non-deterministic choice operator $[]$ then the non-deterministic program $l := 0 \; [] \; l := 1$ would be considered secure under Definition 7. However, the two security definitions (Definition 7 and Definition 2 for MWL thread pools) would be no longer equivalent. Indeed, at no surprise, the completeness theorem (Theorem 4) would not hold. A counterexample is the program $\mathsf{if}\ h = 0\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where $C_1 = l := 0; (l := 1 \; [] \; l := 2)$ and $C_2 = (l := 0; l := 1) \; [] \; (l := 0; l := 2)$. This program is considered secure under the trace-based model (Definition 2) but not secure according to the bisimulation-based Definition 7. Note that, whether one intuitively considers this program as secure or not depends very much on the model of computation one has in mind. For a detailed investigation of this close relation between notions of information flow and models of computation (notions of equivalence) we refer to [26].

Future Work. Plans for future work are centered around exploiting the connection between the two types of security that we have established in the present article. Promising directions include the adaption of intransitive security policies[3] for MWL basing on solutions that were proposed in the context of the assembly kit [20] and progress towards a development method that allows for the stepwise development starting from abstract specifications and ending with concrete programs (cf. [21] for recent progress on the refinement of information flow properties).

Another potentially interesting direction of research is to apply the reduction techniques of [27] combined with the results of this paper to reasoning about probabilistic security properties for event-based systems. Moreover, the integration of synchronization primitives for communication between multiple thread pools together with an according strengthening of the security predicate is another important goal for future work because it is a prerequisite for applying the results of this paper to truly distributed systems.

[3]Intransitive flow policies would provide a way to represent downgrading (and thus, e.g., secure encryption) in the multi-threaded while-language.

Acknowledgments

Thanks are due to Fabio Martinelli for encouraging discussions. We also wish to thank David Sands and Koen Claessen for useful feedback on bisimulation proofs. For useful comments on the presentation, we would like to thank Alexandra Heidger and Axel Schairer.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, January 1999.

[2] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, January 2000.

[3] J. Agat and D. Sands. On Confidentiality and Algorithms. In *Proceedings of 2001 IEEE Symposium on Security and Privacy*, May 2001. To appear.

[4] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.

[5] Ellis S. Cohen. Information Transmission in Computational Systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[6] Ellis S. Cohen. Information Transmission in Sequential Programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[7] B. Davey and H. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[8] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[9] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[10] R. Focardi, R. Gorrieri, and F. Martinelli. Information Flow Analysis in a Discrete-Time Process Algebra. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 170–184, July 3–5 2000.

[11] Riccardo Focardi and Roberto Gorrieri. A Classification of Security Properties for Process Algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[12] Simon N. Foley. A Universal Theory of Information Flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 116–122, Oakland, CA, April 27–29 1987.

[13] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, April 26–28 1982.

[14] J.D. Guttman and M.E. Nadel. "What Needs Securing?". In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 34–57, June 12-15 1988.

[15] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[16] Dale M. Johnson and F. Javier Thayer. Security and the Composition of Machines. In *Proceedings of the Computer Security Foundations Workshop*, pages 72–89, Franconia, NH, June 1988.

[17] K. R. M. Leino and Rajeev Joshi. A Semantic Approach to Secure Information Flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[18] Heiko Mantel. Possibilistic Definitions of Security – An Assembly Kit –. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, UK, July 3–5 2000.

[19] Heiko Mantel. Unwinding Possibilistic Security Properties. In F. Cuppens, Y. Deswarte, D. Gollmann, and M. Waidner, editors, *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, LNCS 1895, pages 238–254, Toulouse, France, October 4-6 2000. Springer.

[20] Heiko Mantel. Information Flow Control and Applications – Bridging a Gap –. In Jose Nuno Olivera and Pamela Zave, editors, *Proceedings of FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, LNCS 2021, pages 153–172, Berlin, Germany, March 12-16 2001. Springer.

[21] Heiko Mantel. Preserving Information Flow Properties under Refinement. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 13-16 2001. To appear.

[22] Daryl McCullough. Specifications for Multi-Level Security and a Hook-Up Property. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 161–166, Oakland, CA, April 27–29 1987.

[23] John McLean. A General Theory of Composition for Trace Sets Closed under Selective Interleaving Functions. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 79–93, Oakland, CA, May 16–18 1994.

[24] A. Roscoe. CSP and Determinism in Security Modelling. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 114–127, Oakland, CA, May 1995.

[25] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through Determinism. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, LNCS 875, pages 33–53, Brighton, UK, November 7–9 1994. Springer.

[26] P.Y.A. Ryan and S.A. Schneider. Process Algebra and Non-interference. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 214–227, Mordano, Italy, June 28–30 1999.

[27] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 200–215, Cambridge, UK, July 3–5 2000.

[28] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[29] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, January 1998.

[30] D. Sutherland. A Model of Information. In *9th National Computer Security Conference*, September 1986.

[31] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security*, 7(2,3):231–253, November 1999.

[32] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):1–21, 1996.

[33] J. Todd Wittbold and Dale M. Johnson. Information Flow in Nondeterministic Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 144–161, Oakland, CA, May 1990.

[34] Aris Zakinthinos and E.S. Lee. A General Theory of Security Properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 94–102, Oakland, CA, May 4–7 1997.