Compositional and Scheduler-Independent Information Flow Security

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

von Diplom-Mathematiker Henning Sudbrock
aus Darmstadt

zur Erlangung des akademischen Grades eines
Doctor rerum naturalium (Dr. rer. nat.)

Referenten der Arbeit:	Prof. Dr. Heiko Mantel
	Prof. Dr. Peter H. Schmitt
Tag der Einreichung:	24. März 2013
Tag der mündlichen Prüfung:	12. Juni 2013

Darmstadt, 2014
Hochschulkennziffer D 17

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen übernommenen Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Darmstadt, im März 2013

Abstract

Software pervades our society deeper with every year. This trend makes software security more and more important. For instance, software systems running critical infrastructures like power plants must withstand criminal or even terrorist attacks, but also smartphone apps used by consumers in their daily routine are usually expected to operate securely.

In particular, before entrusting a program with confidential information (such as, e.g., image or audio data recorded by a smartphone), one wants to be sure that the program is trustworthy and does not leak the secrets to untrusted sinks (such as, e.g., an untrusted server on the Internet). Information flow properties characterize such confidentiality requirements by restricting the flow of confidential information, and an information flow analysis permits to check that a program respects those restrictions.

The problem of information flow in multi-threaded programs is particularly challenging, because information flows can originate in subtle ways from the interplay between threads. Moreover, the existence of such information flows depends on the scheduler, which might not even be known when analyzing a program. To obtain high assurance that no leak is overlooked in an information flow analysis, formally well-founded analyses provide a rigorous solution. Such analyses are proven sound with respect to formal information flow properties that specify precisely what restrictions on information flow mean.

In this thesis, we develop two novel information flow properties for multi-threaded programs, *FSI-security* and *SIFUM-security*. These properties are scheduler-independent, i.e., they characterize secure information flow for different schedulers simultaneously. Moreover, they are compositional, i.e., they permit to break down the analysis of a multi-threaded program to single threads. For both properties we develop a security analysis based on a security type system that is proven sound with respect to the property.

Compared to existing scheduler-independent information flow properties, FSI-security is less restrictive. In particular, FSI-security is the first scheduler-independent information flow property that permits programs with nondeterministic behavior and programs whose control flow depends on secrets. The security analysis based on SIFUM-security is the first provably sound flow-sensitive information flow analysis for multi-threaded programs in the form of a security type system. Flow-sensitivity results in increased analysis precision by taking the order of program statements into account. The key in our development of SIFUM-security and the corresponding flow-sensitive analysis for multi-threaded programs was to adopt assumption-guarantee style reasoning to information flow security.

We integrate FSI-security and SIFUM-security into the novel property *FSIFUM-security*, and we integrate the security analyses for FSI- and SIFUM-security into a security analysis for FSIFUM-security. Thereby, FSIFUM-security and the corresponding analysis inherit the advantages of both FSI- and SIFUM-security.

In addition to developing novel type-based information flow analyses we also explore information flow analysis for multi-threaded programs with program dependence graphs (PDGs) which is used successfully to analyze sequential programs. To this end, we develop a formal connection between PDG-based and type-based information flow analysis for sequential programs. We exploit the connection to transfer concepts from our type-based analysis for multi-threaded programs to PDGs, resulting in a provably sound PDG-based information flow analysis for multi-threaded programs. Beyond this, we also use the connection to transfer concepts from PDGs to type systems and to precisely compare the precision of a type-based and a PDG-based information flow analysis.

Our results provide foundations for more precise and more widely applicable information flow analysis for multi-threaded programs, and we hope that they contribute to a more wide-spread certification of information flow security for concurrent programs.

Zusammenfassung

Software durchdringt unsere Gesellschaft immer tiefer. Dadurch wird Softwaresicherheit immer wichtiger. Beispielsweise müssen Softwaresysteme in kritischen Infrastrukturen kriminellen oder gar terroristischen Attacken standhalten, aber auch von Smartphone-Apps wird ein bestimmtes Sicherheitsniveau erwartet.

Bevor man einem Programm vertrauliche Informationen wie beispielsweise Telefonats-Daten anvertraut möchte man insbesondere sicher sein, dass das Programm die Geheimnisse nicht ungefragt an Dritte beispielsweise im Internet weitergibt. Informationsflusseigenschaften beschreiben solche Vertraulichkeitsanforderungen, indem sie den Fluss vertraulicher Informationen einschränken. Informationsflussanalysen ermöglichen die Überprüfung von Programmen bezüglich dieser Einschränkungen.

Die Informationsflussanalyse nebenläufiger Programme ist besonders schwierig, da durch das Zusammenspiel nebenläufiger Programmteile auf subtile Art und Weise Informationsflüsse entstehen können. Darüber hinaus hängt die Existenz solcher Flüsse vom Scheduler ab, der bei der Analyse möglicherweise unbekannt ist. Um Gewissheit zu erlangen, dass kein Fluss übersehen wird, bieten formal fundierte Analysen eine rigorose Lösung. Solche Analysen garantieren beweisbar sicher, dass Programme einer formalen Informationsflusseigenschaft genügen welche Informationsfluss präzise spezifiziert.

In dieser Arbeit entwickeln wir zwei neue Informationsflusseigenschaften für nebenläufige Programme, *FSI-Security* und *SIFUM-Security*. Diese Eigenschaften sind scheduler-unabhängig, d.h., sie beschreiben Informationsflusssicherheit für verschiedene Scheduler gleichzeitig. Außerdem sind sie kompositional, d.h., sie erlauben es, die Analyse eines Programms auf kleinere Teilprogramme herunterzubrechen. Für beide Eigenschaften entwickeln wir eine Sicherheitsanalyse basierend auf einem Sicherheitstypsystem und beweisen die Korrektheit der Analyse gegenüber der jeweiligen Sicherheitseigenschaft.

FSI-Security ist die erste scheduler-unabhängige Informationsflusseigenschaft die nicht-deterministische Programme sowie Programme mit geheimen Kontrollbedingungen erlaubt. Die Analyse für SIFUM-Security ist die erste beweisbar sichere flusssensitive Informationsflussanalyse für nebenläufige Programme in Form eines Sicherheitstypsystems. Flusssensitivität ermöglicht höhere Präzision indem die Reihenfolge von Programmanweisungen berücksichtigt wird. Der Schlüssel für die Entwicklung von SIFUM-Security war die Verwendung von “Annahmen und Garantien” (“assumptions and guarantees“).

Wir integrieren FSI-Security und SIFUM-Security zur neuen Eigenschaft *FSIFUM-Security*, und wir integrieren die Sicherheitsanalysen für FSI-Security und SIFUM-Security zu einer Sicherheitsanalyse für FSIFUM-Security. Dadurch erben FSIFUM-Security und die zugehörige Analyse die Vorteile von sowohl FSI- als auch SIFUM-Security.

Neben der Entwicklung typ-basierter Analysen betrachten wir auch Analysen für nebenläufige Programme mittels Programmabhängigkeitsgraphen (PDGs), die erfolgreich zur Analyse sequentieller Programme eingesetzt werden. Wir entwickeln eine formale Verbindung zwischen PDG-basierter und typ-basierter Informationsflussanalyse für sequentielle Programme. Darauf basierend überführen wir Konzepte der typ-basierten Analysen für nebenläufige Programme in die Welt der PDGs, was in einer beweisbar korrekten PDG-basierten Informationsflussanalyse für nebenläufige Programme resultiert. Wir nutzen die Verbindung auch, um Konzepte von PDGs in einer typ-basierten Analyse zu verwenden, und um die Präzision einer typ- und einer PDG-basierten Analyse präzise zu vergleichen.

Unsere Resultate bieten die formalen Grundlagen für präzisere und weitreichender einsetzbare Informationsflussanalysen für nebenläufige Programme. Wir hoffen, damit zu einer weiter verbreiteten Zertifizierung von Informationsflusssicherheit nebenläufiger Programme beizutragen.

Publications

We have published excerpts from this thesis as follows.

The scheduler-independent security property FSI-security and the corresponding security type system (Chapter 3) were presented in [MS10]. The scheduler independence result in [MS10] is with respect to a slightly simpler scheduler-dependent security property than in this thesis which does not take inputs and outputs into account. The investigation of the probabilistic Lottery schedulers was not included in [MS10].

The flow-sensitive security type system and the underlying assumption-guarantee based security property, SIFUM-security (Chapter 4), were presented in [MSS11]. The article did not yet contain the scheduler-independence result for SIFUM-security presented in this thesis. Moreover, the article did not contain a security type system that exploits assumptions for tracking values in single threads (developed in Section 4.5.4). The combination of FSI-security and SIFUM-security (Chapter 5) is published in this thesis for the first time.

The investigation of the relation between type-based and PDG-based information flow analyses, comprising the extension of PDG-based analyses to multi-threaded programs (Chapter 6), was presented in [MS13].

Additional refereed publications produced during my PhD studies have not been incorporated into this thesis. One article ([MSK07]) investigates the combination of different information flow analyses for analyzing the information flow security of multi-threaded programs to increase analysis precision, resulting in the idea and an instantiation of the *Combining Calculus*. Other works [MS07, MS09] are concerned with quantitative information flow control in operating systems, investigating formal models of and countermeasures against interrupt-related covert channels.

Acknowledgments

First and foremost, I am very grateful to Heiko Mantel and Peter H. Schmitt, the two reviewers of this thesis. They need to spend both their time and their energy for the review, and I cannot thank them enough for this.

Besides reviewing this thesis, Heiko was also my PhD supervisor. I was lucky to have the opportunity to work in his research group for several years. In particular, many, many fruitful discussions with Heiko and his valuable feedback helped to drive the work presented in this thesis forward. Moreover, Heiko provided a never-ending stream of new and interesting ideas as well as research questions that, on the one hand, provided starting points for new research and, on the other hand, helped to lift existing work to higher levels. This was particularly helpful for the work on scheduler-independent security and on rely-guarantee based security analysis presented in this thesis, which started based on ideas from and evolved in close collaboration with Heiko. In addition to guiding, participating in, and supporting the research Heiko also provided general guidance and support, going beyond my role as research assistant during my time in his research group.

I am also very grateful to David Sands from Chalmers University, Sweden. It was great working together with him and Heiko, developing novel security properties and analyses that are based on rely-guarantee style reasoning. Especially the intense discussions during his visits in Darmstadt resulted in very good progress. David also provided valuable feedback on the work on scheduler-independent information flow security.

Martin Hofmann at LMU München, Peter H. Schmitt, and Gregor Snelting from Karlsruhe Institute of Technology gave me the opportunity to present and discuss results during visits at their chairs, followed by interesting discussions and good feedback.

I thank all my colleagues from the time at RWTH Aachen and TU Darmstadt for lots of interesting discussions about our respective research topics: Markus Aderhold, A. Baskar, Sarah Ereth, Richard Gay, Sylvia Grewe, Jinwei Hu, Tina Kraußer, Alexander Lux, Matthias Perner, Jens Sauer, Dieter Schuster, Heiko Spies, Barbara Sprick, and Artem Starostin. Special thanks go to Alexander, Baskar, and Matthias for proof-reading parts of this thesis. I am especially grateful to Alexander, long-time office mate, with whom I had many good discussions both on a scientific and a personal level. I am also deeply indebted to Artem for his constant yet positive pressure during the last months of this thesis' creation.

Furthermore, I had many interesting contacts with students at RWTH Aachen and TU Darmstadt. Among them, I had good discussions with Daniel Schoepe on proofs of some results of this thesis which Daniel comprehended using the theorem prover Isabelle.

I also thank Gudrun Harris, Elisabeth Johannes, Miriam Rifai-Schön, and Heide Rinnert for their support during their time as secretary of Heiko's research group.

Finally, thanks go to the DFG (German research foundation), which provided funding for parts of the research results that are presented in this thesis under the project FM-SecEng in the Computer Science Action Program (MA 3326/1-2).

Contents

CHAPTER 1

Introduction

> Private information is practically the source of every large modern fortune.
>
> Oscar Wilde

As computer systems become ever more pervasive, people entrust more and more information to them. This includes, in particular, sensitive private information like, for instance, credit card numbers and bank account details, personal messages and photos, or passwords. Moreover, with the growing proliferation of mobile computers like smart phones or tablet PCs, sensible sensor data like GPS-based location information is increasingly entrusted to computer systems. Simultaneously, the number of programs that misuse entrusted information is growing (see, e.g., [Jun11] or [Sop11]).

But how can one be sure that a program is trustworthy enough to be entrusted with confidential information? How can one, for instance, be sure that a program does not leak information about the location of one's smart phone to an untrusted server on the Internet? An analysis of the information flow within an application permits to answer such questions. For instance, one might check that a program that is granted access to both GPS sensor data and to the Internet is implemented in such a way that the program does not send location information to the Internet.

Information flow security has been investigated already since the seventies [Lam73, Den76, Coh77], resulting in the first provably sound information flow analyses for sequential programs in the nineties [VSI96]. For multi-threaded programs, it has turned out that subtle information leaks that do not occur in sequential programs complicate the information flow analysis (e.g., [SS00]). In fact, information flow analyses for multi-threaded programs are usually less satisfactory than such analyses for sequential programs, considering, e.g., precision and efficiency. Even more, already the semantic characterizations of information flow security that provide a basis for modular analyses of multi-threaded programs are less satisfactory, which hinders the improvement of the analyses.

The goal of this thesis is to improve the formal foundations of information flow analysis for multi-threaded programs. This includes improving the state of the art of semantic characterizations of information flow as well as the development of novel information flow analyses. In the remainder of this introduction, we discuss information flow security in more detail, with a focus on information flow security for multi-threaded programs (Sections 1.1 and 1.2). Moreover, we concretize the goal of this thesis and give an overview of the contributions and the organization of this thesis in Sections 1.3–1.4.

1.1 Language-based Information Flow Security

Confidential information processed by computer systems needs to be protected from attackers that try to gain knowledge about this information. Given a computer system, the protection of information can be, roughly, categorized into three parts: (a) protection outside the system when communicating with other systems, (b) protection at the system border when information enters or leaves the system, and (c) protection inside the system when information is processed. Protection outside the system is usually based on *cryptography*. For instance, if information is transmitted between two systems over an untrusted channel, encryption can be employed to ensure that the information remains confidential. *Access control* is used to ensure protection at the system border. This includes, for instance, password-based mechanisms ensuring that system interfaces are only accessed by authorized entities. Protecting information inside the system means ensuring that the system treats secret information in an appropriate way. For instance, a system with legitimate access to confidential information and with access to the Internet should never leak the confidential information to untrusted servers on the Internet.

Traditionally, the focus has been primarily on protecting information at the border of the system and outside the system, which was quite successful using access control and cryptography. However, the need to also ensure a secure treatment of information within a system is illustrated by frequent reports about programs that provide useful functionality and are therefore permitted to access confidential information, but that abuse their permissions to maliciously leak confidential information [Sym11, Bun11]. In allusion to Greek mythology [HomBC], such malicious programs are called *Trojan Horses*.

Information flow control is an approach to ensure that secret information is appropriately treated within a system. To this end, one controls where confidential information flows within a system, restricting, for instance, the flow to untrusted information sinks.[1]

Semantic Characterization of Secure Information Flow. Some information flows in a system are rather obvious from looking at the system's implementation or a more abstract system specification, for instance, when information is directly copied from one variable to another variable. However, information flows may also be quite subtle, for instance, if it depends on some piece of information whether another piece of information is copied. In particular, systems often contain channels over which information might flow that are not intended for communication, called *covert channels* [Lam73]. In consequence, there is danger to overlook information flows when inspecting a system. Semantically well-founded information flow analyses contribute to mitigate this danger. Such analyses

[1] Besides confidentiality, information flow control is also useful for ensuring *integrity* of information by controlling the flow of untrusted information, ensuring that trusted data is not "contaminated" by untrusted information. The third aspect of the traditional "CIA triad" of information security, *availability*, is to our knowledge not approached with information flow control.

are based on semantic characterizations of secure information flow that provide a precise reference point for what secure information flow means.

To our knowledge, the first semantical characterization of secure information flow is Cohen's *Strong Dependency* [Coh77], whereas the probably best known semantical characterization is *Noninterference* by Goguen and Meseguer [GM82]. Noninterference expresses the absence of information flow from secrets to publicly observable outputs by a lack of dependency of these outputs on the secrets. A system satisfies noninterference if public outputs do not differ for system executions with different secret inputs. Based on the idea of noninterference, various semantical characterizations of information flow as well as analyses for these characterizations were developed, both on the level of concrete implementations of programs as well as on the level of more abstract system specifications (see, e.g., [SM03] for an overview concerning concrete implementations and [FG95, Man00] for overviews concerning more abstract specifications).

Language-based Security. In this thesis, we investigate information flow security for confidentiality properties on the level of program implementations. This is part of the area of *language-based security*, where information security is characterized, analyzed, and enforced using techniques from the area of programming languages. A benefit of language-based security is that program code provides a rather clear specification of a program's behavior. This specification can be used as a basis both for the characterization of security and for the program security analysis. Moreover, language-based security benefits from existing technology for programming languages, comprising program semantics and analysis techniques like, e.g., type systems. Language-based techniques can be applied for different aspects of information security (see [SMH01] for an overview); for instance, Java's byte-code verification is a language-based mechanism for enforcing integrity properties. For information flow security, one identifies how information flows through a program based on the program's implementation (given, e.g., as source code, byte code, or assembly code) and, thereby, to detect possible information leakage.

A program may leak secret information in many ways, which are also referred to as different channels. The simplest way to leak information is by *direct leaks* (also referred to as explicit leaks or leakage via direct channels) where information is leaked, for instance, by copying a secret value directly to an output channel. Such leakage is quite easy to detect (for instance, at runtime using dynamic taint propagation [SAB10]). Slightly more subtle are *indirect leaks* (implicit leaks, leakage via indirect channels). Such leakage occurs when the program decides which action to execute based on secret information, as, for instance, in the simple program "$\mathsf{if}\ (\mathsf{secret} > 0)\ \mathsf{then}\ \mathsf{output}(1)\ \mathsf{else}\ \mathsf{output}(0)$". Multi-threaded programs may leak secret information in even more subtle ways, which we illustrate separately in Section 1.2. What constitutes a leak depends on the observational power of the attacker. Usually, one assumes that the attacker knows the program implementation and can at least see public inputs and public outputs of a program. One might also consider more powerful attackers that can observe, for instance, program execution time, memory consumption, program termination, the number of concurrent threads, power consumption, etc. For example, the program "$\mathsf{if}\ (\mathsf{secret} > 0)\ \mathsf{then}\ \{\mathsf{wait}(5\ \mathsf{sec}); \mathsf{output}(0)\}\ \mathsf{else}\ \mathsf{output}(0)$" is insecure if an attacker can measure execution time (the corresponding channel is referred to as a *timing channel*).

Semantic characterizations of secure information flow that capture all possible leaks for a given attacker model are usually specified with "noninterference-like" properties. A "noninterference-like" property follows the idea of noninterference, but does not coincide with the original noninterference definition by Goguen and Meseguer that is defined on the level of system specifications rather than on the level of programs. Stated informally,

noninterference-like properties are usually as follows:

> "A program has secure information flow if any two program executions with equal public inputs result in the same observations for an attacker (even if the secret inputs to the program differ for the two executions)."

There are many variants of such characterizations, for instance, to capture different attackers (e.g., [Aga00, GM05, AHSS08]), or to enable controlled declassification of secrets (e.g., [MS04, MSZ04, MR07, AS07]).

A semantic characterization of secure information flow sets a standard against which one can check the information flow security of a program. Checking a program against such a characterization by hand is time-consuming and error-prone. Hence, it is desirable to automate the check with a reliable analysis. While noninterference-like information flow properties are usually undecidable, it has turned out that they can be approximated quite well using program analysis techniques like, e.g., type systems (e.g., [VSI96, SS00]), program logics (e.g., [AR80, AB04]), or program dependence graphs (e.g., [HUM92, HS09]). The probably most popular approach have been security type systems, starting with Volpano, Smith, and Irvine [VSI96], who were the first to present a semantic noninterference-like security property together with a corresponding analysis technique linked by a soundness result.

Considering sequential languages, security type systems were implemented for rather complex languages, for instance, in the Jif system [Mye99, MNZZ01] (for a Java-like imperative language) or in the Flow Caml system [PS02, Sim03] (for a functional language). Moreover, analyses based on program dependence graphs were implemented for a subset of Java in the Joana system [GH08]. For multi-threaded languages the situation is less satisfactory. We discuss the particular challenges of information flow control for multi-threaded programs in the following section.

1.2 Information Flow in Multi-threaded Programs

The conceptual complexity of concurrent programs makes it particularly desirable to obtain reliable security guarantees. However, securing the information flow in programs with multiple threads has proven non-trivial. One challenge is that, in addition to the various possibilities to leak information in a sequential program, subtle information leaks may be caused by the interplay between the interleaved executions of multiple threads.

Example 1.1. Consider a program with two threads that are implemented as follows, where x and y are Boolean variables and skip denotes a program statement that performs one execution step, but has no effect on the values of any variables.

if (x = true) then	skip;
skip; skip else	skip;
skip fi;	y:=false
y:=true	

Assume that all variables are shared between the threads. Moreover, assume that the program is executed under a scheduler that selects the threads alternately starting with the left thread, switching to the other thread after each execution step (assignments and the evaluation of control conditions each take, like skip-statements, one execution step).

If initially x = true then the left thread assigns true to y in its fourth execution step, after the right thread assigns false to y in its third execution step. Hence, the final value of y is true. However, if initially x = false then the left thread assigns true to y in its

third execution step, prior to the assignment of false to y in the right thread. Hence, the final value of y is false. In consequence, the program copies the value of x into y. $\Diamond$

Intuitively, there is information flow from variable x to variable y in Example 1.1, because information stored in x influences the number of execution steps in the left thread, which, in turn, influences the order in which the assignments to y are executed. Such information leaks are usually called *internal timing leaks*, indicating that the execution time of threads plays a role in the information flow.

Example 1.1 illustrates two further challenges, concerning the treatment of schedulers and the compositionality of information flow properties.

Scheduling. The program in Example 1.1 does not leak information about the value of variable x into variable y if the scheduler is a Round-Robin scheduler that reschedules after every 50 execution steps. Under such a scheduler, the first thread completes execution before the second thread begins execution, and, hence, the final value of y will be true independently of the information stored in x. This illustrates that the information flow security of multi-threaded programs depends on the scheduler. It turns out that, unlike for many other properties of programs, for information flow security it does not suffice to simply consider nondeterministic scheduling, i.e., to assume for the analysis that all conceivable program executions under any scheduler are possible. The underlying reason is that the scheduler dependence of information flow security is an instance of the *refinement paradox* [Jac89] that states that refinements of a system specification (like, e.g., removing impossible schedules) in general does not preserve information flow security.

One solution to the problem of scheduler dependence is to provide different information flow analyses for different schedulers. However, if the scheduler is not known at analysis time this means that one must perform multiple, potentially numerous, information flow analyses. This problem led to the development of *scheduler-independent* information flow properties that are adequate for a whole class of schedulers (e.g., [SS00, ZM03]). However, this led to rather restrictive properties and, in consequence, to restrictive information flow analyses, forbidding, for instance, that the values of control conditions depend on secret information, or forbidding nondeterministic public program output.

Compositionality. Neither of the threads in Example 1.1, each considered as a program with a single thread, leaks information about the value of variable x into variable y. I.e., since both threads together copy the value of x to y under some schedulers, information flow security is not necessarily compositional with respect to the parallel composition of threads. Such compositionality is, however, desirable, because it permits to reduce the complexity of an information flow analysis. Moreover, compositionality results can be exploited in the development of automated information flow analyses.

The advantages of compositionality motivated the development of compositional information flow properties (e.g., [SS00, BC02, ZM03, BPR07]). However, existing approaches have two shortcomings: In some developments, the compositionality results are restrictive. For instance, [ZM03] requires that the composition of threads does not result in nondeterministic output. In other developments, the information flow properties are restrictive. For instance, the properties in [SS00, BC02, BPR07] are too restrictive for flow-sensitive analyses, a type of analyses that has resulted in improved precision for the information flow analysis of sequential programs, (e.g., [HS06, HS09]).

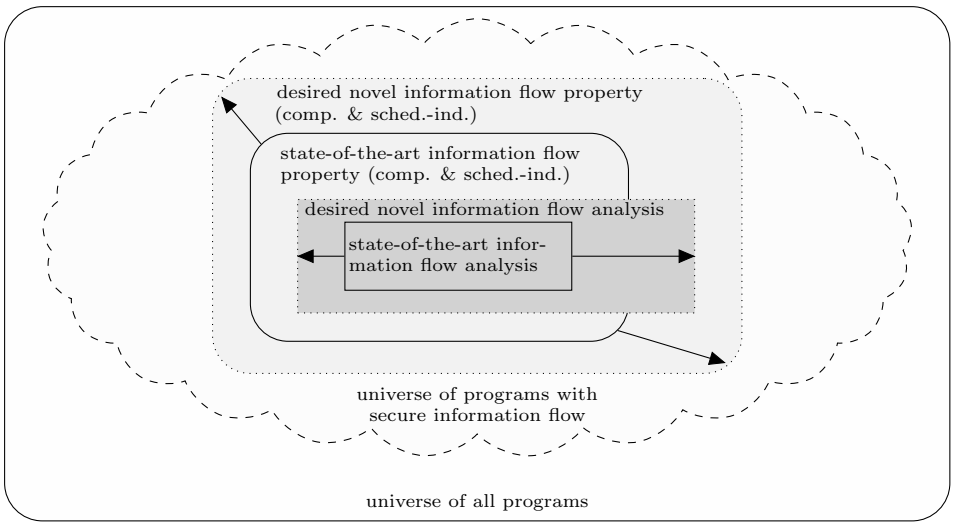


Figure 1.1: Two-step approach to improve information flow analysis

1.3 Improving Information Flow Control for Multi-threaded Programs

Due to the additional challenges for information flow analysis in multi-threaded programs discussed in the previous section, today's information flow analyses for multi-threaded programs are not as satisfactory regarding, e.g., precision, as their counterparts for sequential programs. Moreover, this is not merely a shortcoming of the information flow analyses, but already of the underlying semantic characterizations of information flow.

To improve the state of the art in information flow analysis for multi-threaded programs in this thesis, we therefore follow a two-step approach that starts with the semantic characterization of information flow.

In the first step, we develop novel semantic characterizations of information flow security with two characteristics: Firstly, the novel properties classify a class of multi-threaded programs as secure that is classified as insecure by existing information flow properties although, intuitively, programs in this class have secure information flow. Secondly, the novel properties possess the meta-properties of compositionality and scheduler independence. Figure 1.1 illustrates this step: The cloud indicates the universe of programs that have secure information flow (specified, e.g., by an intuitive notion of information flow security, or by a simple formal notion without the desired meta-properties). The rounded box with solid border indicates the set of programs classified as secure by state-of-the-art compositional and scheduler-independent information flow properties, whereas the rounded box with dashed border indicates this set of programs for the novel properties. (The figure is optimistic in the sense that the novel properties do not necessarily encompass the existing properties completely.)

In the second step, we develop novel provably sound information flow analyses that exploit the improvement in the information flow property from the first step. I.e., the analysis techniques classify programs as secure that are not classified as secure by state-of-the-art information flow properties. Soundness is non-negotiable, i.e., we provide a

soundness result for the novel analysis with respect to the information flow property developed in the first step. Figure 1.1 illustrates this second step by the rectangles that represent sets of programs classified as secure by information flow analyses. Note that the analysis indicated by the larger rectangle is not sound with respect to the property indicated by the smaller rounded box although the analysis only classifies programs as secure that have secure information flow. This illustrates the potential of our approach compared to improving sound analyses based on existing properties.

The remainder of this section sketches how we instantiate these steps in this thesis.

Improving Scheduler-independent Information Flow Analysis. We consider the challenge of scheduler independence in Chapter 3. While many information flow analyses assume a particular scheduler (e.g., a uniform scheduler in [VS99], a Round-Robin scheduler in [RHNS06], or a possibilistic scheduler in [SV98]), there are two main approaches to scheduler-independent semantic characterizations of information flow security. The first approach was introduced by Zdancewic and Myers [ZM03] and adopts the idea of *observational determinism* [RWW94, Ros95] to language-based security. Observational determinism requires that a program's public output is deterministically determined by its public input. The second approach, *strong security* by Sabelfeld and Sands [SS00], requires that the possible behaviors of a program for identical public inputs are stepwise indistinguishable to an observer of public program output. Both approaches provide a semantic basis for an analysis that is independent of the scheduler. However, they are far from satisfactory. Observational determinism forbids nondeterminism in the public output, although intuitively secure programs can have nondeterministic public output. Strong security does not share this deficiency, but implies that a program's execution time must not depend on secrets, even if such timing differences do not influence the program's public outputs.

We developed a compositional and scheduler-independent semantic characterization of information flow security, FSI-security, that (a) permits nondeterminism in public outputs and (b) permits that the execution time of a program depends on secrets (given that the secrets do not influence the program's public outputs). Moreover, in contrast to approaches to scheduler independence by Boudol and Castellani [BC02] and by Russo and Sabelfeld [RS06a], FSI-security does not require a non-standard interface between the scheduler and multi-threaded programs. The existence of such a security property was somewhat surprising in the light of Sabelfeld's result that strong security is the weakest compositional property that implies information flow security for a natural class of schedulers [Sab03]. A key for our development was the identification of a different class of schedulers, the robust schedulers, that also contains many typical schedulers (including, e.g., Round-Robin schedulers and probabilistic schedulers like Lottery schedulers).

We exploit FSI-security in the development of a security type system that is provably sound with respect to FSI-security. The type system successfully exploits the advantages of FSI-security and does not classify programs as insecure only because they produce nondeterministic public output or because their execution time depends on secrets.

Flow-sensitive Information Flow Analysis for Multi-threaded programs. In Chapter 4, we investigate the problem of flow sensitivity for the information flow analysis of multi-threaded programs. In contrast to flow-insensitive analyses, flow-sensitive analyses take the order of program statements into account, which has led to more precise information flow analyses for sequential programs [AB04, HS06, HS09]. However, as we illustrate in Chapter 4, for multi-threaded programs there is a conflict between flow sensitivity and compositionality with respect to the composition of threads. We resolve this conflict

by introducing assumption-guarantee style reasoning in the information flow analysis of multi-threaded programs. To this end, we define a novel semantic characterization of information flow security, SIFUM-security, that is compatible with assumption-guarantee style reasoning. SIFUM-security permits the exploitation of assumptions and guarantees about read and write accesses to shared variables. We show that SIFUM-security is compositional if all assumptions and guarantees are valid, and we also show that SIFUM-security is scheduler-independent. Moreover, we use SIFUM-security as the basis for developing a flow-sensitive information flow analysis in form of a security type system. The analysis is, to our knowledge, the first flow-sensitive information flow analysis for multi-threaded programs. We prove soundness of the security type system with respect to SIFUM-security and illustrate the advantages of the analysis at example programs.

While we consider improvements concerning scheduler independence and compositionality separately in Chapters 3 and 4, we base the developments of FSI- and SIFUM-security on the same approach for defining information flow properties (the "per-approach" [SS99], which we describe in Section 3.3.1). This enables the integration of FSI- and SIFUM-security, which we accomplish in Chapter 5. The integration results in the new information flow property FSIFUM-security and a corresponding security type system that combine the benefits of FSI- and SIFUM-security and the corresponding security type systems.

Information Flow Analysis with Program Dependence Graphs. While the probably most popular approach to information flow analysis in the last 15 years has been the use of security type systems, information flow analyses based on *program dependence graphs* (PDGs) have recently received increased attention (e.g., [HS09, WLS09]). One hope was that PDGs permit more precise analyses, because, for instance, PDGs are inherently flow sensitive. Considering the information flow analysis of multi-threaded programs, however, the use of PDGs had not been formally investigated at all until very recently [GS12]. In particular, no compositional PDG-based analyses for multi-threaded programs existed so far.

In Chapter 6, we investigate the relation between type-based and PDG-based information flow analysis for sequential programs. We exhibit a formal relation between two such analyses that permits to transfer concepts used in type-based analyses to PDG-based analyses and vice versa. We illustrate such a transfer in both directions. In particular, we exploit the relation to develop a provably sound PDG-based information flow analysis for multi-threaded programs by transferring ideas from the flow-sensitive type-based analysis from Chapter 4 to PDGs.

1.4 Organization of the Thesis

In Chapter 2, we introduce necessary preliminaries, a formal execution model for multi-threaded programs, a formal model of schedulers, and a formal security model, as well as a programming language for multi-threaded programs comprising a formal semantics.

Our contributions are presented in Chapters 3–6.

In Chapter 3, we present the novel scheduler-independent information flow property FSI-security. We illustrate advantages of FSI-security over other scheduler-independent information flow properties at examples. Moreover, we introduce the class of robust schedulers and prove that FSI-security is scheduler-independent with respect to such schedulers, and show for different schedulers that they belong to this class. Finally, we present a security type system that provably enforces FSI-security.

Chapter 4 is devoted to flow-sensitive information flow analysis for multi-threaded programs. As a basis for such analyses, we develop an assumption-guarantee style approach to information flow analysis. We present a novel information flow property that is compatible with assumption-guarantee style reasoning, SIFUM-security. Moreover, we exploit assumption-guarantee style reasoning to prove the compositionality of SIFUM-security. Finally, we present a flow-sensitive security type system for multi-threaded programs and prove its soundness with respect to SIFUM-security.

We illustrate in Chapter 5 how the developments from Chapters 3 and 4 can be integrated. In a first step, we integrate FSI-security and SIFUM-security, and prove scheduler independence and compositionality of the resulting security property. In a second step, we integrate the security type systems for FSI-security and SIFUM-security, and prove that the resulting type system enforces the integrated information flow property.

In Chapter 6 we consider information flow analysis for multi-threaded programs beyond security type systems. To this end, we consider analyses based on program dependence graphs (PDGs), which have successfully been applied in the analysis of sequential programs. We construct a bridge between type-based and PDG-based analysis for sequential programs, and investigate how this bridge can be exploited for transferring ideas from type-based to PDG-based analysis and vice versa. This results, in particular, in a PDG-based information flow analysis for multi-threaded programs. Moreover, we exploit the bridge for a precise comparison of the precision of PDG-based and type-based information flow analysis for sequential programs.

We conclude the thesis in Chapter 7, which contains a summary of the thesis and an outlook to directions for future investigations in the area of information flow security for multi-threaded programs.

CHAPTER 2

Execution Model and Security Model

In this chapter, we introduce the formal execution model for multi-threaded programs on which we base our developments in the subsequent chapters. The execution model abstracts from the concrete programming language in which multi-threaded programs are written. For investigating concrete example programs, we instantiate the model with multi-threaded programs specified in a simple imperative programming language. Moreover, the execution model makes the scheduling of threads explicit. The concrete scheduler is a parameter of the model, and we illustrate that the model is expressive enough to deal with typical schedulers by instantiating the model with example schedulers.

Moreover, we introduce a formal security property that characterizes information flow security of programs in the execution model. The security property is defined in the spirit of *Noninterference* [GM82], expressing information flow security by the absence of dependencies of public program outputs on secret program inputs. The formal security property is conceptually rather simple, which simplifies arguing for the property's adequacy. The purpose of this simple information flow security property is to serve as reference point for conceptually more complex compositional and scheduler-independent information flow security properties presented in subsequent chapters.

Overview. Section 2.1 introduces basic mathematical concepts and a notion of probability spaces. Section 2.2 introduces the execution model for multi-threaded programs, which is instantiated with an example programming language in Section 2.3. Section 2.4 introduces the scheduler model and Section 2.5 introduces the security model.

2.1 Mathematical Preliminaries

2.1.1 Basic Concepts and Notation

In this section, we introduce the basic mathematical concepts and corresponding notation used in this thesis.

Sets

Given a set X, we denote the powerset of X with $\mathcal{P}(X)$. Moreover, we denote the cardinality of X with $\left|X\right|$.

We denote the set of natural numbers with $\mathbb{N}$. We consider natural numbers starting with 1, i.e., $\mathbb{N} = \{1, 2, \ldots\}$. Furthermore, we denote the set of integers with $\mathbb{Z}$ and the set of real numbers with $\mathbb{R}$. For $a, b \in \mathbb{R}$ we denote the interval $\{x \in \mathbb{R} \mid a < x \leq b\}$ with $]a, b]$ and the interval $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ with $[a, b]$.

Functions

The notation $f : X \rightharpoonup Y$ indicates that f is a *partial function* that maps each element x in a subset $X' \subseteq X$ to an element $f(x) \in Y$. The set X' is the *domain* of f, which we denote with $dom(f)$. The set $\{y \mid \exists x \in dom(f) : f(x) = y\}$ is the *image* of f, and we denote it with $img(f)$.

The notation $f : X \to Y$ indicates that f is a *total function*, that is, a partial function with domain X. We call total functions simply *functions* and denote the set of all functions $f : X \to Y$ with $X \to Y$.

For $f : X \rightharpoonup Y$ and a set $X' \subseteq dom(f)$ the partial function $f\,|_{X'} \colon X \rightharpoonup Y$ is the unique partial function with $dom(f\,|_{X'}) = X'$ and $f\,|_{X'}\,(x) = f(x)$ for all $x \in X'$.

Given partial functions $f : X \rightharpoonup Y$ and $g : X \rightharpoonup Y$ with $dom(g) \subseteq dom(f)$ we define $f[x \mapsto g(x) \mid x \in dom(g)] : X \rightharpoonup Y$ to be the partial function with domain $dom(f)$ and

$$f[x \mapsto g(x) \mid x \in dom(g)](z) = \begin{cases} f(z) & \text{if } z \notin dom(g), \\ g(z) & \text{if } z \in dom(g). \end{cases}$$

We write $f[x \mapsto y]$ for $f[x \mapsto g(x) \mid x \in dom(g)]$ if $dom(g) = \{x\}$ and $g(x) = y$.

Lists

Given a set X, we denote the set of *finite lists over* X with X^*. We denote the empty list with $\langle\rangle$ and the list of the elements $x_1, x_2, \ldots, x_k \in X$ where $k \in \mathbb{N}$ with $\langle x_1, x_2, \ldots, x_k\rangle$. We denote the set $X^* \setminus \{\langle\rangle\}$ of non-empty finite lists with X^+. We refer to elements of a finite list by their position in the list, and denote the ith element of list l with $l[i]$ (i.e.,$\langle x_1, x_2, \ldots, x_k\rangle[i] = x_i$ for all $i \in \{1, \ldots, k\}$). The *length* of a finite list l, denoted with $\sharp(l)$, is defined by $\sharp(\langle\rangle) = 0$ and $\sharp(\langle x_1, x_2, \ldots, x_k\rangle) = k$. We denote the *concatenation* of two finite lists l_1 and l_2 with $l_1 :: l_2$. The *projection* of a finite list l to a set Y, denoted with $l|_Y$, is defined recursively by $\langle\rangle|_Y = \langle\rangle$, $(\langle x\rangle :: l)|_Y = \langle x\rangle :: l|_Y$ if $x \in Y$, and $(\langle x\rangle :: l)|_Y = l|_Y$ if $x \notin Y$.

We denote the set of infinite lists over a set X with X^ω, and denote the infinite list of the elements $x_1, x_2, \ldots$ with $\langle x_i\rangle_{i\in\mathbb{N}}$. For $i \in \mathbb{N}$ we denote the ith element of an infinite list $l \in X^\omega$ with $l[i]$.

Orders and Lattices

Let X be a set and $\sqsubseteq \subseteq X \times X$ a binary relation on X. The relation $\sqsubseteq$ is a *partial order on* X if it is reflexive (i.e., $\forall x \in X : x \sqsubseteq x$), antisymmetric (i.e., $\forall x, y \in X : (x \sqsubseteq y \ \wedge \ y \sqsubseteq x) \Rightarrow x = y$), and transitive (i.e., $\forall x, y, z \in X : (x \sqsubseteq y \ \wedge \ y \sqsubseteq z) \Rightarrow x \sqsubseteq z$). The pair $(X, \sqsubseteq)$ is a *partially ordered set* if the relation $\sqsubseteq$ is a partial order on X.

Let $(X, \sqsubseteq)$ be a partially ordered set. A *least upper bound* of $x \in X$ and $y \in X$ is an element $z \in X$ such that $x \sqsubseteq z$, $y \sqsubseteq z$, and $z \sqsubseteq w$ for all $w \in X$ with $x \sqsubseteq w$ and $y \sqsubseteq w$.

A *greatest lower bound* of $x \in X$ and $y \in X$ is an element $z \in X$ such that $z \sqsubseteq x$, $z \sqsubseteq y$, and $w \sqsubseteq z$ for all $w \in X$ with $w \sqsubseteq x$ and $w \sqsubseteq y$.

A partially ordered set $(X, \sqsubseteq)$ is a *lattice* if a least upper bound and a greatest lower bound exist for all $x \in X$ and $y \in X$. Given a lattice $(X, \sqsubseteq)$, the least upper bound and the greatest lower bound for $x \in X$ and $y \in X$ are unique. We denote the unique least upper bound and the unique greatest lower bound of $x \in X$ and $y \in X$ with $x \sqcup y$ and $x \sqcap y$, respectively.

2.1.2 Probability Spaces

We use a standard definition of probability spaces as introduced in, e.g., [Dur10] and [MS05].

Definition 2.1. Let Ω be a set. A set $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is a *σ-algebra* over Ω if $\Omega \in \mathcal{F}$, $\Omega \setminus X \in \mathcal{F}$ for each $X \in \mathcal{F}$, and $\bigcup_{i \in I} X_i \in \mathcal{F}$ for each countable family of sets $(X_i)_{i \in I}$ in $\mathcal{F}$. ◊

Definition 2.2. A *probability measure* on a σ-algebra $\mathcal{F}$ over Ω is a function $\rho : \mathcal{F} \to [0, 1]$ with $\rho(\Omega) = 1$ and $\rho(\bigcup_{i \in I} X_i) = \sum_{i \in I} \rho(X_i)$ for each countable family of pairwise disjoint sets $(X_i)_{i \in I}$ in $\mathcal{F}$. ◊

Definition 2.3. A *probability space* is a triple $(\Omega, \mathcal{F}, \rho)$, where Ω is a set, $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is a σ-algebra over Ω, and $\rho : \mathcal{F} \to [0, 1]$ is a probability measure. The set Ω is called the set of *outcomes* and the set $\mathcal{F}$ is called the set of *events* of the probability space. ◊

Lemma 2.1. Let Ω be a set and $\mathcal{E} \subseteq \mathcal{P}(\Omega)$ be a set of pairwise disjoint subsets of Ω with $\bigcup_{E \in \mathcal{E}} E = \Omega$. Then the set $\mathcal{F}_\mathcal{E} = \{\bigcup_{E \in \mathcal{E}'} E \mid \mathcal{E}' \subseteq \mathcal{E} \wedge \mathcal{E}' \text{ is countable}\}$ is the smallest σ-algebra over Ω that contains $\mathcal{E}$.

Let $\zeta : \mathcal{E} \to [0, 1]$ be a function such that $\sum_{E \in \mathcal{E}} \zeta(E) = 1$. Then $\rho_\zeta : \mathcal{F}_\mathcal{E} \to [0, 1]$ defined by $\rho_\zeta(\bigcup_{E \in \mathcal{E}'} E) = \sum_{E \in \mathcal{E}'} \zeta(E)$ for any countable $\mathcal{E}' \subseteq \mathcal{E}$ is the unique probability measure on $\mathcal{F}_\mathcal{E}$ that agrees with ζ on $\mathcal{E}$.

Assume that Ω is countable and that $\mathcal{E} = \{\{e\} \mid e \in \Omega\}$. Then $\mathcal{F}_\mathcal{E} = \mathcal{P}(\Omega)$. Let $\eta : \Omega \to [0, 1]$ be a function such that $\sum_{e \in \Omega} \eta(e) = 1$. Then $\rho_\eta : \mathcal{P}(\Omega) \to [0, 1]$ defined by $\rho_\eta(E) = \sum_{e \in E} \eta(e)$ for any $E \subseteq \Omega$ is the unique probability measure on $\mathcal{F}_\mathcal{E}$ with $\eta(e) = \rho_\eta(\{e\})$ for all $e \in \Omega$. ◊

Definition 2.4. The σ-algebra $\mathcal{F}_\mathcal{E}$ from Lemma 2.1 is the *σ-algebra induced by $\mathcal{E}$*. The probability measures ρ_ζ and ρ_η from Lemma 2.1 are the *probability measures induced by ζ and η*, respectively. ◊

Example 2.1. Consider a program that randomly outputs a sequence of the digits 0 and 1, either stopping eventually or running indefinitely. In this example, we define a probability space that models probabilities that the program behaves in certain ways.

We model the possible set of behaviors of the program with the set $\Omega = A^* \cup A^\omega$ where $A = \{0, 1\}$. Moreover, we model the event that the program outputs a finite sequence with an even number of digits and then stops with the set $\mathsf{EVEN} = \{l \in A^* \mid \sharp(l) \text{ is even}\}$, the event that the program outputs a finite sequence with an odd number of digits and then stops with the set $\mathsf{ODD} = \{l \in A^* \mid \sharp(l) \text{ is odd}\}$, and the event that the program never

stops with the event A^ω. Then the set $\mathcal{F}_\mathcal{E} = \{\{\}, \mathsf{EVEN}, \mathsf{ODD}, A^*, \mathsf{EVEN} \cup A^\omega, \mathsf{ODD} \cup A^\omega, A^\omega, A^* \cup A^\omega\}$ is the σ-algebra over Ω induced by the set $\mathcal{E} = \{\mathsf{EVEN}, \mathsf{ODD}, A^\omega\}$.

Assume that the program runs indefinitely in 90% of all runs, and that the program outputs an even number of digits in all other runs. We model this with the function $\zeta : \mathcal{E} \to [0,1]$ defined by $\zeta(\mathsf{EVEN}) = 0.1$, $\zeta(\mathsf{ODD}) = 0$, and $\zeta(A^\omega) = 0.9$. Let ρ_ζ be the probability measure induced by ζ. Then, for instance, $\rho_\zeta(A^*) = 0.9 + 0 = 0.9$. This expresses that the probability that the program stops eventually (i.e., the program behavior is modeled by an element of the set A^*) is 0.9.

Finally, the triple $(\Omega, \mathcal{F}_\mathcal{E}, \rho_\zeta)$ is a probability space that models probabilities that the program behaves as specified by the events in the set $\mathcal{F}_\mathcal{E}$. ◊

2.2 Execution Model

We consider multi-threaded programs, i.e., programs that may have one or more threads of execution, and assume that the programs are executed on a single processor and that the executions of the single threads are interleaved. We call the collection of the threads of a program during its execution *thread pool*. The size of a thread pool varies during program execution as threads are removed upon termination and new threads may be spawned. In particular, the size of a thread pool has no upper bound. We represent thread pools by finite lists of threads, i.e., the threads are implicitly numbered consecutively by their position in the thread pool.

We assume that multi-threaded programs access the memory by reading and writing *program variables*, and that all program variables are shared by all threads of a program. Besides the shared memory, we do not model any further means for communication between the threads of a program.

In the following, we model executions of multi-threaded programs by successively modeling snapshots during program execution (Section 2.2.1), single execution steps (Section 2.2.2), and complete program executions (Section 2.2.3). The treatment of input and output is modeled in Section 2.2.4.

2.2.1 Snapshots during Program Execution

We model the control states of threads by *commands* in a set *Com* that represent the instructions that remain to be executed by a thread. A distinguished command $stop \in Com$ models that no instructions remain to be executed, i.e., *stop* models the control state of a terminated thread. We model a thread pool by a list $thr \in Thr = (Com \setminus \{stop\})^*$, where $\sharp(thr)$ is the number of threads and $thr[i]$ models the control state of the ith thread for all $i \in \{1, \ldots, \sharp(thr)\}$. Moreover, we model the shared memory by *memories* in the set $Mem = Var \to Val$, where *Var* is a finite set of variables and *Val* is a non-empty set of values. I.e., a memory assigns a value to each variable. Finally, to make scheduling explicit, we model the state of the scheduler with *scheduler states* in a set *sSt*. The sets *Com*, *Var*, *Val*, and *sSt* are parameters of the execution model, and we leave them unspecified for now. We instantiate *Com* and *sSt* for a concrete programming language and for concrete schedulers in Sections 2.3 and 2.4, respectively.

Based on the introduced sets we model three types of snapshots during program execution. We model snapshots of a single thread during its execution by *thread configurations*

$$\langle c, mem \rangle \in Com \times Mem$$

where c is the current control state of the thread and *mem* is the current memory.

Multi-threaded configurations are pairs

$$\langle thr, mem\rangle \in Thr \times Mem$$

that model snapshots during the execution of multiple threads, where thr is the current thread pool and mem is the current shared memory. From the multi-threaded configuration we can derive the corresponding thread configurations for each thread, $\langle thr[i], mem\rangle$.

We model snapshots during the execution of a multi-threaded program under a scheduler with *system configurations*. System configurations are triples

$$\langle thr, mem, sst\rangle \in Thr \times Mem \times sSt$$

where sst is the current state of the scheduler and the multi-threaded configuration $\langle thr, mem\rangle$ is a snapshot of the execution of the multi-threaded program.

In the following, we denote commands with c, memories with mem, thread pools with thr, scheduler states with sst, thread configurations with tc, multi-threaded configurations with mc, and system configurations with sc, all possibly with primes and subscripts. Moreover, we denote the set of thread configurations with $ThreadConf$, the set of multi-threaded configurations with $MultiConf$, and the set of system configurations with $SysConf$. Finally, we introduce selector functions $getThr$, $getMem$, and $getSst$ to retrieve the elements of a system configuration, i.e., $sc = \langle getThr(sc), getMem(sc), getSst(sc)\rangle$.

2.2.2 Execution Steps

We model single execution steps of a multi-threaded program under a scheduler denoted with $\mathcal{S}$ with judgments of the form

$$sc \overset{k,p}{\Longrightarrow}_{\mathcal{S}} sc',$$

where $sc, sc' \in SysConf$, $k \in \{1, \ldots, \sharp(getThr(sc))\}$, and $p \in]0, 1]$. The interpretation of this judgment is that a single execution step under scheduler $\mathcal{S}$ may lead from the system configuration sc to the system configuration sc', where k is the index of the thread that was selected by the scheduler in this execution step, and p is the non-zero probability that this thread was selected by the scheduler.

The rule for deriving judgments of the form $sc \overset{k,p}{\Longrightarrow}_{\mathcal{S}} sc'$ is based on two further judgments that model single execution steps of single threads and the computations of the scheduler, respectively. The judgment

$$\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$$

models that an execution step of a thread with control state c in memory mem results in control state c' and memory mem'. The label $\alpha \in Lab$ is an element of the set $Lab = \{new(thr) \mid thr \in Thr\}$ that represents the creation of new threads during the execution step, where $new(thr)$ models that the initial control states of the new threads are represented by the commands in the list thr. We omit the label $new(thr)$ in the judgment if $thr = \langle\rangle$ is the empty list. We require that the derivation rules for judgments of the form $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ ensure (a) that the distinguished command $stop$ models the control state of a terminated thread, and (b) that the execution steps of single threads are deterministic. This is formalized by the following two requirements:

Requirement 2.1. The judgment $\langle stop, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ is not derivable for any $c' \in Com$, $mem, mem' \in Mem$, and $\alpha \in Lab$. ♢

Requirement 2.2. If $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ and $\langle c, mem\rangle \xrightarrow{\alpha'} \langle c'', mem''\rangle$ are derivable then $\alpha = \alpha'$, $c' = c''$, and $mem' = mem''$. ◊

We introduce derivation rules for the judgments that satisfy the above requirements together with a concrete programming language in Section 2.3.

The judgment

$$(sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst'$$

models that the scheduler $\mathcal{S}$ selects the thread at the kth position in a thread pool to proceed its computation in the next execution step of a program, where the probability of this selection is $p > 0$. The scheduler's decision may depend on the scheduler state *sst* and additional input to the scheduler represented by an element *sin* from a set of scheduler inputs *sIn*. The scheduler state after the decision is sst'. We define derivability of the judgments when introducing our scheduler model in Section 2.4. We require that the probability of the selection and the resulting scheduler state are determined by the selected thread position, which is formalized as follows:

Requirement 2.3. If $(sst, sin) \overset{k,p_1}{\rightsquigarrow}_{\mathcal{S}} sst'_1$ and $(sst, sin) \overset{k,p_2}{\rightsquigarrow}_{\mathcal{S}} sst'_2$ are derivable then $p_1 = p_2$ and $sst'_1 = sst'_2$. ◊

We model the interface of multi-threaded programs to the scheduler with an *observation function* $obs : Thr \times Mem \to sIn$. This function determines the scheduler input based on information about the thread pool and the memory. Based on the observation function, the judgments modeling execution steps of single threads, and the decisions of the scheduler we model the stepwise execution of multi-threaded programs with the following rule:

$$\frac{\begin{array}{cc}(sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst' & \langle thr[k], mem\rangle \xrightarrow{new(thr'')} \langle c', mem'\rangle \\ sin = obs(thr, mem) & thr' = update_k(thr, c', thr'')\end{array}}{\langle thr, mem, sst\rangle \overset{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr', mem', sst'\rangle} \tag{2.1}$$

The function $update_k$ in the fourth premise updates the thread pool *thr* after the execution step. To make things concrete, we assume in the remainder of this thesis that terminated threads are removed from the thread pool and that new threads are inserted in the thread pool directly behind the thread that created the new threads.

Definition 2.5. For $k \in \{1, \ldots, \sharp(thr)\}$ we define

$$update_k(thr, c, thr') = \begin{cases} replace_k(thr, thr') & \text{if } c = stop, \text{ and} \\ replace_k(thr, \langle c\rangle :: thr') & \text{otherwise,}\end{cases}$$

where the partial function $replace_k : A^* \times A^* \rightharpoonup A^*$ is defined by $replace_k(\langle a_1, \ldots, a_n\rangle, l) = \langle a_1, \ldots, a_{k-1}\rangle :: l :: \langle a_{k+1}, \ldots, a_n\rangle$ for $a_1, \ldots, a_n \in A$ with $0 < k \leq n$ and $l \in A^*$. ◊

In the remainder of this thesis, we use each judgment introduced in this section also like a predicate that is true if and only if the judgment is derivable. This permits us to write, for instance, "If $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ then ..." instead of the longer version "If $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ is derivable then ...".

2.2.3 Program Executions

We use traces to model runs of multi-threaded programs.

Definition 2.6. A system configuration $\langle thr, mem, sst\rangle$ is *final* if $\sharp(thr) = 0$. ◊

Definition 2.7. A *terminating trace* is a pair $tr = (str, dtr)$ consisting of a *terminating system trace* $str \in SysConf^+$ and a *terminating decision trace* $dtr \in \mathbb{N}^*$ such that $\sharp(str) = \sharp(dtr) + 1$, $str[\sharp(str)]$ is final, and $str[i]$ is not final and $dtr[i] \in \{1, \ldots, \sharp(getThr(str[i]))\}$ for all $i \in \{1, \ldots, \sharp(str) - 1\}$.

We define the length of a terminating trace $tr = (str, dtr)$ as $\sharp(str)$ and denote it with $\sharp(tr)$. Moreover, we denote the set of terminating traces with $\mathcal{T}_{\Downarrow}$. ◊

Definition 2.8. A *nonterminating trace* is a pair $tr = (str, dtr)$ consisting of a *nonterminating system trace* $str \in SysConf^\omega$ and a *nonterminating decision trace* $dtr \in \mathbb{N}^\omega$ such that $str[i]$ is not final and $dtr[i] \in \{1, \ldots, \sharp(getThr(str[i]))\}$ for all $i \in \mathbb{N}$.

We denote the set of nonterminating traces with $\mathcal{T}_{\not\Downarrow}$. ◊

Definition 2.9. A *trace* is a terminating trace or a nonterminating trace. We denote the set of traces $\mathcal{T}_{\Downarrow} \cup \mathcal{T}_{\not\Downarrow}$ with $\mathcal{T}$. ◊

A trace $tr = (str, dtr)$ models a run of a multi-threaded program. The system trace str models the run of the program where $str[1]$ is a snapshot of the program before starting execution and $str[k+1]$ is a snapshot of the program after k execution steps. Moreover, the decision trace dtr models the scheduling decisions during the run, where $dtr[k]$ is the thread index selected by the scheduler in the kth execution step of the program.

Definition 2.10. Given a set of traces $Tr \subseteq \mathcal{T}$, the *subset of terminating traces* is defined as $Tr_{\Downarrow} = Tr \cap \mathcal{T}_{\Downarrow}$ and the *subset of nonterminating traces* is defined as $Tr_{\not\Downarrow} = Tr \cap \mathcal{T}_{\not\Downarrow}$. Moreover, the *subset of traces terminating in memory mem* is defined as

$$Tr_{\Downarrow mem} = \{(str, dtr) \in Tr_{\Downarrow} \mid getMem(str(\sharp(str))) = mem\}.$$ ◊

We model the possible executions under scheduler $\mathcal{S}$ that start in a given system configuration with a subset of the set of all traces.

Definition 2.11. Given a scheduler $\mathcal{S}$, the set $\mathcal{T}_{\mathcal{S}}(sc)$ of *possible traces starting in a system configuration sc* is defined by $(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)$ if and only if $str[1] = sc$ and one of the following conditions is satisfied:

- $(str, dtr) \in \mathcal{T}_{\Downarrow}$ and for all $k \in \{1, \ldots, \sharp(str) - 1\}$ there exists $p \in]0, 1]$ such that $str[k] \xrightarrow{dtr[k],p}_{\mathcal{S}} str[k+1]$, or
- $(str, dtr) \in \mathcal{T}_{\not\Downarrow}$ and for all $k \in \mathbb{N}$ there exists $p \in]0, 1]$ such that $str[k] \xrightarrow{dtr[k],p}_{\mathcal{S}} str[k+1]$. ◊

We associate a probability with each terminating trace as follows:

Definition 2.12. Let $tr = (str, dtr) \in \mathcal{T}_{\Downarrow}$. The *probability* $\rho(tr)$ is defined by

$$\rho(tr) = \prod_{i=1}^{\sharp(dtr)} p_{i,dtr(i)}$$

where $p_{i,k}$ is the probability p for which $str[i] \xrightarrow{k,p}_{\mathcal{S}} str[i+1]$ is derivable, and where the product is defined to be 1 if $\sharp(dtr) = 0$. ◊

The number $\rho(tr)$ for $tr = (str, dtr)$ models the probability that the run of a multi-threaded program results in the sequence of snapshots modeled by str, and the scheduler decisions that lead to this sequence of snapshots is modeled by dtr. Note that this probability is not necessarily equal to the probability that the run of the multi-threaded program results in the sequence of snapshots modeled by str, because different scheduler decisions might lead to the same sequence of snapshots. The probability that a program run results in the sequence of snapshots modeled by str under any sequence of scheduler decisions is the sum $\sum_{\{tr \in \mathcal{T}_{\Downarrow} \mid \exists dtr \in \mathbb{N}^* : tr = (str, dtr)\}} \rho(tr)$. We do not define probabilities on system traces, because the definition on traces is slightly simpler and nevertheless sufficient for defining the security model in Section 2.5.

We describe in which sense the probabilities defined in Definition 2.12 give rise to a probability space for each system configuration after defining schedulers in Section 2.4 (cf. Definition 2.15).

2.2.4 Program Input and Program Output

We assume that programs obtain one or more inputs before beginning the execution and output one or more results after the execution terminates. We model the inputs by the initial values of variables in a set $Var_{in} \subseteq Var$, and the outputs by the final values of variables in a set $Var_{out} \subseteq Var$, where $Var_{in} \cap Var_{out} = \{\}$. We do not specify which initial values the variables in the set $Var \setminus Var_{in}$ have. These values could, for instance, be default values, or be determined nondeterministically.

Remark 2.1 (Input and Output during Program Execution). While we do not model outputs during the program execution directly, such intermediate outputs can be encoded in our model with a variable $out \in Var_{out}$ to which information is only appended during program execution. I.e., whenever $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ then either $mem'(out) = mem(out)$, or $mem'(out) = mem(out)\hat{}v$ for some $v \in Val$ (where $\hat{}$ denotes concatenation of values). Moreover, intermediate inputs to a program can be encoded in the model with a variable $in \in Var_{in}$ that stores a sequence of values, and an operation that reads an input by consuming the first value of this sequence. This captures scenarios where input is non-interactive, i.e., input provided during program execution does not depend on earlier program inputs. The treatment of interactive program input in the context of information flow security is investigated by Clark and Hunt in [CH08]. ◊

2.3 A Multi-threaded Programming Language

To investigate concrete example programs we instantiate the execution model with a simple multi-threaded programming language. Besides supporting standard programming constructs (assignments, conditionals, and loops), the programming language features an operation for dynamic thread creation.

2.3.1 Syntax

We define the syntax of the programming language with the following grammar:

$$
\begin{array}{lcl}
e & ::= & v \mid x \mid \mathsf{op}(e, \ldots, e) \\
c & ::= & \mathsf{skip} \mid x{:=}e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \\
 & & \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \mid \mathsf{spawn}(c, \ldots, c)
\end{array}
$$

where $v \in \mathit{Val}$ and $x \in \mathit{Var}$. An expression e is constructed from values, variables, and operators. We do not specify the set of values and the set of operators, and use typical values like numbers $1, 2, \ldots$ and the Boolean values true and false and operators like $+$, $-$, $=$, and $<$ using the standard infix notation in examples. We denote the set of all expressions specified by the grammar with Exp.

The language specified by the grammar for c permits to specify the instructions that are executed by a single thread, where skip is an instruction that does not modify the memory, $x{:=}e$ is an assignment of the value of expression e to variable x, $\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi}$ permits conditional execution, $\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}$ is for programming loops, and $\mathsf{spawn}(c, \ldots, c)$ permits to spawn new threads.

Remark 2.2. The programming language does not feature specific synchronization primitives. Nevertheless, synchronization can be used by programs written in the language because synchronization primitives can be implemented without leaving the language fragment. For instance, mutexes can be implemented using Peterson's algorithm for mutual exclusion [Pet81]. ◊

2.3.2 Semantics

We assume that expression evaluation is total, atomic, and unambiguous, and model expression evaluation with a function $\mathit{eval} : \mathit{Exp} \times \mathit{Mem} \to \mathit{Val}$ such that $\mathit{eval}(e, \mathit{mem})$ is the value of expression e in the memory mem. We denote with $\mathit{vars}(e)$ the set of variables on which the value of e depends. I.e.,

$$[\forall x \in \mathit{vars}(e) : \mathit{mem}_1(x) = \mathit{mem}_2(x)] \Rightarrow \mathit{eval}(e, \mathit{mem}_1) = \mathit{eval}(e, \mathit{mem}_2)$$

holds for all $e \in \mathit{Exp}$ and all $\mathit{mem}_1, \mathit{mem}_2 \in \mathit{Mem}$.

For the semantics of the programming language, we instantiate the set Com of commands with the set of all terms that are defined by the above grammar for c which we extend with $c = \mathit{stop}$ to represent a terminated thread. The intuition is that a command $c \neq \mathit{stop}$ models the control state of a thread by a specification of the instructions that remain to be executed by the thread. We define an operational semantics by a set of derivation rules for the judgments of the form $\langle c, \mathit{mem}\rangle \xrightarrow{\alpha}\!\!\triangleright \langle c', \mathit{mem}'\rangle$ (from Section 2.2.2). The derivation rules are displayed in Figure 2.1. The first four rules define a standard semantics for skip-statements, assignments, conditionals, and while loops, and the fifth rule models the semantics for dynamic thread creation. For the semantics of sequential composition we use evaluation contexts like, e.g., used by Abadi and Plotkin in [AP09]. Evaluation contexts are defined by the grammar

$$\mathcal{E} ::= \bullet \mid \mathcal{E}; c$$

where $c \in \mathit{Com} \setminus \{\mathit{stop}\}$ is a command. The command $\mathcal{E}[c]$ is defined as c if $\mathcal{E} = \bullet$ and as $\mathcal{E}'[c]; c'$ if $\mathcal{E} = \mathcal{E}'; c'$. I.e., $\mathcal{E}[c]$ is obtained from the evaluation context $\mathcal{E}$ by "filling the hole $\bullet$ with c". Intuitively, c specifies the instructions that shall next be executed when executing $\mathcal{E}[c]$. This is formalized by the last two derivation rules in Figure 2.1.

Example 2.2. Consider the command $c = \mathsf{skip}; x{:=}0$ and the evaluation context $\mathcal{E} = \bullet; x{:=}0$. Then $c = \mathcal{E}[\mathsf{skip}]$.

From the first rule in Figure 2.1 we derive $\langle \mathsf{skip}, \mathit{mem}\rangle \rightarrowtriangle \langle \mathit{stop}, \mathit{mem}\rangle$, and, hence, with the last rule in Figure 2.1 that $\langle c, \mathit{mem}\rangle \rightarrowtriangle \langle \mathcal{E}[\mathit{stop}], \mathit{mem}\rangle = \langle \mathit{stop}; x{:=}0, \mathit{mem}\rangle$. Moreover, we derive with the last but one rule in Figure 2.1 that $\langle \mathit{stop}; x{:=}0, \mathit{mem}\rangle \rightarrowtriangle \langle x{:=}0, \mathit{mem}\rangle$. ◊

$$\frac{}{\langle \mathsf{skip}, mem\rangle \rightarrowtriangle \langle stop, mem\rangle} \qquad \frac{eval(e, mem) = v}{\langle x{:=}e, mem\rangle \rightarrowtriangle \langle stop, mem[x := v]\rangle}$$

$$\frac{eval(e, mem) = \mathsf{true}}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}, mem\rangle \rightarrowtriangle \langle c_1, mem\rangle}$$

$$\frac{eval(e, mem) = \mathsf{false}}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}, mem\rangle \rightarrowtriangle \langle c_2, mem\rangle}$$

$$\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}, mem\rangle \rightarrowtriangle \langle \mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\ \mathsf{else}\ stop\ \mathsf{fi}, mem\rangle}$$

$$\frac{}{\langle \mathsf{spawn}(thr), mem\rangle \xrightarrow{new(thr)}\triangleright \langle stop, mem\rangle}$$

$$\frac{}{\langle stop; c, mem\rangle \rightarrowtriangle \langle c, mem\rangle} \qquad \frac{\langle c, mem\rangle \xrightarrow{\alpha}\triangleright \langle c', mem'\rangle}{\langle \mathcal{E}[c], mem\rangle \xrightarrow{\alpha}\triangleright \langle \mathcal{E}[c'], mem'\rangle}$$

Figure 2.1: Operational semantics of the programming language

2.4 Scheduling

During the execution of a multi-threaded program the scheduler repeatedly decides which thread shall next proceed with the computation. Scheduling algorithms differ not only in how they make this decision, but also in the information on which they base their decision. For instance, a uniform scheduler needs to know the current number of threads to randomly choose among the available threads with equal probability. A Round-Robin scheduler iterates over the list of available threads in a cyclic fashion. Beyond knowing the number of threads this requires that the scheduler remembers its scheduling decision from the previous step. A scheduler might even need to know part of the program's memory, for instance, in priority-based scheduling if priorities are first-class values that may be read and modified by the program itself. To cover these and various other possibilities, we assume that schedulers base their decision on their current internal state and on information about the current thread pool and program memory that is provided by the system to the scheduler.

We model the information that schedulers obtain about the thread pool and the memory by the elements of a set *sIn* of *scheduler inputs*. We leave this set unspecified, and only assume that it is at least possible to retrieve the current number of threads from an input $sin \in sIn$. We denote this number with $\sharp(sin)$.

We model the behavior of a scheduler with a labeled transition relation on a set *sSt* of *scheduler states*, where the set of labels is $sIn \times \mathbb{N}\times]0, 1]$. The label (sin, k, p) indicates that *sin* is the input to the scheduler, k is the position of the thread selected by the scheduler, and p is the positive probability of this selection.

Definition 2.13. A *scheduler* is a tuple $(sSt, sst_0, \rightarrow)$ consisting of a set of states *sSt*, an initial state $sst_0 \in sSt$, and a transition relation $\rightarrow \subseteq sSt \times (sIn \times \mathbb{N}\times]0, 1]) \times sSt$ for

which the following condition is satisfied:

– There exist families of functions $(\rho_{sst,sin} : \mathbb{N} \to [0,1])_{(sst,sin)\in sSt\times sIn}$ and $(next_{sst,sin} : \mathbb{N} \to sSt)_{(sst,sin)\in sSt\times sIn}$ such that
  * if $\sharp(sin) > 0$ then $1 = \sum_{k=1}^{\sharp(sin)} \rho_{sst,sin}(k)$ and $\rho_{sst,sin}(k) = 0$ for all $k > \sharp(sin)$, and
  * $(sst, (sin, k, p), sst') \in \to$ if and only if $next_{sst,sin}(k) = sst'$, $\rho_{sst,sin}(k) = p$, and $p > 0$. ◊

The properties of the functions $\rho_{sst,sin}$ ensure that each of these functions induces a probability measure on the σ-algebra $\mathcal{P}(\mathbb{N})$ over $\mathbb{N}$ (cf. Definition 2.4). The value $\rho_{sst,sin}(k)$ models the probability that the scheduler, in state sst and given scheduler input sin, selects the k^{th} thread. Moreover, the property on the transition relation ensures that the scheduler selects among the available threads. From the definition of schedulers it follows that the families of functions $(next_{sst,sin})_{(sst,sin)\in sSt\times sIn}$ and $(\rho_{sst,sin})_{(sst,sin)\in sSt\times sIn}$ are uniquely determined by the transition relation $\to$. In the following, we denote the functions $next_{sst,sin}$ and $\rho_{sst,sin}$ for scheduler $\mathcal{S}$ with $next^{\mathcal{S}}_{sst,sin}$ and $\rho^{\mathcal{S}}_{sst,sin}$, respectively. Moreover, we define

$$\rho^{\mathcal{S}}_{\langle thr,mem,sst\rangle} = \rho^{\mathcal{S}}_{sst,obs(thr,mem)},$$

i.e., $\rho^{\mathcal{S}}_{sc}(k)$ models the probability that scheduler $\mathcal{S}$ selects the thread at position k in the system configuration sc.

This scheduler model is sufficiently expressive for common deterministic schedulers such as *Round-Robin schedulers*, and may also be used to model probabilistic schedulers like, e.g., the *uniform scheduler* or *lottery schedulers*.

Example 2.3. A Round-Robin scheduler can be modeled by the tuple

$$RR = (sSt_{RR}, sst_{RR,0}, \to_{RR}),$$

where sSt_{RR} is the set of functions $\{choice, size\} \to \mathbb{N}$, and $sst_{RR,0}$ is the function with $sst_{RR,0}(choice) = 1$ and $sst_{RR,0}(size) = 1$. The scheduler variables *choice* and *size* store from the previous step which thread position was selected and what size the thread pool had. The transition relation is defined by $(sst, (sin, k, p), sst') \in \to_{RR}$ if and only if

$$k = \big[(sst(choice) + (\sharp(sin) - sst(size))) \mod \sharp(sin)\big] + 1,$$

$sst'(choice) = k$, $sst'(size) = \sharp(sin)$, and $p = 1$.

The condition on k ensures, firstly, that no thread is skipped if the current thread terminates (in this case $\sharp(sin) - sst(size)$ equals -1) and, secondly, that newly created threads obtain their term only after all other threads have been scheduled (in this case $\sharp(sin) - sst(size)$ equals the number of newly created threads). ◊

Theorem 2.1. The scheduler RR from Example 2.3 satisfies the conditions imposed on schedulers by Definition 2.13. ◊

Proof. Define

$$\rho^{RR}_{sst,sin}(k) = \begin{cases} 1 & \text{if } k = \big[(sst(choice) + (\sharp(sin) - sst(size))) \mod \sharp(sin)\big] + 1, \\ 0 & \text{otherwise.} \end{cases}$$

Then $1 = \sum_{k=1}^{\sharp(sin)} \rho^{RR}_{sst,sin}(k)$ and $\rho^{RR}_{sst,sin}(k) = 0$ for all $k > \sharp(sin)$. Moreover, define

$$next^{RR}_{sst,sin}(k) = \begin{cases} sst[choice \mapsto k][size \mapsto \sharp(sin)] & \text{if } \rho^{RR}_{sst,sin}(k) = 1, \\ sst & \text{otherwise.} \end{cases}$$

From the definitions of $\rho^{RR}_{sst,sin}$, $next^{RR}_{sst,sin}$, and $\rightarrow_{RR}$ it follows that $(sst, (sin, k, p), sst') \in \rightarrow_{RR}$ if and only if $next^{RR}_{sst,sin}(k) = sst'$ and $\rho^{RR}_{sst,sin}(k) = p = 1$. Hence, RR satisfies the conditions imposed by Definition 2.13. □

Example 2.4. A uniform scheduler can be modeled by the tuple

$$Uni = (\{s\}, s, \rightarrow_{Uni}\},$$

where $(s, (sin, k, p), s) \in \rightarrow_{Uni}$ if and only if $1 \leq k \leq \sharp(sin)$ and $p = 1/\sharp(sin)$. ◊

Theorem 2.2. The scheduler *Uni* from Example 2.4 satisfies the conditions imposed on schedulers by Definition 2.13. ◊

Proof. Define

$$\rho^{Uni}_{sst,sin}(k) = \begin{cases} 1/\sharp(sin) & \text{if } 1 \leq k \leq \sharp(sin), \\ 0 & \text{otherwise.} \end{cases}$$

Then $1 = \sum_{k=1}^{\sharp(sin)} \rho^{Uni}_{sst,sin}(k)$ and $\rho^{Uni}_{sst,sin}(k) = 0$ for all $k > \sharp(sin)$. Moreover, define $next^{Uni}_{sst,sin}(k) = sst$. From the definitions of $\rho^{Uni}_{sst,sin}$, $next^{Uni}_{sst,sin}$, and $\rightarrow_{Uni}$ it follows that $(sst, (sin, k, p), sst') \in \rightarrow_{Uni}$ if and only if $next^{Uni}_{sst,sin}(k) = sst'$ and $\rho^{Uni}_{sst,sin}(k) = p = 1/\sharp(sin)$. Hence, *Uni* satisfies the conditions imposed by Definition 2.13. □

Example 2.5. Lottery schedulers [WW94] assign a positive number of *lottery tickets* to each thread. Thread selection is determined by holding a lottery where one ticket is drawn from the set of all tickets, and the thread to which the ticket is assigned is selected. By assigning a lower respectively higher number of tickets to a thread the thread's CPU share can be decreased respectively increased. This permits to increase the priority of threads. In particular, if the number of tickets can be changed by the program at runtime then the program can dynamically adapt the priorities.

To model a lottery scheduler we assume that the number of tickets of the thread at position i is stored in the shared memory and can be dynamically adapted by a multi-threaded program, and that this part of the shared memory is provided to the scheduler as part of the scheduler input. We denote the number of tickets of the thread at position i encoded in scheduler input sin with $tickets_i(sin)$, and require that $tickets_i(sin) > 0$ for all thread positions i.

We model a lottery scheduler by the tuple $Lot = (\{s\}, s, \rightarrow_{Lot})$, where the transition relation is defined by $(s, (sin, k, p), s) \in \rightarrow_{Lot}$ if and only if $k < \sharp(sin)$ and $p = tickets_k(sin)/\left[\sum_{i=1}^{\sharp(sin)} tickets_i(sin)\right]$. ◊

Theorem 2.3. The scheduler *Lot* from Example 2.5 satisfies the conditions imposed on schedulers by Definition 2.13. ◊

Proof. Define

$$\rho^{Lot}_{sst,sin}(k) = \begin{cases} tickets_k(sin) / \left[\sum_{i=1}^{\sharp(sin)} tickets_i(sin)\right] & \text{if } 1 \leq k \leq \sharp(sin), \\ 0 & \text{otherwise.} \end{cases}$$

Then $\sum_{k=1}^{\sharp(sin)} \rho^{Lot}_{sst,sin}(k)$ equals $\left[\sum_{i=1}^{\sharp(sin)} tickets_i(sin)\right] / \left[\sum_{i=1}^{\sharp(sin)} tickets_i(sin)\right] = 1$, and $\rho^{Lot}_{sst,sin}(k) = 0$ for all $k > \sharp(sin)$. Moreover, define $next^{Lot}_{sst,sin}(k) = sst$. From the definitions of $\rho^{Lot}_{sst,sin}$, $next^{Lot}_{sst,sin}$, and $\rightarrow_{Lot}$ it follows that $(sst, (sin, k, p), sst') \in \rightarrow_{Lot}$ if and only if $next^{Lot}_{sst,sin}(k) = sst'$ and $\rho^{Lot}_{sst,sin}(k) = p = tickets_k(sin) / \left[\sum_{i=1}^{\sharp(sin)} tickets_i(sin)\right]$. Hence, *Lot* satisfies the conditions imposed by Definition 2.13. □

Given a scheduler $\mathcal{S}$ we can now define derivability of the judgments of the form $(sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst'$.

Definition 2.14. Let $\mathcal{S} = (sSt, sst_0, \rightarrow)$ be a scheduler. The judgment $(sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst'$ is derivable if and only if $(sst, (sin, k, p), sst') \in \rightarrow$. ◊

Definitions 2.13 and 2.14 ensure that if $(sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst'$ is derivable then p and sst' are uniquely determined by sst, sin and k, i.e., Requirement 2.3 is satisfied:

Theorem 2.4. Requirement 2.3 is satisfied for every scheduler $\mathcal{S}$. ◊

Proof. Assume that $(sst, sin) \overset{k,p_1}{\rightsquigarrow}_{\mathcal{S}} sst'_1$ and $(sst, sin) \overset{k,p_2}{\rightsquigarrow}_{\mathcal{S}} sst'_2$ are derivable. Then $(sst, (sin, k, p_1), sst'_1) \in \rightarrow$ and $(sst, (sin, k, p_2), sst'_2) \in \rightarrow$ by Definition 2.14. Hence, by Definition 2.13, $sst'_1 = next_{sst,sin}(k) = sst'_2$ and $p_1 = \rho_{sst,sin}(k) = p'_2$. □

Moreover, for each scheduler $\mathcal{S}$ and each system configuration sc, the trace probabilities from Definition 2.12 permit to define a probability space where each outcome is a trace in the set $\mathcal{T}_{\mathcal{S}}(sc)$. The formal definition is based on the notions of induced σ-algebras and induced probability measures from Definition 2.4.

Definition 2.15. Let sc be a system configuration and $\mathcal{S}$ be a scheduler. We define the probability space $(\Omega(\mathcal{S}, sc), \mathcal{F}(\mathcal{S}, sc), \rho(\mathcal{S}, sc))$ as follows:

- $\Omega(\mathcal{S}, sc) = \mathcal{T}_{\mathcal{S}}(sc)$;
- $\mathcal{F}(\mathcal{S}, sc)$ is the σ-algebra induced by the set $\{\{tr\} \mid tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}\} \cup \{\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}\}$;
- $\rho(\mathcal{S}, sc)$ is the probability measure induced by the function ζ defined by
  - $\zeta(\{tr\}) = \rho(tr)$ for $tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}$ and
  - $\zeta(\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}) = 1 - \sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \zeta(\{tr\})$. ◊

Intuitively, for $tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}$ the set $\{tr\}$ models the event that the program behaves as indicated by the terminating trace tr, and the probability $\rho(\mathcal{S}, sc)(\{tr\})$ is the probability that this event occurs. Moreover, the set $\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}$ models the event that the program does not terminate when starting in sc, and the probability $\rho(\mathcal{S}, sc)(\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow})$ is the probability that this event occurs. If the system configuration sc is clear from the context we write $\rho_{\mathcal{S}}$ for the probability measure $\rho(\mathcal{S}, sc)$.

Theorem 2.5. The triple $(\Omega(\mathcal{S}, sc), \mathcal{F}(\mathcal{S}, sc), \rho(\mathcal{S}, sc))$ from Definition 2.15 satisfies the conditions imposed on probability spaces (cf. Definition 2.3) for every system configuration sc and every scheduler $\mathcal{S}$. ◊

Proof. The set $\mathcal{F}(\mathcal{S}, sc)$ is a σ-algebra over the set $\Omega(\mathcal{S}, sc)$ by definition, namely the smallest σ-algebra that contains the set $\{\{tr\} \mid tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}\} \cup \{\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}\} \subseteq \mathcal{P}(\Omega(\mathcal{S}, sc))$.

The function $\rho(\mathcal{S}, sc)$ is defined as the probability measure induced by the function ζ, and, hence, by Lemma 2.1 the function $\rho(\mathcal{S}, sc)$ is a probability measure on $\mathcal{F}(\mathcal{S}, sc)$ if the range of ζ is the set $[0, 1]$ and $\sum_{E \in \{\{tr\} \mid tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}\} \cup \{\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}\}} \zeta(E) = 1$. By definition of ζ, this sum equals $(\sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \zeta(\{tr\})) + (1 - \sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \zeta(\{tr\}))$, which equals 1. It remains to show that the range of ζ is the set $[0, 1]$. We first show that $0 \leq \zeta(\{tr\}) \leq 1$ for all $tr \in \mathcal{T}_{\mathcal{S}}(sc)$. By definition, $\zeta(\{tr\}) = \rho(tr)$. By Definition 2.12, $\rho(tr) = \prod_{i=1}^{\sharp(dtr)} p_{i,dtr(i)}$, where $p_{i,k}$ is the probability p for which $str[i] \stackrel{k,p}{\Longrightarrow}_{\mathcal{S}} str[i+1]$ is derivable, and where the product is defined to be 1 if $\sharp(dtr) = 0$. All the $p_{i,k}$ are in the interval $[0, 1]$ by definition of the judgments of the form $sc \stackrel{k,p}{\Longrightarrow}_{\mathcal{S}} sc'$. Hence, $\rho(tr) \in [0, 1]$. Now we show that $\zeta(\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}) = 1 - \sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \zeta(\{tr\}) \in [0, 1]$ by showing that $\sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \zeta(\{tr\}) \leq 1$. Let str and dtr such that $tr = (str, dtr)$. By Definition 2.12, this sum equals

$$\sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \left(\prod_{i=1}^{\sharp(dtr)} p_{i,dtr(i)} \right),$$

which we rewrite as

$$\lim_{n \to \infty} \sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow} \wedge \sharp(tr) \leq n} \left(\prod_{i=1}^{\sharp(dtr)} p_{i,dtr(i)} \right).$$

Consider the sum in this sequence for some fixed n: Each summand corresponds to the probability of some terminating trace tr of length smaller or equal to n. We can represent these traces using a tree whose nodes are system configurations. The root of the tree is the system configuration sc, and there is an edge from a system configuration sc_1 to another configuration sc_2 marked with probability p if $sc_1 \stackrel{k,p}{\Longrightarrow}_{\mathcal{S}} sc_2$ is derivable for some k. We mark each edge of this tree with the corresponding probability p. Then the sum of the probabilities on outgoing edges from some system configuration is smaller or equal to 1. Hence, the sum of all the probabilities in the sum for any fixed n is smaller or equal to 1.

In consequence, the limit of the sequence consisting of the sums for all $n \in \mathbb{N}$ is smaller or equal to 1, i.e., $\sum_{tr \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}} \zeta(\{tr\}) \leq 1$. □

Definition 2.16. Let $\mathcal{S} = (sSt, sst_0, \rightarrow)$ be a scheduler. A system configuration sc is *terminating under* $\mathcal{S}$ if $\mathcal{T}_{\mathcal{S}}(sc) \subseteq \mathcal{T}_{\Downarrow}$. A thread pool thr is *terminating under* $\mathcal{S}$ if $\langle thr, mem, sst_0 \rangle$ is terminating under $\mathcal{S}$ for all $mem \in Mem$. ◊

Theorem 2.6. If the system configuration sc is terminating under $\mathcal{S}$ then

$$\rho(\mathcal{S}, sc)(\mathcal{T}_{\mathcal{S}}(sc)_{\not\Downarrow}) = 0. \qquad \Diamond$$

Proof. This follows from the definition of the probability measure $\rho(\mathcal{S}, sc)$. □

2.5 Security Model

In this section, we define security policies, we formally define a simple noninterference-like information flow property for the execution model introduced in the previous sections, and we argue for the adequacy of the property. This information flow property shall serve as reference point for more complex security properties in subsequent chapters.

2.5.1 Security Policy and Attacker Model

A security policy classifies the inputs and outputs of a program with respect to their level of confidentiality. As usual, we model different levels of confidentiality with *security domains* in a *security lattice* [Den76].

Definition 2.17. A *security lattice* is a lattice $(\mathcal{D}, \sqsubseteq)$ where $\mathcal{D}$ is a finite set of security domains. ◊

The interpretation is that if $d_1 \sqsubseteq d_2$ and $d_1 \neq d_2$ then d_1 represents a lower level of confidentiality than d_2.

Example 2.6. A classical example (e.g., [Wei69]) is the security lattice with four linearly ordered security domains *unclassified* $\sqsubseteq$ *confidential* $\sqsubseteq$ *secret* $\sqsubseteq$ *top secret*. ◊

Example 2.7. We denote with $\mathbb{2}$ the two-level security lattice $(\mathcal{D}_\mathbb{2}, \sqsubseteq_\mathbb{2})$ where $\mathcal{D}_\mathbb{2} = \{low, high\}$ and $low \sqsubseteq_\mathbb{2} high$. This lattice may be used to represent a low and a high level of confidentiality.

One example interpretation is that *high* represents the confidentiality level of a user's personal information stored on the user's computing device, and *low* represents the confidentiality level of information on the Internet. ◊

Definition 2.18. A *multi-level security policy* is a tuple $(\mathcal{D}, \sqsubseteq, dma)$, where $(\mathcal{D}, \sqsubseteq)$ is a security lattice and $dma : (Var_{in} \cup Var_{out}) \to \mathcal{D}$ is a *domain assignment*. ◊

Domain assignments associate input and output variables with security domains, capturing the level of confidentiality of each input to and each output from a program.

Remark 2.3. We use the non-standard symbol *dma* for domain assignments instead of the frequently used symbol *dom* to prevent confusion with the domain of partial functions.◊

We assume that the access to a program's inputs and outputs is restricted by an access control system, such that an *attacker with clearance* d can only observe inputs and outputs at security domain d or below, i.e., the initial values of the variables $\{x \in Var_{in} \mid dma(x) \sqsubseteq d\}$ and the final values of the variables $\{x \in Var_{out} \mid dma(x) \sqsubseteq d\}$. The intuitive security requirement of a multi-level security policy $(\mathcal{D}, \sqsubseteq, dma)$ is that for each $d \in \mathcal{D}$ an attacker with clearance d cannot deduce information about the inputs that he cannot observe directly (i.e., the initial values of variables in the set $\{x \in Var_{in} \mid dma(x) \not\sqsubseteq d\}$) from his observations of inputs and outputs.

Example 2.8. Consider a program that obtains confidential financial data via the single input variable $x \in Var_{in}$ and that writes outputs to a server on the network into the single output variable $y \in Var_{out}$. Consider furthermore the multi-level security policy $(\mathcal{D}_\mathbb{2}, \sqsubseteq_\mathbb{2}, dma)$ with $dma(x) = high$ and $dma(y) = low$. Then an attacker with clearance *low* can only see the output to the server. Moreover, the attacker must not be able to deduce information about the financial data from his observation. ◊

In the remainder of this thesis we restrict our investigations to multi-level security policies where the security lattice is the two-level lattice $\mathbb{2}$. This choice is justified by the following observation: The intuitive security requirement imposed by a multi-level security policy only distinguishes between variables x with $dma(x) \sqsubseteq d$ and variables y with $dma(y) \not\sqsubseteq d$. Hence, the intuitive security requirement for the policy $(\mathcal{D}, \sqsubseteq, dma)$ is equivalent to the conjunction of the intuitive requirements for the policies in the family $(\mathcal{D}_{\mathbb{2}}, \sqsubseteq_{\mathbb{2}}, dma_d)_{d \in \mathcal{D}}$ with

$$dma_d(x) = \begin{cases} low & \text{if } dma(x) \sqsubseteq d, \\ high & \text{otherwise.} \end{cases}$$

2.5.2 Formal Security Characterization

We formalize the intuitive requirement for a security policy $(\mathcal{D}_{\mathbb{2}}, \sqsubseteq_{\mathbb{2}}, dma)$ based on the idea of *Noninterference* [GM82]. A program is noninterferent if different program runs with the same public input produce the same public outputs, regardless of secret input.

In the following, we denote the set of input variables with security domain *high* with $HI = \{x \in Var_{in} \mid dma(x) = high\}$ and the set of output variables with security domain *low* with $LO = \{x \in Var_{out} \mid dma(x) = low\}$. Moreover, for $X \subseteq Var$ we say that mem_1 and mem_2 are X-equal (denoted with $mem_1 =_X mem_2$) if and only if $mem_1(x) = mem_2(x)$ for all $x \in X$. X-equality is an equivalence relation, and we denote the equivalence class of mem with respect to the relation $=_X$ with $[mem]_X = \{mem' \mid mem =_X mem'\}$. In particular, if mem_1 and mem_2 model two memories before program execution and $mem_1 =_{Var \setminus HI} mem_2$ then the attacker cannot distinguish the two memories by observing only public inputs. Moreover, if mem_1 and mem_2 model two memories after program termination and $mem_1 =_{LO} mem_2$ then the attacker cannot distinguish the memories by observing only public outputs.

We formalize a noninterference-like information flow property as follows:

Definition 2.19. Let $\mathcal{S}$ be a scheduler. A thread pool thr is *$\mathcal{S}$-noninterferent* if the following implication is satisfied for all memories mem_1, mem_2, and mem:

$$mem_1 =_{Var \setminus HI} mem_2 \wedge sc_1 = \langle thr, sst_0, mem_1 \rangle \wedge sc_2 = \langle thr, sst_0, mem_2 \rangle \\ \Rightarrow \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'}) \quad \Diamond$$

The two sums in the above definition express the probabilities that an execution of the thread pool thr starting in memories mem_1 and mem_2, respectively, terminates in a memory that agrees on public outputs with the memory mem.

I.e., our notion of $\mathcal{S}$-noninterference guarantees in particular that whether an $\mathcal{S}$-noninterferent program terminates under scheduler $\mathcal{S}$ with given low outputs is not influenced by the initial values of high inputs. This means, $\mathcal{S}$-noninterference implies that the attacker cannot deduce anything about high program inputs from his observation of low program outputs. This implication still holds if the attacker knows the code of the program and the scheduling algorithm. Moreover, the implication also holds if the attacker observes multiple runs of the program and determines the frequency of his observations.

Moreover, $\mathcal{S}$-noninterference is a *termination-sensitive* information flow property, i.e., from knowing whether a run of an $\mathcal{S}$-noninterferent thread pool terminates or not an attacker cannot deduce information about the secret program inputs. Even the probability that an $\mathcal{S}$-noninterferent thread pool terminates is not influenced by secret program input:

Theorem 2.7. Let $\mathcal{S}$ be a scheduler, let mem_1 and mem_2 be two memories such that $mem_1 =_{Var \setminus HI} mem_2$, and let thr be an $\mathcal{S}$-noninterferent thread pool.
Then $\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(\langle thr, sst_0, mem_1\rangle)_{\not\Downarrow}) = \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(\langle thr, sst_0, mem_2\rangle)_{\not\Downarrow})$. ◊

Proof. Let sc be a system configuration. Then $\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow} = \bigcup_{mem' \in Mem} \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'}$ by Definition 2.10. In consequence, $\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow} = \bigcup_{M \in Mem/=_{LO}} (\bigcup_{mem' \in M} \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'})$ where $Mem/=_{LO}$ denotes the set of equivalence classes of the equivalence relation $=_{LO}$. Since $\rho_{\mathcal{S}}$ is a probability measure and the sets $\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}$ are disjoint for different memories mem' it follows that $\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow}) = \sum_{M \in Mem/=_{LO}} (\sum_{mem' \in M} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'}))$.

Let $sc_i = \langle thr, sst_0, mem_i\rangle$ for $i \in \{1, 2\}$. Since thr is $\mathcal{S}$-noninterferent, the equality $\sum_{mem' \in M} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in M} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$ holds for all $M \in Mem/=_{LO}$. Hence, $\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow}) = \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow})$. Moreover, by the definition of $\rho_{\mathcal{S}}$ (cf. Definition 2.15), $\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_i)_{\not\Downarrow}) = 1 - \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_i)_{\Downarrow})$ for $i \in \{1, 2\}$. Hence, $\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\not\Downarrow}) = \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\not\Downarrow})$. □

Remark 2.4. Note that $\mathcal{S}$-noninterference does not guarantee information flow security with respect to physical side channels like power consumption channels or timing channels. Such channels are outside the scope of this thesis. ◊

2.5.3 Examples

We illustrate the definition of $\mathcal{S}$-noninterference as well as its limitations at examples in the programming language introduced in Section 2.3. In the following examples as well as in the remainder of this thesis we assume a domain assignment that assigns domain *low* to output variables $l, l', l_1, l_2 \ldots$ and domain *high* to input variables $h, h', h_1, h_2, \ldots$

Example 2.9. (Direct leak) The program

$$l{:=}h$$

directly copies the secret value of h into the public variable l. Intuitively, the program has insecure information flow, because the final value of l contains all information about the initial value of h.

In fact, the thread pool $\langle l{:=}h\rangle$ is not $\mathcal{S}$-noninterferent for any scheduler: Consider two memories mem_1 and mem_2 with $mem_1(h) \neq mem_2(h)$ and $mem_1 =_{Var \setminus HI} mem_2$, and let $sc_1 = \langle\langle l{:=}h\rangle, sst_0, mem_1\rangle$ and $sc_2 = \langle\langle l{:=}h\rangle, sst_0, mem_2\rangle$. Then the definition of $\mathcal{S}$-noninterference for $\langle l{:=}h\rangle$ requires that the equality $\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$ holds for all $mem \in Mem$. Consider this equality for $mem = mem_1[l \mapsto mem_1(h)]$. Then the only trace starting in sc_1 terminates in mem. Hence, the left side of the required equality equals 1. Moreover, the only trace starting in sc_2 terminates in $mem_2[l \mapsto mem_2(h)]$, which is not contained in the equivalence class $[mem]_{LO}$ because $mem(l) = mem_1(h) \neq mem_2(h)$. In consequence, the right side of the required equality equals 0. Hence, the required equality is not satisfied. ◊

Example 2.10. (Indirect leak) Assume that l and h store Boolean values. The program

$$\mathsf{if}\ (h = \mathsf{true})\ \mathsf{then}\ l{:=}\mathsf{true}\ \mathsf{else}\ l{:=}\mathsf{false}\ \mathsf{fi}$$

copies the initial value of h into l indirectly, i.e., without direct assignment. Intuitively, the program has insecure information flow, because as in the program from Example 2.9 the final value of l contains all information about the initial value of h.

In fact, the thread pool $thr = \langle \mathsf{if}\ (h = \mathsf{true})\ \mathsf{then}\ l{:=}\mathsf{true}\ \mathsf{else}\ l{:=}\mathsf{false}\ \mathsf{fi}\rangle$ is not $\mathcal{S}$-noninterferent for any scheduler. This follows by the same argument as in Example 2.9, since also for this program every trace starting in the system configuration $\langle thr, sst_0, mem\rangle$ terminates in the memory $mem[l \mapsto mem(h)]$. ◊

Example 2.11. (Termination leak) The program

$$\mathsf{while}\ (h = \mathsf{true})\ \mathsf{do}\ \mathsf{skip}\ \mathsf{od}$$

terminates if and only if the initial value of h is false.

One sees as follows that this program is not $\mathcal{S}$-noninterferent for any scheduler $\mathcal{S}$: Let $thr = \langle \mathsf{while}\ (h = \mathsf{true})\ \mathsf{do}\ \mathsf{skip}\ \mathsf{od}\rangle$, $sc_1 = \langle thr, sst_0, mem_1\rangle$ and $sc_2 = \langle thr, sst_0, mem_2\rangle$ where mem_1 and mem_2 are memories with $mem_1(h) = \mathsf{true}$, $mem_2(h) = \mathsf{false}$, and $mem_1(x) = mem_2(x)$ for all $x \neq h$. Then $\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem} = \{\}$ for all memories mem, whereas $\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem_2}$ contains one trace. Hence, $\sum_{mem' \in [mem_2]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = 0$ and $\sum_{mem' \in [mem_2]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'}) > 0$. In consequence, thr is not $\mathcal{S}$-noninterferent.◊

Example 2.11 illustrates that $\mathcal{S}$-noninterference is a *termination-sensitive* information flow property, i.e., $\mathcal{S}$-noninterference requires that the termination behavior of a thread pool is not influenced by secret program input (cf. Theorem 2.7).

Example 2.12. (Timing leak) Consider the thread pool

$$\langle \mathsf{while}\ (h > 0)\ \mathsf{do}\ h{:=}h - 1\ \mathsf{od}\rangle,$$

where h stores Integer values that are either positive or 0. The number of execution steps during an execution of the thread pool depends on the initial value of h. If each execution step takes the same amount of time, an attacker might, hence, deduce information about this value by measuring execution time.

Nevertheless, the thread pool is $\mathcal{S}$-noninterferent for any scheduler. Intuitively, this is because the final value of any low variable l does not depend on the initial value of h. To show this formally, let $sc_i = \langle\langle \mathsf{while}\ (h > 0)\ \mathsf{do}\ h{:=}h - 1\ \mathsf{od}\rangle, sst_0, mem_i\rangle$ for $i \in \{1, 2\}$ be two system configurations with $mem_1 =_{Var \setminus HI} mem_2$. Then there is exactly one trace starting in sc_1 and sc_2, respectively, which terminates in memory $mem_1' = mem_1[h \mapsto 0]$ and $mem_2' = mem_2[h \mapsto 0]$, respectively. It follows from $mem_1 =_{Var \setminus HI} mem_2$ and $Var_{in} \cap Var_{out} = \{\}$ that $mem_1' =_{LO} mem_2'$. Hence, the sum $\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'})$ equals the sum $\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$. ◊

Example 2.12 illustrates that $\mathcal{S}$-noninterference is a *timing-insensitive* information flow property in the sense that the number of execution steps of an $\mathcal{S}$-noninterferent program might depend on secret program inputs. I.e., $\mathcal{S}$-noninterference does not guarantee information flow security if the attacker has a stopwatch to measure execution time and tries to deduce secret information from this additional observation of a program run.

However, $\mathcal{S}$-noninterference forbids so-called *internal timing leaks*, i.e., leaks that arise because the final value of a variable depends on the internal timing of a thread.

Example 2.13. (Internal timing leak) Consider the thread pool

$$\langle \mathsf{if}\ (h = \mathsf{true})\ \mathsf{then}\ \mathsf{skip}\ \mathsf{else}\ \mathsf{skip}; \mathsf{skip}\ \mathsf{fi}; l{:=}\mathsf{true}\ ,\ \mathsf{skip}; \mathsf{skip}; l{:=}\mathsf{false}\rangle.$$

This is the thread pool already encountered in Example 1.1 in Section 1.2 (the variables are renamed here to indicate the security domain of each variable). Assume that this

thread pool is scheduled under the Round-Robin scheduler RR (see Example 2.3). Then the first thread will execute the assignment to l in its third execution step (and, hence, before the assignment to l in the second thread) if the initial value of h is true, and in its fourth execution step (and, hence, after the assignment to l in the second thread) otherwise. In consequence, the thread pool copies the initial value of h into l. The underlying reason is that the number of execution steps before the assignment to l in the first thread depends on the value of h. I.e., even if the attacker does not observe execution time the timing influences the attacker's observation because it affects the order of the assignments to l.

That this thread pool thr is not RR-noninterferent follows by the same argument as in Example 2.9, since every trace starting in the system configuration $\langle thr, sst_{RR,0}, mem\rangle$ terminates in the memory $mem[l \mapsto mem(h)]$. ◊

Information flow can also occur due to the communication between threads.

Example 2.14. (Leakage by thread communication) Consider thread pool thr defined by

$$\langle l{:=}h; l{:=}0 \ , \ l'{:=}l\rangle.$$

The value of h is leaked into l' if and only if the assignment to l' in the second thread is scheduled between the two assignments in the first thread. This illustrates that the security of a program might depend on the scheduler. In particular, this thread pool is not RR-noninterferent (where RR is the Round-Robin scheduler from Example 2.3), but it is $\mathcal{S}$-noninterferent for a scheduler that always selects the first thread in a thread pool.

This is seen as follows: Let $sc_1 = \langle thr, sst_0, mem_1\rangle$ and $sc_2 = \langle thr, sst_0, mem_2\rangle$ be two system configurations with $mem_1 =_{Var\setminus HI} mem_2$. Under scheduler RR there is exactly one trace starting in sc_1 and sc_2, respectively, terminating in memories mem_1' with $mem_1'(l') = mem_1(h)$ and mem_2' with $mem_2'(l') = mem_2(h)$, respectively. Hence, if $mem_1(h) \neq mem_2(h)$ then $mem_1' \neq_{LO} mem_2'$, and, hence, the equality required by the definition of $\mathcal{S}$-noninterference for $mem = mem_1'$ is not satisfied. I.e., thr is not RR-noninterferent. For a scheduler that always selects the first thread in a thread pool, call it $First$, there is also exactly one trace starting in sc_1 and sc_2, respectively, but terminating in memories $mem_1' = mem_1[l \mapsto 0][l' \mapsto 0]$ and $mem_2' = mem_2[l \mapsto 0][l' \mapsto 0]$, respectively. Hence, $mem_1' =_{LO} mem_2'$, and, hence, the equality required by the definition of $\mathcal{S}$-noninterference is satisfied for all memories mem. Thus, thr is $First$-noninterferent.◊

Since both thread pools $\langle l{:=}h; l{:=}0\rangle$ and $\langle l'{:=}l\rangle$ are $\mathcal{S}$-noninterferent for any scheduler, Example 2.14 illustrates that $\mathcal{S}$-noninterference is not compositional with respect to the composition of thread pools. We define compositional properties that imply $\mathcal{S}$-noninterference in the remainder of this thesis.

Example 2.15. (Probabilistic Leak) Consider the thread pool

$$\langle l{:=}\mathsf{false}, l{:=}\mathsf{true}, l{:=}h\rangle$$

where h and l are Boolean input and output variables, respectively.

Assume that the thread pool is scheduled under a scheduler that may select any thread at each execution step. Then the set of possible final values of l equals $\{\mathsf{true}, \mathsf{false}\}$, regardless of the initial value of h. However, if the scheduler is such that the probability of scheduling the first thread in a thread pool is large, say, 0.9, then the third thread in the thread pool will be selected last with probability larger than or equal to $0.9 * 0.9 = 0.81$, and, hence, the probability that the final value of l equals the initial value of h is larger

than or equal to 0.81. In consequence, the probability that the final value of l does not equal the initial value of h is smaller than or equal to $1 - 0.81 = 0.19$.

That the thread pool is not $\mathcal{S}$-noninterferent for such a scheduler is seen as follows: If $mem_1(h) = \mathsf{true}$ and $mem_2(h) = \mathsf{false}$ in the system configurations sc_1 and sc_2 in the definition of $\mathcal{S}$-noninterference, then $\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem_1[l \mapsto \mathsf{true}]}) > 0.81$ and $0.19 > \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem_2[l \mapsto \mathsf{true}]})$. Hence, the sums $\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'})$ and $\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$ are not equal, where $mem' = mem_1[l \mapsto \mathsf{true}]$.

I.e., while the set of possible final values of l does not depend on the initial value of h, the probabilities for the possible final values might depend on the initial value of h. This could be exploited by an attacker who observes multiple program runs to learn the frequency of each possible final value. ◊

Due to probabilistic leaks as in Example 2.15 we do not define $\mathcal{S}$-noninterference with a *possibilistic* approach (like, e.g., in [MSK07]), but use a *probabilistic* approach instead.

2.6 Summary

In this chapter we introduced the execution model and an example multi-threaded programming language. Moreover, we defined a simple, noninterference-like information flow property, $\mathcal{S}$-noninterference. Now we can move to the first central chapter of this thesis, where we investigate scheduler-independent information flow security.

CHAPTER 3

A More Flexible Scheduler-independent Security Analysis

3.1 Introduction

The information flow security of multi-threaded programs does not only depend on the programs' source code, but also on the execution environment. In particular, information flow security of multi-threaded programs depends on the scheduler. In consequence, an information flow analysis that is only sound for program executions under a particular scheduler is not sufficient if programs might be executed in different execution environments with potentially different schedulers. In such situations, it is desirable to use an information flow analysis that provides reliable security guarantees for all schedulers under which the program might run. Such an analysis is also desirable if programs will be executed under a fixed scheduler which is not known at analysis time. Information flow analyses checking that programs have secure information flow under a whole class of schedulers rather than under a particular scheduler, as well as the underlying information flow properties, are called *scheduler independent*.

In this chapter, we develop a novel scheduler-independent information flow property, *FSI-security*. We define FSI-security without referring to the scheduler, and show that FSI-secure programs satisfy the scheduler-dependent security property $\mathcal{S}$-noninterference (defined in Section 2.5) for a natural class of schedulers comprising, e.g., Round-Robin schedulers and Lottery schedulers. Moreover, we show that FSI-security is preserved under the composition of thread pools. We exploit the compositionality of FSI-security in the development of a provably sound security type system that enforces FSI-security.

The goal of developing FSI-security was to overcome restrictions of other scheduler-independent information flow properties. Previously developed scheduler-independent properties suffer from three main deficiencies: Some properties are not suitable for programs with nondeterministic public outputs. Such properties are not suitable for programs with useful nondeterminism that occurs, e.g., if multiple threads concurrently append to a public log file. Some properties are not suitable for programs whose runtime

depends on secrets, even if this dependency does not cause differences in the public output. Other properties require a non-standard interface between program and scheduler. FSI-security is the first information flow property that is suitable for programs with nondeterministic public output whose runtime depends on secrets, and that is scheduler-independent for common schedulers. The existence of such a property was somewhat surprising in the light of Sabelfeld's result in [Sab03] that a more restrictive information flow property, strong security, is the least restrictive compositional information flow property that is scheduler-independent for a natural class of schedulers. We overcame the limits implied by this result by identifying another natural class of schedulers, the *robust schedulers*, that also contains many relevant schedulers.

We conclude the section by illustrating the benefits of FSI-security as well as the corresponding security type system at an example program, and discussing the relation to other scheduler-independent security properties in more detail.

Overview. In Section 3.2 we introduce scheduler independence. In Section 3.3 we define our novel scheduler-independent semantic characterization of information flow, for which we present a security type system in Section 3.4. We illustrate the type system with an example security analysis in Section 3.5. Section 3.6 summarizes the results and discusses the relation to other approaches to scheduler-independent information flow security.

3.2 Scheduler-independent Information Flow Security

The security of multi-threaded programs depends on the scheduler. In other words, there are multi-threaded programs that have secure information flow when they are executed under one scheduler, but leak secret information when they are executed under another scheduler. The following example illustrates this phenomenon.

Example 3.1. Consider the thread pool $thr = \langle c_1, c_2 \rangle$, with c_1 and c_2 defined as follows:

$$c_1 = \mathsf{while}\ (h_1 > 0)\ \mathsf{do}\ h_1{:=}h_1 - 1\ \mathsf{od}; l{:=}\mathsf{true}$$

$$c_2 = \mathsf{while}\ (h_2 > 0)\ \mathsf{do}\ h_2{:=}h_2 - 1\ \mathsf{od}; l{:=}\mathsf{false}$$

The execution of this thread pool under the Round-Robin scheduler from Example 2.3 leaks the value of the expression $h_1 > h_2$ into variable l. This is because under the Round-Robin scheduler the assignment to l in c_1 is executed after the assignment to l in c_2 if and only if the body of the loop in c_1 is executed more often than the body of the loop in c_2, i.e., if and only if the expression $h_1 > h_2$ evaluates to true. In consequence, the final value of l is true if and only if the expression $h_1 > h_2$ evaluates to true.

Consider another scheduler, *First*, that always selects the first thread in a thread pool. If the thread pool thr is executed under the scheduler *First* then the final value of l is false regardless of the initial values of variables, because the second thread is executed after the execution of the first thread, and, hence, false is assigned to l after true is assigned to l.◊

Theorem 3.1. The thread pool thr from Example 3.1 is *First*-noninterferent but not RR-noninterferent. ◊

Proof. Let mem_1 and mem_2 be two memories with $mem_1 =_{Var \setminus HI} mem_2$. Then, by the definition of *First* and the program semantics, the sets $\mathcal{T}_{First}(\langle thr, sst_0^{First}, mem_1 \rangle)$ and $\mathcal{T}_{First}(\langle thr, sst_0^{First}, mem_2 \rangle)$ both contain exactly one trace (where sst_0^{First} is the initial scheduler state of *First*), which we denote with tr_1^{First} and tr_2^{First}. Hence, $\rho_{First}(tr_1^{First}) =$

$\rho_{First}(tr_2^{First}) = 1$. Moreover, tr_1^{First} terminates in memory $mem_1[l \mapsto \mathsf{false}][h_1 \mapsto \max(h_1, 0)][h_2 \mapsto \max(h_2, 0)]$, and tr_2^{First} terminates in memory $mem_2[l \mapsto \mathsf{false}][h_1 \mapsto \max(h_1, 0)][h_2 \mapsto \max(h_2, 0)]$. These two memories are related by the relation $=_{LO}$, because $mem_1 =_{Var \setminus HI} mem_2$ and $LO \subseteq Var \setminus HI$. Hence, by the definition of $\mathcal{S}$-noninterference, *thr* is *First*-noninterferent.

Let now mem_1 and mem_2 be memories with $mem_1 =_{Var \setminus HI} mem_2$, $mem_1(h_1) = 1$, $mem_1(h_2) = 0$, $mem_2(h_1) = 0$, and $mem_2(h_2) = 1$. Then, by the definition of *RR* and the program semantics, $\mathcal{T}_{RR}(\langle thr, sst_0^{RR}, mem_1\rangle)$ and $\mathcal{T}_{RR}(\langle thr, sst_0^{RR}, mem_2\rangle)$ both contain exactly one trace (where sst_0^{RR} is the initial scheduler state of *RR*), which we denote with tr_1^{RR} and tr_2^{RR}. Hence, $\rho_{RR}(tr_1^{RR}) = \rho_{RR}(tr_2^{RR}) = 1$. Moreover, tr_1^{RR} terminates in memory $mem_1[l \mapsto \mathsf{true}][h_1 \mapsto 0][h_2 \mapsto 0]$, and tr_2^{RR} terminates in memory $mem_2[l \mapsto \mathsf{false}][h_1 \mapsto 0][h_2 \mapsto 0]$. These two memories are not related by the relation $=_{LO}$, because $mem_1(l) \neq mem_2(l)$. Hence, by the definition of $\mathcal{S}$-noninterference, *thr* is not *RR*-noninterferent. □

As illustrated by Example 3.1, a multi-threaded program might leak secrets even if one has convinced oneself that the program has secure information flow with an analysis that assumes a particular scheduler. In consequence, it is desirable to characterize information flow security with a *scheduler-independent* information flow property, which has the characteristic that programs satisfying the scheduler-independent property have secure information flow under each scheduler in a sufficiently large class of schedulers.

Definition 3.1. An information flow property *IFP* on thread pools is *scheduler independent with respect to $\mathcal{S}$-noninterference for a set of thread pools TP and a set of schedulers $\mathscr{S}$* if whenever $thr \in TP$ satisfies *IFP* then *thr* is $\mathcal{S}$-noninterferent for all $\mathcal{S} \in \mathscr{S}$.◊

It is obviously straightforward to define a scheduler-independent information flow property for a set of schedulers $\mathscr{S}$, namely the property that is satisfied for a thread pool if and only if the thread pool is $\mathcal{S}$-noninterferent for all $\mathcal{S} \in \mathscr{S}$. A disadvantage of this property is, however, that it might not be obvious how to verify that the property is satisfied for a given thread pool for large or even infinite $\mathscr{S}$. In the remainder of this chapter we develop a scheduler-independent information flow property for a natural class of schedulers that is defined without referring to concrete schedulers, and we use the property as the basis for a type-based scheduler-independent information flow analysis.

Remark 3.1. For many properties, including, for instance, *safety properties* [Lam77], for a scheduler-independent analysis it suffices to consider program execution under the *possibilistic scheduler*, i.e., the scheduler that nondeterministically selects any possible thread. This is, however, not sufficient for information flow security. The underlying reason is the *refinement paradox*, i.e., that refinements of a system specification might have insecure information flow even if the original specification has secure information flow [Jac89]. Replacing the possibilistic scheduler with a scheduler that will never schedule the threads in certain orders results in a refinement of the specification of program execution which, in fact, may turn a secure program into an insecure program. ◊

3.3 A Novel Scheduler-independent Security Property

Before defining a novel scheduler-independent information flow property in Section 3.3.2 we describe the general approach for defining the property in Section 3.3.1. Subsequently, we show the scheduler independence of the new property with respect to the novel class of robust schedulers (Sections 3.3.3 and 3.3.4).

3.3.1 The Per Approach

To define the scheduler-independent information flow property, we follow the common approach to define information flow security using a *partial equivalence relation* (short: per) on thread pools [SS99].

Definition 3.2. A *partial equivalence relation* is a symmetric and transitive binary relation. ◊

The idea is to define a partial equivalence relation that relates two thread pools only if they compute equal public outputs when being executed in memories that only differ in secret inputs, and to define a thread pool as secure if it is related to itself. This ensures that public outputs of a secure thread pool do not depend on secret inputs. Intuitively, the relation models an "indistinguishability" of thread pools in the sense that executions of related thread pools with equal public inputs are "indistinguishable" for an attacker who sees public inputs and outputs but who cannot see secret inputs and outputs.

To define the partial equivalence relation, we follow a line of security definitions (e.g., [SS00, Sab01, BC02, MR07]) that define partial equivalence relations using a bisimulation relation that models a stepwise "indistinguishability" of thread pools, ensuring that execution steps of related thread pools are indistinguishable with respect to modifications of "public" variables. The intention is that the set of public variables comprises public input and public output variables, and that public variables store public but no secret information. To this end, one extends the domain assignment dma from the set $Var_{in} \cup Var_{out}$ to the set Var such that variables in the set $L = \{x \in Var \mid dma(x) = low\}$ are the "public" variables. The bisimulation is defined such that single execution steps of bisimilar thread pools in L-equal memories (i.e., memories that differ only on non-public variables) result in bisimilar thread pools and L-equal memories. Using an inductive argument, it follows that sequences of execution steps of bisimilar thread pools in L-equal memories result in L-equal memories. This ensures that thread pools that are bisimilar to themselves satisfy some noninterference-like information flow property (like, e.g., $\mathcal{S}$-noninterference from Section 2.5).

In the remainder of this thesis, we assume that the domain assignment dma assigns a security domain to every variable, and we denote the sets $\{x \in Var \mid dma(x) = low\}$ and $\{x \in Var \mid dma(x) = high\}$ with L and H, respectively.

Remark 3.2. Assigning a security domain to all variables and requiring that the intermediate values of public variables do not depend on the initial values of secrets is too restrictive for programs that use some variables to store both secret and public information during an execution. We do not consider this issue in this chapter, and present a solution that permits such a more flexible use of variables in multi-threaded programs without giving up compositionality in Chapter 4. ◊

3.3.2 FSI-Security

For the definition of FSI-security we distinguish between *high and low threads*. A high thread never modifies the values of low variables. Moreover, this must still be true if other threads influence the execution of the thread by modifying the shared memory. All other threads are low. To formally define high and low threads, we use a notion of reachability for thread configurations that takes memory modifications by other threads into account.

Definition 3.3. Let tc be a thread configuration. The set $loc\text{-}reach(tc) \subseteq Com \times Mem$ of *thread configurations locally reachable from* tc is inductively defined as follows:

- $tc \in \textit{loc-reach}(tc)$,
- $\forall c', \mathit{mem}', \mathit{mem}'' : \langle c', \mathit{mem}' \rangle \in \textit{loc-reach}(tc) \Rightarrow \langle c', \mathit{mem}'' \rangle \in \textit{loc-reach}(tc)$, and
- $\forall tc', c'', \mathit{mem}'', \mathit{thr} : [tc' \in \textit{loc-reach}(tc) \ \wedge \ tc' \xrightarrow{\mathit{new}(\mathit{thr})} \langle c'', \mathit{mem}'' \rangle] \Rightarrow$
  $[\langle c'', \mathit{mem}'' \rangle \in \textit{loc-reach}(tc) \wedge \forall i \in \{1, \ldots, \sharp(\mathit{thr})\} : \langle \mathit{thr}[i], \mathit{mem}'' \rangle \in \textit{loc-reach}(tc)]$. ◊

Definition 3.4. A command c is *high* if and only if the following condition is satisfied:

$$\forall c', c'' \in \mathit{Com} : \forall \mathit{mem}, \mathit{mem}', \mathit{mem}'' \in \mathit{Mem} : \forall \alpha \in \mathit{Lab} :$$
$$[\langle c', \mathit{mem}' \rangle \in \textit{loc-reach}(\langle c, \mathit{mem} \rangle) \wedge \langle c', \mathit{mem}' \rangle \xrightarrow{\alpha} \langle c'', \mathit{mem}'' \rangle] \Rightarrow \mathit{mem}' =_L \mathit{mem}''$$

We denote the set of high commands with HCom. A command is *low* if it is not high, and we denote the set of low commands with LCom (i.e., $\mathit{LCom} = \mathit{Com} \setminus \mathit{HCom}$). ◊

Example 3.2. The command h:=0 is high, and the command l:=0 is low. Moreover, the command h:=false; if (h) then l:=0 else skip fi is low. ◊

The third command in Example 3.2 illustrates that a command that never assigns to a low variable if executed in isolation need not be high. Low commands may result in high commands after an execution step, like, e.g., the low command l:=0; h:=0. High commands, in contrast, cannot result in a low command after an execution step.

Theorem 3.2. Let $c \in \mathit{HCom}$ with $\langle c, \mathit{mem} \rangle \xrightarrow{\mathit{new}(\mathit{thr})} \langle c', \mathit{mem}' \rangle$. Then $c' \in \mathit{HCom}$ and $\mathit{thr}[i] \in \mathit{HCom}$ for all $i \in \{1, \ldots, \sharp(\mathit{thr})\}$. ◊

Proof. For all memories mem and mem' it follows from the definition of local reachability that $\textit{loc-reach}(\langle c', \mathit{mem}' \rangle) \subseteq \textit{loc-reach}(\langle c, \mathit{mem} \rangle)$. Thus, by the definition of high commands, $c' \in \mathit{HCom}$ follows from $c \in \mathit{HCom}$. That $\mathit{thr}[i] \in \mathit{HCom}$ for all $i \in \{1, \ldots, \sharp(\mathit{thr})\}$ follows analogously from $\textit{loc-reach}(\langle \mathit{thr}[i], \mathit{mem}' \rangle) \subseteq \textit{loc-reach}(\langle c, \mathit{mem} \rangle)$. □

In the following, we refer to a thread with control state represented by a high command as *high thread*, and to all other threads as *low thread*.

Definition 3.5. A *low match* of thread pools thr_1 and thr_2 is an order-preserving bijection $f : \{i \in \{1, \ldots, \sharp(\mathit{thr}_1)\} \mid \mathit{thr}_1[i] \in \mathit{LCom}\} \to \{i \in \{1, \ldots, \sharp(\mathit{thr}_2)\} \mid \mathit{thr}_2[i] \in \mathit{LCom}\}$. ◊

That is, a low match of thr_1 and thr_2 maps the position of the ith low thread in thr_1 to the position of the ith low thread in thr_2.

Example 3.3. Let $c_l, c_l' \in \mathit{LCom}$ and $c_h \in \mathit{HCom}$. Then the function $f : \{1, 3\} \to \{1, 2\}$ with $f(1) = 1$ and $f(3) = 2$ is a low match of the thread pools $\langle c_l, c_h, c_l' \rangle$ and $\langle c_l', c_l \rangle$. ◊

Theorem 3.3. There exists a low match of thread pools thr_1 and thr_2 if and only if thr_1 and thr_2 have the same number of low threads, i.e., $\sharp(\mathit{thr}_1|_{\mathit{LCom}}) = \sharp(\mathit{thr}_2|_{\mathit{LCom}})$. If a low match exists then it is unique and given by the function

$$\textit{l-match}_{\mathit{thr}_1, \mathit{thr}_2}(k_1) = \min\Big\{ k_2 \in \mathbb{N} \mid \big|\{l \leq k_1 \mid \mathit{thr}_1[l] \in \mathit{LCom}\}\big| = \big|\{l \leq k_2 \mid \mathit{thr}_2[l] \in \mathit{LCom}\}\big| \Big\}. \quad ◊$$

Proof. An order-preserving bijection between two finite subsets of $\mathbb{N}$ exists if and only if the sets have equal cardinality, and $\sharp(thr|_{LCom}) = \big|\{i \in \{1,\ldots,\sharp(thr)\} \mid thr[i] \in LCom\}\big|$.

If an order-preserving bijection exists between two finite subsets of $\mathbb{N}$ then it is unique and maps the kth-smallest number in one set to the kth-smallest number in the other set. The number $k_1 \in \{i \in \{1,\ldots,\sharp(thr_1)\} \mid thr_1[i] \in LCom\}$ is the kth-smallest number in that set if $k = \big|\{l \leq k_1 \mid thr_1[l] \in LCom\}\big|$. Moreover, the kth-smallest number in the set $\{i \in \{1,\ldots,\sharp(thr_2)\} \mid thr_2[i] \in LCom\}$ is the minimal k_2 such that $k = \big|\{l \leq k_2 \mid thr_2[l] \in LCom\}\big|$. This results in the given characterization of low matches. □

Now we are ready to define the bisimulation relation on which we base the definition of our novel security property, FSI-security.

Definition 3.6. A symmetric relation $\mathcal{R}$ on thread pools with an equal number of low threads is a *low bisimulation modulo low matching* if and only if the following conditions are satisfied for all $thr_1, thr_1', thr_2 \in Thr$, all $mem_1, mem_2 \in Mem$, all $k_1 \in \{1,\ldots,\sharp(thr_1)\}$, and all $c_1' \in Com$ with $thr_1 \mathcal{R} thr_2$, $mem_1 =_L mem_2$, and $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$ where $\alpha_1 = new(thr_1')$:

1. if $thr_1[k_1] \in LCom$ then there exist $c_2' \in Com$, $mem_2' \in Mem$, and $thr_2' \in Thr$ with
   a) $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$,
   b) $mem_1' =_L mem_2'$,
   c) $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, and
   d) $update_{k_1}(thr_1, c_1', \alpha_1) \mathcal{R} update_{k_2}(thr_2, c_2', \alpha_2)$

   where $\alpha_2 = new(thr_2')$ and $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$; and
2. if $thr_1[k_1] \in HCom$ then $update_{k_1}(thr_1, c_1', \alpha_1) \mathcal{R} thr_2$.

The relation $\sim_{lm}$ is the union of all low bisimulations modulo low matching. ◊

Theorem 3.4. The relation $\sim_{lm}$ is the largest low bisimulation modulo low matching. ◊

Proof. If $\sim_{lm}$ is a low bisimulation modulo low matching it follows from its definition that it is the largest such relation. It remains to show that $\sim_{lm}$ is a low bisimulation modulo low matching.

The relation $\sim_{lm}$ is symmetric and relates only thread pools with an equal number of low threads because it is the union of relations with these properties. Assume that $thr_1 \sim_{lm} thr_2$, $mem_1 =_L mem_2$, $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$, and $\alpha_1 = new(thr_1')$. By the definition of $\sim_{lm}$ there exists a low bisimulation modulo low matching $\mathcal{R}$ such that $thr_1 \mathcal{R} thr_2$. Hence, if $thr_1[k_1] \in LCom$ then there exist $\alpha_2 = new(thr_2')$, $c_2' \in Com$, and $mem_2' \in Mem$ such that $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$, $mem_1' =_L mem_2'$, $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, and $update_{k_1}(thr_1, c_1', \alpha_1) \mathcal{R} update_{k_2}(thr_2, c_2', \alpha_2)$ where $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$. Thus, $update_{k_1}(thr_1, c_1', \alpha_1) \sim_{lm} update_{k_2}(thr_2, c_2', \alpha_2)$ by the definition of $\sim_{lm}$. Moreover, $update_{k_1}(thr_1, c_1', \alpha_1) \mathcal{R} thr_2$ holds if $thr_1[k_1] \in HCom$, and, in consequence, $update_{k_1}(thr_1, c_1', \alpha_1) \sim_{lm} thr_2$. □

Remark 3.3. It is also possible to define the relation $\sim_{lm}$ as the greatest fixed point of a function that maps binary relations on thread pools to binary relations on thread pools, thereby following another well-known approach for defining bisimulation relations (see, e.g., [San09, Section 2.3]). Such a definition makes fixed point theory directly available for reasoning about $\sim_{lm}$, e.g., yielding Theorem 3.4 as a corollary of the Knaster-Tarski theorem. In this thesis, we do not exploit this alternative way of defining $\sim_{lm}$. ◊

Theorem 3.5. The relation $\sim_{lm}$ is a partial equivalence relation. ◊

Proof. Symmetry follows from Theorem 3.4 and the symmetry of low bisimulations modulo low matching.

To prove transitivity, define a relation $\mathcal{R}$ on thread pools by $thr_1 \mathcal{R} thr_2$ if and only if there exists thr' such that $thr_1 \sim_{lm} thr'$ and $thr' \sim_{lm} thr_2$. Relation $\mathcal{R}$ is a low bisimulation modulo low matching by Lemma A.2 (see Appendix A.1). Assume now that $thr_1 \sim_{lm} thr_2$ and $thr_2 \sim_{lm} thr_3$. Then $thr_1 \mathcal{R} thr_3$. Since $\mathcal{R}$ is a low bisimulation modulo low matching it follows from the definition of $\sim_{lm}$ that $thr_1 \sim_{lm} thr_3$. □

Low bisimulations modulo low matching characterize the following indistinguishability of thread pools: If two thread pools are related by a low bisimulation modulo low matching then their executions are indistinguishable to an observer who sees only modifications of low variables, given that each step of a low thread in one execution is matched by a step of the matching low thread in the other execution. This is due to Condition (1) in Definition 3.6, which ensures that steps of matching low threads equally modify the values of low variables and result in bisimilar thread pools, and to Condition (2) in Definition 3.6 which ensures that steps of high threads (that do not modify the values of low variables by definition) result in bisimilar thread pools. In consequence, by observing modifications of low variables an observer cannot distinguish the executions. This remains true if the memory is modified between steps as long as the modified memories are again L-equal (due to the quantification over all L-equal memories in Definition 3.6).

Definition 3.7. *A thread pool thr is FSI-secure* if $thr \sim_{lm} thr$. *A command c is FSI-secure* if the thread pool $\langle c \rangle$ is FSI-secure. ◊

From the above intuition of low bisimulations modulo low matchings it follows that the FSI-security of a thread pool implies that two executions of the thread pool that start in L-equal memories perform the same sequence of modifications of low variables, given that each step of a low thread in one execution is matched by a step of the matching low thread in the other execution. If this matching condition holds for executions under a given scheduler, then it follows that an attacker who sees only the initial and final values of low variables cannot deduce information about the initial values of high variables. In consequence, an attacker who sees only the initial values of low input and the final values of low output variables cannot deduce information about the initial values of high input variables. We establish that FSI-security indeed implies $\mathcal{S}$-noninterference for a natural class of schedulers in Section 3.3.4, i.e., that FSI-security is a scheduler-independent information flow property for that class of schedulers. This motivates the expansion "flexible scheduler-independent security" of the acronym "FSI-security".

Moreover, FSI-security is a compositional information flow property with respect to the parallel composition of thread pools.

Theorem 3.6. Let $thr_1, thr_2, thr_1', $ and thr_2' be thread pools such that $thr_1 \sim_{lm} thr_1'$ and $thr_2 \sim_{lm} thr_2'$. Then $thr_1 :: thr_2 \sim_{lm} thr_1' :: thr_2'$.

In consequence, if thr_1 and thr_2 are FSI-secure thread pools then their parallel composition $thr_1 :: thr_2$ is FSI-secure. ◊

Proof. We define the relation $\mathcal{R}$ by $thr_1 \mathcal{R} thr_2$ if and only if there exist $thr_{1,1}$, $thr_{1,2}$, $thr_{2,1}$, and $thr_{2,2}$ such that $thr_1 = thr_{1,1} :: thr_{1,2}$, $thr_2 = thr_{2,1} :: thr_{2,2}$, $thr_{1,1} \sim_{lm} thr_{2,1}$, and $thr_{1,2} \sim_{lm} thr_{2,2}$.

By Lemma A.3 (see Appendix A.1), $\mathcal{R}$ is a low bisimulation modulo low matching. Moreover, $thr_1 :: thr_2 \mathcal{R} thr_1' :: thr_2'$. Hence, since $\mathcal{R}$ is a low bisimulation modulo low matching, $thr_1 :: thr_2 \sim_{lm} thr_1' :: thr_2'$ follows from the definition of $\sim_{lm}$.

That the parallel composition of FSI-secure thread pools is FSI-secure follows with the definition of FSI-security. □

Theorem 3.6 ensures that a modular security analysis of multi-threaded programs with respect to FSI-security that reduces the analysis of thread pools to single threads is sound. Moreover, Theorem 3.6 shows that FSI-security is compatible with programs that have nondeterministic public output, as illustrated by the following simplistic example.

Example 3.4. Assume that $\mathsf{log} \in \mathit{Var}_{out}$ and $\mathit{dma}(\mathsf{log}) = \mathit{low}$. Then the thread pool

$$\langle \mathsf{log}{:=}\mathsf{log} + \text{"}\mathsf{message1}\text{"} \ , \ \mathsf{log}{:=}\mathsf{log} + \text{"}\mathsf{message2}\text{"} \rangle$$

has nondeterministic public output. The thread pool is FSI-secure by Theorem 3.6, because both $\langle \mathsf{log}{:=}\mathsf{log} + \text{"}\mathsf{message1}\text{"} \rangle$ and $\langle \mathsf{log}{:=}\mathsf{log} + \text{"}\mathsf{message2}\text{"} \rangle$ are FSI-secure. ◊

Compatibility with programs that have nondeterministic public output is an advantage of FSI-security over scheduler-independent information flow properties based on the idea of observational nondeterminism (e.g., [ZM03, HWS06]) that are not satisfied for such programs. (A more detailed comparison to such properties is provided in Section 3.6.)

Moreover, the presence of high threads in a thread pool does not influence the FSI-security of the thread pool.

Theorem 3.7. Let $\mathit{thr}_1, \mathit{thr}_2 \in \mathit{Thr}$. Then $\mathit{thr}_1 \sim_{lm} \mathit{thr}_2$ if and only if $(\mathit{thr}_1|_{\mathit{LCom}}) \sim_{lm} (\mathit{thr}_2|_{\mathit{LCom}})$. ◊

Proof. We define the relation $\mathcal{R}'$ on thread pools by $\mathit{thr}_1 \ \mathcal{R}' \ \mathit{thr}_2$ if and only if there exists thr_2' such that $\mathit{thr}_1|_{\mathit{LCom}} = \mathit{thr}_2|_{\mathit{LCom}}$ and $\mathit{thr}_1 \sim_{lm} \mathit{thr}_2'$. Moreover, we define the relation $\mathcal{R}$ as the symmetric closure of $\mathcal{R}'$. Then $\mathcal{R}$ is a low bisimulation modulo low matching by Lemma A.4 (see Appendix A.1).

If $\mathit{thr}_1 \sim_{lm} \mathit{thr}_2$ it follows from symmetry and transitivity of $\sim_{lm}$ (Theorem 3.5) that $\mathit{thr}_1 \sim_{lm} \mathit{thr}_1$ and $\mathit{thr}_2 \sim_{lm} \mathit{thr}_2$. Hence, $\mathit{thr}_1 \ \mathcal{R} \ \mathit{thr}_1|_{\mathit{LCom}}$ and $\mathit{thr}_2 \ \mathcal{R} \ \mathit{thr}_2|_{\mathit{LCom}}$. Since $\mathcal{R}$ is a low bisimulation modulo low matching it follows from the definition of $\sim_{lm}$ that $\mathit{thr}_1 \sim_{lm} \mathit{thr}_1|_{\mathit{LCom}}$ and $\mathit{thr}_2 \sim_{lm} \mathit{thr}_2|_{\mathit{LCom}}$. Hence, by symmetry and transitivity of $\sim_{lm}$ and $\mathit{thr}_1 \sim_{lm} \mathit{thr}_2$ it follows that $\mathit{thr}_1|_{\mathit{LCom}} \sim_{lm} \mathit{thr}_2|_{\mathit{LCom}}$.

If $\mathit{thr}_1|_{\mathit{LCom}} \sim_{lm} \mathit{thr}_2|_{\mathit{LCom}}$ it follows that $\mathit{thr}_1|_{\mathit{LCom}} \ \mathcal{R} \ \mathit{thr}_2$ (set $\mathit{thr}_2' = \mathit{thr}_2|_{\mathit{LCom}}$ in the definition of $\mathcal{R}$ and exploit that $(\mathit{thr}|_{\mathit{LCom}})|_{\mathit{LCom}} = \mathit{thr}|_{\mathit{LCom}}$ for all thread pools thr). Hence, since $\mathcal{R}$ is a low bisimulation modulo low matching, it follows that $\mathit{thr}_1|_{\mathit{LCom}} \sim_{lm} \mathit{thr}_2$. But then $\mathit{thr}_1 \ \mathcal{R} \ \mathit{thr}_2$ (set $\mathit{thr}_2' = \mathit{thr}_1|_{\mathit{LCom}}$ in the definition of $\mathcal{R}$). Thus, $\mathit{thr}_1 \sim_{lm} \mathit{thr}_2$. □

In particular, it follows directly from Theorem 3.7 that high commands are FSI-secure.

Theorem 3.8. Let $c \in \mathit{HCom}$. Then c is FSI-secure. ◊

Proof. Trivially, $\langle\rangle \sim_{lm} \langle\rangle$ is satisfied. Hence, $\langle c \rangle \sim_{lm} \langle c \rangle$ follows from Theorem 3.7 because $\langle c \rangle|_{\mathit{LCom}} = \langle\rangle$ for $c \in \mathit{HCom}$. □

Theorem 3.8 shows that FSI-security is compatible with programs whose number of execution steps depends on secret information, as illustrated by the following example.

Example 3.5. Let $c' \in \mathit{HCom}$. Then the command $\mathsf{while}\ (h > 0)\ \mathsf{do}\ c'; h{:=}h - 1\ \mathsf{od}$ that executes c' for decreasing h is FSI-secure by Theorem 3.8. ◊

Compatibility with programs for which the number of execution steps depends on secret information constitutes a major improvement over the strong security condition from [SS00] which classifies such programs as insecure. (A more detailed comparison to strong security is provided in Section 3.6.)

3.3.3 Robust Schedulers

Our goal is to establish that FSI-security is scheduler independent for a natural class of schedulers. If secrets leak to the scheduler via the interface between programs and the scheduler then the schedule might depend on these secrets, and, in consequence, the program's public output might depend on secrets. To prevent such leakage via the scheduler, we require that the interface between programs and the scheduler is restricted.

Definition 3.8. Let $X \subseteq Var$. The observation function obs is *confined to* X if it satisfies the following condition for all multi-threaded configurations $\langle thr_1, mem_1\rangle$ and $\langle thr_2, mem_2\rangle$:

$$(\sharp(thr_1) = \sharp(thr_2) \wedge mem_1 =_X mem_2) \Rightarrow obs(\langle thr_1, mem_1\rangle) = obs(\langle thr_2, mem_2\rangle) \quad \Diamond$$

If the observation function is confined to the set of variables X then the observation function provides at most information about the number of threads and the values of variables in X. I.e., if the observation function is confined to L then the scheduler does not obtain information about the values of high variables or information about the control state of the threads via the interface between program and scheduler. We assume in the following that the observation function is confined to the set L.

Confining the observation function to L is not sufficient for the scheduler independence of FSI-security, because the number of threads in an FSI-secure thread pool might depend on secrets. A simple solution to obtain scheduler independence is to require that schedulers cannot see the number of threads by confining the observation function further. However, this would exclude standard schedulers, like, for instance, Round-Robin schedulers. Another possible solution is to strengthen the definition of FSI-security such that the number of threads in an FSI-secure program never depends on secrets. But then all programs where the number of threads might depend on secrets would be rejected, even those programs that are intuitively secure under standard schedulers.

The key to a solution to this problem was (a) the insight that FSI-security implies $\mathcal{S}$-noninterference if the scheduling order of low threads under $\mathcal{S}$ does not depend on the number of high threads in the thread pool, and (b) the observation that various typical schedulers satisfy this property on the scheduling order. In fact, it is even sufficient to require that the scheduling order of low threads does not depend on the presence of high threads in the thread pool. We formalize this requirement in the following, which leads to the novel class of *robust schedulers*. We formalize the class of robust schedulers in Definition 3.12 based on the auxiliary notion of $\mathcal{S}$-simulations formalized in Definition 3.11.

Definition 3.9. Let $sc = \langle thr, mem, sst\rangle$ be a system configuration and let $\mathcal{S}$ be a scheduler. The *probability that $\mathcal{S}$ selects a low thread in sc* is defined as

$$l\text{-}\rho_{\mathcal{S}}(sc) = \sum_{\{k \mid thr[k] \in LCom\}} \rho^{\mathcal{S}}_{sc}(k). \quad \Diamond$$

Definition 3.10. We extend the projection of thread pools to low commands to the projection of system configurations to low commands by

$$\langle thr, mem, sst\rangle|_{LCom} = \langle thr|_{LCom}, mem, sst\rangle. \qquad \Diamond$$

Definition 3.11. Let $\mathcal{S} = (sSt, sst_0, \rightarrow)$ be a scheduler. A binary relation on system configurations $\prec$ is an *$\mathcal{S}$-simulation* if and only if whenever $sc_1 \prec sc_2$ with $thr_1 = getThr(sc_1)$ and $thr_2 = getThr(sc_2)$ then $\sharp(thr_1|_{LCom}) = \sharp(thr_2|_{LCom}) = \sharp(thr_2)$ and for all $k_1 \in \{1, \ldots, \sharp(thr_1)\}$, $p_1 \in]0, 1]$, and $sc_1' \in SysConf$ with $sc_1 \xrightarrow{k_1,p_1}_{\mathcal{S}} sc_1'$ the following conditions are satisfied:

1. if $thr_1[k_1] \in LCom$ then there exists $sc_2' \in SysConf$ such that
   a) $sc_2 \xrightarrow{k_2,p_2}_{\mathcal{S}} sc_2'$ with $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$ and $p_2 = p_1/l\text{-}\rho_{\mathcal{S}}(sc_1)$, and
   b) $sc_1' \prec sc_2'|_{LCom}$; and
2. if $thr_1[k_1] \in HCom$ then $sc_1' \prec sc_2$.

The relation $\prec_{\mathcal{S}}$ is the union of all $\mathcal{S}$-simulations. $\Diamond$

Theorem 3.9. For every scheduler $\mathcal{S}$ the relation $\prec_{\mathcal{S}}$ is an $\mathcal{S}$-simulation. $\Diamond$

Proof. The theorem follows directly from the definition of $\prec_{\mathcal{S}}$ as a union of $\mathcal{S}$-simulations and the definition of $\mathcal{S}$-simulations. □

$\mathcal{S}$-simulations relate arbitrary system configurations with system configurations that do not contain high threads (this follows from the requirement $\sharp(thr_2|_{LCom}) = \sharp(thr_2)$ in Definition 3.11). If two system configurations sc_1 and sc_2 are related by an $\mathcal{S}$-simulation, then the sequence in which the low threads in sc_1 are scheduled by $\mathcal{S}$ is also possible for the threads in sc_2. This leads to the definition of robust schedulers.

Definition 3.12. The scheduler $\mathcal{S} = (sSt, sst_0, \rightarrow)$ is *robust* if

$$\langle thr, mem, sst_0\rangle \prec_{\mathcal{S}} \langle thr, mem, sst_0\rangle|_{LCom}$$

for each FSI-secure thread pool thr and each memory mem. $\Diamond$

Intuitively, if a scheduler is robust then the scheduling order of low threads during an execution of an FSI-secure thread pool remains unchanged when one removes all high threads from the thread pool. We make the requirement in the definition of robustness only for FSI-secure thread pools, because this enlarges the class of robust schedulers without losing scheduler independence of FSI-security for robust schedulers.

The class of robust schedulers contains various typical schedulers, including the Round-Robin, the uniform, and the lottery scheduler introduced in Section 2.4.

Theorem 3.10. The Round-Robin scheduler RR (see Example 2.3) is robust. $\Diamond$

Theorem 3.11. The uniform scheduler Uni (see Example 2.4) is robust. $\Diamond$

Theorem 3.12. The Lottery scheduler Lot (see Example 2.5) is robust if the variables that store the number of tickets for each thread are low. $\Diamond$

Intuitively, these schedulers are robust because the order in which they select low threads does not change when removing or adding high threads to the thread pool. To prove Theorems 3.10 – 3.12 we define appropriate simulation relations and show that they are actually $\mathcal{S}$-simulations for the respective schedulers. In the following proof sketches we show how we define the simulation relations, because these definitions are the key to the proofs, and refer to Appendix A.2 for the proofs that the relations are $\mathcal{S}$-simulations.

Proof sketch for Theorem 3.10. Define $\langle thr_1, mem_1, sst_1\rangle \prec \langle thr_2, mem_2, sst_2\rangle$ if and only if the following conditions are satisfied:

1. $thr_1 \sim_{lm} thr_2$,
2. $mem_1 =_L mem_2$,
3. $thr_1|_{LCom} = thr_2$, and
4. if $k_1 = [(sst_1(choice) + (\sharp(thr_1) - sst_1(size)) \mod \sharp(thr_1)] + 1$ and $thr_1[k_1] \in LCom$ then $l\text{-}match_{thr_1,thr_2}(k_1) = [(sst_2(choice) + (\sharp(thr_2) - sst_2(size)) \mod \sharp(thr_2)] + 1$.

The relation $\prec$ is a *RR*-simulation by Lemma A.6 (see Appendix A.2).

Let thr be an FSI-secure thread pool and mem a memory. Then

$$\langle thr, mem, sst_{0,RR}\rangle \prec \langle thr, mem, sst_{0,RR}\rangle|_{LCom}$$

(where Item 1 in the definition of $\prec$ follows from Lemma A.5). □

Proof sketch for Theorem 3.11. Define $\langle thr_1, mem_1, sst_1\rangle \prec \langle thr_2, mem_2, sst_2\rangle$ if and only if the following conditions are satisfied:

1. $thr_1 \sim_{lm} thr_2$,
2. $mem_1 =_L mem_2$, and
3. $thr_1|_{LCom} = thr_2$.

The relation $\prec$ is a *Uni*-simulation by Lemma A.7 (see Appendix A.2).

Let thr be an FSI-secure thread pool and mem a memory. Then

$$\langle thr, mem, sst_{0,Uni}\rangle \prec \langle thr, mem, sst_{0,Uni}\rangle|_{LCom}$$

(where Item 1 in the definition of $\prec$ follows from Lemma A.5). □

Proof sketch for Theorem 3.12. Consider the same relation $\prec$ as in the proof of Theorem 3.11 above. The relation is a *Lot*-simulation by Lemma A.8 (see Appendix A.2).

Let thr be an FSI-secure thread pool and mem a memory. Then

$$\langle thr, mem, sst_{0,Lot}\rangle \prec \langle thr, mem, sst_{0,Lot}\rangle|_{LCom}$$

(where Item 1 in the definition of $\prec$ follows from Lemma A.5). □

The above examples illustrate that typical schedulers are robust. Non-robust schedulers exhibit different scheduling orders when adding or removing threads from the thread pool. This is nicely illustrated by a class of schedulers that we call *frog schedulers*. Frog schedulers derive from Round-Robin schedulers, but, in contrast to Round-Robin schedulers, frog schedulers "jump" over one or more threads when selecting the next thread. To our knowledge, frog schedulers are not used in practice for scheduling programs.

Example 3.6. We define a *frog scheduler* by modifying the definition of the Round-Robin scheduler from Example 2.3, replacing the equation

$$k = \big[(sst(choice) + (\sharp(sin) - sst(size))) \mod \sharp(sin)\big] + 1$$

by

$$k = \big[(sst(choice) + (\sharp(sin) - sst(size))) \mod \sharp(sin)\big] + 2.$$

This frog scheduler is not robust: In a thread pool with two low threads the scheduler selects the first thread until this thread terminates. When adding a high thread, the frog scheduler firstly selects the first low thread, then the high thread, and then the second low thread (assuming that the first low thread did not terminate after one execution step).◊

3.3.4 Scheduler-independence Result

We are now ready to establish the main theorem of this chapter, the scheduler independence result for FSI-security.

Theorem 3.13. Assume that the observation function *obs* is confined to L. Then every terminating FSI-secure thread pool is $\mathcal{S}$-noninterferent (cf. Definition 2.19) for every robust scheduler $\mathcal{S}$. ◊

Theorem 3.13 shows that terminating FSI-secure thread pools satisfy the simple noninterference-like security condition $\mathcal{S}$-noninterference for all robust schedulers. In particular, Theorem 3.13 reduces the adequacy argument for FSI-security to the adequacy argument for $\mathcal{S}$-noninterference. Theorems 3.10–3.12 indicate that FSI-security is scheduler-independent for a practically relevant class of schedulers. Moreover, it follows from Theorem 3.13 that FSI-security is scheduler independent in the sense of Definition 3.1.

Corollary 3.1. Assume that the observation function *obs* is confined to L. Then FSI-security is scheduler independent with respect to $\mathcal{S}$-noninterference for the set of all terminating thread pools and the set of robust schedulers. ◊

Proof. The corollary follows immediately from Theorem 3.13 and Definition 3.1. □

The proof of Theorem 3.13 uses an induction over the number of execution steps of a terminating thread pool under a robust scheduler $\mathcal{S}$. To obtain a sufficiently strong induction hypothesis we prove the following generalization of the theorem which relates trace probabilities for thread pools that are low bisimilar modulo low matching:

Theorem 3.14. Let $\mathcal{S}$ be a robust scheduler, and assume that *obs* is confined to L. Moreover, let $sc_1 = \langle thr_1, mem_1, sst_1\rangle$ and $sc_2 = \langle thr_2, mem_2, sst_2\rangle$ be terminating system configurations, and let $sc_{1,p} = \langle thr_{1,p}, mem_{1,p}, sst_p\rangle$ and $sc_{2,p} = \langle thr_{2,p}, mem_{2,p}, sst_p\rangle$ be system configurations such that the following conditions are satisfied:

1. $thr_1 \sim_{lm} thr_2$, $thr_1 \sim_{lm} thr_{1,p}$, and $thr_2 \sim_{lm} thr_{2,p}$,
2. $mem_1 =_L mem_2$, $mem_1 =_L mem_{1,p}$, and $mem_2 =_L mem_{2,p}$, and
3. $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and $sc_2 \prec_{\mathcal{S}} sc_{2,p}$.

Then the following holds for all $mem \in Mem$:

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$$

◊

$$[\mathsf{EXP}]\ \frac{}{\vdash e : \bigsqcup_{x \in vars(e)} dma(x)}$$

Figure 3.1: Typing rule for expressions

Intuitively the assumptions in Theorem 3.14 express relations between system configurations that are reached after n execution steps of low threads from four initial system configurations with L-equal memories: Configurations sc_1 and sc_2 result from two initial configurations, and $sc_{1,p}$ and $sc_{2,p}$ result from the same configurations from which all high threads have been removed. A proof of Theorem 3.14 is provided in Appendix A.3.

Proof of Theorem 3.13. Let $\mathcal{S}$ be a robust scheduler. Moreover, let thr be a terminating FSI-secure thread pool and let mem_1 and mem_2 be memories such that $mem_1 =_{Var \setminus HI} mem_2$. Then $mem_1 =_L mem_2$ because $L \subseteq Var \setminus HI$. Moreover, by Lemma A.5, $thr \sim_{lm} thr|_{LCom}$. Hence, all requirements of Theorem 3.14 are satisfied when defining $sc_1 = \langle thr, sst_0, mem_1 \rangle$, $sc_2 = \langle thr, sst_0, mem_2 \rangle$, $sc_{1,p} = \langle thr|_{LCom}, mem_1, sst_0 \rangle$, and $sc_{2,p} = \langle thr|_{LCom}, mem_2, sst_0 \rangle$. Hence,

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$$

for all $mem \in Mem$. I.e., thr is $\mathcal{S}$-noninterferent. □

Remark 3.4. FSI-security is a termination-insensitive security property. For example, the command $c = \mathsf{while}\ h = 0\ \mathsf{do\ skip\ od}$ is FSI-secure, although the termination of c depends on the initial value of h. In contrast, $\mathcal{S}$-noninterference is a termination-sensitive property (cf. Theorem 2.7), and, in particular, c is not $\mathcal{S}$-noninterferent for any scheduler $\mathcal{S}$. This is why we established the scheduler independence result for FSI-security for the set of terminating programs. Checking the termination behavior of a program is possible with existing analysis techniques (see, e.g., [CS02, OBvEG10]).

3.4 A Type-based Security Analysis

We present a security type system to illustrate how an automated analysis of programs with respect to FSI-security could be implemented. The security type system is for the programming language from Section 2.3. For the formalization of the security type system, we assume that the domain assignment for variables is given by dma.

The type system consists of judgments for expressions, commands, and thread pools. Typing judgments for expressions are of the form

$$\vdash e : d$$

with $e \in Exp$ and $d \in \{low, high\}$. The interpretation of the judgments is that d is an upper bound on the security level of the value of expression e (cf. typing rule [EXP] in Figure 3.1). I.e., the value of e might depend on the value of a variable x with domain $high$ if $d = high$, whereas the value of e only depends on variables with domain low if $d = low$.

Typing judgments for commands are of the form

$$\vdash c : (mdf, stp)$$

$$[\mathsf{STOP}]\frac{}{\vdash \mathit{stop} : (\mathit{high}, \mathit{low})} \qquad [\mathsf{SKIP}]\frac{}{\vdash \mathsf{skip} : (\mathit{high}, \mathit{low})}$$

$$[\mathsf{ASS}]\frac{\vdash e : d \qquad d \sqsubseteq \mathit{dma}(x)}{\vdash x{:=}e : (\mathit{dma}(x), \mathit{low})} \qquad [\mathsf{SPAWN}]\frac{\forall i \in \{1, \ldots, k\} : \vdash c_i : (\mathit{mdf}, \mathit{stp}_i)}{\vdash \mathsf{spawn}(c_1, \ldots, c_k) : (\mathit{mdf}, \mathit{low})}$$

$$[\mathsf{IF}]\frac{\vdash c_1 : (\mathit{mdf}, \mathit{stp}) \qquad \vdash c_2 : (\mathit{mdf}, \mathit{stp}) \qquad \vdash e : d \qquad d \sqsubseteq \mathit{mdf}}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi} : (\mathit{mdf}, \mathit{stp} \sqcup d)}$$

$$[\mathsf{WHILE}]\frac{\vdash c : (\mathit{mdf}, \mathit{stp}) \qquad \vdash e : d \qquad \mathit{stp} \sqcup d \sqsubseteq \mathit{mdf}}{\vdash \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} : (\mathit{mdf}, \mathit{stp} \sqcup d)}$$

$$[\mathsf{SEQ}]\frac{\vdash c_1 : (\mathit{mdf}_1, \mathit{stp}_1) \qquad \vdash c_2 : (\mathit{mdf}_2, \mathit{stp}_2) \qquad \mathit{stp}_1 \sqsubseteq \mathit{mdf}_2}{\vdash c_1; c_2 : (\mathit{mdf}_1 \sqcap \mathit{mdf}_2, \mathit{stp}_1 \sqcup \mathit{stp}_2)}$$

$$[\mathsf{SUB}]\frac{\vdash c : (\mathit{mdf}', \mathit{stp}') \qquad \mathit{mdf} \sqsubseteq \mathit{mdf}' \qquad \mathit{stp}' \sqsubseteq \mathit{stp}}{\vdash c : (\mathit{mdf}, \mathit{stp})}$$

Figure 3.2: Typing rules for commands

where $c \in \mathit{Com}$ and $\mathit{mdf}, \mathit{stp} \in \{\mathit{low}, \mathit{high}\}$. The interpretation of the judgments is as follows: The security domain mdf (for "modify") is a lower bound on the security domain of the variables which a thread with control state modeled by c modifies. I.e., if $\mathit{mdf} = \mathit{high}$, then the command c is a high command, while if $\mathit{mdf} = \mathit{low}$ then c might be a high or a low command. Moreover, the security domain stp (for "steps") is an upper bound on the security domain of variables on whose values the number of execution steps of a thread with control state modeled by c depends (not counting execution steps of spawned threads). I.e., if $\mathit{stp} = \mathit{low}$ then the number of execution steps does not depend on the values of high variables, and if $\mathit{stp} = \mathit{high}$ then the number of execution steps might depend on the values of high variables.

The typing rules for commands are displayed in Figure 3.2. Subtyping is covariant in the first component of the pair $(\mathit{mdf}, \mathit{stp})$ (compare rule [SUB]), because the interpretation is that mdf is a lower bound, and contravariant in the second component, because the interpretation is that stp is an upper bound. To prevent direct information leaks, rule [ASS] ensures that assignments of high expressions to low variables are not typable. Moreover, to prevent indirect information leaks rules [IF] and [WHILE] prevent typability of commands that assign to low variables under high control conditions by requiring $d \sqsubseteq \mathit{mdf}$. Rule [SPAWN] permits to type commands that dynamically spawn threads. Since a spawn-command always takes one execution step, the typing rule states that $\mathit{stp} = \mathit{low}$. Moreover, the typing rule states that $\mathit{mdf} = \mathit{high}$ only if each command in the list of spawned threads is typable with $\mathit{mdf} = \mathit{high}$, i.e., if all spawned threads modify only high variables. The conditions $\mathit{stp}_1 \sqsubseteq \mathit{mdf}_2$ and $\mathit{stp} \sqsubseteq \mathit{mdf}$ in rules [SEQ] and [WHILE], respectively, ensure that if a thread is low then the number of execution steps that it performs does not depend on the values of high variables. These restrictions on the typability of sequentially composed threads and loops are essential for the type system's soundness, because they ensure that lock-step execution is possible for threads that execute low commands.

Definition 3.13. A command c is *typable with type* $(\mathit{mdf}, \mathit{stp})$ if and only if $\vdash c : (\mathit{mdf}, \mathit{stp})$

is derivable in the type system. A command c is *typable* if there exist $mdf, stp \in \{low, high\}$ such that $\vdash c : (mdf, stp)$ is derivable in the type system. ◊

Before introducing typing judgments for thread pools, we establish a soundness result for the typing rules for commands.

Theorem 3.15. Let $c \in Com$. If c is typable then c is FSI-secure. ◊

To prove Theorem 3.15, we establish three lemmas about typability, concerning subject reduction, high threads, and execution steps in L-equal memories.

Lemma 3.1. Let $c \in Com$. Assume that c is typable with type (mdf, stp) and that $\langle c, mem\rangle \xrightarrow{new(thr)} \langle c', mem'\rangle$. Then c' is typable with (mdf', stp') where $mdf \sqsubseteq mdf'$ and $stp' \sqsubseteq stp$. Moreover, $thr[i]$ is typable for all $i \in \{1, \ldots, \sharp(thr)\}$. ◊

Lemma 3.2. Let $c \in Com$. Assume that c is typable with type $(high, stp)$ for some $stp \in \{low, high\}$. Then $c \in HCom$. ◊

Lemma 3.3. Let $c \in Com$. Assume that c is typable with type (mdf, stp). Assume furthermore that $\langle c, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$ and $\langle c, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$ for L-equal memories mem_1 and mem_2. Then the following conditions are satisfied:
- If $stp = low$ then $c_1' = c_2'$, $\alpha_1 = \alpha_2$, and $mem_1' =_L mem_2'$.
- If $c_1' \neq c_2'$ then $stp = high$. ◊

The proof for each of the three lemmas is by induction over the derivation of typing judgments. We provide detailed proofs in Appendix A.4.

Now we sketch the proof of Theorem 3.15.

Proof Sketch. We define the relation $\mathcal{R}$ on thread pools by $thr_1 \mathcal{R} thr_2$ if and only if the following conditions are satisfied:
- thr_1 and thr_2 contain the same number of low threads and
- if $i \in \{1, \ldots, \sharp(thr_1)\}$ and $thr_1[i] \in LCom$ then $thr_1[i] = thr_2[l\text{-}match_{thr_1, thr_2}(i)]$ and $thr_1[i]$ is typable.

Then $\langle c\rangle \mathcal{R} \langle c\rangle$ if c is typable. To show that typable commands are FSI-secure it hence suffices to show that $\mathcal{R}$ is a low bisimulation modulo low matching.

A detailed proof that $\mathcal{R}$ is a low bisimulation modulo low matching is available in Appendix A.4, here we only sketch the proof. We must show that whenever $thr_1 \mathcal{R} thr_2$ then (a) if $thr_1[i] \in LCom$ then an execution step of $thr_1[i]$ can be matched by an execution step of $thr_2[l\text{-}match_{thr_1, thr_2}(i)]$ in each L-equal memory, resulting in related thread pools and L-equal memories, and (b) if $thr_1[i] \in HCom$ then an execution step of $thr_1[i]$ results in related thread pools. While (b) follows directly from the definition of $\mathcal{R}$, for proving (a) we exploit that $\vdash thr_1[i] : (mdf, stp)$ is derivable by the definition of $\mathcal{R}$ and do a proof by induction on the derivation of this judgment. Note that $mdf = low$, because otherwise $thr_1[i] \in HCom$ would hold by Lemma 3.2.

We sketch the proof for three more interesting cases, namely derivations via rules [ASS], [IF], and [SEQ].

If the derivation is via [ASS] (i.e., $thr_1[i] = x{:=}e$) the typing rule ensures that the security level of e is low, and, hence, that the value of e only depends on low variables. In consequence, executing the assignment in L-equal memories results in L-equal memories.

If the derivation is via [IF] (i.e., $thr_1[i] =$ if e then c_1 else c_2 fi) it follows from $mdf = low$ and rule [IF] that e is typable with low, i.e., the value of e depends only on

$$[\mathsf{PAR}]\ \frac{\forall i \in \{1,\ldots,n\} : \vdash thr[i] : (mdf_i, stp_i)}{\vdash thr}$$

Figure 3.3: Typing rule for thread pools

low variables. Hence, transitions of $thr_1[i]$ in L-equal memories result either both in c_1 or both in c_2, which are also typable by Lemma 3.1.

If the derivation is via [SEQ] (i.e., $thr_1[i] = c_1; c_2$) then $\vdash c_1 : (mdf_1, stp_1)$ and $\vdash c_2 : (mdf_2, stp_2)$ are derivable. Assume that $c_1 \neq stop$ (for $c_1 = stop$ the proof is straightforward). Then, by the induction hypothesis, steps of c_1 in L-equal memories result in L-equal memories. It remains to show that the resulting commands, call them $c_1''; c_2$ and $c_2''; c_2$, are either both high or are equal, low, and typable. By the induction hypothesis, c_1'' and c_2'' are either both high or are equal, low, and typable. If they are equal it is straightforward to see that $c_1''; c_2$ and $c_2''; c_2$ are either both high or equal, low, and typable using rule [SEQ]. If $c_1'' \neq c_2''$ are both high, then $stp_1 = high$ by Lemma 3.3. In consequence, by typing rule [SEQ] it follows that $mdf_2 = high$, i.e., c_2 is a high command by Lemma 3.2. Hence, $c_1''; c_2$ and $c_2''; c_2$ are both high commands. □

Typing judgments for thread pools are of the form

$$\vdash thr,$$

where $thr \in Thr$ is a thread pool. The corresponding typing rule in Figure 3.3 requires that $thr[i]$ is typable for all $i \in \{1,\ldots,\sharp(thr)\}$. The soundness of the type system for thread pools then follows directly from the compositionality of FSI-security and Theorem 3.15.

Theorem 3.16. If the judgment $\vdash thr$ is derivable then thr is FSI-secure. ◊

Proof. Since $\vdash thr$ is derivable, $thr[i]$ is typable for all $i \in \{1,\ldots,\sharp(thr)\}$ by rule [PAR]. Hence, by Theorem 3.15, $thr[i]$ is FSI-secure for all $i \in \{1,\ldots,\sharp(thr)\}$. In consequence, by compositionality of FSI-security (Theorem 3.6) thr is FSI-secure. □

Remark 3.5. The type systems proposed in [Smi01, BC02, Smi03, ABC07] are similar to our type system because they restrict the assignments a program may perform after executing a conditional or a loop with a high control condition. However, [Smi01, Smi03, ABC07] guarantee soundness only for one scheduler-specific security property. The type system in [BC02] is for a language that allows dynamic thread creation only in a limited form (no threads may be created inside loops), and the article assumes that threads idle after their termination instead of being removed from the thread pool. Our type system and its soundness result do not share these limitations. The scheduler independence result from [BC02] will be further compared to the result in this thesis in Section 3.6. ◊

3.5 Example Security Analysis

We illustrate the benefits of FSI-security and the security type system at the security analysis of a simple, yet realistic multi-threaded example program.

Consider the code fragment in Figure 3.4, which is part of an application managing the personal finances of the user of the program. The input to the program is stored in

Initial thread :
spawn(*writeStockPricesToDB*);
spawn(*writeFundsPricesToDB*);
spawn(*computeAccountOverview*)

writeStockPricesToDB:
i_l:=0;
while (i_l < getSize(stockPricesIn$_l$))
do
databaseOut$_l$:=databaseOut$_l$
+getTitleAt(stockPricesIn$_l$, i_l)
+getPriceAt(stockPricesIn$_l$, i_l);
i_l:= i_l+1 od

writeFundsPricesToDB:
j_l:=0;
while (j_l < getSize(fundsPricesIn$_l$))
do
databaseOut$_l$:=databaseOut$_l$
+getTitleAt(fundsPricesIn$_l$, j_l)
+getPriceAt(fundsPricesIn$_l$, j_l);
j_l:= j_l+1 od

computeAccountOverview:
k_h:= 0; overviewOut$_h$:= "";
while (k_h < getSize(userPortfolioIn$_h$)) do
title$_h$:=getTitleAt(userPortfolioIn$_h$, k_h);
if (isStock(title$_h$)) then
price$_h$:=getPrice(stockPricesIn$_l$, title$_h$)
∗ getQuantityAt(userPortfolioIn$_h$, k_h)
else
price$_h$:=getPrice(fundsPricesIn$_l$, title$_h$)
∗ getQuantityAt(userPortfolioIn$_h$, k_h)
fi;
oldPrice$_h$:=
getLastPrice(databaseIn$_l$, title$_h$);
if (oldPrice$_h$ ≤ price$_h$)
then tendency$_h$:= "up"
else tendency$_h$:= "down"
fi;
overviewOut$_h$:=overviewOut$_h$+title$_h$
+price$_h$+tendency$_h$;
k_h:= k_h+1
od

Figure 3.4: Example code fragment

the input variables stockPricesIn$_l$ and fundsPricesIn$_l$ (up-to-date pricing information about stocks and funds, respectively), databaseIn$_l$ (a database with historical pricing information that is administered by the program), and userPortfolioIn$_h$ (containing information about the user's custody account). The output computed by the program is stored in the output variables databaseOut$_l$ (novel pricing information to be stored in the database) and overviewOut$_h$ (an overview of the user's custody account).

To compute the novel pricing information to be stored in the database, the program spawns two threads executing the commands *writeStockPricesToDB* and *writeFundsPricesToDB*, respectively. A third spawned thread (*computeAccountOverview*) generates the overview of the user's custody account and stores it in the variable overviewOut$_h$. The three tasks are spawned in separate threads to not block any computations following this code fragment in the overall application.

We model the data used in the example program by String values. To access data encoded in String values, we use selector expressions like, e.g., getTitleAt or getPriceAt.

The subscripts of variables indicate whether the domain assignment classifies a variable as public (l) or as secret (h). Information about historical pricing information (stored within the database) and information about up-to-date pricing information is classified as public, while the user's portfolio as well as the computed overview of the portfolio are classified as secret. The variables that do not model inputs or outputs are classified in a way that permits to type-check the program with the type system from Section 3.4. For instance, variable title$_h$ is classified as secret because it is used to store information that is derived from the secret portfolio, and variable i_l is classified as public because it is used to iterate over the entries of the public variable stockPricesIn$_l$.

We use the security type system to show that the program is FSI-secure.

Theorem 3.17. The program from Figure 3.4 is FSI-secure. ◊

Proof. The initial thread as well as the threads executing the commands *writeStockPrices-ToDB* and *writeFundsPricesToDB* are typable with the type (low, low) (they or threads that they spawn write to low variables, and they do only contain low control conditions). Moreover, the command *computeAccountOverview* is typable with the type $(high, high)$ (it assigns only to high variables and contains high control conditions). □

In contrast, the program is rejected by other information flow analyses that guarantee the absence of insecure information flows under common schedulers, because the underlying scheduler-independent information flow properties are not satisfied for the program. Properties based on the idea of observational determinism [ZM03] are violated because the order in which entries are written to the database is nondeterministic (the order depends on the order in which the threads *writeStockPricesToDB* and *writeFundsPri-cesToDB* are selected by the scheduler). Strong security [SS00] is violated because the number of execution steps of the loop in the thread *computeAccountOverview* depends on secret information.

Since the program is terminating, it is $\mathcal{S}$-noninterferent for robust $\mathcal{S}$ by Theorem 3.13. This shows that although the number of execution steps depends on secret information the public outputs do not. In particular, the order of entries in variable $\mathsf{databaseOut}_l$ does not depend on secret information.

3.6 Summary and Comparison to Related Work

In this chapter, we have presented the novel information flow security property *FSI-security*, and we have shown that FSI-security is scheduler independent with respect to a class of relevant schedulers, the robust schedulers.

We defined FSI-security using a standard approach that is based on partial equivalence relations (pers) and bisimulations. Using a bisimulation-based security definition, firstly, permitted us to easily derive a compositionality result for FSI-security. Secondly, since pers and bisimulations are also used for other advances in the characterization of information flow security, we are confident that it is possible to integrate FSI-security with other information flow properties defined using the same approach, like, e.g., properties that permit controlled declassification as proposed in [MR07]. In Chapter 5 we show how to integrate FSI-security with a per-based information flow property that exploits assumption-guarantee based reasoning and that we present in the following chapter.

With the development of FSI-security we were able to overcome three major deficiencies of other approaches to scheduler-independent information flow security: Incompatibility with programs that have nondeterministic public output, incompatibility with programs for which the number of execution steps depends on secrets, and incompatibility with standard scheduler interfaces. Moreover, we were not only able to overcome these deficiencies in the security property, but also in a provably sound type system. In the following, we discuss these improvements with respect to other approaches to scheduler-independent information flow security in more detail. An overview of the benefits of FSI-security and the approaches discussed in the following is displayed in Table 3.1.

Observational Determinism. The idea of *observational determinism* goes back to McLean [McL92] and Roscoe et al [RWW94, Ros95], who proposed information flow properties not at the level of program implementations but for more abstract specifications. The idea

	Scheduler-independence Result	Standard Scheduler Interface	Compatible with Nondeterministic Public Output	Compatible with Timing Differences Depending on Secrets	Compatible with Low Assignments after High Loop Guard (but Blocks During Loop)	Compositionality Result
[VS98]	✗	✓	✓	✓	✓	✓
[SS00]	✓	✓	✓	✗	✗	✓
[ZM03, HWS06]	✓	✓	✗	✓	✗	✓[1]
[BC02]	✗	✗	✓	✓	✗	✓
[RS06a, BRRS07, RS09]	✓	✗	✓	✓	✓	✓
[RS06b]	✗[2]	✗	✓	✓	✓	✓
FSI-security	✓	✓	✓	✓	✗	✓

[1] Compositionality requires that the composed threads do not have races.
[2] It seems plausible that a scheduler-independence result could be established.

Table 3.1: Comparison of approaches to information flow security with scheduling

was adapted to a language-based setting in [ZM03, HWS06]. Observational determinism requires that public observations of program executions are deterministic regardless of the interleaving of threads and the values of secret variables. If this requirement is satisfied, reducing the possible interleavings by assuming a concrete scheduler cannot result in a dependency of public observations on secrets. Observational determinism has the drawback that it forbids useful nondeterminism which occurs, for instance, when multiple threads append data to the same public variable (as in the example program analyzed in Section 3.5).

Strong Security. Sabelfeld and Sands [SS00] introduce the property *strong security* which is scheduler independent for a natural class of schedulers. Strong security is quite restrictive, because it requires that the runtime of a program must not depend on secret data. This drawback appeared unavoidable because Sabelfeld [Sab03] proved that strong security is the weakest compositional property that implies information flow security for the natural class of schedulers. Hence, strong security was used, despite its restrictiveness, as the foundation of many later developments (e.g., [MS03, FRS05, KM07]) and has been generalized in various ways, e.g., for distributed systems [MS03] or to control declassification [LM09]. While [SS00] proposes a security type system that can transform some insecure programs into strongly secure programs, only a subset of the intuitively

secure programs is amenable to this approach. For instance, the security type system does not transform the FSI-secure example from Section 3.5 into a strongly secure program.

Remark 3.6. The combining calculus [MSK07] (joint work of the author of this thesis with Heiko Mantel and Tina Kraußer) was a first step towards combining approaches based on observational determinism and strong security by making the combination of different analysis techniques in an information flow analysis feasible. However, the information flow property considered in [MSK07] is not scheduler independent with respect to standard schedulers like Round-Robin schedulers, and no scheduler independence result has so far been established for the combining calculus. ◊

Controlled Thread Systems. Boudol and Castellani [BC02] proposed a security type system for *controlled thread systems* that consist of a thread pool and a scheduler. If a controlled thread system is typable, then the thread pool is secure under the scheduler. In contrast to FSI-security, the approach requires the size of a thread pool to remain fixed during a program run. In particular, dynamic thread creation is not supported, and threads remain in the thread pool even after their termination (and may still be selected by the scheduler). Boudol and Castellani argue that if the termination of certain threads would be signaled to the scheduler then controlled thread systems writing public variables cannot be typed. This is a non-standard restriction, because schedulers typically need to know the number of running threads when selecting the next thread.

Non-standard Scheduler Interfaces. As a different approach to relax the security property while remaining scheduler-independent, Russo and Sabelfeld [RS06a, BRRS07, RS09] proposed to use non-standard schedulers that provide a customized interface to the scheduled threads. Via two special commands, programs can *hide* (and at a later point *unhide*) a thread; the schedulers guarantee that during the execution of hidden threads no other threads are scheduled. The approach allows to securely execute programs containing threads that assign to low variables after performing computations whose runtime depends on secrets (hiding the thread during those computations), but at the cost that a scheduler with a non-standard interface must be used. Such threads are rejected by FSI-security, because they may cause information leakage when being executed under currently available schedulers.

Program Transformations and Cooperative Scheduling. Another approach that, like the approach discussed in the previous paragraph, prevents scheduling during computations whose runtime depends on secrets is followed by Russo and Sabelfeld [RS06b]. They propose a program transformation that introduces yield-statements into a program, where each yield-statement instructs the scheduler to select another thread, such that no yield-statements occur during computations depending on secrets, and rescheduling only occurs at yield-statements. The approach is implemented for a Round-Robin scheduler, but the article argues that it is applicable for a wide class of schedulers. The transformation entails that computations on secrets block all remaining threads. This is particularly critical when these computations are time-consuming. In contrast, FSI-security allows any computations to be interleaved with the executions of other threads.

CHAPTER 4

A Flow-sensitive Security Analysis

4.1 Introduction

Many information flow analyses are *flow-insensitive*. Flow-insensitive analyses do not take the order of program statements into account [NNH99, Sec. 2.5.6], i.e., they yield the same result for programs $P_1; P_2$ and $P_2; P_1$ where the semicolon denotes sequential composition of programs. In contrast, a *flow-sensitive* information flow analysis might give different results for the programs $P_1; P_2$ and $P_2; P_1$. Hence, flow-sensitivity permits higher precision for information flow analysis because the information flow in a program might change when reordering program statements. To increase precision, flow-sensitive information flow analyses were developed for sequential programs, e.g., using program logics [AB04], security type systems [HS06], and program dependence graphs [HS09].

In this chapter, our main goal is to develop a provably sound flow-sensitive information flow analysis for multi-threaded programs. A key challenge in this development is that for information flow analysis for multi-threaded programs there is a conflict between flow-sensitivity and compositionality. To solve this conflict, we propose an approach based on assumption-guarantee style reasoning. We define a novel information flow property, SIFUM-security, that exploits assumptions and guarantees about how threads access shared variables and that, thereby, enables a compositional and flow-sensitive analysis. We relate SIFUM-security to our reference property, $\mathcal{S}$-noninterference, with a scheduler independence result. Moreover, we develop a provably sound flow-sensitive security type system that enforces SIFUM-security. It is, to our knowledge, the first provably sound flow-sensitive information flow analysis for multi-threaded programs.

SIFUM-security and the flow-sensitive security type system use assumptions made for single threads about read and write accesses of other threads, and they use guarantees provided about a thread's own read and write accesses. SIFUM-security permits to use a flow-sensitive information flow analysis as long as valid assumptions are made. Moreover, for those parts of a program for which no assumptions are made the analysis can safely fall back to flow-insensitive reasoning. In addition to the security analysis one must verify that assumptions and guarantees are valid, and we sketch how this task can be approached

with the help of existing verification techniques. We illustrate the applicability of the flow-sensitive analysis at realistic examples.

Overview. In Section 4.2 we discuss the challenge of flow-sensitive information flow analysis for multi-threaded programs in more detail, and we informally introduce our solution based on assumption-guarantee style reasoning in Section 4.3. In Section 4.4 we define a semantic characterization of information flow that is compatible with assumption-guarantee style reasoning and show that it is compositional and scheduler-independent. In Section 4.5 we present a flow-sensitive security type system based on the semantic characterization, and illustrate benefits of the flow-sensitive analysis in Section 4.6. Section 4.7 summarizes the results and discusses the relation to other works.

4.2 Flow-sensitive Analysis and Multi-threading

A flow-sensitive analysis, in principle, permits higher precision than a flow-insensitive analysis if one wants to check a property of programs that depends on the order of the program statements. This is in particular so for information flow security. For instance, in contrast to a flow-insensitive information flow analysis a flow-sensitive analysis can exploit that a program overwrites the value of a secret variable with a public value before it copies the secret variable to a public sink. The following example illustrates this.

Example 4.1. Consider the program $h{:=}0; l{:=}h$, where variable h contains a secret input and l is a low output variable. Intuitively, the program has secure information flow, because it overwrites the secret in h with a constant before it copies the value of h to l.

Nevertheless, a flow-insensitive information flow analysis will not classify the program as secure, because the program obtained by reversing the order of the assignments, $l{:=}h; h{:=}0$, is insecure (it copies the secret in h into l). ◊

In contrast to a flow-insensitive analysis, a flow-sensitive analysis might also tolerate that a program copies a secret value to a variable classified as public, as long as the program overwrites the value of this public variable at a later point with a public value.

Example 4.2. Consider the program $l{:=}h; l{:=}0$, where l is a low output variable and h contains a secret value. Intuitively, the program has secure information flow, because it overwrites l with a constant value after copying the secret in h to l.

A flow-insensitive information flow analysis rejects the program because the program obtained by reordering the assignments, $l{:=}0; l{:=}h$, leaks the secret value in h into l. ◊

Several flow-sensitive information flow analyses for sequential programs classify the programs in Examples 4.1 and 4.2 as secure (for instance, using program logics (e.g., [AB04]), security type systems (e.g., [HS06]), or program dependence graphs (e.g., [HS09])).

In the following, we illustrate that a key challenge for developing a flow-sensitive information flow analysis for multi-threaded programs is that flow-sensitive reasoning as illustrated in the above examples conflicts with the desire that the parallel composition of any two programs classified as secure should also have secure information flow.

Example 4.3. The programs $h{:=}0; l{:=}h$ and $h{:=}h'$ are both classified as secure by existing flow-sensitive analyses for sequential programs (where h and h' contain secrets and l is a public output variable). However, executing the programs in parallel leaks the secret in h' into l if the assignment $h{:=}h'$ is executed between the other two assignments. ◊

Example 4.3 illustrates that it is unsound to exploit that a secret variable is overwritten with a public value before being copied to a public output, if in between another thread writes a secret value into that variable.

The next example illustrates that there is danger if one permits assignments of secrets to public variables even if the secret is later overwritten with a public value, because another thread might copy the intermediate secret value into another public variable.

Example 4.4. The programs $l{:=}h; l{:=}0$ and $l'{:=}l$ are both classified as secure by existing flow-sensitive analyses for sequential programs (where h contains a secret and l and l' are public output variables). However, executing the programs in parallel leaks the secret in h into l' if the assignment $l'{:=}l$ is executed between the other two assignments. ◊

Examples 4.3 and 4.4 illustrate that the parallel composition of two programs classified as secure by a flow-sensitive analysis for sequential programs might leak secret information. I.e., there is a conflict between flow-sensitivity and compositionality.

In the following sections, we develop a solution that permits flow-sensitive compositional analysis based on assumption-guarantee style reasoning.

4.3 Assumptions and Guarantees

To develop an information flow analysis for multi-threaded programs that is both compositional and flow-sensitive we exploit that programmers usually ensure that threads in a multi-threaded program access the shared memory in a coordinated way. For instance, if a variable is used in one thread for a computation and modifications of that variable by other threads would interfere with the computation in undesired ways, programmers ensure that other threads do not write the variable during the computation. Moreover, if other threads shall not access intermediate values of a variable during a computation in one thread, programmers ensure that the other threads only read the variable at defined points during the computation. To ensure such coordinated memory accesses, programmers might employ some form of synchronization.

To exploit such coordinated access to the shared memory in the information flow analysis, we develop an analysis that exploits assumption-guarantee style reasoning. Reconsider the thread $h{:=}0; l{:=}h$ from Example 4.1. Under the assumption that other threads do not write the value of h between the two assignments it is safe to exploit the order of the assignments in the information flow analysis. Moreover, for the thread $l{:=}h; l{:=}0$ from Example 4.2 it is safe to exploit the order of the assignments under the assumption that other threads do not read l between the assignments. This motivates the use of two types of assumptions in the analysis, namely assumptions of the form

"During some interval of the thread's execution other threads do not read *variable x"*,

and assumptions of the form

"During some interval of the thread's execution other threads do not write *variable x"*.

Such assumptions are sufficient to cover scenarios in which a thread exclusively accesses shared variables during some parts of its execution, and shares these variables with other threads at selected points during the execution. The first type of assumption is useful for the analysis of programs like in Example 4.2 that temporarily store secret values in low variables. The second type of assumption is useful for the analysis of programs like in Example 4.1 that store public values in high variables.

Moreover, we consider corresponding guarantees that express that a thread refrains from certain accesses to the shared memory. The guarantees are of the form

"During some interval of the thread's execution the thread does not read *variable x"*,

and

"During some interval of the thread's execution the thread does not write *variable x"*.

Where do the assumptions and guarantees come from? They could, for instance, be specified before the information flow analysis using annotations in the program source code, capturing the intended pattern of variable accesses. The annotations could be added, e.g., by the programmer or by the security analyst. In this case, the validity of assumptions and guarantees needs to be verified to ensure that an information flow analysis that exploits them is correct. The assumptions and guarantees could also be inferred using a program analysis, capturing some actual pattern of variable accesses. In this case, a soundness result for this program analysis could guarantee that the inferred assumptions and guarantees are valid. For the definition of a formal information flow property that exploits assumptions and guarantees we abstract from the way in which assumptions and guarantees are obtained by enriching the program semantics from Section 2 with information about assumptions and guarantees.

4.4 A Security Property with Assumptions and Guarantees

To formalize a security property that is compatible with assumptions and guarantees (as introduced informally in the previous section) we extend our formal model of program execution with information about assumptions and guarantees. This information is used solely for defining an information flow property and for proving soundness of assumption-guarantee based information flow analyses with respect to that property, and we extend the execution model such that information about assumptions and guarantees does not influence the transitions modeling program execution.

4.4.1 Modes

The assumptions and guarantees made for a thread in the information flow analysis may differ for different points in the thread's execution. We model the assumptions and guarantees made for a thread at some program point by associating each variable with a collection of *modes*. Each mode represents an assumption or a guarantee for the corresponding variable. This modeling approach borrows from the idea of access modes in file systems, where a file is associated with access modes representing how the file may be accessed. Similar to files being associated with multiple access modes (e.g., both read and write access), multiple modes representing assumptions and guarantees may be associated with a variable (using, e.g., two modes to represent the guarantee that the thread neither reads nor writes the variable).

Definition 4.1. A *mode* is an element of the set

$$Mod = \{asm\text{-}no\text{-}r, asm\text{-}no\text{-}w, guar\text{-}no\text{-}r, guar\text{-}no\text{-}w\}.$$

A *mode state* is a function $mds : Mod \to \mathcal{P}(Var)$. We denote the set of mode states with Mds. ◊

If $x \in mds(\textit{asm-no-r})$ or $x \in mds(\textit{asm-no-w})$ the mode state mds represents the assumption that no other thread reads or writes the variable x before the next execution step of the thread, respectively. Moreover, if $x' \in mds(\textit{guar-no-r})$ or $x' \in mds(\textit{guar-no-w})$ the mode state mds represents the guarantee that the thread itself does not read or write the variable x' in its next execution step, respectively.

To model that different assumptions are made and different guarantees are provided at different points of a thread's execution, we extend our model of snapshots during a thread's execution with mode states.

Definition 4.2. A *thread configuration with modes* is a triple $\langle c, mds, mem\rangle$, where $c \in Com$, $mds \in Mds$, and $mem \in Mem$. ◊

Moreover, we introduce judgments of the form

$$\langle c, mds, mem\rangle \xrightarrow{\alpha}_{\triangleright} \langle c', mds', mem'\rangle,$$

where $\alpha \in Lab_m = \{new(thr, mdss) \mid thr \in Thr \wedge mdss \in Mds^* \wedge \sharp(thr) = \sharp(mdss)\}$. In addition to modeling the single execution steps of a thread, the judgments model changes of the assumptions and guarantees between two points in the thread's execution. The event $new(thr, mdss) \in Lab_m$ represents the creation of new threads, where the list of mode states $mdss$ models the initial assumptions and guarantees for each new thread.

We use modes solely for the information flow analysis, and the modes shall have no impact on the program execution. We formalize this with the following requirement on the relation between judgments of the form $\langle c, mds, mem\rangle \xrightarrow{\alpha}_{\triangleright} \langle c', mds', mem'\rangle$ and judgments of the form $\langle c, mem\rangle \xrightarrow{\alpha}_{\triangleright} \langle c', mem'\rangle$.

Requirement 4.1. If $\langle c, mds, mem\rangle \xrightarrow{\alpha}_{\triangleright} \langle c', mds', mem'\rangle$ with $\alpha = new(thr, mdss)$ then $\langle c, mem\rangle \xrightarrow{\alpha'}_{\triangleright} \langle c', mem'\rangle$ with $\alpha' = new(thr)$.

Moreover, whenever $\langle c, mem\rangle \xrightarrow{\alpha}_{\triangleright} \langle c', mem'\rangle$ with $\alpha = new(thr)$, then for each mode state mds there exist a mode state mds' and a list of mode states $mdss'$ such that $\langle c, mds, mem\rangle \xrightarrow{\alpha'}_{\triangleright} \langle c', mds', mem'\rangle$ with $\alpha' = new(thr, mdss)$. ◊

We define derivation rules for the new judgments for an extension of the programming language from Section 2.3 in Section 4.5.1, taking care that Requirement 4.1 is satisfied.

Moreover, we extend our model of snapshots during the execution of multi-threaded programs with modes.

Definition 4.3. A *system configuration with modes* is a tuple $\langle thr, mdss, mem, sst\rangle$ where $thr \in Thr$, $mdss \in Mds^*$, $mem \in Mem$, $sst \in sSt$, and $\sharp(thr) = \sharp(mdss)$. ◊

Moreover, we introduce judgments of the form

$$(thr, mdss, mem, sst) \overset{k,p}{\Longrightarrow}_{\mathcal{S}} (thr', mdss', mem', sst')$$

and define derivability for these judgments with the derivation rule displayed in Figure 4.1. The rule is an extension of Rule (2.1) from Section 2.2.2 to judgments with modes. The rule specifies that the list of mode states is updated when the size of the thread pool changes in accordance with the update of the thread pool.

In the remainder, we denote thread configurations with modes with *tcm* and system configurations with modes with *scm*, both possibly with primes and subscripts.

$$\frac{\begin{array}{c}\langle thr[k], mdss[k], mem\rangle \xrightarrow{new(thr'', mdss'')} \langle c', mds', mem'\rangle \\ sin = obs(thr, mem) \quad (sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst' \quad thr' = update_k(thr, c', thr'') \\ c' = stop \Rightarrow mdss' = replace_k(mdss, mdss'') \\ c' \neq stop \Rightarrow mdss' = replace_k(mdss, \langle mds'\rangle :: mdss'')\end{array}}{\langle thr, mdss, mem, sst\rangle \overset{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr', mdss', mem', sst'\rangle}$$

Figure 4.1: Derivation rule for transitions of system configurations with modes

4.4.2 SIFUM-Security

We define a security property that is compatible with assumptions and guarantees, basing the definition on the Per-approach with bisimulation relations [SS99] (cf. Section 3.3.1). Like in Chapter 3, we extend the domain assignment *dma* to the set *Var* such that a security domain is also assigned to variables that do not represent input or output, and we denote the set of all variables with security domain *low* with L. In contrast to Chapter 3, we define a partial equivalence relation on commands (and not on thread pools) that models "indistinguishability" of commands with respect to executions in L-equal memories, and define a command as secure if and only if it is related to itself.

Usually, information flow properties that are defined based on a bisimulation-based partial equivalence relation not only require that two runs of a secure program starting in L-equal memories result in L-equal final memories, but require L-equality also at intermediate execution points (like, e.g., FSI-security in Section 3.3 and the properties in [SS00], [Sab03], [ZM03], and [MS04]). This requirement can be used to prove the compositionality of the resulting security property by exploiting that no thread ever stores secrets in low variables. We exploit assumptions that other threads do not read variables to relax this requirement at intermediate execution points without loosing compositionality. To this end, we permit a thread to store secrets in a low variable as long as one makes a no-read assumption for that thread and that variable. To capture this formally, we introduce the following relaxed variant of L-equality that requires equality only for low variables without mode *asm-no-r*.

Definition 4.4. Memories $mem_1, mem_2 \in Mem$ are *L-equal modulo mode state $mds \in Mds$* (denoted by $mem_1 =_L^{mds} mem_2$) if and only if

$$mem_1(x) \;=_{L \setminus mds(asm\text{-}no\text{-}r)}\; mem_2(x).$$

Moreover, in contrast to previous definitions of information flow properties that are based on partial equivalence relations and bisimulations we follow a two-step approach to distinguish explicitly between execution steps of a thread's environment and execution steps of a thread itself: In the first step, we consider memory modifications by the environment in a closure condition on the bisimulation that exploits no-write assumptions. In the second step, we consider the execution steps of a single thread in a bisimulation definition that exploits no-read assumptions.

Definition 4.5. A binary relation $\mathcal{R}$ on thread configurations with modes that have equal mode states is *closed under globally consistent changes* if for all $c_1, c_2 \in Com$, $mds \in Mds$, and $mem_1, mem_2 \in Mem$ with $\langle c_1, mds, mem_1\rangle \;\mathcal{R}\; \langle c_2, mds, mem_2\rangle$ the following conditions are satisfied for all $x \in Var$:

1. $(x \notin mds(asm\text{-}no\text{-}w) \wedge dma(x) = high) \Rightarrow$
   $\forall v_1, v_2 \in Val : \langle c_1, mds, mem_1[x \mapsto v_1]\rangle \;\mathcal{R}\; \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$

2. $(x \notin mds(\textit{asm-no-w}) \wedge dma(x) = low) \Rightarrow$
$\forall v \in Val : \langle c_1, mds, mem_1[x \mapsto v]\rangle \mathcal{R} \langle c_2, mds, mem_2[x \mapsto v]\rangle$ ◊

The two conditions in Definition 4.5 characterize the possible modifications of the memory by other threads that (a) respect the no-write assumptions modeled by the mode state and (b) do not store secrets in low variables.

Definition 4.6. A symmetric binary relation $\mathcal{R}$ on thread configurations with modes that have equal mode states is a *strong low bisimulation modulo modes* if it is closed under globally consistent changes, and if for all $c_1, c_2 \in Com$, $mds \in Mds$, and $mem_1, mem_2 \in Mem$ with $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ the following conditions are satisfied:

1. $mem_1 =_L^{mds} mem_2$, and
2. for all $c_1' \in Com$, $mds' \in Mds$, $mem_1' \in Mem$, and $\alpha_1 = new(thr_1, mdss_1) \in Lab_m$, if $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$ then there exist $c_2' \in Com$, $mem_2' \in Mem$, and $\alpha_2 = new(thr_2, mdss_2) \in Lab_m$ such that the following conditions are satisfied:
   a) $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
   b) $\langle c_1', mds', mem_1'\rangle \mathcal{R} \langle c_2', mds', mem_2'\rangle$,
   c) $\sharp(thr_1) = \sharp(thr_2)$, and
   d) $\langle thr_1[i], mdss_1[i], mem_1'\rangle \mathcal{R} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

The relation $\sim_{mm}$ is the union of all strong low bisimulations modulo modes. ◊

Theorem 4.1. The relation $\sim_{mm}$ is a strong low bisimulation modulo modes. ◊

Theorem 4.2. The relation $\sim_{mm}$ is a partial equivalence relation (i.e., transitive and symmetric). ◊

The proofs of the two theorems are straightforward, they go along the same lines as the proofs for the corresponding theorems for low bisimulations modulo low matching (Theorems 3.4 and 3.5 in Section 3.3.2). Detailed proofs are available in Appendix A.5.

Strong low bisimulations modulo modes characterize an indistinguishability of snapshots during two executions of a thread: If $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ then the executions of c_1 and c_2 starting in mem_1 and mem_2 are indistinguishable to an observer who sees the values of all low variables without no-read assumption. This is due to Condition (1) in Definition 4.6 that ensures L-equality of memories modulo the mode state, and to Condition (2) that ensures that execution steps result in related configurations and, hence, contain memories that are L-equal modulo the mode state. Repeating this argument inductively shows that the same still holds after any number of execution steps. In consequence, it is impossible to distinguish the executions based on the values of low variables without no-read assumption. Since strong low bisimulations modulo modes are closed under globally consistent changes, the indistinguishability still holds if other threads that respect the no-write assumptions and that do not store secrets in low variables are executed concurrently. Even more, the indistinguishability still holds if the other threads respect no-write assumptions and store secrets only in low variables that the thread does not read, because those variables do not affect the execution.

Based on strong low bisimulations modulo modes we define a partial equivalence relation on commands that we use to define the novel information flow property.

Definition 4.7. Two commands c_1 and c_2 are *SIFUM-indistinguishable for mode state* mds (denoted by $c_1 \sim_{mm}^{mds} c_2$) if and only if $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$ for all memories mem_1 and mem_2 with $mem_1 =_L mem_2$. ◊

Definition 4.8. Command c is *SIFUM-secure for mode state mds* if and only if $c \sim_{mm}^{mds} c.\Diamond$

The acronym "SIFUM" expands to "secure information flow using modes". We illustrate strong low bisimilarity modulo modes and SIFUM-security at the simplistic examples from Section 4.2 and discuss more realistic examples after introducing a flow-sensitive type system enforcing SIFUM-security in Section 4.6. For the following examples, we assume that the semantics of the example programs is such that modes do not change during program execution. (The semantics with modes is formalized in Section 4.5.1.)

Example 4.5. Reconsider the command $c = h{:=}0; l{:=}h$ from Example 4.1. We construct a strong low bisimulation modulo modes $\mathcal{R}$ such that $\langle c, mds, mem_1\rangle \mathcal{R} \langle c, mds, mem_2\rangle$ if $h \in mds(\textit{asm-no-w})$ and $mem_1 =_L mem_2$. We define $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ if and only if $h \in mds(\textit{asm-no-w})$ and one of the following conditions is satisfied:

- $c_1 = c_2 = c$ and $mem_1 =_L mem_2$,
- $c_1 = c_2 = \textit{stop}; l{:=}h$, $mem_1 =_L mem_2$, and $mem_1(h) = mem_2(h) = 0$,
- $c_1 = c_2 = l{:=}h$, $mem_1 =_L mem_2$, and $mem_1(h) = mem_2(h) = 0$, or
- $c_1 = c_2 = \textit{stop}$ and $mem_1 =_L mem_2$.

It is straightforward to show that $\mathcal{R}$ is indeed a strong low bisimulation modulo modes. Hence, $h{:=}0; l{:=}h$ is SIFUM-secure for initial mode states mds with $h \in mds(\textit{asm-no-w})$. If $h \in mds(\textit{asm-no-w})$ would not be required in the definition of $\mathcal{R}$ then $\mathcal{R}$ would not be closed under globally consistent changes, because $\langle l{:=}h, mds, mem_1\rangle \mathcal{R} \langle l{:=}h, mds, mem_2\rangle$ does not hold for arbitrary L-equal mem_1 and mem_2. In fact, if $h \notin mds(\textit{asm-no-w})$ then $h{:=}0; l{:=}h$ is only SIFUM-secure for mds if $l \in mds(\textit{asm-no-r})$. $\Diamond$

Example 4.6. Reconsider the program $l{:=}h; l{:=}0$ from Example 4.2. The command is SIFUM-secure for all initial mode states mds with $l \in mds(\textit{asm-no-r})$. This can be shown like in Example 4.5 by constructing a suitable strong low bisimulation modulo modes. In difference to Example 4.5, a suitable relation relates configurations where the memories are not necessarily equal for l. $\Diamond$

The above examples illustrate that the two examples from Section 4.2 that are classified as secure by flow-sensitive analyses for sequential programs are also classified as SIFUM-secure if one makes suitable assumptions about other threads. Intuitively, this expresses that the execution of the threads in the examples does not lead to information leaks if the threads are executed in parallel with SIFUM-secure threads that respect the assumptions.

In fact, SIFUM-security adequately captures information flow security given that assumptions and guarantees capture correctly how threads and the scheduler access the shared memory. We provide formal justification in Section 4.4.4.

We use SIFUM-security as a reference point for flow-sensitive information flow analyses exploiting assumptions and guarantees. SIFUM-security supports flow-sensitive reasoning in two directions: Firstly, it permits that low variables store secrets as long as a no-read assumption is made, and, in consequence, flow-sensitive reasoning like in Example 4.1 is sound in such situations. Moreover, if a no-write assumption is made for a variable one can be sure that the security level of the value of that variable does not change, and, hence, flow-sensitive reasoning like in Example 4.2 is possible. Beyond this, we establish a compositionality result for SIFUM-security in the following sections, which permits modular information flow analyses.

4.4.3 Sound Modes

A thread pool that contains only SIFUM-secure commands is $\mathcal{S}$-noninterferent for any scheduler if assumptions and guarantees are used in a sound way. This means that
(a) whenever an assumption is made for a thread then this assumption is matched by the corresponding guarantee in all other threads,
(b) the guarantees made for a thread are valid for that thread,
(c) no no-read assumptions are made after a thread terminates, and
(d) the observation function *obs* is confined to a subset of low variables for which no no-read assumptions are made for any thread.

Items (a) and (b) ensure that the assumptions for one thread are valid with respect to memory accesses by other threads. Besides such accesses, a variable might also be read if it is output upon program termination or if it is used to interface the program with the scheduler. Items (c) and (d) ensure that no-read assumptions are also valid with respect to such variable accesses. We make this notion of soundness of assumptions and guarantees precise in Definition 4.17, based on a formalization of Item (a) in Definition 4.10, of Item (b) in Definition 4.14, of Item (c) in Definition 4.15, and of Item (d) in Definition 4.16.

Definition 4.9. Let *scm* be a system configuration with modes. We inductively define the set $reach_{\mathcal{S}}(scm)$ of *system configurations with modes reachable from scm under scheduler* $\mathcal{S}$ as follows:
- $scm \in reach_{\mathcal{S}}(scm)$, and
- if $scm' \in reach_{\mathcal{S}}(scm)$ and $scm' \overset{k,p}{\Longrightarrow}_{\mathcal{S}} scm''$ for some k and p then $scm'' \in reach_{\mathcal{S}}(scm)$.

The *set of lists of mode states reachable from scm under* $\mathcal{S}$ is defined as

$$\{mdss \in Mds^* \mid \exists thr, mem, sst : \langle thr, mdss, mem, sst\rangle \in reach_{\mathcal{S}}(scm)\}. \qquad \Diamond$$

Definition 4.10. We say that a list of mode states *mdss has compatible modes* if and only if for all $i \in \{1, \ldots, \sharp(mdss)\}$ and all $x \in Var$ the following conditions are satisfied:
- $x \in mdss[i](asm\text{-}no\text{-}r) \Rightarrow \forall j \neq i : x \in mds[j](guar\text{-}no\text{-}r)$, and
- $x \in mdss[i](asm\text{-}no\text{-}w) \Rightarrow \forall j \neq i : x \in mds[j](guar\text{-}no\text{-}w)$.

A set of lists of mode states has compatible modes if each of its elements has compatible modes. The *system configuration with modes scm has compatible modes under scheduler* $\mathcal{S}$ if the set of lists of mode states reachable from *scm* under $\mathcal{S}$ has compatible modes.$\Diamond$

Definition 4.11. Let $c \in Com$ and $x \in Var$. We say that *c does not read x* if and only if for all $c' \in Com$, $mem, mem' \in Mem$, and $\alpha \in Lab$ with $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$ one of the following conditions is satisfied:
- $\forall v \in Val : \langle c, mem[x \mapsto v]\rangle \xrightarrow{\alpha} \langle c', mem'[x \mapsto v]\rangle$, or
- $\forall v \in Val : \langle c, mem[x \mapsto v]\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$. $\Diamond$

Intuitively, the first condition captures the case that the value of x does not change in the execution step, and the second condition captures the case that the value of x changes.

Definition 4.12. Let $c \in Com$ and $x \in Var$. We say that *c does not modify x* if $mem(x) = mem'(x)$ for all $c' \in Com$, $mem, mem' \in Mem$, and $\alpha \in Lab$ with $\langle c, mem\rangle \xrightarrow{\alpha} \langle c', mem'\rangle$.$\Diamond$

To define what it means that a thread respects its guarantees we refine the approximation of the set of thread configurations that are locally reachable from a configuration (see Section 3.3.2). To this end, we adapt local reachability to our setting with modes, exploiting the no-write assumptions to refine local reachability.

Definition 4.13. Let tcm be a thread configuration with modes. The set $\mathit{loc\text{-}reach}(tcm) \subseteq \mathit{Com} \times \mathit{Mds} \times \mathit{Mem}$ of *thread configurations with modes that are locally reachable from* tcm is inductively defined as follows:

- $tcm \in \mathit{loc\text{-}reach}(tcm)$,
- $\forall c', mds', mem', mem'' : [\langle c', mds', mem'\rangle \in \mathit{loc\text{-}reach}(tcm) \wedge mem' =_{mds'(\mathit{asm\text{-}no\text{-}w})} mem''] \Rightarrow \langle c', mds', mem''\rangle \in \mathit{loc\text{-}reach}(tcm)$, and
- $\forall tcm', c'', mds'', mem'', thr, mdss :$
  $[tcm' \in \mathit{loc\text{-}reach}(tcm) \wedge tcm' \xrightarrow{new(thr,mdss)} \langle c'', mds'', mem''\rangle] \Rightarrow$
  $\big[\langle c'', mds'', mem''\rangle \in \mathit{loc\text{-}reach}(tcm) \wedge$
  $\forall i \in \{1, \ldots, \sharp(thr)\} : \langle thr[i], mdss[i], mem''\rangle \in \mathit{loc\text{-}reach}(tcm)\big]$. ◊

Definition 4.14. A thread configuration with modes tcm *respects guarantees* if for all $\langle c', mds', mem'\rangle \in \mathit{loc\text{-}reach}(tcm)$ and all $x \in \mathit{Var}$ the following conditions are satisfied:

- $x \in mds'(\mathit{guar\text{-}no\text{-}r}) \Rightarrow c'$ does not read x, and
- $x \in mds'(\mathit{guar\text{-}no\text{-}w}) \Rightarrow c'$ does not modify x. ◊

Definition 4.15. A thread configuration with modes tcm *has no no-read assumptions at thread termination* if $mds'(\mathit{asm\text{-}no\text{-}r}) = \{\}$ for all $\langle \mathit{stop}, mds', mem'\rangle \in \mathit{loc\text{-}reach}(tcm)$.◊

Definition 4.16. Let $X \subseteq \mathit{Var}$ be the largest set of variables such that obs is confined to X. A thread configuration with modes tcm *has modes compatible with* obs if $mds'(\mathit{asm\text{-}no\text{-}r}) \cap X = \{\}$ for all $\langle c', mds', mem'\rangle \in \mathit{loc\text{-}reach}(tcm)$. ◊

Definition 4.17. Let thr be a thread pool, $mdss$ be a list of mode states, and $\mathcal{S} = (sSt, sst_0, \rightarrow)$ be a scheduler. The pair $(thr, mdss)$ has *sound modes for* $\mathcal{S}$ if the configuration $\langle thr, mdss, mem, sst_0\rangle$ has compatible modes under $\mathcal{S}$ and, for all $i \in \{1, \ldots, \sharp(thr)\}$, the configuration $\langle thr[i], mdss[i], mem\rangle$ respects guarantees, has no no-read assumptions at thread termination, and has modes compatible with obs. ◊

The definition of sound modes expresses that all assumptions and guarantees are actually valid for a multi-threaded program. This definition captures the necessary requirements on assumptions and guarantees for establishing compositionality and scheduler independence of SIFUM-security in the following section.

4.4.4 Compositionality and Scheduler-independence Result

For any scheduler $\mathcal{S}$, the parallel composition of SIFUM-secure threads is $\mathcal{S}$-noninterferent if the thread pool has sound modes for $\mathcal{S}$ and the observation function is confined to L.

Theorem 4.3. Let thr be a thread pool, $mdss$ a list of mode states, and $\mathcal{S}$ a scheduler such that the following conditions are satisfied:

- $thr[i]$ is SIFUM-secure for $mdss[i]$ for all $i \in \{1, \ldots, \sharp(thr)\}$, and
- the pair $(thr, mdss)$ has sound modes for $\mathcal{S}$.

Assume furthermore that *obs* is confined to L. Then *thr* is $\mathcal{S}$-noninterferent. $\Diamond$

Theorem 4.3 states both compositionality and scheduler-independence. The theorem shows, firstly, that SIFUM-security adequately captures information flow security in the sense of the noninterference-like property $\mathcal{S}$-noninterference. Secondly, the theorem shows that this is still the case when composing multiple SIFUM-secure threads in parallel, as long as the assumptions and guarantees made for the threads satisfy the sanity requirements formalized in the previous section.

In the remainder of this section, we discuss the proof of Theorem 4.3. The basic idea is to firstly show that the probability that an execution of the thread pool *thr* under $\mathcal{S}$ terminates with given public outputs after any fixed number k of execution steps does not depend on the secret inputs if modes are sound for $\mathcal{S}$. From this intermediate result it follows immediately that the probability to terminate with given public outputs after any number of execution steps does not depend on the secret inputs (by summation over the probability for each possible number of execution steps k).

In the proof, we exploit that execution steps of a thread configuration can be matched in thread configurations that are strong low bisimilar modulo modes. However, it is not straightforward to lift this property from single threads to thread pools. The reason is that memory modifications of one thread may "destroy" strong low bisimilarity modulo modes for another thread, or, formally, $\langle thr_1[i], mdss[i], mem_1\rangle \sim_{mm} \langle thr_2[i], mdss[i], mem_2\rangle$ does not imply $\langle thr_1[i], mdss[i], mem_1'\rangle \sim_{mm} \langle thr_2[i], mdss[i], mem_2'\rangle$, where mem_1' and mem_2' result from execution steps of another pair of threads $thr_1[j]$ and $thr_2[j]$. The problem is that the closure of $\sim_{mm}$ under globally consistent changes only ensures that this implication is valid if the values of low variables are modified equally by $thr_1[j]$ and $thr_2[j]$, which need not be the case if $mdss[j]$ contains a no-read assumption for a low variable. However, intuitively, the possible execution steps of $thr_1[i]$ and $thr_2[i]$ do not depend on those variables because under the assumption of sound modes $thr_1[i]$ and $thr_2[i]$ do not read them. In particular, there exist memories mem_1'' and mem_2'' such that $\langle thr_1[i], mdss[i], mem_1''\rangle \sim_{mm} \langle thr_2[i], mdss[i], mem_2''\rangle$ and such that mem_1' and mem_1'' as well as mem_2' and mem_2'' differ only on variables that $thr_1[i]$ and $thr_2[i]$ do not read. The following definition formally captures the properties of these memories.

Definition 4.18. Let $tcm_1 = \langle thr_1, mdss, mem_1\rangle$ and $tcm_2 = \langle thr_2, mdss, mem_2\rangle$ be thread configurations with modes with the same number of threads, and let $n = \sharp(thr_1) = \sharp(thr_2)$. We say that lists of memories $mems_1$ and $mems_2$ with $\sharp(mems_1) = \sharp(mems_2) = n$ *make tcm_1 and tcm_2 compatible with strong low bisimulation modulo modes* if and only if the following conditions are satisfied (where $X_i = \{x \in Var \mid mems_1[i](x) \neq mem_1(x) \vee mems_2[i](x) \neq mem_2(x)\}$ for each $i \in \{1, \ldots, n\}$):

1. $\forall i \in \{1, \ldots, n\} : \forall f \in Var \rightharpoonup Val : dom(f) = X_i \Rightarrow \langle thr_1[i], mdss[i], mems_1[i][x \mapsto f(x) \mid x \in X_i]\rangle \sim_{mm} \langle thr_2[i], mdss[i], mems_2[i][x \mapsto f(x) \mid x \in X_i]\rangle$,
2. $\forall i \in \{1, \ldots, n\} : \forall x \in Var : [mem_1(x) = mem_2(x) \vee x \in H] \Rightarrow x \notin X_i$, and
3. $(n = 0 \wedge mem_1 =_L mem_2) \vee (\forall x \in Var : \exists i \in \{1, \ldots, n\} : x \notin X_i)$. $\Diamond$

Intuitively, the set X_i in Definition 4.18 contains those variables for which the memories $mems_1[i]$ or $mems_2[i]$ differ from the actual memories mem_1 and mem_2, respectively. Item 1 ensures that the thread configurations for the threads at position i are strong low bisimilar modulo modes for memories mem_1 and mem_2 after redefining the values of variables in X_i. Item 2 ensures that the memories only differ from the actual memories for low variables for which the actual memories differ (i.e., for which some thread makes

a no-read assumption). Finally, Item 3 ensures that for each variable there is at least one thread index i for which the memories $mems_1[i]$ or $mems_2[i]$ are equal to the actual memories. Intuitively, this is the index of the thread that last modified that variable.

The existence of memories that make two multi-threaded configurations compatible with strong low bisimulation modulo modes ensures, firstly, that for all i the threads $thr_1[i]$ and $thr_2[i]$ do not read variables in the set X_i from Definition 4.18, and, secondly, that for all i an execution step of $thr_1[i]$ in mem_1 can be matched by an execution step of $thr_2[i]$ in mem_2 such that there are lists of memories that make the resulting configurations compatible with strong low bisimulations modulo modes. These properties are established by the following lemmas.

Lemma 4.1. Let $\langle thr_1, mdss, mem_1, sst\rangle$ and $\langle thr_2, mdss, mem_2, sst\rangle$ be configurations such that $(thr_1, mdss)$ and $(thr_2, mdss)$ have sound modes for $\mathcal{S}$, and let $mems_1$ and $mems_2$ be lists of memories that make $\langle thr_1, mdss, mem_1\rangle$ and $\langle thr_2, mdss, mem_2\rangle$ compatible with strong low bisimulation modulo modes.

Then for all $i \in \{1, \ldots, \sharp(thr_1)\}$ the commands $thr_1[i]$ and $thr_2[i]$ do not read any variable in $X_i = \{x \in Var \mid mems_1[i](x) \neq mem_1(x) \vee mems_2[i](x) \neq mem_2(x)\}$. ◊

Proof Sketch. (A detailed proof is available in Appendix A.6.)

If $x \in X_i$ then there exists $j \neq i$ with $x \notin X_j$ by Items 2 and 3 in Definition 4.18. It follows with Item 1 in Definition 4.18 for index j that $x \in mdss[j](\textit{asm-no-r})$. Hence, $x \in mdss[i](\textit{guar-no-r})$ due to sound modes, and, hence, also due to sound modes $thr_1[i]$ and $thr_2[i]$ do not read x. □

Lemma 4.2. Let $\langle thr_1, mdss, mem_1, sst\rangle$ and $\langle thr_2, mdss, mem_2, sst\rangle$ be configurations such that $(thr_1, mdss)$ and $(thr_2, mdss)$ have sound modes for $\mathcal{S}$, and assume that *obs* is confined to L. Assume that $\sharp(thr_1) = \sharp(thr_2)$, that $mems_1$ and $mems_2$ make $\langle thr_1, mdss, mem_1\rangle$ and $\langle thr_2, mdss, mem_2\rangle$ compatible with strong low bisimulation modulo modes, and that $\langle thr_1, mdss, mem_1, sst\rangle \overset{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr_1', mdss', mem_1', sst'\rangle$.

Then there exist thr_2', mem_2', $mems_1'$, and $mems_2'$ such that $\sharp(thr_1') = \sharp(thr_2')$, such that $mems_1'$ and $mems_2'$ make $\langle thr_1', mdss', mem_1'\rangle$ and $\langle thr_2', mdss', mem_2'\rangle$ compatible with strong low bisimulation modulo modes, and such that $\langle thr_2, mdss, mem_2, sst\rangle \overset{k,p}{\Longrightarrow}_{\mathcal{S}}$ $\langle thr_2', mdss', mem_2', sst'\rangle$. ◊

Proof Sketch. (A detailed proof is available in Appendix A.6.)

By Lemma 4.1, mem_1 and $mems_1[k]$ differ only in variables that $thr_1[k]$ does not read. Hence, the execution step of $thr_1[k]$ in mem_1 implies that the same execution step is also possible in $(mems_1[k])[x \mapsto f(x) \mid x \in X_k]$ for an arbitrary partial function f with domain $X_k = \{x \in Var \mid mems_1[k](x) \neq mem_1(x) \vee mems_2[k](x) \neq mem_2(x)\}$. Due to Item 3 in Definition 4.18 and the definition of strong low bisimulations modulo modes this execution step can be matched by an execution step of $thr_2[k]$ in $(mems_2[k])[x \mapsto f(x) \mid x \in X_k]$. Again by Lemma 4.1 $(mems_2[k])[x \mapsto f(x) \mid x \in X_k]$ and mem_2 differ only in variables that $thr_2[k]$ does not read, from which we obtain the execution step of $thr_2[k]$ in mem_2 that we need to exhibit.

The lists of memories $mems_1'$ and $mems_2'$ can be constructed from $mems_1$ and $mems_2$ based on the modifications of the memory by $thr_1[k]$ and $thr_2[k]$. □

Theorem 4.4. Let $\langle thr_1, mdss, mem_1, sst\rangle$ and $\langle thr_2, mdss, mem_2, sst\rangle$ be configurations such that $(thr_1, mdss)$ and $(thr_2, mdss)$ have sound modes for $\mathcal{S}$, and assume that *obs* is confined to L. Assume furthermore that $\sharp(thr_1) = \sharp(thr_2)$ and that lists of memories

$mems_1$ and $mems_2$ make $\langle thr_1, mdss, mem_1\rangle$ and $\langle thr_2, mdss, mem_2\rangle$ compatible with strong low bisimulation modulo modes.

Then the following equation is satisfied for all $k \in \mathbb{N}$, all $mem \in Mem$, $sc_1 = \langle thr_1, mem_1, sst\rangle$, and $sc_2 = \langle thr_2, mem_2, sst\rangle$:

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\{tr \in \mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'} \mid \sharp(tr) = k\}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\{tr \in \mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'} \mid \sharp(tr) = k\})$$

$\Diamond$

Proof Sketch. The proof is by induction on k (for a detailed proof see Appendix A.6).

In the base case, $k = 1$ holds. For arbitrary sc the set $\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'}$ contains at most one trace of length 1. The set contains a trace of length 1 if and only if $getMem(sc) = mem'$ and sc is final, i.e., $\sharp(getThr(sc)) = 0$. In this case, the probability of the single trace is 1. Hence, the equality that we have to prove is satisfied if $\sharp(thr_1) > 0$, because then sc_1 and sc_2 are both not final. If $\sharp(thr_1) = 0$ the equality is satisfied because then $mem_1 =_L mem_2$ by assumption.

In the step case, it follows from Lemma 4.2 that a step of $\langle thr_1, mdss, mem_1, sst\rangle$ can be matched by a step of $\langle thr_2, mdss, mem_2, sst\rangle$ with the same probability, resulting in two system configurations with modes for which the assumptions of the theorem are again satisfied. Hence, the induction hypothesis for $k - 1$ can be applied, from which the statement for k follows immediately. □

Finally, we can prove Theorem 4.3 by exploiting Theorem 4.4.

Proof of Theorem 4.3. Let $mem_1 =_{Var \setminus HI} mem_2$. Then $mem_1 =_L mem_2$ because $L \subseteq Var \setminus HI$. Hence, if $\sharp(thr) = 0$ the theorem follows trivially.

If $\sharp(thr) > 0$ define $mems_1$ and $mems_2$ by $mems_1[i] = mem_1$ and $mems_2[i] = mem_2$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$. Then the assumptions of Theorem 4.4 are satisfied for $mems_1$ and $mems_2$, and, hence, the following holds for all $k \in \mathbb{N}$, where $sc_1 = \langle thr, mem_1, sst_0\rangle$ and $sc_2 = \langle thr, mem_2, sst_0\rangle$:

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\{tr \in \mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'} \mid \sharp(tr) = k\}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\{tr \in \mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'} \mid \sharp(tr) = k\})$$

Thus, thr is $\mathcal{S}$-noninterferent, i.e., the following equation is satisfied for all $mem \in Mem$:

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'}).$$

□

4.5 A Type-based Flow-sensitive Security Analysis

In this section, we present a flow-sensitive security type system for multi-threaded programs and establish the soundness of the type system with respect to SIFUM-security. Moreover, at the end of the section we illustrate with an extension of the type system how SIFUM-security can be exploited beyond flow-sensitive security types.

The type system is based on assumptions and guarantees that are specified by annotations in the program. To enable such specifications, we extend the programming language from Section 2.3 with annotations.

$$\frac{\langle c, mem\rangle \xrightarrow{new(thr)}\triangleright \langle c', mem'\rangle \quad \sharp(mdss) = \sharp(thr) \quad \forall i \in \{1,\ldots,\sharp(thr)\} : mdss[i] = \mathit{spawn\text{-}mds}(mds)}{\langle \mathcal{E}[c], mds, mem\rangle \xrightarrow{new(thr,mdss)}\triangleright \langle \mathcal{E}[c'], mds, mem'\rangle}$$

$$\frac{\langle c, \mathit{mds\text{-}update}(mds, ann), mem\rangle \xrightarrow{\alpha}\triangleright \langle c', mds', mem'\rangle}{\langle \mathcal{E}[/\!\!/ ann /\!\!/\ c], mds, mem\rangle \xrightarrow{\alpha}\triangleright \langle \mathcal{E}[c'], mds', mem'\rangle}$$

Figure 4.2: Semantics for commands with annotations

4.5.1 Annotations for Specifying Assumptions and Guarantees

We extend the syntax of the programming language from Section 2.3 with annotations that specify for which parts of a thread one makes assumptions about memory accesses by other threads, and for which parts one states guarantees for a thread. Annotations are enclosed in comments ($/\!/ \ldots /\!/$), indicating that the annotations do not influence the runtime behavior of programs. The extended syntax is defined by the following grammar:

$$\begin{aligned} ann ::= \;& \mathsf{acq}(m, x) \mid \mathsf{rel}(m, x) \\ c ::= \;& \mathsf{skip} \mid x{:=}e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \\ & \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od} \mid \mathsf{spawn}(c, \ldots, c) \mid \\ & /\!/ ann /\!/\ c, \end{aligned}$$

where $m \in \mathit{Mod}$, $x \in \mathit{Var}$, and $e \in \mathit{Exp}$. The annotation $\mathsf{acq}(m, x)$ indicates that mode m is acquired for variable x, and the annotation $\mathsf{rel}(m, x)$ indicates that mode m is released for variable x. Each command may be annotated with multiple annotations, e.g., two annotations may be used for acquiring a mode for two different variables.

For the semantics of the extended programming language, we instantiate the set Com of commands with the set of all terms that are defined by the above grammar for c to which we add the command stop. We formalize the semantics by a calculus for the judgments of the form $\langle c, mds, mem\rangle \xrightarrow{\alpha}\triangleright \langle c', mds', mem'\rangle$ such that Requirement 4.1 from Section 4.4.1 is satisfied, i.e., such that modes do not influence the runtime behavior of programs. The derivation rules are defined in Figure 4.2. They are based on the calculus for the judgments $\langle c, mem\rangle \xrightarrow{\alpha}\triangleright \langle c', mem'\rangle$ that formalizes the semantics of programs without annotations.

The annotations directly preceding an instruction are evaluated together with the first execution step of that instruction. This ensures that the evaluation of annotations does not introduce additional execution steps. Moreover, the evaluation of annotations affects the mode state as specified by the function $\mathit{mds\text{-}update}$.

Definition 4.19. Let mds be a mode state and ann an annotation. We define the mode state $\mathit{mds\text{-}update}(mds, ann)$ by

$$\mathit{mds\text{-}update}(mds, ann) = \begin{cases} mds[m \mapsto mds(m) \cup \{x\}] & \text{if } ann = \mathsf{acq}(m, x), \\ mds[m \mapsto mds(m) \setminus \{x\}] & \text{if } ann = \mathsf{rel}(m, x). \end{cases} \qquad \lozenge$$

When new threads are spawned, the initial mode states of the spawned threads are determined based on the mode state of the spawning thread.

Definition 4.20. Let mds be a mode state. We define the mode state $\mathit{spawn\text{-}mds}(mds)$ by
- $\mathit{spawn\text{-}mds}(mds)(m) = \{\}$ if $m \in \{\mathit{asm\text{-}no\text{-}r}, \mathit{asm\text{-}no\text{-}w}\}$,
- $\mathit{spawn\text{-}mds}(mds)(\mathit{guar\text{-}no\text{-}r}) = mds(\mathit{asm\text{-}no\text{-}r}) \cup mds(\mathit{guar\text{-}no\text{-}r})$, and
- $\mathit{spawn\text{-}mds}(mds)(\mathit{guar\text{-}no\text{-}w}) = mds(\mathit{asm\text{-}no\text{-}w}) \cup mds(\mathit{guar\text{-}no\text{-}w})$. ◊

Definition 4.20 formalizes that no assumptions are made initially for a spawned thread. Moreover, all guarantees made for the spawning thread are also made for the spawned thread. Finally, if one makes assumptions for the spawning thread, the corresponding guarantees are stated for the spawned thread. This ensures that the modes of the spawned thread are compatible with the modes of the spawning thread. Any assumptions for the spawned thread can be specified with annotations in the code of the spawned thread.

Remark 4.1. Definition 4.20 formalizes a simple way for specifying the initial modes for spawned threads. In some scenarios, more complex schemes might be useful, like, for instance, handing over an assumption from the spawning thread to a spawned thread. To handle such situations one could, for instance, extend the syntax of the programming language with special annotations for spawn-commands. Another possibility is to interpret release-annotations of instructions that spawn exactly one thread as "transfer a mode from the spawning thread to the spawned thread". ◊

The evaluation of annotations only affects mode states, and not the control state of the thread or the shared memory. I.e., Requirement 4.1 from Section 4.4.1 is satisfied.

Theorem 4.5. Requirement 4.1 is satisfied for the calculus from Figure 4.2, i.e., the following conditions are satisfied:
- If $\langle c, mds, mem\rangle \xrightarrow{new(thr,mdss)} \langle c', mds', mem'\rangle$ then $\langle c, mem\rangle \xrightarrow{new(thr)} \langle c', mem'\rangle$.
- If $\langle c, mem\rangle \xrightarrow{new(thr)} \langle c', mem'\rangle$ then for each mode state mds there exist a mode state mds' and a list of mode states $mdss'$ such that $\langle c, mds, mem\rangle \xrightarrow{new(thr,mdss)} \langle c', mds', mem'\rangle$. ◊

Proof. The theorem follows directly from the derivation rules for judgments of the form $\langle c, mds_1, mem\rangle \xrightarrow{new(thr,mdss)} \langle c', mds_1', mem'\rangle$ which are based on the derivation rules for judgments of the form $\langle c, mem\rangle \xrightarrow{new(thr)} \langle c', mem'\rangle$. □

4.5.2 The Security Type System

For defining the type system we assume that the domain assignment is given by dma. The type system contains different typing judgments for expressions, commands, and thread pools. Typing judgments for expressions have the form $\Gamma \vdash e : d$ where $\Gamma : \mathit{Var} \to \{low, high\}$. We call the functions in the set $\mathit{Var} \to \{low, high\}$ *environments*. The interpretation of the judgment is that d is an upper bound on the security level of the value of e, given that $\Gamma(x)$ is an upper bound on the security level of the value of x for all $x \in \mathit{Var}$. The derivation rule for typing judgments for expressions is displayed at the top of Figure 4.3.

Typing judgments for commands are of the form

$$\vdash \Lambda\ \{c\}\ \Lambda',$$

where c is a command and $\Lambda, \Lambda' : \mathit{Var} \rightharpoonup \{low, high\}$. We call the partial functions in the set $\mathit{Var} \rightharpoonup \{low, high\}$ *partial environments*. A partial environment Λ captures

upper bounds on the security level of the values of variables in $dom(\Lambda)$. For each variable $x \notin dom(\Lambda)$ we assume that the upper bound is $dma(x)$. The upper bounds for all variables are then given by the following extension of the partial environment Λ:

Definition 4.21. Let Λ be a partial environment. Then the *total environment for* Λ is the environment $\Lambda\langle\cdot\rangle : Var \to \{low, high\}$ that is defined by

$$\Lambda\langle x\rangle = \begin{cases} \Lambda(x) & \text{if } x \in dom(\Lambda), \\ dma(x) & \text{otherwise.} \end{cases}$$

We use the total environment to describe the interpretation of the judgment $\vdash \Lambda\ \{c\}\ \Lambda'$: If $dom(\Lambda)$ is a lower bound on the set of low variables with no-read and high variables with no-write assumption before c is executed then $dom(\Lambda')$ is such a lower bound after the execution of c. Moreover, if $\Lambda\langle x\rangle$ is an upper bound on the security level of the value of x for all variables before the execution of c then $\Lambda'\langle x\rangle$ is such an upper bound after the execution of c. Moreover, this still holds if other SIFUM-secure threads that respect the assumptions made for c modify the memory during the execution of c.

The typing judgments permit to exploit the order of program statements in the typing rules because they encode variations of upper bounds on security levels that are caused by the execution of a command. The reason for the requirement that the upper bound is given by $dma(x)$ if x is not in the domain of the partial environment is twofold: If x is a low variable, then SIFUM-security requires that x stores secrets only if a no-read assumption is made for x, and, hence, if no such assumption is made then domain *low* must be an upper bound for x. Moreover, if x is a high variable without no-write assumption then other threads might write secrets into x, and, hence, the upper bound must be *high*.

The derivation rules for judgments of the form $\vdash \Lambda\ \{c\}\ \Lambda'$ are displayed in Figure 4.3.

When typing an annotated command, rule [ANNO] ensures that the domain of the partial environment is adjusted based on the annotation, using the function *adjust-pe* (for "adjust partial environment").

Definition 4.22. Let Λ be a partial environment and *ann* an annotation. Then $\Lambda' = \mathit{adjust\text{-}pe}(\Lambda, \mathit{ann})$ is the partial environment defined by

$$dom(\Lambda') = \begin{cases} dom(\Lambda) \cup \{x\} & \mathit{ann} = \mathsf{acq}(\mathit{asm\text{-}no\text{-}r}, x) \wedge dma(x) = low \\ dom(\Lambda) \cup \{x\} & \mathit{ann} = \mathsf{acq}(\mathit{asm\text{-}no\text{-}w}, x) \wedge dma(x) = high \\ dom(\Lambda) \setminus \{x\} & \mathit{ann} = \mathsf{rel}(\mathit{asm\text{-}no\text{-}r}, x) \wedge dma(x) = low \\ dom(\Lambda) \setminus \{x\} & \mathit{ann} = \mathsf{rel}(\mathit{asm\text{-}no\text{-}w}, x) \wedge dma(x) = high \\ dom(\Lambda) & \text{otherwise} \end{cases}$$

and $\Lambda'(x) = \Lambda\langle x\rangle$ for all $x \in dom(\Lambda')$. ◊

The second premise in rule [ANNO] ensures that the upper bound on the security level for a variable, $\Lambda\langle x\rangle$, does not decrease when adjusting the partial environment. Decreasing $\Lambda\langle x\rangle$ is only possible by decreasing $\Lambda(x)$ in rule [ASS_2]. If the security level of a variable has been reset to its original security level $dma(x)$, i.e., if it is safe to stop making the no-read assumption for x, then rule [ANNO] can remove x from $dom(\Lambda)$.

The typing rule [ASS_1] is applicable for assignments to variables that are not in the domain of Λ, i.e., for variables for which the upper bound on the security level is fixed to $dma(x)$, while rule [ASS_2] is applicable for assignments to variables in the domain of Λ, i.e., for variables for which the upper bound on the security level is not fixed. If

$$[\mathsf{EXP}]\ \frac{}{\Gamma \vdash e : \bigsqcup_{x \in vars(e)} \Gamma(x)} \qquad [\mathsf{STOP}]\ \frac{}{\vdash \Lambda\ \{stop\}\ \Lambda} \qquad [\mathsf{SKIP}]\ \frac{}{\vdash \Lambda\ \{\mathsf{skip}\}\ \Lambda}$$

$$[\mathsf{ASS}_1]\ \frac{x \notin dom(\Lambda) \quad \Lambda\langle\cdot\rangle \vdash e : d \quad d \sqsubseteq dma(x)}{\vdash \Lambda\ \{x{:=}e\}\ \Lambda} \qquad [\mathsf{ASS}_2]\ \frac{x \in dom(\Lambda) \quad \Lambda\langle\cdot\rangle \vdash e : d}{\vdash \Lambda\ \{x{:=}e\}\ \Lambda[x \mapsto d]}$$

$$[\mathsf{IF}]\ \frac{\begin{array}{c}\vdash \Lambda\ \{c_1\}\ \Lambda' \qquad \vdash \Lambda\ \{c_2\}\ \Lambda' \\ \Lambda\langle\cdot\rangle \vdash e : high \Rightarrow \big[(\forall mds : mds \text{ consistent with } \Lambda \Rightarrow c_1 \sim_{mm}^{mds} c_2) \\ \wedge\ (\forall x \in dom(\Lambda') : \Lambda'(x) = high)\big]\end{array}}{\vdash \Lambda\ \{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\ \Lambda'}$$

$$[\mathsf{WHILE}]\ \frac{\Lambda\langle\cdot\rangle \vdash e : low \quad \vdash \Lambda\ \{c\}\ \Lambda}{\vdash \Lambda\ \{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\}\ \Lambda} \qquad [\mathsf{SEQ}]\ \frac{\vdash \Lambda\ \{c_1\}\ \Lambda' \quad \vdash \Lambda'\ \{c_2\}\ \Lambda''}{\vdash \Lambda\ \{c_1; c_2\}\ \Lambda''}$$

$$[\mathsf{SPAWN}]\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, \sharp(thr)\} : \vdash \Lambda_0\ \{thr[i]\}\ \Lambda_i \\ \forall x \in Var : \Lambda\langle x\rangle \sqsubseteq dma(x)\end{array}}{\vdash \Lambda\ \{\mathsf{spawn}(thr)\}\ \Lambda}$$

$$[\mathsf{ANNO}]\ \frac{\Lambda' = adjust\text{-}pe(\Lambda, ann) \quad \forall x \in Var : \Lambda\langle x\rangle \sqsubseteq \Lambda'\langle x\rangle \quad \vdash \Lambda'\ \{c\}\ \Lambda''}{\vdash \Lambda\ \{/\!/ann/\!/\ c\}\ \Lambda''}$$

$$[\mathsf{SUB}]\ \frac{\vdash \Lambda_1\ \{c\}\ \Lambda_1' \quad \Lambda_2 \sqsubseteq \Lambda_1 \quad \Lambda_1' \sqsubseteq \Lambda_2'}{\vdash \Lambda_2\ \{c\}\ \Lambda_2'}$$

Figure 4.3: Flow-sensitive security type system

$x \in dom(\Lambda)$ then an assignment $x{:=}e$ is typable with rule [ASS_2], and the new upper bound on the security level of x is stored in $\Lambda(x)$. If, however, $x \notin dom(\Lambda)$ then the assignment $x{:=}e$ is not typable if $dma(x) = low$ and the upper bound on the security level of e is *high*. This prevents that public variables without no-read assumption are assigned secret values.

The rule for conditionals, [IF], distinguishes between control conditions that might depend on secret information and control conditions that only depend on public information. Besides typability of the branches no additional requirement is imposed if the control condition only depends on public information. If the control condition might depend on secrets, the third premise of rule [IF] requires that the two branches are strong low bisimilar modulo modes. This prevents indirect information leaks. The condition that *mds* is consistent with Λ expresses that *mds* contains no-read and no-write assumptions only for low and high variables in $dom(\Lambda)$, respectively.

Definition 4.23. *The mode state mds is consistent with the partial environment* Λ if and only if for all $x \in dom(\Lambda)$ one of the following conditions is satisfied:

- $dma(x) = low$ and $x \in mds(asm\text{-}no\text{-}r)$, or
- $dma(x) = high$ and $x \in mds(asm\text{-}no\text{-}w)$. ◊

Remark 4.2. To approximate the third premise of rule [IF] syntactically, ideas proposed by Mantel and Sands [MS04] for a similar approximation can be applied. The basic idea

is to check that both branches take the same number of execution steps, that assignments to low variables without no-read assumption occur simultaneously in both branches, and that these assignments equally modify the variable. ◊

Rule [WHILE] requires that the loop guard depends only on public information. This ensures that the number of execution steps of the loop does not depend on secret information, which is essential for the SIFUM-security of the loop.

Remark 4.3. The restriction that loop guards do not depend on secrets is not needed in the type system for FSI-security (see Chapter 3). We show in chapter 5 how to integrate FSI-security with SIFUM-security as well as how to integrate the respective type systems. In the integration, this restriction on loop guards is relaxed. Another approach to relaxing the restriction on loop guards is proposed by Volpano and Smith [VS98] and further explored by Russo and Sabelfeld [RS06b], where one adds a (non-standard) *protect*-statement to the programming language which ensures that no rescheduling occurs during the execution of a protected loop. It is safe to type loops with secret guards that are protected in this fashion as long as no public variables are written in the loop body.◊

Rule [SPAWN] permits to type commands that spawn new threads. The rule requires that all spawned threads are also typable, where the domain of the partial environment before the execution of a spawned thread is empty (the partial environment Λ_0 is defined as the unique partial environment with $dom(\Lambda_0) = \{\}$).

For subtyping (rule [SUB]) we define an order on partial environments.

Definition 4.24. We define the relation $\sqsubseteq$ on partial environments by $\Lambda \sqsubseteq \Lambda'$ if and only if $dom(\Lambda) \supseteq dom(\Lambda')$ and $\Lambda\langle x\rangle \sqsubseteq \Lambda'\langle x\rangle$ for all $x \in Var$. ◊

The order captures that the domain of partial environments is a lower bound on the variables with suitable assumptions and that the total environment for Λ, the environment $\Lambda\langle\cdot\rangle$, captures upper bounds on the security levels of the variables' values.

We establish a subject reduction result for type derivations for commands as well as a soundness result for type derivations for commands with respect to strong low bisimulations modulo modes.

Theorem 4.6. Let $c \in Com$. Assume that the judgment $\vdash \Lambda\ \{c\}\ \Lambda'$ is derivable and that $\langle c, mds, mem\rangle \xrightarrow{\alpha} \langle c', mds', mem'\rangle$. Then there exists Λ'' such that the following conditions are satisfied:
- $\vdash \Lambda''\ \{c'\}\ \Lambda'$, and
- if mds is consistent with Λ then mds' is consistent with Λ''. ◊

Proof Sketch. The proof is by induction on the derivation of $\vdash \Lambda\ \{c\}\ \Lambda'$. A detailed proof is available in Appendix A.7. □

Theorem 4.7. Let $c \in Com$. Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derivable, and let mds be a mode state consistent with Λ. Let mem_1 and mem_2 be memories with $mem_1(x) = mem_2(x)$ for all $x \in Var$ with $\Lambda\langle x\rangle = low$. Then $\langle c, mds, mem_1\rangle \sim_{mm} \langle c, mds, mem_2\rangle$. ◊

Proof Sketch. To prove the theorem we construct a family of relations on thread configurations with modes $(\mathcal{R}^{\Lambda})_{\Lambda: Var \rightharpoonup \{low, high\}}$ consisting of a relation for each partial environment Λ such that (a) the union of these relations is a strong low bisimulation modulo modes and (b) $\langle c, mds, mem_1\rangle\ \mathcal{R}^{\Lambda'}\ \langle c, mds, mem_2\rangle$ is satisfied.

$$[\mathsf{PAR}_{\mathcal{S}}]\ \frac{\begin{array}{c}\forall i \in \{1,\ldots,\sharp(thr)\} : \vdash \Lambda_i\ \{thr[i]\}\ \Lambda_i' \\ \forall i \in \{1,\ldots,\sharp(thr)\} : \forall x \in \mathit{Var} : \mathit{dma}(x) \sqsubseteq \Lambda_i\langle x\rangle \\ \forall i \in \{1,\ldots,\sharp(thr)\} : \quad \mathit{mdss}[i] \text{ is consistent with } \Lambda_i \\ (thr, \mathit{mdss}) \text{ has sound modes for } \mathcal{S}\end{array}}{\vdash thr}$$

Figure 4.4: Typing rule for thread pools

Intuitively, two configurations are related if their commands are typable with partial environments Λ and Λ', their mode states are consistent with Λ, and their memories are equal for all variables x with $\Lambda\langle x\rangle = \mathit{low}$. To take the semantic side condition in rule [IF] into account, the relation also relates configurations that are related by $\sim_{mm}$ as long as the commands are typable with partial environments Λ_1 and Λ' and with partial environments Λ_2 and Λ', respectively.

A formal definition of the family of relations and the proof that their union is a strong low bisimulation modulo modes is available in Appendix A.7. □

Corollary 4.1. Let $c \in \mathit{Com}$, let Λ and Λ' be such that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derivable, and let mds be a mode state. Assume that mds is consistent with Λ, and that $\mathit{dma}(x) \sqsubseteq \Lambda\langle x\rangle$ for all $x \in \mathit{Var}$. Then c is SIFUM-secure for mode state mds. ◊

Proof. To show that c is SIFUM-secure for mds we must show that $\langle c, \mathit{mds}, \mathit{mem}_1\rangle \sim_{mm} \langle c, \mathit{mds}, \mathit{mem}_2\rangle$ for all mem_1 and mem_2 with $\mathit{mem}_1 =_L \mathit{mem}_2$.

Let mem_1 and mem_2 be two memories with $\mathit{mem}_1 =_L \mathit{mem}_2$. Let $x \in \mathit{Var}$ with $\Lambda\langle x\rangle = \mathit{low}$. Then it follows from the assumption $\mathit{dma}(x) \sqsubseteq \Lambda\langle x\rangle$ that $\mathit{dma}(x) = \mathit{low}$. Hence, it follows from $\mathit{mem}_1 =_L \mathit{mem}_2$ that $\mathit{mem}_1(x) = \mathit{mem}_2(x)$. I.e., all assumptions of Theorem 4.7 hold, and, in consequence, $\langle c, \mathit{mds}, \mathit{mem}_1\rangle \sim_{mm} \langle c, \mathit{mds}, \mathit{mem}_2\rangle$. □

Corollary 4.2. Let $c \in \mathit{Com}$ and let Λ' be such that $\vdash \Lambda_0\ \{c\}\ \Lambda'$ is derivable (where Λ_0 is the partial environment with $\mathit{dom}(\Lambda_0) = \{\}$). Then c is SIFUM-secure for any mode state.◊

Proof. By Definition 4.23, every mode state is consistent with Λ_0. Moreover, by Definition 4.21, $\Lambda_0\langle x\rangle = \mathit{dma}(x)$ for all $x \in \mathit{Var}$. Hence, by Corollary 4.1, c is SIFUM-secure for any mode state. □

Typing judgments for thread pools have the form $\vdash thr$. The corresponding derivation rule is parametric in a scheduler. It is displayed in Figure 4.4. The premises of the rule ensure, firstly, that every command in the thread pool is typable with suitable partial environments. In consequence, every command is SIFUM-secure for some mode state by Corollary 4.1. Secondly, the premises ensure that the thread pool has sound modes for these mode states and the scheduler. In combination, this ensures that typable thread pools are $\mathcal{S}$-noninterferent.

Theorem 4.8. Assume that the observation function is confined to L. If the judgment $\vdash thr$ is derivable with typing rule $[\mathsf{PAR}_{\mathcal{S}}]$ then the thread pool thr is $\mathcal{S}$-noninterferent.◊

Proof. Since $\vdash thr$ is derivable, by the typing rule $[\mathsf{PAR}_{\mathcal{S}}]$ there exist a list of mode states mdss and partial environments $\Lambda_1, \ldots, \Lambda_{\sharp(thr)}$ and $\Lambda_1', \ldots, \Lambda_{\sharp(thr)}'$ such that $\vdash \Lambda_i\ \{thr[i]\}\ \Lambda_i'$ is derivable, $\mathit{dma}(x) \sqsubseteq \Lambda_i\langle x\rangle$ for all $x \in \mathit{Var}$, and $\mathit{mdss}[i]$ is consistent with

Λ_i for all $i \in \{1, \ldots, \sharp(thr)\}$, and such that $(thr, mdss)$ has sound modes for $\mathcal{S}$. Hence, by Corollary 4.1, it holds for all $i \in \{1, \ldots, \sharp(thr)\}$ that $thr[i]$ is SIFUM-secure for mode state $mdss[i]$. Since modes are sound for $\mathcal{S}$, with Theorem 4.3 it follows that thr is $\mathcal{S}$-noninterferent. □

4.5.3 Enforcing Sound Use of Modes

The typing rule [$\mathsf{PAR}_{\mathcal{S}}$] requires that thread pools and lists of mode states have sound modes, i.e., that the set of mode state lists reachable from a configuration has compatible modes, and that thread configurations with modes respect guarantees, have no no-read assumptions at thread termination, and have modes compatible with obs. This verification is not the focus of this thesis. To illustrate how a verification of sound modes could be done we sketch a simple approach in this section.

For verifying that guarantees are respected, that no no-read assumptions occur at thread termination, and that modes are compatible with obs, an upper bound on the assumptions and guarantees made at each program point can be computed for an annotated command with a simple type system. Judgments are simply triples of the form $\vdash mds\ \{c\}\ mds'$. The interpretation of the judgment is that if mds gives an upper bound on assumptions and guarantees before c executes, then mds' gives an upper bound on assumptions and guarantees after the execution of c. We provide a straightforward type system for computing this approximation of modes and verifying that guarantees are respected, that no no-read assumptions occur at thread termination, and that modes are compatible with obs in Appendix B (including a soundness proof).

For verifying the compatibility of all reachable mode state lists, one can compute another approximation of modes at each program point for an annotated program using a simple type system, where the approximation is an upper bound on the assumptions and a lower bound on the guarantees. One can then check compatibility of reachable mode state lists by using a *may-happen-in-parallel* analysis (see, e.g., [MR93, NA98, NAC99]) to determine all pairs of program points which may execute concurrently, and, using the upper bound on assumptions and the lower bound on guarantees for each of these program points, to check whether the modes are compatible.

4.5.4 Extending the Type System with Value Tracking

So far, we exploited assumptions and guarantees to determine upper bounds on the security levels of variables in a flow-sensitive manner. In addition, no-write assumptions can also be used to approximate the possible values of variables in a flow-sensitive manner, because no-write assumptions ensure that the set of possible values is not modified between execution steps by other threads. The benefit of such tracking of values is illustrated by the following simplistic example:

Example 4.7. Consider the following command:

$/\!/\mathsf{acq}(asm\text{-}no\text{-}w, \mathsf{debug})/\!/$
debug:=false;
if (debug) then log:=log + secret else skip fi

Due to the assumption that other threads do not write the variable debug it is safe to assume that the value of debug is false when the control condition of the conditional is

evaluated, and that, hence, the branch with the insecure assignment is not executed. In fact, the command is SIFUM-secure. ◊

No-write assumptions are, for instance, used in the spirit of tracking values in program analyses based on concurrent separation logic like, e.g., by Dodds, Feng, Parkinson, and Vafeiadis in [DFPV09]. They exploit assumptions to ensure that the truth values of formulas in a Hoare-like logic remain unchanged between execution steps of a thread. In this section, we illustrate how the type system from Section 4.5.2 can be extended with value tracking, which has to our knowledge not been used for the information flow analysis of multi-threaded programs so far. In combination with the partial environments this even permits to determine upper bounds on the security levels of variables that depend on the current values of variables.

To extend the type system, we consider *partial memories* $pm : \mathit{Var} \rightharpoonup \mathit{Val}$ that we use to map only variables with no-write assumption to a value, while the values of other variables are not specified. We extend the evaluation of expressions to partial memories by $\mathit{eval}(e, pm) = \{\mathit{eval}(e, \mathit{mem}) \mid \mathit{mem} \in \mathit{Mem} \wedge \forall x \in \mathit{dom}(pm) : \mathit{mem}(x) = pm(x)\}$, i.e., $\mathit{eval}(e, pm)$ contains the values to which e might evaluate given the values of variables in $\mathit{dom}(pm)$.

We modify the typing judgments from Section 4.5.2 by using *dependent partial environments* instead of partial environments. A dependent partial environment is a partial function $\Delta : (\mathit{Var} \rightharpoonup \mathit{Val}) \rightharpoonup (\mathit{Var} \rightharpoonup \{\mathit{low}, \mathit{high}\})$ with $\mathit{dom}(\Delta) \neq \{\}$ such that all $pm \in \mathit{dom}(\Delta)$ have the same domain. The extended judgments are of the form

$$\vdash \Delta\ \{c\}\ \Delta',$$

where Δ, Δ' are dependent partial environments. The interpretation of the judgments is as follows: Assume that the domain of the partial memories in $\mathit{dom}(\Delta)$ is a lower bound on the set of variables with no-write assumption before the execution of c. Then the domain of the partial memories in $\mathit{dom}(\Delta')$ is a lower bound on this set after the execution of c. Moreover, if $\mathit{dom}(\Delta)$ is a lower bound on the set of possible values of variables with no-write assumption before the execution of c then $\mathit{dom}(\Delta')$ is a lower bound on this set after the execution of c. Finally, if the values of variables with no-write assumption are as specified by pm then $\Delta(pm)\langle\cdot\rangle$ and $\Delta'(pm)\langle\cdot\rangle$ give upper bounds on the security domains of variables like in the type system from Section 4.5.2.

Example 4.8. Let Δ be a dependent partial environment with $\mathit{dom}(\Delta) = \{pm_1, pm_2\}$, where $\mathit{dom}(pm_1) = \mathit{dom}(pm_2) = \{x\}$. This encodes that a no-write assumption is made at least for the variable x.

Moreover, assume that $pm_1(x) = 1$ and that $pm_2(x) = 2$. This encodes that x might have value 1 or value 2, and that it is certain that x does not have any other value.

Assume furthermore that $\Delta(pm_1) = \Lambda_1$ and $\Delta(pm_2) = \Lambda_2$, where $\mathit{dom}(\Lambda_1) = \mathit{dom}(\Lambda_2) = \{y\}$ and $\mathit{dma}(y) = \mathit{low}$. This encodes that regardless of the value of x a no-read assumption is made for y.

Finally, assume that $\Lambda_1(y) = \mathit{low}$ and $\Lambda_2(y) = \mathit{high}$. This encodes that if the value of x equals 1 then the security level of the value of y is guaranteed to be *low*, while the security level of y might be *high* if the value of x equals 2. ◊

We define the typing rules by extending the rules from Figure 4.3 in Section 4.5.2. The extended rules are displayed in Figure 4.5.

Rule [ASS] ensures that an assignment $x{:=}e$ is not typable if $\mathit{dma}(x) = \mathit{low}$, the upper bound on the security level of e is *high*, and x is not in the domain of the partial

$$[\mathsf{EXP}]\ \frac{}{\Gamma \vdash e : \bigsqcup_{x \in vars(e)} \Gamma(x)} \qquad [\mathsf{STOP}]\ \frac{}{\vdash \Delta\ \{stop\}\ \Delta} \qquad [\mathsf{SKIP}]\ \frac{}{\vdash \Delta\ \{\mathsf{skip}\}\ \Delta}$$

$$[\mathsf{ASS}]\ \frac{\forall pm \in dom(\Delta) : \forall x \notin dom(\Delta(pm)) : \Delta(pm)\langle\cdot\rangle \vdash e : d_{pm} \Rightarrow d_{pm} \sqsubseteq dma(x)}{\vdash \Delta\ \{x{:=}e\}\ \mathit{update\text{-}dpe}(\Delta, x, e)}$$

$$[\mathsf{IF}]\ \frac{\begin{array}{c}\Delta_1 = \Delta\,|_{\{pm \mid \mathsf{true} \in eval(e,pm)\}} \qquad dom(\Delta_1) \neq \{\} \Rightarrow \vdash \Delta_1\ \{c_1\}\ \Delta' \\ \Delta_2 = \Delta\,|_{\{pm \mid \mathsf{false} \in eval(e,pm)\}} \qquad dom(\Delta_2) \neq \{\} \Rightarrow \vdash \Delta_2\ \{c_2\}\ \Delta' \\ pm \in dom(\Delta) \Rightarrow \Delta(pm)\langle\cdot\rangle \vdash e : low\end{array}}{\vdash \Delta\ \{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\ \Delta'}$$

$$[\mathsf{WHILE}]\ \frac{pm \in dom(\Delta) \Rightarrow \Delta(pm)\langle\cdot\rangle \vdash e : low \quad \vdash \Delta\ \{c\}\ \Delta}{\vdash \Delta\ \{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\}\ \Delta}$$

$$[\mathsf{SEQ}]\ \frac{\vdash \Delta\ \{c_1\}\ \Delta' \quad \vdash \Delta'\ \{c_2\}\ \Delta''}{\vdash \Delta\ \{c_1; c_2\}\ \Delta''}$$

$$[\mathsf{SPAWN}]\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, \sharp(thr)\} : \vdash \Delta_0\ \{thr[i]\}\ \Delta_i \\ pm \in dom(\Delta \Rightarrow \forall x \in Var : \Delta(pm)\langle x\rangle \sqsubseteq dma(x)\end{array}}{\vdash \Delta\ \{\mathsf{spawn}(thr)\}\ \Delta}$$

$$[\mathsf{ANNO}]\ \frac{\begin{array}{c}\Delta' = \mathit{adjust\text{-}dpe}(\Delta, ann) \qquad \vdash \Delta'\ \{c\}\ \Delta'' \\ pm \in dom(\Delta) \Rightarrow \forall x \in Var : \Delta(pm)\langle x\rangle \sqsubseteq \Delta'(pm)\langle x\rangle\end{array}}{\vdash \Delta\ \{/\!\!/ ann /\!\!/\ c\}\ \Delta''}$$

$$[\mathsf{SUB}]\ \frac{\Delta_2 \sqsubseteq \Delta_1 \quad \Delta_1' \sqsubseteq \Delta_2' \quad \vdash \Delta_1\ \{c\}\ \Delta_1'}{\vdash \Delta_2\ \{c\}\ \Delta_2'}$$

Figure 4.5: Flow-sensitive security type system with value tracking

environment for any of the possible partial memories pm. This corresponds to rule [ASS_1] from the original type system. Moreover, to construct the partial environment after the assignment the domain of Δ is updated by the function *update-dpe* (for "update dependent partial environment") based on the assignment, and each partial environment is updated like in rules [ASS_1] and [ASS_2] in the original type system.

Definition 4.25. Let Δ be a dependent partial environment, $x \in Var$, and $e \in Exp$. We define the dependent partial environment $\mathit{update\text{-}dpe}(\Delta, x, e)$ as follows, where for each $pm \in dom(\Delta)$ the security domain d_{pm} is such that $\Delta(pm)\langle\cdot\rangle \vdash e : d_{pm}$ is derivable and the partial environment Λ'_{pm} equals $\Delta(pm)$ if $x \notin dom(\Delta(pm))$ and $\Delta(pm)[x \mapsto d_{pm}]$ otherwise, and X is the domain of the partial memories in $dom(\Delta)$:

- If $x \notin X$ then $dom(\mathit{update\text{-}dpe}(\Delta, x, e)) = dom(\Delta)$, and $(\mathit{update\text{-}dpe}(\Delta, x, e))(pm) = \Lambda'_{pm}$ for all $pm \in dom(\Delta)$.
- If $x \in X$ then $dom(\mathit{update\text{-}dpe}(\Delta, x, e)) = \{pm[x \mapsto v] \mid pm \in dom(\Delta) \wedge v \in eval(e, pm)\}$, and $(\mathit{update\text{-}dpe}(\Delta, x, e))(pm[x \mapsto v]) = \bigsqcup_{\{pm \in dom(\Delta) \mid v \in eval(e,pm)\}} \Lambda'_{pm}$ for all $pm[x \mapsto v] \in dom(\mathit{update\text{-}dpe}(\Delta, x, e))$, where $(\Lambda \sqcup \Lambda')(x) = \Lambda(x) \sqcup \Lambda'(x)$. $\Diamond$

Intuitively, Definition 4.25 ensures that the value of the assigned variable x is updated in the partial memories in the domain of the dependent partial environment only if x is

in the domain of these partial memories. This happens in Item 2 of the definition. Item 2 moreover ensures that if the upper bounds on the security levels depended on the value of x before the assignment then the least upper bound of the different upper bounds before the assignment is used after the assignment. For instance, if the upper bound for a variable y was *high* if the value of x equals 0 and *low* if the value of x equals 1, then the upper bound for y will be set to *high* by the function *update-dpe*.

Rule [IF] exploits that the branches need only be analyzed if the control condition evaluates to true or false, respectively, given the values of variables with no-write assumption.

Rule [ANNO] updates the domains of the functions when assumptions are acquired or released. We model this by extending the function *adjust-pe* from partial environments to dependent partial environments.

Definition 4.26. Let Δ be a dependent partial environment, and let $X = dom(pm)$ for some $pm \in dom(\Delta)$. Then $\Delta' = \textit{adjust-dpe}(\Delta, ann)$ is defined as follows:

- If $ann = \mathsf{acq}(\textit{asm-no-w}, x)$ then $dom(\Delta') = \{pm \mid dom(pm) = X \cup \{x\} \wedge pm|_X \in dom(\Delta)\}$, and $\Delta'(pm) = \textit{adjust-pe}(\Delta(pm|_X), ann)$ for all $pm \in dom(\Delta')$.
- If $ann = \mathsf{rel}(\textit{asm-no-w}, x)$ then $dom(\Delta') = \{pm|_{X\setminus\{x\}} \mid pm \in dom(\Delta)\}$, and $\Delta'(pm') = \textit{adjust-pe}(\bigsqcup_{\{pm \in dom(\Delta) \mid pm|_{X\setminus\{x\}} = pm'\}} \Delta(pm), ann)$ for all $pm' \in dom(\Delta')$.
- If $ann \neq \mathsf{acq}(\textit{asm-no-w}, x)$ and $ann \neq \mathsf{rel}(\textit{asm-no-w}, x)$ then $dom(\Delta') = dom(\Delta)$ and $\Delta'(pm) = \textit{adjust-pe}(\Delta(pm), ann)$ for all $pm \in dom(\Delta')$. ◊

Intuitively, the function *adjust-dpe* ensures that if a no-write assumption is acquired for variable x then x is included in the domain of the partial memories in the dependent partial environment (cf. the first item in the definition). Moreover, if the no-write assumption is released then the function *adjust-dpe* ensures that x is removed from the domain of the partial memories (cf. the second item in the definition). If in this case the upper bounds given by the partial environments depended on the value of x then the function *adjust-dpe* ensures that these partial environments are suitably combined.

For subtyping we extend the order on partial to dependent partial environments.

Definition 4.27. We define the order $\sqsubseteq$ on dependent partial environments by $\Delta \sqsubseteq \Delta'$ if and only if $X \supseteq X'$ where X is the domain of the partial memories in $dom(\Delta)$ and X' is the domain of the partial memories in $dom(\Delta')$, and, for all $pm \in dom(\Delta)$, $\Delta(pm) \sqsubseteq \Delta'(pm|_{X'})$. ◊

We derive a soundness result in analogy to the soundness result for the type system from Section 4.5.2, where we say that Δ is consistent with a mode state *mds* if and only if $X \subseteq \{x \in \mathit{Var} \mid x \in \textit{mds}(\textit{asm-no-w})\}$ where X is the domain of the partial memories in $dom(\Delta)$, and $\Delta(pm)$ is consistent with *mds* for all $pm \in dom(\Delta)$.

Theorem 4.9. Assume that $\vdash \Delta\ \{c\}\ \Delta'$ is derivable, let *mds* be a mode state consistent with Δ, let $pm \in dom(\Delta)$, and let X be the domain of the partial memories in $dom(\Delta)$. Let $mem_1, mem_2 \in \mathit{Mem}$ be such that $mem_1(x) = mem_2(x) = pm(x)$ for all $x \in dom(pm)$. Then $\langle c, \textit{mds}, mem_1\rangle \sim_{mm} \langle c, \textit{mds}, mem_2\rangle$ if $mem_1(x) = mem_2(x)$ for all $x \in \mathit{Var}$ with $\Delta(pm)\langle x\rangle = \mathit{low}$. ◊

A proof of Theorem 4.9 (see Appendix A.7) is obtained by adapting the proof of the corresponding result for the type system from Section 4.5.2 (Theorem 4.7). The major differences are (a) in the definition of the bisimulation relation, which now requires that the memories in the related configurations, restricted to the domain of the partial memories in the domain of the dependent partial environment, are in the domain of the

dependent partial environment), and (b) in the induction steps for derivations with rules [ASS] and rule [IF].

Example 4.9. The command from Example 4.7 is typable with the type system, and, hence, SIFUM-secure by Theorem 4.9. ◊

Note that the program from Example 4.7 is rejected by many existing compositional analysis techniques like in [VS99, SS00, RS06a] and the analysis from Chapter 3. While the compositional analysis techniques in [ZM03, MSK07] classify programs like in Example 4.9 as secure, they impose the restriction that the variable log must not be concurrently written by the environment, which is restrictive because logging is often performed in multiple threads of a multi-threaded program. In contrast, for Example 4.7 SIFUM-security does not restrict concurrent modifications of the log.

4.6 Benefits of the Flow-sensitive Analysis

We illustrate the benefits of the type-based analyses from Section 4.5 at two small, yet realistic examples. With the first example we illustrate how the flow-sensitive security types are exploited. The second example illustrates how the combination of flow-sensitive security types and value tracking is exploited.

Example 4.10. Consider a thread that executes the command displayed in Figure 4.6. The thread subsequently swaps the values of the two secret variables key1 and key2, and the values of the two public variables pub1 and pub2. The swaps are performed using a temporary variable with mode *asm-no-r* during the execution of c_{temp}. This assumption is, for instance, natural if temp is used as a local variable in the thread. ◊

Theorem 4.10. Assume that $dma(\mathsf{key1}) = dma(\mathsf{key2}) = high$ and that $dma(\mathsf{pub1}) = dma(\mathsf{pub2}) = low$. Then command c_{temp} from Example 4.10 is SIFUM-secure for mds_0 (defined by $mds_0(m) = \{\}$ for all $m \in Mod$). ◊

Proof. We prove the statement by deriving the judgment $\vdash \Lambda_0\ \{c_{temp}\}\ \Lambda_0$ in the type system without value tracking (from Section 4.5.2). Let Λ^{low} and Λ^{high} be partial environments defined by $dom(\Lambda^{low}) = dom(\Lambda^{high}) = \{\mathsf{temp}\}$, $\Lambda^{low}(\mathsf{temp}) = low$, and $\Lambda^{high}(\mathsf{temp}) = high$. Then the judgments
(1) $\vdash \Lambda_0\ \{/\!\!/\mathsf{acq}(\textit{asm-no-r}, \mathsf{temp})/\!\!/; \mathsf{temp}{:=}\mathsf{key1}\}\ \Lambda^{high}$,
(2) $\vdash \Lambda^{high}\ \{\mathsf{key1}{:=}\mathsf{key2}\}\ \Lambda^{high}$,
(3) $\vdash \Lambda^{high}\ \{\mathsf{key2}{:=}\mathsf{temp}\}\ \Lambda^{high}$,
(4) $\vdash \Lambda^{high}\ \{\mathsf{temp}{:=}\mathsf{pub1}\}\ \Lambda^{low}$,
(5) $\vdash \Lambda^{low}\ \{\mathsf{pub1}{:=}\mathsf{pub2}\}\ \Lambda^{low}$, and
(6) $\vdash \Lambda^{low}\ \{\mathsf{pub2}{:=}\mathsf{temp}/\!\!/\mathsf{rel}(\textit{asm-no-r}, \mathsf{temp})/\!\!/\}\ \Lambda_0$

are derivable using rules [ANNO], [ASS_2], [SKIP], and [SEQ] for (1), [ASS_1] for (2), (3), and (5), [ASS_2] for (4), and [ANNO] and [ASS_1] for (6). By five applications of rule [SEQ] it follows that $\vdash \Lambda_0\ \{c_{temp}\}\ \Lambda_0$ is derivable.

In consequence, c_{temp} is SIFUM-secure for mds_0 by Theorem 4.7. □

Example 4.10 illustrates that flow-sensitive reasoning permits to successfully analyze secure programs that were rejected by existing compositional analysis techniques like, e.g., the analyses in [VS99, SS00, RS06a] and the analysis from Chapter 3.

In the next example, we illustrate how using both no-read and no-write assumptions can be exploited in the information flow analysis of more complex application scenarios.

$c_{temp} =$
  $/\!/\mathsf{acq}(asm\text{-}no\text{-}r, \mathsf{temp})/\!/;$
  temp:=key1; key1:=key2; key2:=temp;
  temp:=pub1; pub1:=pub2; pub2:=temp
  $/\!/\mathsf{rel}(asm\text{-}no\text{-}r, \mathsf{temp})/\!/$

Figure 4.6: Thread reusing temporary variable

Example 4.11. We consider a multi-threaded server. The command c_{srv} in Figure 4.7 implements a thread that is part of the server. The thread serves a client that requests information from the server. The contents of the variables request and src are set up by the main server thread (which we do not display here) specifying the data that is requested and the network address of the requester, respectively. This technique is used in multi-threaded server implementations like NULL HTTPD [Nul] that spawn worker threads working on data structures set up by the main server thread. To simplify the setting, we assume identifiers for two categories of data ("secret-data" and "public-data") and two network addresses ("local" and "nonlocal"). The variables secretData and publicData contain the data identified by "secret-data" and "public-data", respectively. The variables localout and nonlocalout represent output channels to the network addresses "local" and "nonlocal", respectively. The command c_{srv} operates in three steps: It firstly checks whether the request's source is authorized to access the requested data, where the policy is that "secret-data" may only be sent to "local". Afterwards, it computes the answer to the request (which is either the requested data or an error message if the authorization failed). Finally, it sends the answer to the channel identified by src, deletes the answer, and logs the request. We use a mutex variable mutex[1] to ensure exclusive access to variables that are shared with other threads (as for instance request and src which are shared with the main server thread). If other threads only access these variables when holding the mutex variable mutex, they provide guarantees matching the assumptions specified for c_{srv}. $\Diamond$

Theorem 4.11. Assume that $dma(\mathsf{secretData}) = dma(\mathsf{localout}) = high$ and $dma(x) = low$ for all other variables. Then the command c_{srv} from Example 4.11 (omitting the mutex operations) is SIFUM-secure for mds_0 (defined by $mds_0(m) = \{\}$ for all $m \in Mod$).[2] $\Diamond$

Proof Sketch. For the proof we use the type system with value tracking from Section 4.5.4, showing that the judgment $\vdash \Delta_0\ \{c_{srv}\}\ \Delta_0$ is derivable (where Δ_0 is defined as the dependent partial environment with domain $\{pm_0\}$ and $\Delta_0(pm_0) = \Lambda_0$, where both pm_0 and Λ_0 have domain $\{\}$). We illustrate the type derivation by sketching the dependent partial environments at the intermediate program points of c_{srv}.

- Due to the four annotations, the domain of the partial memories in the domain of the dependent partial environment is set to $\{\mathsf{src}, \mathsf{request}, \mathsf{auth}\}$ by rule [ANNO]. Moreover, the domain of the partial environments in the image of the dependent partial environment is set to $\{\mathsf{answer}\}$.
- After the first conditional, the domain of the dependent partial environment contains all partial memories pm with one of the following two properties:

[1] Given that our simple language does not support mutexes as primitives, the operations for acquiring and releasing mutexes can be implemented using, e.g., Peterson's algorithm [Pet81] without leaving the language fragment.

[2] Assuming that the mutex operations are actually implemented with Peterson's algorithm using low flag variables the command including these operations is also SIFUM-secure.

```
c_srv =
 acquire(mutex);
 //acq(asm-no-w, src)// //acq(asm-no-w, request)//
 //acq(asm-no-w, auth)// //acq(asm-no-r, answer)//
 if (src ≠ "local" ∧ request = "secret-data")
  then auth:=false
  else auth:=true
 fi;
 if (auth = false)
  then answer:="not authorized"
  else if (request = "public-data")
    then answer:=publicData
    else answer:=secretData
  fi
 fi;
 if (src = "local")
  then localout:=answer
  else nonlocalout:=answer
 fi;
 answer:="";
 log:=log + src + request;
 //rel(asm-no-w, src)// //rel(asm-no-w, request)//
 //rel(asm-no-w, auth)// //rel(asm-no-r, answer)//
 release(mutex)
```

Figure 4.7: Worker thread of multi-threaded server

  - $pm(\mathsf{src}) \neq$ "local", $pm(\mathsf{request}) =$ "secret-data", and $pm(\mathsf{auth}) = \mathsf{false}$, or
  - $pm(\mathsf{src}) =$ "local" or $pm(\mathsf{request}) \neq$ "secret-data", and $pm(\mathsf{auth}) = \mathsf{true}$.

  In both cases the dependent partial environment maps pm to the partial environment that assigns *low* to answer.
- After the second conditional, the domain of the dependent partial environment contains all partial memories pm with one of the following properties:
  - $pm(\mathsf{src}) \neq$ "local", $pm(\mathsf{request}) =$ "secret-data", $pm(\mathsf{auth}) = \mathsf{false}$, and $pm(\mathsf{answer}) =$ "not authorized", or
  - $pm(\mathsf{src}) =$ "local" or $pm(\mathsf{request}) \neq$ "secret-data", and $pm(\mathsf{auth}) = \mathsf{true}$.

  The partial memory pm is mapped to the partial environment that maps answer to *high* if and only if $pm(\mathsf{auth}) = \mathsf{true}$ and $pm(\mathsf{request}) \neq$ "public-data". Otherwise, pm is mapped to the partial environment that maps answer to *low*.
- The last conditional does not change the dependent partial environment. Note, however, that the conditional is only typable because the partial environment maps answer to *high* if and only if src $\neq$ "local". If answer would be mapped to *high* while src = "local" then the type system would require to type the assignment of answer to localout, which is not possible if answer is mapped to *high*. □

The thread would not be typable without the no-write assumptions for src, request, and auth. In fact, if other threads could modify these variables, for instance, the value of request could be modified from "public-data" to "secret-data" after checking the authorization. This modification would result in secretData being sent to nonlocalout although this is not

authorized, constituting a type of attack also referred to as a time-of-check-to-time-of-use (TOCTTOU) attack [LBMC94].

Example 4.11 illustrates how the flow-sensitive information flow analysis can be exploited in the security analysis in realistic multi-threaded example scenarios. We are not aware of a compositional analysis technique for multi-threaded programs that allows a successful information flow security analysis for the multi-threaded server in Example 4.11.

4.7 Summary and Comparison to Related Work

In this chapter, we have presented the first flow-sensitive security type system for analyzing information flow security of multi-threaded programs. The key for developing such a type system was the introduction of assumption-guarantee style reasoning into the information flow analysis. Exploiting assumptions and guarantees about variable accesses of threads permits to do flow-sensitive analysis without giving up compositionality, and, hence, without giving up a modular security analysis.

Moreover, exploiting assumptions and guarantees permits to exploit information about the synchronization between threads, because, typically, synchronization ensures that variables are accessed in a controlled way. Other approaches do not exploit synchronization in the security analysis, but rather only consider synchronization as a potential source of forbidden information leakage (e.g., [Sab01, RS09]). Our assumption-guarantee based approach permits to consider both positive and negative influences of synchronization on the information flow in a program.

As a basis for the flow-sensitive security type system we developed a novel information flow property, *SIFUM-security*, that is compatible with assumption-guarantee style reasoning. SIFUM-security abstracts from how assumptions and guarantees are determined. This permits the manual specification of assumptions and guarantees, but also the computation of assumptions and guarantees by a program analysis. Moreover, it leaves freedom of the choice of synchronization patterns that are used to establish the validity of assumptions and guarantees.

We have shown that SIFUM-security is compositional and scheduler independent. The only requirements for this result are that assumptions and guarantees are used in a sound way, i.e., that assumptions are matched by valid guarantees in all other threads, and that assumptions are not violated by memory accesses of the scheduler or when the program outputs values. Whether assumptions and guarantees are sound can be checked based on existing program analysis techniques that compute may-happen-in-parallel information.

Finally, we have illustrated at realistic example programs that our flow-sensitive information flow analysis permits to successfully analyze secure programs that were rejected by existing analysis techniques. In particular, the flow-sensitive analysis permits to successfully classify programs as secure that are not classified as secure by existing flow-insensitive information flow analyses for multi-threaded programs like, e.g., the analyses in [VS98, VS99, SS00, Smi01, BC02, ZM03, RS06a].

Interestingly, in contrast to more modern information flow analyses for multi-threaded programs, the to our knowledge first information flow analysis for multi-threaded programs proposed by Andrews and Reitman in [RA79, AR80] supports flow-sensitive reasoning. Andrews and Reitman consider a flow logic that supports the derivation of noninterference proofs for programs. They sketch an extension to concurrent programs based on the Owicki-Gries method [OG76]. The idea is that one must show that any derivation for a given thread is unaffected by the assignment statements in any other thread. As well as being non-compositional, a consequence of this approach is an assumption that

any assignment in one thread may run concurrently with any other statement. This would make the opportunities to exploit flow sensitivity limited to thread local variables. Moreover, the approach neither provides a semantically justified soundness result nor a formal semantics for the underlying programming language.

There are some other approaches that use the general idea of assumption-guarantee reasoning to perform modular security analyses. Jürjens [Jür00] studied a probabilistic noninterference for trace-based message-passing processes, and used assumption-guarantee reasoning in the proof of compositionality. In contrast to our approach, assumptions and guarantees are not explicitly exploited in the analysis, but rather used as a proof technique in the compositionality result. Guttman et al [GTC+04] annotate Strand-space representations of protocol participants with assumption-guarantee statements related to trust. In common with the approach here a soundness of assumptions and guarantees is required for annotated protocols. Garg et al [GFKD10] place heavy emphasis on assumption-guarantee proof rules in reasoning about safety-property-based security properties in a concurrent first-order functional language.

Finally, assumptions similar to those used in this thesis have been proposed by others in contexts that were not security-specific. As our focus is information flow security, we do not consider in depth the problem of verifying the soundness of modes. One influential line of work stems from Boyland's *fractional permissions* [Boy03] that permit to divide permissions among multiple threads: Full permissions permit both writing and reading, partial permissions permit only reading. I.e., full permissions correspond to the combination of our no-read and no-write assumption, and partial permissions to the combination of our no-write assumption and no-write guarantee. Other kinds of fractional permissions have also been investigated: In [DFPV09], fractional *deny*-permissions, roughly, correspond to the combination of our no-write assumption and guarantee, and fractional *guar*-permissions to neither making no-write assumptions nor providing no-write guarantees. Based on ideas from concurrent separation logic [O'H04], compositional logics for reasoning with fractional permissions have been developed, where a key to compositionality is to exploit specific synchronization primitives that permit to safely transfer permissions between threads. For instance, [DFPV09] uses synchronization points provided by fork/join synchronization to transfer permissions between dynamically forked threads, and [HG11] allows to transfer permissions at synchronization points established by barrier synchronization. Other approaches exploit atomic statements [PBO07] and conditional critical regions [O'H04]. Our approach is, in contrast, not specific to some particular synchronization primitive. The price we pay for this generality is that we cannot describe the soundness of modes in a compositional way. The closest form of assumption to those used in the present paper are the thread-level *exclusive ownership* (nobody else will read or write) and *read ownership* policies used by Martin et al [MHC+10]. Ownership is acquired and released via annotations. The soundness of the annotations are not verified however, but are checked using a runtime monitor.

CHAPTER 5

Integrating Flexible Scheduler Independence and Flow-sensitivity

5.1 Introduction

In Chapters 3 and 4 we developed information flow security analyses that improved the state of the art in two directions: The analysis from Chapter 3 permits a more flexible scheduler-independent analysis and, for instance, does not reject all secure programs with control conditions that depend on secrets. Moreover, the analysis from Chapter 4 is flow-sensitive and thereby increases analysis precision. However, both improvements are needed for the successful analysis of some secure programs.

To combine both improvements, in this chapter we integrate our solutions from Chapters 3 and 4. We firstly integrate the security properties FSI-security and SIFUM-security, resulting in a novel compositional and scheduler-independent security property that inherits benefits of FSI-security (scheduler independence without unconditionally rejecting programs with secret control conditions) and SIFUM-security (support for flow-sensitive information flow analyses). Secondly, we integrate the security type system for FSI-security and the flow-sensitive security type system for SIFUM-security, resulting in a scheduler-independent flow-sensitive information flow analysis that does not unconditionally reject programs with secret control conditions.

Overview. We integrate FSI-security and SIFUM-security in Section 5.2, and we integrate the security type systems for these properties in Section 5.3. Section 5.4 briefly summarizes the results.

5.2 Integration of FSI-Security and SIFUM-Security

To integrate FSI-security and SIFUM-security, we use the definition of strong low bisimulations modulo modes from Section 4.4 as a starting point that we modify based on low bisimulations modulo low matching from Section 3.3.

Definition 5.1. A symmetric binary relation $\mathcal{R}$ on thread configurations with modes that have equal mode states is a *low bisimulation modulo low matching and modes* if it is closed under globally consistent changes, and if for all $c_1, c_2 \in Com$, $mds \in Mds$, and $mem_1, mem_2 \in Mem$ with $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ the following conditions are satisfied:

1. $mem_1 =_L^{mds} mem_2$, and
2. if $c_1 \notin HCom$ then for all $c_1' \in Com$, $mds' \in Mds$, $mem_1' \in Mem$, and $\alpha_1 = new(thr_1, mdss_1) \in Lab_m$, if $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$ then there exist $c_2' \in Com$, $mem_2' \in Mem$, and $\alpha_2 = new(thr_2, mdss_2) \in Lab_m$ such that the following conditions are satisfied:
   a) $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
   b) $\langle c_1', mds_1', mem_1'\rangle \mathcal{R} \langle c_2', mds_2', mem_2'\rangle$, and
   c) there exists an injective partial function $match : \{1, \ldots, \sharp(thr_1)\} \rightharpoonup \{1, \ldots, \sharp(thr_2)\}$ such that $\langle thr_1[i], mdss_1[i], mem_1'\rangle \mathcal{R} \langle thr_2[match(i)], mdss_2[match(i)], mem_2'\rangle$ for all $i \in dom(match)$, $thr_1[i] \in HCom$ for all $i \notin dom(match)$, and $thr_2[i] \in HCom$ for all $i \notin img(match)$.

The relation $\sim_{lmm}$ is the union of all low bisimulations modulo low matching and modes.$\Diamond$

Theorem 5.1. Relation $\sim_{lmm}$ is a low bisimulation modulo low matching and modes. $\Diamond$

Theorem 5.2. Relation $\sim_{lmm}$ is a partial equivalence relation. $\Diamond$

The proofs for Theorems 5.1 and 5.2 are along the same lines as the proofs for the corresponding theorems for low bisimulations modulo low matching and strong low bisimulations modulo modes, i.e., Theorems 3.4, 3.5, 4.1, and 4.2.

In contrast to the definitions of low bisimulations modulo low matching and strong low bisimulations modulo modes (Definitions 3.6 and 4.6), Definition 5.1 does not place a requirement on the execution steps of high threads. Although we could have required that after an execution step of a high thread in one of two bisimilar configurations one obtains again two bisimilar configurations, we chose to omit such a requirement to simplify the definition. Omitting the requirement does not weaken the resulting bisimulation relation, in analogy to the result that we have shown for low bisimulations modulo low matching in Theorem 3.7 stating that two thread pools are bisimular modulo low matching if and only if they are bisimilar modulo low matching after removing all high threads. In fact, later in this section we show that Definition 5.1 leads to a security definition that implies $\mathcal{S}$-noninterference, like FSI-security and SIFUM-security (cf. Theorem 5.3).

Intuitively, the partial function *match* in Item (2c) of Definition 5.1 corresponds to the low match in the definition of low bisimulations modulo low matching. In contrast to low bisimulations modulo low matching, Definition 5.1 does not require that every low thread is matched by a low thread. The reason is that the definition of high and low threads (Definition 3.4) does not exploit no-write assumptions. In consequence, a low thread might never assign to low variables due to the assumptions made for the thread. Definition 5.1 permits to match such low threads with high threads. An example for such a low thread is given by the following example:

Example 5.1. Consider the command

$$/\!/\mathsf{acq}(\mathit{asm\text{-}no\text{-}w}, x)/\!/ x{:=}\mathsf{false}; \mathsf{if}\ x = \mathsf{true}\ \mathsf{then}\ l{:=}1\ \mathsf{else}\ \mathsf{skip}\ \mathsf{fi}.$$

The command is a low command, although it will never assign the low variable l given that concurrent threads respect the no-write assumption for x. $\Diamond$

Like for strong low bisimulations modulo modes, we use low bisimulations modulo low matching and modes to define an indistinguishability relation on commands and a security property for commands.

Definition 5.2. Two commands c_1 and c_2 are *FSIFUM-indistinguishable for mode state mds* (denoted by $c_1 \sim_{lmm}^{mds} c_2$) if and only if $\langle c_1, mds, mem_1\rangle \sim_{lmm} \langle c_2, mds, mem_2\rangle$ for all memories mem_1, mem_2 with $mem_1 =_L mem_2$. ◊

Definition 5.3. A command c is *FSIFUM-secure for mode state mds* if and only if $c \sim_{lmm}^{mds} c$. ◊

The name FSIFUM-security indicates that the security property results from integrating FSI-security and SIFUM-security. According to one's preference, one can pronounce it either *F-S-I-FUM*-security or *F-SIFUM*-security. The following compositionality and scheduler-independence theorem results from integrating the corresponding theorems for FSI-security and SIFUM-security, Theorems 3.13 and 4.3.

Theorem 5.3. Let *thr* be a terminating thread pool, *mdss* a list of mode states, and $\mathcal{S}$ a robust scheduler such that the following conditions are satisfied:

1. $thr[i]$ is FSIFUM-secure for $mdss[i]$ for all $i \in \{1, \ldots, \sharp(thr)\}$ and
2. the pair $(thr, mdss)$ has sound modes for $\mathcal{S}$.

Assume furthermore that *obs* is confined to L. Then *thr* is $\mathcal{S}$-noninterferent. ◊

The proof (see Appendix A.8) is obtained by integrating the proofs of Theorem 3.13 and of Theorem 4.3.

FSIFUM-security is no more restrictive than FSI-security for commands in which one does not specify any assumptions and guarantees. We capture such commands by the following definition.

Definition 5.4. A command c *does not affect the mode state* if for all commands c' and c'', all memories mem, mem', and mem'', all mode states mds' and mds'', and all labels α the following condition is satisfied:

$$[\langle c', mem'\rangle \in \textit{loc-reach}(\langle c, mem\rangle) \wedge \langle c', mds', mem'\rangle \xrightarrow{\alpha}\!\triangleright \langle c'', mds'', mem''\rangle] \Rightarrow mds' = mds'' \quad ◊$$

Theorem 5.4. Let c be an FSI-secure command that does not affect the mode state. Then c is FSIFUM-secure for any mode state. ◊

Proof. Define relation $\mathcal{R}$ on thread configurations by $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ if and only if $\langle c_1\rangle \sim_{lm} \langle c_2\rangle$, $mem_1 =_L mem_2$, and both c_1 and c_2 do not affect the mode state. Then $\mathcal{R}$ is a low bisimulation modulo low matching and modes (see Lemma A.15 in Appendix A.8).

Let $mem_1 =_L mem_2$. It follows from the FSI-security of c that $\langle c\rangle \sim_{lm} \langle c\rangle$. Hence, $\langle c, mds, mem_1\rangle \mathcal{R} \langle c, mds, mem_2\rangle$. Since $\mathcal{R}$ is a low bisimulation modulo low matching and modes, $\langle c, mds, mem_1\rangle \sim_{lmm} \langle c, mds, mem_2\rangle$. Hence, c is FSIFUM-secure. □

Moreover, FSIFUM-security is also no more restrictive than SIFUM-security.

Theorem 5.5. Let c be a command that is SIFUM-secure for mode state *mds*. Then c is FSIFUM-secure for mode state *mds*. ◊

Proof. If $mem_1 =_L mem_2$ it follows from the SIFUM-security of c for mode state mds that $\langle c, mds, mem_1\rangle \sim_{mm} \langle c, mds, mem_2\rangle$. Moreover, the relation $\sim_{mm}$ is a low bisimulation modulo low matching and modes (see Lemma A.16 in Appendix A.8). In consequence, $\langle c, mds, mem_1\rangle \sim_{lmm} \langle c, mds, mem_2\rangle$ for all $mem_1 =_L mem_2$. Hence, c is FSIFUM-secure for mode state mds. □

Moreover, FSIFUM-security is strictly more permissive than both FSI-security and SIFUM-security, because it permits both flow-sensitive reasoning as well as control conditions that depend on secrets. This is illustrated by the following simplistic example:

Example 5.2. Consider the following command:

$$\begin{array}{l} /\!/\mathsf{acq}(\mathit{asm\text{-}no\text{-}r}, l)/\!/; \\ l{:=}h; l{:=}0; \\ \mathsf{if}\ h > 0\ \mathsf{then\ skip; skip\ else\ skip\ fi}; \\ /\!/\mathsf{rel}(\mathit{asm\text{-}no\text{-}r}, l)/\!/ \end{array}$$

The command is FSIFUM-secure, but neither is the command with annotations removed FSI-secure nor is it SIFUM-secure. (We show that the command is FSIFUM-secure in the following section, using a security type system for FSIFUM-security.) ◊

I.e., while Theorem 5.3 shows that FSIFUM-security preserves the compositionality and the scheduler-independence of FSI-security and SIFUM-security and Theorems 5.4 and 5.5 show that there is no loss in precision with respect to FSI-security and SIFUM-security, Example 5.2 shows that FSIFUM-security even permits to exploit the advantages regarding precision of both FSI-security and SIFUM-security in combination.

5.3 Integration of the Security Type Systems

To analyze programs with respect to FSIFUM-security, we integrate the type system for FSI-security from Section 3.4 and the type system for SIFUM-security from Section 4.5. It turns out that it is sufficient to simply combine the ingredients of both kinds of typing judgment for commands in typing judgments of the form

$$\vdash \Lambda\ \{c\}\ \Lambda' : (\mathit{mdf}, \mathit{stp}),$$

where $\Lambda, \Lambda' : \mathit{Var} \rightharpoonup \{\mathit{low}, \mathit{high}\}$ are partial environments (like in the type system for SIFUM-security), c is a command, and $\mathit{mdf}, \mathit{stp} \in \{\mathit{low}, \mathit{high}\}$ (like in the type system for FSI-security). The interpretation of the judgment is derived directly from the interpretations of the judgments $\vdash c : (\mathit{mdf}, \mathit{stp})$ and $\vdash \Lambda\ \{c\}\ \Lambda'$. (I.e., $\mathit{dom}(\Lambda)$ is a lower bound on the set of low variables with no-read and high variables with no-write assumption after the execution of c if $\mathit{dom}(\Lambda)$ is such a lower bound before the execution of c, $\Lambda'\langle x\rangle$ is an upper bound on the security level of x after the execution of c if $\Lambda\langle\cdot\rangle$ gives such upper bounds before the execution of c, mdf is a lower bound on the security level of variables modified by c, and stp is an upper bound on the security level of variables on which the number of execution steps of c depends.)

The derivation rules for the judgments are displayed in Figure 5.1. The rules are obtained by merging the conditions of the respective typing rules from the type systems for FSI-security in Section 3.4 and for SIFUM-security in Section 4.5.

Note that in rule [IF] we do not use a semantic side condition for conditionals with a high control condition like in the security type system for SIFUM-security. The reason

is the following: In the soundness proof of the security type system for FSI-security we exploit that the typing rules ensure that after executing a high control condition a typable program no longer assigns to low variables. We exploit this also in the soundness proof of the type system for FSIFUM-security. The semantic side condition from the type system for SIFUM-security, however, permits assignments to low variables within the branches of a conditional with a high guard. A consequence of removing the semantic side condition is that some programs that are typable in the type system for SIFUM-security are not typable in the type system for FSIFUM-security (namely such programs that assign to low variables after a high guard, and that are hence only typable in the type system for SIFUM-security using the semantic side condition).

The type system for commands is sound with respect to FSIFUM-security.

Theorem 5.6. Assume that $\vdash \Lambda \{c\} \Lambda' : (mdf, stp)$ is derivable, and let mds be a mode state consistent with Λ. Moreover, let mem_1, mem_2 be memories such that $mem_1(x) = mem_2(x)$ for all $x \in Var$ with $\Lambda\langle x\rangle = low$. Then $\langle c, mds, mem_1\rangle \sim_{lmm} \langle c, mds, mem_2\rangle$.$\Diamond$

The proof (see Appendix A.9) is along the same lines as the proofs for Theorems 3.15 and 4.7, constructing a bisimulation relation that relates typable commands and showing that this relation is a low bisimulation modulo low matching and modes.

Typing judgments for concurrent programs have the form $\vdash thr$. The single derivation rule (also displayed in Figure 5.1) is derived directly from the corresponding derivation rule in the security type system for SIFUM-security.

Theorem 5.7. Assume that obs is confined to L and that $\mathcal{S}$ is a robust scheduler. If thr is terminating and $\vdash thr$ is derivable with rule [$\mathsf{PAR}_{\mathcal{S}}$] then thr is $\mathcal{S}$-noninterferent. $\Diamond$

The proof (see Appendix A.9) is along the same lines as the proof of the corresponding Theorem for SIFUM-security, Theorem 4.8.

Example 5.3. Reconsider the command from Example 5.2. It is typable with the rules [ANNO], [ASS_2], [IF], [SKIP], and [SEQ]. Hence, it is FSIFUM-secure by Theorem 5.6. $\Diamond$

Moreover, it can be easily seen that the security type system for FSIFUM-security subsumes the security type system for FSI-security from Section 3.4 (assuming that one does not make any assumptions and guarantees that might not be sound), and that it subsumes the security type system for SIFUM-security from Section 4.5 if one removes the semantic side condition in rule [IF] of that type system.

5.4 Summary

In this chapter, we have integrated our solutions from Chapters 3 and 4, resulting in a novel compositional and scheduler-independent information flow property, FSIFUM-security. FSIFUM-security subsumes FSI-security and SIFUM-security. Moreover, we integrated the two corresponding security type systems, resulting in a provably sound security type system for FSIFUM-security. The results illustrate that the benefits of FSI-security and SIFUM-security can be combined in a single information flow property.

The integration was rather straightforward, which was due to using the Per-approach in the definitions of both FSI-security and of SIFUM-security. Also the proofs for the compositionality, scheduler-independence and type system soundness results could be adopted directly from the existing proofs for FSI-security and SIFUM-security. This gives hope that other properties based on the Per-approach (like properties supporting control of declassification as, e.g., in [MR07]) could also be integrated.

$$[\mathsf{EXP}]\ \frac{}{\Gamma \vdash e : \bigsqcup_{x \in vars(e)} \Gamma(x)}$$

$$[\mathsf{STOP}]\ \frac{}{\vdash \Lambda\ \{stop\}\ \Lambda : (high, low)} \qquad [\mathsf{SKIP}]\ \frac{}{\vdash \Lambda\ \{\mathsf{skip}\}\ \Lambda : (high, low)}$$

$$[\mathsf{ASS}_1]\ \frac{x \notin dom(\Lambda) \qquad \Lambda\langle\cdot\rangle \vdash e : d \qquad d \sqsubseteq dma(x)}{\vdash \Lambda\ \{x{:=}e\}\ \Lambda : (dma(x), low)}$$

$$[\mathsf{ASS}_2]\ \frac{x \in dom(\Lambda) \qquad \Lambda\langle\cdot\rangle \vdash e : d}{\vdash \Lambda\ \{x{:=}e\}\ \Lambda[x \mapsto d] : (dma(x), low)}$$

$$[\mathsf{IF}]\ \frac{\begin{array}{cc}\vdash \Lambda\ \{c_1\}\ \Lambda' : (mdf, stp) & \vdash \Lambda\ \{c_2\}\ \Lambda' : (mdf, stp) \\ \Lambda\langle\cdot\rangle \vdash e : d & d \sqsubseteq mdf\end{array}}{\vdash \Lambda\ \{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\ \Lambda' : (mdf, stp \sqcup d)}$$

$$[\mathsf{WHILE}]\ \frac{\Lambda\langle\cdot\rangle \vdash e : d \qquad \vdash \Lambda\ \{c\}\ \Lambda : (mdf, stp) \qquad stp \sqcup d \sqsubseteq mdf}{\vdash \Lambda\ \{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\}\ \Lambda : (mdf, stp \sqcup d)}$$

$$[\mathsf{SEQ}]\ \frac{\vdash \Lambda\ \{c_1\}\ \Lambda' : (mdf_1, stp_1) \qquad \vdash \Lambda'\ \{c_2\}\ \Lambda'' : (mdf_2, stp_2) \qquad stp_1 \sqsubseteq mdf_2}{\vdash \Lambda\ \{c_1; c_2\}\ \Lambda'' : (mdf_1 \sqcap mdf_2, stp_1 \sqcup stp_2)}$$

$$[\mathsf{SPAWN}]\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, \sharp(thr)\} : \vdash \Lambda_0\ \{thr[i]\}\ \Lambda_i : (mdf, stp_i) \\ \forall x \in Var : \Lambda\langle x\rangle \sqsubseteq dma(x)\end{array}}{\vdash \Lambda\ \{\mathsf{spawn}(thr)\}\ \Lambda : (mdf, low)}$$

$$[\mathsf{ANNO}]\ \frac{\begin{array}{c}\Lambda' = adjust\text{-}pe(\Lambda, ann) \qquad \vdash \Lambda'\ \{c\}\ \Lambda'' : (mdf, stp) \\ \forall x \in Var : \Lambda\langle x\rangle \sqsubseteq \Lambda'\langle x\rangle\end{array}}{\vdash \Lambda\ \{/\!\!/ ann /\!\!/\ c\}\ \Lambda'' : (mdf, stp)}$$

$$[\mathsf{SUB}]\ \frac{\begin{array}{cc}\vdash \Lambda_1\ \{c\}\ \Lambda_1' : (mdf_1, stp_1) & \Lambda_2 \sqsubseteq \Lambda_1 \qquad \Lambda_1' \sqsubseteq \Lambda_2' \\ mdf_2 \sqsubseteq mdf_1 & stp_1 \sqsubseteq stp_2\end{array}}{\vdash \Lambda_2\ \{c\}\ \Lambda_2' : (mdf_2, stp_2)}$$

$$[\mathsf{PAR}_{\mathcal{S}}]\ \frac{\begin{array}{c}\forall i \in \{1, \ldots, \sharp(thr)\} : \vdash \Lambda_i\ \{thr[i]\}\ \Lambda_i' : (mdf_i, stp_i) \\ \forall i \in \{1, \ldots, \sharp(thr)\} : \forall x \in Var : dma(x) \sqsubseteq \Lambda_i\langle x\rangle \\ \forall i \in \{1, \ldots, \sharp(thr)\} :\ mdss[i] \text{ is consistent with } \Lambda_i \\ (thr, mdss) \text{ has sound modes for } \mathcal{S}\end{array}}{\vdash thr}$$

Figure 5.1: Security type system for FSIFUM-security

CHAPTER 6

A Bridge to PDG-based Information Flow Analysis

6.1 Introduction

In Chapters 3–5 we developed the information flow analyses for the novel information flow properties using security type systems. Security type systems are so far probably the most popular approach to language-based information flow analysis. Starting with Volpano, Smith, and Irvine in the nineties [VSI96], type-based information flow analyses were developed for programs containing various language features comprising procedures (e.g., [VS97, HS11]), concurrency (e.g., [SV98, MSS11]), and objects (e.g., [Mye99, BN02]). Moreover, security type systems were proposed for certifying a variety of information flow properties, including timing-sensitive properties (e.g., [SS00]), timing-insensitive properties (e.g., [BC02] and the type systems from this thesis), and properties supporting declassification (e.g., [MS04, MR07]).

Other information flow analysis techniques were also investigated. For instance, Hsieh, Unger, and Mata-Toledo already proposed at the beginning of the nineties to use program dependence graphs (PDGs) in an information flow analysis [HUM92]. A PDG [FOW87] is a graph-based representation of dependencies in a program. PDGs have been extended over time to programs with various languages features including procedures (e.g., [HRB90, RHSR94]), concurrency (e.g., [Che93, Kri98]), and objects (e.g., [MMKM94, HS09]). The idea of PDG-based information flow analysis recently received renewed attention, resulting in, e.g., a PDG-based information flow analysis for object-oriented programs and PDG-based analysis that supports declassification [HS09, Ham10].

It has been conjectured that using PDGs in an information flow analysis leads to a better precision than what can be achieved with security type systems. However, the connection between type-based and PDG-based security analyses had not been studied in detail so far. In this chapter, we investigate the relationship between type-based and PDG-based information flow analysis. The goal of the investigation is to (a) gain a better understanding of the relationship of these analysis techniques and (b) to investigate

possibilities to exploit the relationship in the development of novel information flow analyses, in particular for multi-threaded programs.

As a first step, we introduce a formal connection between a type-based and a PDG-based information flow analysis for sequential programs. To be able to establish a precise relation we consider two analyses that are fully formalized, namely a type-based analysis from [HS06] and a PDG-based analysis from [WLS09]. We exploit the connection for showing that the two analyses have not only roughly the same precision, but have exactly the same precision. Secondly, we investigate how the formal connection can be used for transferring ideas from PDG-based to type-based information flow analysis and vice versa. We illustrate this possibility in both directions: Firstly, we transfer the concept of summary edges to type systems. This leads to a context-sensitive type-based analysis. We show that the security type system from [HS11] captures the intuition of summary edges precisely. Secondly, we derive a novel compositional and scheduler-independent PDG-based information flow analysis that is suitable for multi-threaded programs. To this end, we exploit the ideas developed in Chapter 4 of this thesis. The analysis is the first PDG-based information flow analysis for multi- threaded programs that is shown to be compositional, and, thereby, enables a modular information flow analysis. Moreover, it is the first such analysis that supports programs with nondeterministic public output.

Overview. We introduce a type-based and a PDG-based information flow analysis for sequential programs in Sections 6.2 and 6.3, and establish a formal relation between these analyses in Section 6.4. In Section 6.5 we illustrate the transfer of the concept of summary edges from PDG-based to type-based analysis, and in Section 6.6 we develop a PDG-based analysis for multi-threaded programs by transferring assumption-guarantee based reasoning from type systems to PDGs. Section 6.7 summarizes and discusses the relation to other PDG-based analyses for concurrent programs.

6.2 Type-based Analysis for Sequential Programs

As a starting point, we consider the security type system for sequential programs from Hunt and Sands [HS06]. This type system provides, in contrast to many other type-based information flow analyses for sequential programs, a flow-sensitive analysis.[1] The type system permits to type sequential programs written in the programming language from Section 2.3 excluding the primitive for spawning new threads. We denote the corresponding set of commands with Com_{seq}.

Typing judgments for expressions are like in Section 4.5.2, except that an arbitrary security lattice $\mathcal{D}$ is used instead of the two-level security lattice $\{low, high\}$: The judgments are of the form $\Gamma \vdash e : d$ where $\Gamma : Var \to \mathcal{D}$, and the interpretation of the judgment is that d is an upper bound on the security level of the value of e, given that $\Gamma(x)$ is an upper bound on the security level of the value of x for all $x \in Var$.

Typing judgments for commands have the form $pc \vdash \ \Gamma \, \{c\} \, \Gamma'$, where $pc \in \mathcal{D}$ is a security domain and $c \in Com_{seq}$ is a command. Moreover, $\Gamma, \Gamma' : Var \to \mathcal{D}$ are environments that map variables to security domains. The interpretation of the judgment is as follows: Assume that pc is an upper bound on the security level of all information that determines whether command c is executed or not and that, for each $x \in Var$, $\Gamma(x)$ is an upper bound on the security level of the value of x before c is executed. Then, for each $y \in Var$, $\Gamma'(y)$ is an upper bound on the security level of the value of y after the execution of c.

[1] We discuss flow-sensitivity in more detail in Section 4.2.

$$[\mathsf{EXP}]\ \frac{}{\Gamma \vdash e : \bigsqcup_{x \in vars(e)} \Gamma(x)} \qquad [\mathsf{SKIP}]\ \frac{}{pc \vdash\ \Gamma\ \{\mathsf{skip}\}\ \Gamma}$$

$$[\mathsf{ASS}]\ \frac{\Gamma \vdash e : d}{pc \vdash\ \Gamma\ \{x{:=}e\}\ \Gamma[x \mapsto pc \sqcup d]} \qquad [\mathsf{SEQ}]\ \frac{pc \vdash\ \Gamma\ \{c_1\}\ \Gamma' \qquad pc \vdash\ \Gamma'\ \{c_2\}\ \Gamma''}{pc \vdash\ \Gamma\ \{c_1; c_2\}\ \Gamma''}$$

$$[\mathsf{IF}]\ \frac{\Gamma \vdash e : d \qquad pc \sqcup d \vdash\ \Gamma\ \{c_1\}\ \Gamma_1' \qquad pc \sqcup d \vdash\ \Gamma\ \{c_2\}\ \Gamma_2'}{pc \vdash\ \Gamma\ \{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\ \Gamma_1' \sqcup \Gamma_2'}$$

$$[\mathsf{WHILE}]\ \frac{\begin{array}{ccc} \Gamma_0' = \Gamma & \Gamma_0' \vdash e : d_0 & pc \sqcup d_0 \vdash\ \Gamma_0'\ \{c\}\ \Gamma_0'' \\ \Gamma_1' = \Gamma_0'' \sqcup \Gamma & \Gamma_1' \vdash e : d_1 & pc \sqcup d_1 \vdash\ \Gamma_1'\ \{c\}\ \Gamma_1'' \\ & \vdots & \\ \Gamma_k' = \Gamma_{k-1}'' \sqcup \Gamma & \Gamma_k' \vdash e : d_k & pc \sqcup d_k \vdash\ \Gamma_k'\ \{c\}\ \Gamma_k'' \\ \Gamma_{k+1}' = \Gamma_k'' \sqcup \Gamma & \Gamma_{k+1}' = \Gamma_k' & k \in \mathbb{N} \end{array}}{pc \vdash\ \Gamma\ \{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\}\ \Gamma_k'}$$

Figure 6.1: Type system from [HS06]

The typing rules from [HS06] are displayed in Figure 6.1, where the least upper bound operator on $\mathcal{D}$ is extended to environments by $(\Gamma \sqcup \Gamma')(x) := \Gamma(x) \sqcup \Gamma'(x)$. Unlike in the security type systems in the previous chapters, there are no restrictions on conditionals and loops with high control conditions, because there is no danger of internal timing leaks in sequential programs. Moreover, to capture implicit information flows the security level of control conditions is propagated via pc to the variables which are assigned to under the control condition. The typing rule for loops is somewhat more complex than the other rules, because it determines the environment after the loop by a fixed point computation. Thereby, the typing rules ensure that for any given c, Γ, and pc there is an environment Γ' such that $pc \vdash\ \Gamma\ \{c\}\ \Gamma'$ is derivable. Moreover, this environment is uniquely determined by c, Γ, and pc [HS06, Theorem 4.1].

For the corresponding type-based information flow analysis, we instantiate the security lattice with the two-level security lattice.

Definition 6.1. Let $c \in Com_{seq}$, $\Gamma(x) = dma(x)$ for all $x \in Var_{in}$, $\Gamma(x) = low$ for all $x \in Var \setminus Var_{in}$, and let Γ' be the unique environment such that $low \vdash\ \Gamma\ \{c\}\ \Gamma'$ is derivable. The command c is *accepted by the type-based analysis* if $\Gamma'(x) = low$ for all public output variables $x \in LO$. ◊

Example 6.1. Consider the command c specified by the code at the right, where $Var_{in} = \{y, z\}$ and $Var_{out} = \{x\}$. Assume that $dma(x) = dma(y) = low$ and $dma(z) = high$. Let $\Gamma(x) = dma(x)$ for all $x \in Var$. Then the judgment $low \vdash\ \Gamma\ \{c\}\ \Gamma'$ is derivable if and only if $\Gamma'(x) = \Gamma'(y) = \Gamma'(z) = high$ and $\Gamma'(x') = \Gamma(x')$ for all other $x' \in Var$. Since $x \in LO$ and $\Gamma'(x) = high$ command c is not accepted by the type-based analysis. ◊

```
1.  if (z < 0) then
2.    while (y > 0) do
3.      y:=y + z od else
4.    skip fi;
5.  x:=y
```

Theorem 6.1. If a command $c \in Com_{seq}$ is accepted by the type-based analysis then c is $\mathcal{S}$-noninterferent for any scheduler $\mathcal{S}$.[2] ◊

Proof. The theorem follows from Theorem 3.3 in [HS06]. □

6.3 PDG-based Analysis for Sequential Programs

PDG-based information flow analyses, firstly proposed in [HUM92], exploit that the absence of certain paths in a *program dependence graph* (PDG) [FOW87] is a sufficient condition for the information flow security of a program. In this section, we recall the PDG-based analysis from [WLS09] for the language Com_{seq}.

The PDG-based analysis is defined based on the control flow graph (CFG) of a program. To define the PDG-based analysis, we firstly introduce the CFG of a program and subsequently define the PDG of a program based on its CFG.

6.3.1 Control Flow Graphs

Definition 6.2. A *directed graph* is a pair (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges.

A *path p from node n_1 to node n_k* is a non-empty sequence of nodes $\langle n_1, \ldots, n_k \rangle \in N^+$ where $(n_i, n_{i+1}) \in E$ for all $i \in \{1, \ldots, k-1\}$. We say that a node n is *on the path* $\langle n_1, \ldots, n_k \rangle$ if $n = n_i$ for some $i \in \{1, \ldots, k\}$. ◊

Definition 6.3. A *control flow graph with def and use sets* is a tuple

$$(N, E, def, use)$$

where (N, E) is a directed graph, N contains two distinguished nodes *start* and *stop*, and $def, use : N \to \mathcal{P}(Var)$ are functions. ◊

Nodes *start* and *stop* represent program start and termination, respectively, and the remaining nodes represent program statements and control conditions. An edge $(n, n') \in E$ models that n' might immediately follow n in a program execution. Finally, the functions *def* and *use* define the *def set* and *use set* for each node $n \in N$, where the sets $def(n)$ and $use(n)$ contain all variables that are defined and used, respectively, at n. In the following, we simply write "CFG" instead of "CFG with def and use sets."

We provide examples of CFGs after defining the construction of CFGs for programs in the sequential fragment of the programming language from Section 2.3, i.e., for commands in the set Com_{seq}. Our construction of CFGs follows Wasserrab and Lochbihler [WL08]. As usual, the i^{th} statement respectively control condition is represented by a node $i \in \mathbb{N}$. To define the set of nodes of the control flow graph for a command, we define the number of statements and control conditions of a command, as well as the ith statement or control condition of a command, abusing the corresponding notation for lists.

Definition 6.4. For $c \in Com_{seq}$ we define $\sharp(c)$ recursively by $\sharp(\mathsf{skip}) = 1$, $\sharp(x{:=}e) = 1$, $\sharp(c_1; c_2) = \sharp(c_1) + \sharp(c_2)$, $\sharp(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}) = 1 + \sharp(c_1) + \sharp(c_2)$, and $\sharp(\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}) = 1 + \sharp(c)$. ◊

The number $\sharp(c)$ is the number of statements and control conditions of the command c.

[2] Note that the choice of the scheduler is irrelevant for sequential programs.

Definition 6.5. For $c \in Com_{seq}$ and $1 \leq i \leq \sharp(c)$ we denote with $c[i]$ the i^{th} statement or control condition in c, which is recursively defined as follows:
- If $c = \mathsf{skip}$ or $c = x{:=}e$ then $c[1] = c$.
- If $c = c_1; c_2$ then $c[i] = c_1[i]$ for $1 \leq i \leq \sharp(c_1)$ and $c[i] = c_2[i{-}\sharp(c_1)]$ for $\sharp(c_1) < i \leq \sharp(c)$.
- If $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ then $c[1] = e$, $c[i] = c_1[i{-}1]$ for $1 < i \leq 1 + \sharp(c_1)$, and $c[i] = c_2[i{-}1{-}\sharp(c_1)]$ for $1 + \sharp(c_1) < i \leq \sharp(c)$.
- If $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$ then $c[1] = e$ and $c[i] = c_1[i{-}1]$ for $1 < i \leq \sharp(c)$. ◊

Note that $c[i]$ is either an expression, an assignment, or a skip-statement. We now construct the CFG for a command.

Definition 6.6. For $c \in Com_{seq}$, we define $N_c = \{1, \ldots, \sharp(c)\} \cup \{start, stop\}$. ◊

We define the operator $\ominus : N_c \times \mathbb{Z} \to \mathbb{Z} \cup \{start, stop\}$ by $n \ominus z = n - z$ if $n \in \mathbb{N}$ and $n \ominus z = n$ if $n \in \{start, stop\}$.

Definition 6.7. For $c \in Com_{seq}$ the set $E_c \subseteq N_c \times N_c$ is defined recursively as follows:
- $E_{\mathsf{skip}} = E_{x:=e} =$
  $\{(start, stop), (start, 1), (1, stop)\}$,
- $E_{c_1;c_2} =$
  $\{(start, stop)\} \cup$
  $\{(n, n') \mid (n, n') \in E_{c_1} \wedge n' \neq stop\} \cup$
  $\{(n, n') \mid (n \ominus \sharp(c_1), n' \ominus \sharp(c_1)) \in E_{c_2} \wedge n \neq start\} \cup$
  $\{(n, n') \mid (n, stop) \in E_{c_1} \wedge (start, n' \ominus \sharp(c_1)) \in E_{c_2} \wedge n \neq start \wedge n' \neq stop\}$,
- $E_{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} =$
  $\{(start, stop), (start, 1)\} \cup$
  $\{(1, n') \mid (start, n' \ominus 1) \in E_{c_1} \wedge n' \neq stop\} \cup$
  $\{(1, n') \mid (start, n' \ominus (1{+}\sharp(c_1))) \in E_{c_2} \wedge n' \neq stop\} \cup$
  $\{(n, n') \mid (n \ominus 1, n' \ominus 1) \in E_{c_1} \wedge n \neq start\} \cup$
  $\{(n, n') \mid (n \ominus (1{+}\sharp(c_1)), n' \ominus (1{+}\sharp(c_1))) \in E_{c_2} \wedge n \neq start\}$, and
- $E_{\mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}} =$
  $\{(start, stop), (start, 1), (1, stop)\} \cup$
  $\{(n, n') \mid (n \ominus 1, n' \ominus 1) \in E_c \wedge n \neq start \wedge n' \neq stop\} \cup$
  $\{(1, n') \mid (start, n' \ominus 1) \in E_c \wedge n' \neq stop\} \cup$
  $\{(n, 1) \mid (n \ominus 1, stop) \in E_c \wedge n \neq start\}$.

◊

Definition 6.8. For $c \in Com_{seq}$ we define functions $def_c, use_c : N_c \to \mathcal{P}(Var)$ by
- $def_c(n) = \{x\}$ if $n \in \{1, \ldots, \sharp(c)\}$ and $c[n] = x{:=}e$, and $def_c(n) = \{\}$ otherwise, and
- $use_c(n) = vars(e)$ if $n \in \{1, \ldots, \sharp(c)\}$ and $c[n] = x{:=}e$ or $c[n] = e$, and $use_c(n) = \{\}$ otherwise. ◊

Definition 6.9. Let $c \in Com_{seq}$. We define the *control flow graph of* c by

$$CFG_c = (N_c, E_c, def_c, use_c). \qquad \text{◊}$$

Note that, by definition, an edge from *start* to *stop* is contained in E_c. This edge models, as usual, the possibility that c is not executed.

We augment CFGs by two nodes *in* and *out* to capture the program's interaction with its environment. Two sets of variables $I, O \subseteq Var$ specify which variables may be initialized by the environment before the program executes and which variables may be read by the environment after the execution. This results in the following variant of CFGs:

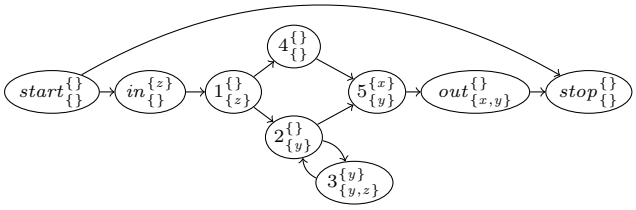


Figure 6.2: The CFG of the command in Example 6.1

Definition 6.10. Let $CFG = (N, E, def, use)$ be a control flow graph and $I, O \subseteq Var$. Then $CFG^{I,O}$ is defined as (N', E', def', use') where

- $N' = N \cup \{in, out\}$,
- $E' = \{(start, stop), (start, in), (out, stop)\} \cup \{(in, n') \mid (start, n') \in E \wedge n' \neq stop\} \cup \{(n, out) \mid (n, stop) \in E \wedge n \neq start\} \cup \{(n, n') \in E \mid n \notin \{start, stop\} \wedge n' \notin \{start, stop\}\}$,
- $def'(in) = I$, $use'(out) = O$, and $use'(in) = def'(out) = \{\}$, and
- $def'(n) = def(n)$ and $use'(n) = use(n)$ for $n \in N$. ◊

Definitions 6.9 and 6.10 both specialize the general notion of control flow graphs (see Definition 6.3). In the remainder of this thesis, we use the abbreviation CFG for arbitrary control flow graphs (including those that satisfy Definition 6.9 or 6.10).

We use a graphical representation for displaying CFGs where we depict nodes with ellipses and edges with solid arrows. For each node n we label the corresponding ellipse with n^X_Y where $X = def(n)$ and $Y = use(n)$.

Example 6.2. Command c from Example 6.1 contains three statements and two control conditions (i.e., $\sharp(c) = 5$). Hence, $N_c = \{1, \ldots, 5, start, stop\}$. Nodes 1–5 represent the statements and control conditions in Lines 1–5 of the program, respectively. For $I = \{z\}$ and $O = \{x, y\}$ (i.e., $I = H$ and $O = L$ for the domain assignment from Example 6.1) the control flow graph $CFG_c^{I,O}$ is displayed in Figure 6.2. ◊

6.3.2 PDG-based Analysis

PDGs are directed graphs that represent dependencies in imperative programs [FOW87]. We construct the PDG from a CFG based on the following notions of data dependency and control dependency.

Definition 6.11. Let (N, E, def, use) be a CFG and $n, n' \in N$. If $x \in def(n)$ we say that *the definition of x at n reaches n'* if there is a path p from n to n' such that $x \notin def(n'')$ for every node n'' on p with $n'' \neq n$ and $n'' \neq n'$.

Node n' is *data dependent* on node n if there exists $x \in Var$ such that $x \in def(n)$, $x \in use(n')$, and the definition of x at n reaches n'. ◊

Intuitively, a node n' is data dependent on a node n if n' uses a variable that was written at n and has not been overwritten until being used at n'.

Example 6.3. Consider the CFG depicted in Figure 6.2. The definition of y at Node 3 reaches Node 5 because $\langle 3, 2, 5\rangle$ is a path in the CFG and $y \notin def(2)$. In addition,

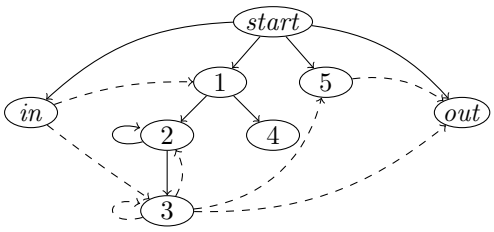


Figure 6.3: PDG of CFG from Figure 6.2

$y \in def(3)$ and $y \in use(5)$, and, hence, Node 5 is data dependent on Node 3. Moreover, Node 2 is also data dependent on Node 3 and Node 3 is data dependent on itself. ◊

Definition 6.12. Let (N, E, def, use) be a CFG. Node n' *postdominates* node n if $n \neq n'$ and every path from n to *stop* contains n'.

Node n' is *control dependent* on node n if n' does not postdominate n and there is a path p from n to n' such that n' postdominates all nodes n'' on p with $n'' \notin \{n, n'\}$. ◊

Intuitively, a node n' is control dependent on a node n if it depends on n whether n' is executed or not.

Example 6.4. Consider the CFG depicted in Figure 6.2. Node 5 postdominates Node 1 because Node 5 is on every path from Node 1 to Node *stop*. Hence, Node 5 is not control dependent on Node 1. Nodes 2, 3, and 4 do not postdominate Node 1. Since Node 3 does not postdominate Node 2 and all paths from Node 1 to Node 3 contain Node 2, Node 3 is not control dependent on Node 1. However, Node 3 is control dependent on Node 2. Moreover, Nodes 2 and 4 are control dependent on Node 1, because $\langle 1, 2\rangle$ and $\langle 1, 4\rangle$ are paths in the CFG. Finally, Node 2 is control dependent on itself, because Node 2 postdominates Node 3 and $\langle 2, 3, 2\rangle$ is a path in the CFG. I.e., a node might be control dependent on itself. ◊

Remark 6.1. The definition of control dependency captures direct control dependency (like the dependency of the loop body on the loop guard in the above example), but not the indirect dependency of the loop body on the guard of the conditional that encloses the loop. Such indirect dependencies are reflected by the notion of paths in a PDG. ◊

Definition 6.13. Let $CFG = (N, E, def, use)$ be a control flow graph. The directed graph (N', E') is the *PDG of CFG* if $N' = N$ and $(n, n') \in E'$ if and only if n' is data dependent or control dependent on n in CFG. We denote this graph with $PDG(CFG)$. ◊

We use the usual graphical representation for displaying PDGs in the remainder of this thesis, depicting Node n with an ellipse labeled with n, edges that reflect control dependency with solid arrows, and edges that reflect data dependency with dashed arrows. Moreover, we do not display nodes that have neither in- nor outgoing edges.

Example 6.5. Consider the CFG from Figure 6.2. The PDG constructed from this CFG is displayed in Figure 6.3. (Node *stop* is not displayed because it has neither in- nor outgoing edges.) ◊

The PDG-based information flow analysis from [WLS09] for a command $c \in Com_{seq}$ is based on the PDG of $CFG_c^{HI,LO}$ (cf. Definition 6.10), i.e., the CFG where node *in* represents writing high input variables before program execution and node *out* represents reading low output variables after program execution.

Definition 6.14. The command $c \in Com_{seq}$ is *accepted by the PDG-based analysis* if and only if there is no path from *in* to *out* in $PDG(CFG_c^{HI,LO})$. ◊

Example 6.6. Consider command c and domain assignment from Example 6.1. Assume that $Var_{in} = \{y, z\}$ and $Var_{out} = \{x\}$. The program dependence graph $PDG(CFG_c^{HI,LO})$ is depicted in Figure 6.3. It contains a path from Node *in* to Node *out*, (e.g., the path $\langle in, 3, out\rangle$). In consequence, c is not accepted by the PDG-based analysis. ◊

Theorem 6.2. If a command $c \in Com_{seq}$ is accepted by the PDG-based analysis then c is $\mathcal{S}$-noninterferent for any scheduler.[3] ◊

Proof. The theorem follows from [WLS09, Theorem 8]. □

6.4 Relating Type-based and PDG-based Analysis

While both the type-based analysis from Section 6.2 and the PDG-based analysis from Section 6.3 are sound, both analyses are also incomplete. I.e., for each analysis there are programs that satisfy $\mathcal{S}$-noninterference, but that are not accepted by the analysis. In fact, developing a complete analysis is impossible, because $\mathcal{S}$-noninterference is undecidable (this can be proved in a standard way by showing that the decidability of the property would imply the decidability of the halting problem [SM03]). This raises the question if any of the two analyses is more precise than the other. In this section, we answer this question. To this end, we establish the following relation between the two analyses.

Theorem 6.3. Let $c \in Com_{seq}$, $y \in Var$, and Γ be an environment. Let Γ' be the unique environment such that $low \vdash \Gamma\ \{c\}\ \Gamma'$ is derivable in the type system from Section 6.2. Moreover, let X be the set of all $x \in Var$ such that there exists a path from *in* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$. Then $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$. ◊

Proof Sketch. In the proof of the theorem (see Appendix A.10) we establish a more general theorem where Γ' is such that $pc \vdash \Gamma\ \{c\}\ \Gamma'$ is derivable for some arbitrary security domain pc. In this case, $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$ if there is no path from *start* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ except the path $\langle start, out\rangle$, and $\Gamma'(y) = pc \sqcup \left(\bigsqcup_{x \in X} \Gamma(x)\right)$ otherwise. The proof is by induction on the structure of c. □

Theorem 6.3 is the key to establishing the following theorem that relates the precision of the sound type-based analysis from Section 6.2 with the precision of the sound PDG-based analysis from Section 6.3.

Theorem 6.4. A command $c \in Com_{seq}$ is accepted by the type-based analysis for sequential programs from Section 6.2 if and only if it is accepted by the PDG-based analysis for sequential programs from Section 6.3. ◊

[3] Note that, like for the type-based analysis for sequential programs from Section 6.2, the choice of the scheduler is irrelevant for sequential programs.

Proof. Our proof is by contraposition.

Let $\Gamma(x) = dma(x)$ for all $x \in Var_{in}$, $\Gamma(x) = low$ for all $x \in Var \setminus Var_{in}$, and let Γ' be the unique environment such that $low \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable in the type system from Section 6.2.

Assume firstly that c is not accepted by the type-based analysis, i.e., $\Gamma'(y) = high$ for some $y \in LO$. Then, by Theorem 6.3, there exists a variable x with $\Gamma(x) = high$ and a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. By the definition of Γ it follows that $x \in HI$. Hence, this path is also a path in $PDG(CFG_c^{HI,LO})$. Thus, c is not accepted by the PDG-based analysis.

Assume now that c is not accepted by the PDG-based analysis, i.e., there is a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{HI,LO})$. But then there exist variables $x \in HI$ and $y \in LO$ such that there is a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. Hence, by Theorem 6.3, $\Gamma'(y) = high$. Thus, c is not accepted by the type-based analysis. □

Theorem 6.4 demonstrates that a type-based information flow analysis, despite its conceptual simplicity, can have the same precision as a PDG-based information flow analysis. Moreover, since both analyses are equally precise, one can select between them based on other features of the analyses. For instance, if one needs a compositional analysis one would select the type-based analysis. Moreover, if a program is not accepted by the analyses one could use the PDG-based analysis to localize the source of potential information leakage by inspecting the path in the PDG that leads to the rejection of the program. This path identifies the program statements and control conditions that might cause information leakage.

Theorem 6.4 not only clarifies the connection between type-based and PDG-based information flow analyses, but also provides a bridge to transfer concepts from one tradition of information flow analysis to the other. In the two subsequent sections, we illustrate how concepts can be transferred across this bridge in both directions, in particular, for deriving a PDG-based analysis for multi-threaded programs (in Section 6.6).

6.5 Context-sensitive Interprocedural Analysis

For interprocedural information flow analysis, PDGs that are extended with *summary edges* [HRB90] at nodes representing procedure calls permit a *context-sensitive* PDG-based analysis. Context-sensitive analyses take the calling context of procedure calls into account to improve the precision of the analysis.

Example 6.7. Consider the program

$$\mathsf{temp}{:=}h; \mathsf{swap}(\mathsf{temp}, h'); \mathsf{temp}{:=}l; \mathsf{swap}(\mathsf{temp}, l')$$

where h and h' are high variables, l, l', and temp are low variables, and procedure swap swaps the values of its arguments. A context-sensitive analysis can detect that the program does not leak secret information into the final values of l, l', and temp. A context-sensitive analysis is needed to detect this, because temp must be treated as secret at the first call site, but as public at the second call site. ◊

Using PDGs with summary edges for context-sensitive information flow analysis is proposed in [HS09]. A formalization of and soundness results for such an analysis are provided in [Was09, Was10] (following the formalization of summary edges in [RHSR94]). In this section, we investigate how the idea underlying the notion of summary edges can

be adopted to a type-based analysis. As we will show, there already exists a concept in type-based analyses that corresponds to summary edges. The main theorem of this section is that the resulting context-sensitive security type system accepts the same programs as the PDG-based analysis with summary edges.

6.5.1 Programs and Control Flow Graphs with Procedures

We consider programs with call-by-value, return-by-value procedures. Let P be a finite set of procedure names. The signature for procedure name $\mathsf{p} \in P$ is the pair $(l_\mathsf{p}, m_\mathsf{p}) \in \mathbb{N} \times \mathbb{N}$ where l_p is the number of in parameters and m_p the number of out parameters of procedure p. A procedure p is defined by the term

$$((x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}), (y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}), c_\mathsf{p}),$$

where the variables $x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}$ are the *formal in parameters* and the variables $y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}$ are the *formal out parameters* of p, and the command c_p is the body of p. Like in the previous sections, we consider sequential programs modeled by commands in Com_{seq}. Here, in addition, commands (including the bodies of the procedures) may contain procedure calls, and we extend the grammar for commands from Section 2.3 by

$$c \quad ::= \quad \ldots \mid \mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}}),$$

where $\mathsf{p} \in P$, $e_1, \ldots, e_{l_\mathsf{p}} \in Exp$, and $z_1, \ldots, z_{m_\mathsf{p}} \in Var$ are distinct variables. The expressions e_i are the *actual in parameters* of the call and the variables z_i its *actual out parameters*. We require that for each $\mathsf{p} \in P$ the command c_p, i.e., the body of procedure p, only contains variables in the set $\{x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}\} \cup \{y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}\}$. When procedure p is executed, the initial values of the variables $\{x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}\}$ are determined by the actual in parameters of the procedure call, and the variables $\{y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}\}$ have a default initial value. (A formal semantics for the language with procedures is defined in Appendix C.)

The following definition extends the construction of CFGs from Definitions 6.4–6.9 to commands with procedure calls.

Definition 6.15. For $c = \mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})$ we define $\sharp(c) = 1$ and $c[1] = c$. Moreover, we modify the definition of N_c as follows:

$$\begin{array}{rcl} N_c & = & \{start, stop\} \cup \{1, \ldots, \sharp(c)\} \\ & \cup & \{a\text{-}in_{n,i,\mathsf{p}} \mid c[n] = \mathsf{p}(\ldots) \wedge i \in \{1, \ldots, l_\mathsf{p}\}\} \\ & \cup & \{a\text{-}out_{n,i,\mathsf{p}} \mid c[n] = \mathsf{p}(\ldots) \wedge i \in \{1, \ldots, m_\mathsf{p}\}\}. \end{array}$$

We extend the operator $\ominus$ to the additional nodes by defining $a\text{-}in_{n,i,\mathsf{p}} \ominus k = a\text{-}in_{n-k,i,\mathsf{p}}$ and $a\text{-}out_{n,i,\mathsf{p}} \ominus k = a\text{-}out_{n-k,i,\mathsf{p}}$. Moreover, we extend the recursive definition of E_c to procedure calls as follows:

– $E_{\mathsf{p}(e_1,\ldots,e_{l_\mathsf{p}};z_1,\ldots,z_{m_\mathsf{p}})} =$
$\{(start, stop)\} \cup$
$\{(start, 1), (1, a\text{-}in_{1,1,\mathsf{p}}), (a\text{-}out_{1,m_\mathsf{p},\mathsf{p}}, stop)\} \cup$
$\{(a\text{-}in_{1,i,\mathsf{p}}, a\text{-}in_{1,i+1,\mathsf{p}}) \mid i \in \{1, \ldots, l_\mathsf{p} - 1\}\} \cup$
$\{(a\text{-}in_{1,l_\mathsf{p},\mathsf{p}}, a\text{-}out_{1,1,\mathsf{p}})\} \cup$
$\{(a\text{-}out_{1,i,\mathsf{p}}, a\text{-}out_{1,i+1,\mathsf{p}}) \mid i \in \{1, \ldots, m_\mathsf{p} - 1\}\}.$

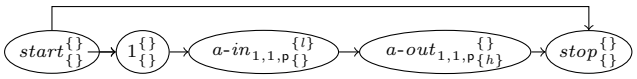
If $c[n] = \mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})$ we define

– $use_c(a\text{-}in_{n,i,\mathsf{p}}) = vars(e_i)$,

– $def_c(a\text{-}out_{n,i,\mathsf{p}}) = \{z_i\}$, and
– $def_c(n) = use_c(n) = def_c(a\text{-}in_{n,i,\mathsf{p}}) = use_c(a\text{-}out_{n,i,\mathsf{p}}) = \{\}$. ◊

Nodes $a\text{-}in_{n,i,\mathsf{p}}$ and $a\text{-}out_{n,i,\mathsf{p}}$ in the CFG of a command with procedure calls represent writing the value of the ith actual in parameter to the ith formal in parameter of procedure p before the execution of p and writing the value of the ith formal out parameter to the ith actual out parameter after the execution of p, respectively. The parameter n indicates the node in the CFG that represents the corresponding procedure call.

Example 6.8. Let addOne be a procedure with signature $(1, 1)$ that is defined by the term $((x), (y), y{:=}x + 1)$. I.e., procedure addOne adds 1 to its in parameter and stores the result in its out parameter. Let $c = \mathsf{addOne}(l; h)$. Then the graphical representation of CFG_c is as follows:



6.5.2 PDG-based Interprocedural Analysis

We define a variant of PDGs for programs with procedures, the *system dependence graphs* (SDGs) [HRB90], upon which we then define a PDG-based interprocedural information flow analysis. To define the SDG we use *procedure dependence graphs* that are constructed from PDGs by adding nodes representing formal in and formal out parameters, respectively.

Definition 6.16. The *procedure dependence graph* PDG_{p} for procedure p is the directed graph $(N_{\mathsf{p}}, E_{\mathsf{p}})$ where $N_{\mathsf{p}} = (N_{c_{\mathsf{p}}} \times \{\mathsf{p}\}) \cup \{f\text{-}in_1^{\mathsf{p}}, \ldots, f\text{-}in_{l_{\mathsf{p}}}^{\mathsf{p}}\} \cup \{f\text{-}out_1^{\mathsf{p}}, \ldots, f\text{-}out_{m_{\mathsf{p}}}^{\mathsf{p}}\}$ and $(n, n') \in E_{\mathsf{p}}$ if and only if one of the following conditions is satisfied:
– $n = (n'', \mathsf{p})$, $n' = (n''', \mathsf{p})$, and (n'', n''') is an edge in $PDG(CFG_{c_{\mathsf{p}}})$,
– $n = f\text{-}in_i^{\mathsf{p}}$ and $n' = (n'', \mathsf{p})$ where $n'' \in \{1, \ldots, \sharp(c_{\mathsf{p}})\}$ and n'' is data dependent on Node in in the control flow graph $CFG_{c_{\mathsf{p}}}^{\{x_i^{\mathsf{p}}\},\{\}}$,
– $n = (n'', \mathsf{p})$ where $n'' \in \{1, \ldots, \sharp(c_{\mathsf{p}})\}$, $n' = f\text{-}out_i^{\mathsf{p}}$, and Node out is data dependent on n'' in the control flow graph $CFG_{c_{\mathsf{p}}}^{\{\},\{y_i^{\mathsf{p}}\}}$, or
– $n = f\text{-}in_i^{\mathsf{p}}$, $n' = f\text{-}out_j^{\mathsf{p}}$, and Node out is data dependent on Node in in the control flow graph $CFG_{c_{\mathsf{p}}}^{\{x_i^{\mathsf{p}}\},\{y_j^{\mathsf{p}}\}}$. ◊

Nodes $f\text{-}in_i^{\mathsf{p}}$ and $f\text{-}out_i^{\mathsf{p}}$ represent writing the initial value of the procedure's ith formal in parameter before and reading the final value of the procedure's ith formal out parameter after the procedure execution, respectively. The edges defined in the first item in Definition 6.16 are simply carried over from the CFG of the procedure body, and model the dependencies within the procedure body. The edges defined in the other three items in Definition 6.16 model dependencies on the procedure's formal in parameters as well as dependencies of the procedure's formal out parameters.

Example 6.9. Reconsider the definition of procedure addOne from Example 6.8. The set E_{addOne} contains the edge $(f\text{-}in_1^{\mathsf{addOne}}, (1, \mathsf{addOne}))$ because Node 1 is data dependent on Node in in $CFG_{y:=x+1}^{\{x\},\{\}}$. Moreover, the graph PDG_{addOne} contains the edge $((1, \mathsf{addOne}), f\text{-}out_1^{\mathsf{addOne}})$ because Node out is data dependent on Node 1 in $CFG_{y:=x+1}^{\{\},\{y\}}$.◊

The SDG contains the nodes and edges of each procedure dependence graph and edges that connect nodes representing corresponding actual and formal in and out parameters.

Definition 6.17. The set of edges E_{SDG} is defined by $(n, n') \in E_{SDG}$ if and only if there exist k, n'', p, and p' such that one of the following conditions is satisfied:

- $n = (a\text{-}in_{n'',k,\mathsf{p}'}, \mathsf{p})$ and $n' = f\text{-}in_k^{\mathsf{p}'}$ or
- $n = f\text{-}out_k^{\mathsf{p}'}$ and $n' = (a\text{-}out_{n'',k,\mathsf{p}'}, \mathsf{p})$.

The *system dependence graph* (SDG) is the directed graph (N, E) with $N = \bigcup_{\mathsf{p}\in P} N_{\mathsf{p}}$ and $E = \left(\bigcup_{\mathsf{p}\in P} E_{\mathsf{p}}\right) \cup E_{SDG}$. ◊

Now we are ready to define PDGs with summary edges.

Definition 6.18. Let (N, E) be an SDG and $A = \{(a\text{-}in_{n,i,\mathsf{p}'}, \mathsf{p}) \mid n \in N \wedge i \in \{1, \ldots, l_{\mathsf{p}'}\} \wedge \mathsf{p}, \mathsf{p}' \in P\} \cup \{(a\text{-}out_{n,i,\mathsf{p}'}, \mathsf{p}) \mid n \in N \wedge i \in \{1, \ldots, m_{\mathsf{p}'}\} \wedge \mathsf{p}, \mathsf{p}' \in P\}$. A list of nodes $r \in A^*$ is *realizable* if it is generated by the following grammar:

$$R \quad ::= \quad \langle\rangle \mid R :: \langle(a\text{-}in_{n,i,\mathsf{p}'}, \mathsf{p})\rangle :: R :: \langle(a\text{-}out_{n,j,\mathsf{p}'}, \mathsf{p})\rangle$$

A path $p \in N^+$ is realizable if $p|_A$ is realizable. ◊

Definition 6.19. Let $c \in Com_{seq}$, $I, O \subseteq Var$, and $(N, E) = PDG(CFG_c^{I,O})$. The *program dependence graph with summary edges for c, I, and O* is defined by (N', E'), where $N = N'$, $E' = E \cup E''$, and $(n, n') \in E''$ if and only if there exist n'', i, j, and p such that $n = a\text{-}in_{n'',i,\mathsf{p}}$, $n' = a\text{-}out_{n'',j,\mathsf{p}}$, and the SDG contains a realizable path from $f\text{-}in_i^{\mathsf{p}}$ to $f\text{-}out_j^{\mathsf{p}}$. We denote this program dependence graph with $PDG^*(CFG_c^{I,O})$. ◊

The edges that are in $PDG^*(CFG_c^{I,O})$ but not in $PDG(CFG_c^{I,O})$ are called *summary edges*. Intuitively, the summary edge $(a\text{-}in_{n,i,\mathsf{p}}, a\text{-}out_{n,j,\mathsf{p}})$ captures that the execution of procedure p leads to a dependency of the jth actual out on the ith actual in parameter.

Example 6.10. Assume that $P = \{\mathsf{addOne}, \mathsf{proc}\}$, that procedure addOne is defined like in Example 6.8, and that procedure proc is defined by the term

$$((x_1, x_2), (y_1, y_2), \mathsf{addOne}(x_1, x_1); y_2{:=}x_1; y_1{:=}x_2).$$

I.e., procedure proc stores its two in parameters in its two out parameters in reverse order, adding one to the first in parameter.

Then the SDG contains a realizable path from $f\text{-}in_1^{\mathsf{proc}}$ to $f\text{-}out_2^{\mathsf{proc}}$. This path represents the dependency of the second formal out parameter of proc, y_2, on the first formal in parameter of proc, x_1. The path is highlighted by the bold arrows in the graphical representation of the SDG in Figure 6.4.

Consider the command $c = \mathsf{proc}(h_1, h_2; h_1, h_2); \mathsf{proc}(l_1, l_2; l_1, l_2)$. Due to the realizable path from $f\text{-}in_1^{\mathsf{proc}}$ to $f\text{-}out_2^{\mathsf{proc}}$ in the SDG, the graph $PDG^*(CFG_c^{I,O})$ contains the summary edges $(a\text{-}in_{1,1,\mathsf{proc}}, a\text{-}out_{1,2,\mathsf{proc}})$ and $(a\text{-}in_{2,1,\mathsf{proc}}, a\text{-}out_{2,2,\mathsf{proc}})$ (for arbitrary sets of variables I and O). ◊

Definition 6.20. A command c is *accepted by the interprocedural PDG-based analysis* if there is no path from *in* to *out* in $PDG^*(CFG_c^{HI,LO})$. ◊

Example 6.11. Reconsider the procedures and the command c from Example 6.10 and assume that $dma(h_1) = dma(h_2) = high$, $dma(l_1) = dma(l_2) = low$, $Var_{in} = \{h_1, h_2\}$, and $Var_{out} = \{l_1, l_2\}$. The PDG with summary edges $PDG^*(CFG_c^{HI,LO})$ is depicted in Figure 6.5. This graphical representation illustrates that there is no path from *in* to *out*. Hence, c is accepted by the interprocedural PDG-based analysis. ◊

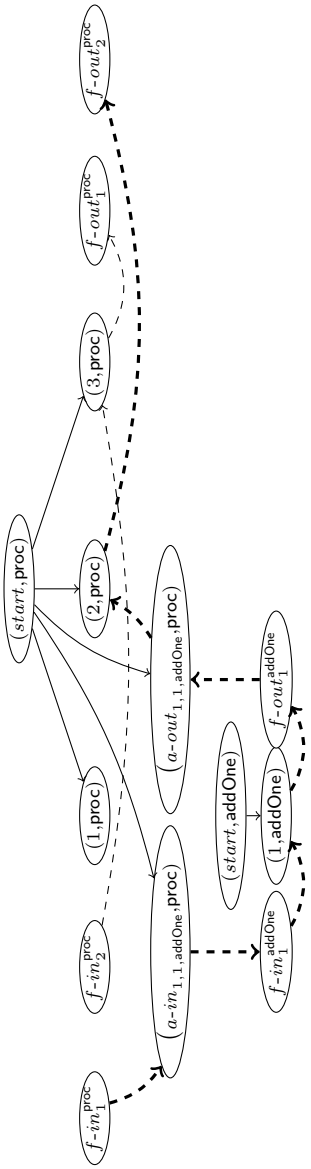


Figure 6.4: System dependence graph for Example 6.10, bold edges highlight a realizable path

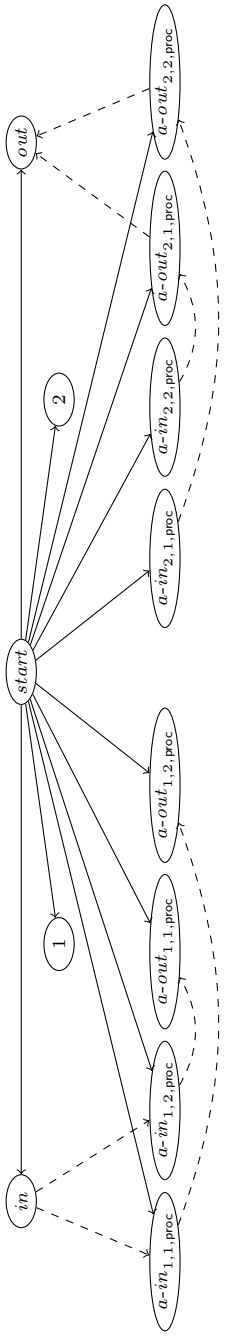


Figure 6.5: PDG with summary edges ($PDG^*(CFG_c^{H,L})$) for Example 6.11

6.5.3 Type-based Interprocedural Analysis

The PDG-based interprocedural information flow analysis from Section 6.5.2 uses the concept of summary edges to make the analysis context-sensitive. Can the idea of summary edges also be adopted in type-based information flow analysis?

Intuitively, given a node n that represents a call of procedure p there is a summary edge $(a\text{-}in_{n,i,\mathsf{p}}, a\text{-}out_{n,j,\mathsf{p}})$ if the jth out parameter of the call might depend on the ith in parameter of the call. This information can be encoded by a function $\Psi_\mathsf{p} : \{y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}\} \to \mathcal{P}(\{x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}\})$, with the intuition that $x_i^\mathsf{p} \in \Psi_\mathsf{p}(y_j^\mathsf{p})$ if the jth out parameter of p might depend on the ith in parameter of p.

We used such functions to develop a type-based information flow analysis for programs with procedures based on the type system from Section 6.2. We were surprised to discover that the solution essentially corresponds to an existing type system that is outlined by Hunt and Sands in [HS11]. In the following, we adapt their type system to the notation used in this thesis and establish the main theorem of this section, namely that the resulting interprocedural type-based information flow analysis accepts exactly the same commands as the interprocedural PDG-based information flow analysis from Section 6.5.2.

The typing judgments have the form $pc \vdash_\Psi \Gamma\,\{c\}\,\Gamma'$ where $\Psi = (\Psi_\mathsf{p})_{\mathsf{p}\in P}$ is a tuple of functions $\Psi_\mathsf{p} : \{y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}\} \to \mathcal{P}(\{x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}\})$. The set of typing rules contains all rules from Figure 6.1 in Section 6.2 in which each judgment of the form $pc \vdash\ \Gamma\,\{c\}\,\Gamma'$ is replaced syntactically with the judgment $pc \vdash_\Psi \Gamma\,\{c\}\,\Gamma'$, as well as the following typing rule for procedure calls:

$$[\mathsf{call}]\ \frac{\begin{array}{c}\Gamma'(x) = \Gamma(x) \text{ for } x \notin \{z_1, \ldots, z_{m_\mathsf{p}}\} \\ \Gamma'(z_j) = \bigsqcup\{\Gamma(x) \mid \exists i : x_i^\mathsf{p} \in \Psi_\mathsf{p}(y_j^\mathsf{p}) \wedge x \in vars(e_i)\} \sqcup pc\end{array}}{pc \vdash_\Psi \Gamma\,\{\mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})\}\,\Gamma'}$$

The rule's first condition captures that the variables that are not among the actual out parameters are not modified by the procedure p. The second condition captures that the security level of the actual out parameter z_i after the execution of p is determined (a) by the security level pc and (b) by the security levels of the variables that occur free in an actual in parameter on which the actual out parameter z_i depends according to Ψ_p.

Similar to the type system from Section 6.2, for each pc, Ψ, Γ, and c there exists a unique environment Γ' such that $pc \vdash_\Psi \Gamma\,\{c\}\,\Gamma'$ is derivable.

The key insight for computing the tuple Ψ that will be used in the type-based information flow analysis is the following: Dependencies of actual out parameters on actual in parameters that are introduced by a procedure call can be computed using the type system where the security lattice is instantiated with the *universal lattice* $(\mathcal{P}(Var), \subseteq)$. Intuitively, if $\{\} \vdash\ \Gamma_{id}\,\{c\}\,\Gamma'$ is derivable where $\Gamma_{id}(x) = \{x\}$ for all $x \in Var$ then the value of y after executing c depends at most on the initial values of the variables in the set $\Gamma'(y)$. This observation is exploited in the following iterative fixed-point computation of the tuple Ψ that is used in the type-based information flow analysis.

Definition 6.21. We define an infinite list of tuples of functions $(\Psi_i)_{i\in\mathbb{N}}$ recursively as follows: For all $\mathsf{p} \in P$ and $y \in \{y_1^\mathsf{p}, \ldots, y_{m_\mathsf{p}}^\mathsf{p}\}$, $(\Psi_1)_\mathsf{p}(y) = \{\}$ and $(\Psi_{i+1})_\mathsf{p}(y) = \Gamma'(y) \cap \{x_1^\mathsf{p}, \ldots, x_{l_\mathsf{p}}^\mathsf{p}\}$, where Γ' is the unique environment such that $\{\} \vdash_{\Psi_i} \Gamma_{id}\,\{c_\mathsf{p}\}\,\Gamma'$ is derivable.

Since $(\Psi_i)_\mathsf{p}(y) \subseteq (\Psi_{i+1})_\mathsf{p}(y)$ for all i, p, and y and the set of variables occurring in a program is finite there exists $k \in \mathbb{N}$ such that $\Psi_{k+i} = \Psi_k$ for all $i \in \mathbb{N}$. We define $\Psi^* = \Psi_k$ for the minimal such k. ◊

Definition 6.22. Let $c \in \mathit{Com}_{seq}$ and let Γ' be the unique environment for which the judgment $\{\} \vdash_{\Psi^*} \Gamma_{id}\,\{c\}\,\Gamma'$ is derivable. The command c is *accepted by the interprocedural type-based analysis* if $\Gamma'(x) \cap \mathit{HI} = \{\}$ for all $x \in \mathit{LO}$. ◊

Intuitively, requiring that $\Gamma'(x) \cap \mathit{HI} = \{\}$ for all $x \in \mathit{LO}$ ensures that the final value of each low output variable does not depend on the initial value of any high input variable.

To prove that the interprocedural type-based analysis accepts precisely the same commands as the interprocedural PDG-based analysis we lift Theorem 6.3 to commands with procedure calls.

Theorem 6.5. Let $c \in \mathit{Com}_{seq}$, $y \in \mathit{Var}$, and Γ be an environment. Let Γ' be the unique environment such that $\{\} \vdash_{\Psi^*} \Gamma\,\{c\}\,\Gamma'$ is derivable in the type system, and let X be the set of all $x \in \mathit{Var}$ such that there exists a path from *in* to *out* in $\mathit{PDG}^*(\mathit{CFG}_c^{\{x\},\{y\}})$. Then $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$. ◊

Proof Sketch. In the proof (see Appendix A.10) we lift the inductive proof of Theorem 6.3 to commands with procedure calls. The key step is to define an approximation of the set of summary edges for each $k \in \mathbb{N}$. The kth approximation contains only those summary edges that are due to realizable paths where the derivation for showing that the path is realizable has at most height k. We then prove a variant of the theorem where Ψ^* is replaced with Ψ_k and the PDG with summary edges is replaced by a PDG to which we only add the summary edges in the kth approximation of summary edges. The theorem for the actual set of summary edges and the tuple Ψ^* then follows from the variant of the theorem for large enough k. □

Theorem 6.6. A program is accepted by the interprocedural PDG-based analysis if and only if it is accepted by the interprocedural type-based analysis. ◊

Proof Sketch. The proof is like the proof of Theorem 6.4, exploiting Theorem 6.5 instead of Theorem 6.3. □

Like for the comparison of the type-based and the PDG-based analysis in Section 6.4, the interprocedural type-based and the interprocedural PDG-based analysis have the same precision. The result indicates that the context-sensitivity of PDG-based information flow analysis is no compelling argument for abandoning type-based information flow analysis in favor of PDG-based information flow analysis.

6.6 A PDG-based Analysis for Multi-threaded Programs

In this section, we exploit the connection between type-based and PDG-based analysis for sequential programs from Section 6.4 to transfer ideas from type-based analysis for multi-threaded programs to PDGs. To this end, we develop a novel provably sound PDG-based analysis for multi-threaded programs by adopting assumption-guarantee style reasoning from the flow-sensitive type-based analysis for multi-threaded programs from Section 4.5.2. Thereby, advantages of the type-based information flow analysis over existing PDG-based information flow analyses for multi-threaded programs carry over to the novel PDG-based analysis (we discuss the advantages in Section 6.7).

Like in the type-based analysis from Section 4.5.2 that provides typing rules for single commands we define a PDG-based analysis for single commands that is then used to analyze multi-threaded programs in a compositional manner. To this end, for a

command c we extend the set of edges of the program dependence graph $PDG(CFG_c^{H,L})$, resulting in a novel program dependence graph that we denote with $PDG^{\parallel}(CFG_c^{H,L})$. The additional edges model dependencies that arise if c is executed in a multi-threaded program in which the assumptions made for c are valid and that do not necessarily exist if c is executed in isolation. For simplicity, we assume that each command is in the set Com_{seq}, i.e., we consider programs that may contain multiple threads but that do not dynamically create new threads during execution. We denote the corresponding set of thread pools $(Com_{seq} \setminus \{stop\})^*$ with Thr_{seq}.

Before making the definition of $PDG^{\parallel}(CFG_c^{H,L})$ precise, we make two observations explicit that we exploit to transfer ideas from the type system to the PDG. To this end, we compare the flow-sensitive type system for sequential programs from Section 6.2 with the flow-sensitive type system for multi-threaded programs from Section 4.5.

Observation 6.1. In both type systems, the typing judgments capture upper bounds on the security levels of the values of variables. In contrast to the type system from Section 6.2, the type system from Section 4.5 requires that the upper bound for variable x is fixed to $dma(x)$ unless a suitable assumption is made for x (i.e., a no-write assumption for high variables or a no-read assumption for low variables). Fixing the upper bound to $dma(x)$ reflects that (a) low variables that might be read by other threads must not store secrets because the secrets might be leaked via other threads (i.e., the upper bound must be *low*), and (b) other threads might write secrets into high variables without no-write assumption (and, hence, the upper bound cannot be *low*). ◊

Observation 6.2. In contrast to the type system from Section 6.2, the type system from Section 4.5 imposes restrictions on the control conditions of conditionals and loops. With the exception of the semantic side condition in rule [IF], the type system requires that the security level of control conditions is *low*. The rationale for this restriction is that secret control conditions might result in internal timing leaks. ◊

We now define the novel variant of program dependence graphs, $PDG^{\parallel}(CFG_c^{H,L})$, by adding edges that capture the additional restrictions imposed by the type-based analysis for multi-threaded programs from Section 4.5 compared to the type-based analysis for sequential programs from Section 6.2. In the remainder of this section we use concepts like, for instance, modes, that are used in the definition of the flow-sensitive type-based information flow analysis for multi-threaded programs, for which we refer to Chapter 4.

Definition 6.23. Let $m \in Mod$ be a mode and $x \in Var$. Command c *does not acquire respectively release* m *for* x if c is not of the form $/\!\!/ann/\!\!/ c'$. The command $/\!\!/\mathsf{acq}(x,m)/\!\!/ c$ *acquires* m *for* x if and only if c does not release m for x. The command $/\!\!/\mathsf{rel}(x,m)/\!\!/ c$ *releases* m *for* x if and only if c does not acquire m for x.

For $c \in Com_{seq}$ and $mds \in Mds$ we define functions $CFG\text{-}mds_{c,mds} : N_c \to Mds$ by $x \in CFG\text{-}mds_{c,mds}(n)(m)$ if and only if for all paths $\langle start, \ldots, n\rangle$ in CFG_c one of the following two conditions is satisfied:

- there exists n' on the path such that $c[n']$ acquires m for x, and for all nodes n'' following n' on the path $c[n'']$ does not release m for x, or
- $x \in mds(m)$ and for all nodes n' on the path $c[n']$ does not release m for x. ◊

Intuitively, $CFG\text{-}mds_{c,mds}(n)$ is a lower bound on the modes at the program statement or control condition in c represented by node n, assuming initial mode state mds.

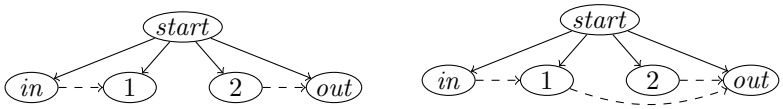


Figure 6.6: PDGs for Example 6.12 (to the left with, to the right without assumption)

Definition 6.24. Let $c \in Com_{seq}$, $mds \in Mds$, and $(N, E) = PDG(CFG_c^{H,L})$. We define the program dependence graph $PDG^{\parallel}(CFG_c^{H,L}, mds)$ by $(N, E \cup E')$ where $(n, n') \in E'$ if and only if one of the following conditions is satisfied:

1. $n = in$, there exists $x \in use_c(n')$ with $dma(x) = high$, and there exists $n'' \in N$ with $x \notin CFG\text{-}mds_{c,mds}(n'')(asm\text{-}no\text{-}w)$, such that there is a path p from n'' to n' with $x \notin def_c(n''')$ for every node n''' on p with $n''' \neq n''$ and $n''' \neq n'$,
2. $n' = out$, there exists $x \in def_c(n)$ with $dma(x) = low$, and there exists $n'' \in N$ with $x \notin CFG\text{-}mds_{c,mds}(n'')(asm\text{-}no\text{-}r)$, such that there is a path p from n to n'' with $x \notin def_c(n''')$ for every node n''' on p with $n''' \neq n$ and $n''' \neq n''$, or
3. $n \in \{1, \ldots, \sharp(c)\}$, $c[n] \in Exp$, and $n' = out$. ◊

The edges defined in Items 1 and 2 of Definition 6.24 capture dependencies on the initial values of high variables and of the final values of low variables that might occur if c is executed as a thread in a multi-threaded program. The assumptions (represented by modes) are exploited to exclude dependencies that only exist if c is executed in a multi-threaded programs for which the assumptions are not valid. The edges defined in Item 3 capture dependencies on control conditions that might result from internal timing leaks.

The definition of the additional edges is inspired by Observations 6.1 and 6.2. According to Observation 6.1, secret values in low variables that might be read by other threads may be leaked into the final values of low variables. This is captured by the edge (n, out) whenever Node n defines a low variable such that the value might eventually be read by another thread. Moreover, according to Observation 6.1, writing a public value into a high variable does not guarantee that the value remains public if other threads might write the value. This is captured by edges (in, n) whenever Node n uses a high variable whose value might have been written by another thread. Finally, according to Observation 6.2, conditionals with high control conditions might lead to internal timing leaks, which is captured by the edges (n, out) whenever n represents a control condition.

Example 6.12. Consider the following command, where h is a secret input variable and l is a public output variable:

$$c = /\!/\mathsf{acq}(asm\text{-}no\text{-}r, l)/\!/; l{:=}h; l{:=}0; /\!/\mathsf{rel}(asm\text{-}no\text{-}r, l)/\!/$$

Then $PDG^{\parallel}(CFG_c^{H,L}, mds_0) = PDG(CFG_c^{H,L})$, where $mds_0(m) = \{\}$ for all $m \in Mod$. This PDG is displayed at the left of Figure 6.6.

Let $c' = l{:=}h; l{:=}0$. Then $PDG^{\parallel}(CFG_{c'}^{H,L}, mds_0)$ contains an edge from the node representing the assignment $l{:=}h$ to Node out (see the PDG at the right of Figure 6.6). This edge is due to Item 2 in Definition 6.24. Intuitively, the edge captures that the value assigned to l in c' might be leaked to low outputs via other threads because no no-read assumption is made. In particular, $PDG^{\parallel}(CFG_{c'}^{H,L}, mds_0) \neq PDG(CFG_{c'}^{H,L})$. ◊

Definition 6.25. Let $thr \in Thr_{seq}$ and let $mdss \in Mds^*$. The thread pool thr is *accepted by the PDG-based analysis for multi-threaded programs for the list of mode states mds*

and scheduler $\mathcal{S}$ if and only if for each $i \in \{1, \ldots, \sharp(thr)\}$ there is no path from *in* to *out* in the program dependence graph $PDG^{\parallel}(CFG^{H,L}_{thr[i]}, mdss[i])$ and $(thr, mdss)$ has sound modes for $\mathcal{S}$. ◊

Example 6.13. Reconsider the commands c and c' from Example 6.12. The thread pool $\langle c \rangle$ is accepted by the PDG-based analysis for multi-threaded programs for the initial mode state mds_0. However, the thread pool $\langle c' \rangle$ is not accepted because due to the additional edge described in Example 6.12 the graph $PDG^{\parallel}(CFG^{H,L}_{c'}, mds_0)$ contains a path from Node *in* to Node *out*. ◊

Not accepting $\langle c' \rangle$ is crucial for the compositionality of the PDG-based analysis because another thread executing $l' := l$ could copy the intermediate secret value of l into a public variable l'. We prove the soundness of the PDG-based analysis for multi-threaded program by relating it to the soundness of the security type system from Section 4.5.

Theorem 6.7. Assume that the observation function *obs* is confined to L. If the thread pool *thr* is accepted by the PDG-based analysis for multi-threaded programs for the list of mode states *mdss* then the thread pool is $\mathcal{S}$-noninterferent for any scheduler $\mathcal{S}$. ◊

Proof Sketch. To prove the theorem, we show that $\vdash thr$ is derivable in the type system from Section 4.5. Then the theorem follows immediately from the soundness result for the type system (Theorem 4.8).

The proof (see Appendix A.10) shows that the non-derivability of the judgment $\vdash \Lambda_i \{thr[i]\} \Lambda'_i$ for a partial environment Λ_i consistent with $mdss[i]$ guarantees the existence of a path from *in* to *out* in the program dependence graph $PDG^{\parallel}(CFG^{H,L}_{thr[i]}, mdss[i])$ (exploiting the relation between PDG-based and type-based analysis from Theorem 6.3). In consequence, for all $i \in \{1, \ldots, \sharp(thr)\}$ there exists Λ'_i such that $\vdash \Lambda_i \{thr[i]\} \Lambda'_i$ is derivable, and, hence, all conditions of the typing rule [PAR] are satisfied. □

The novel PDG-based analysis is compositional, which follows directly from its definition (Definition 6.25). In consequence, it permits concurrent writes to public variables. This is an advantage over existing PDG-based analyses for multi-threaded programs that forbid such useful nondeterministic program behavior (see the comparison to related work in Section 6.7). Moreover, the novel PDG-based analysis is also scheduler-independent. This follows directly from Theorem 6.7 that establishes soundness of the analysis with respect to a scheduler-independent information flow property.

6.7 Summary and Comparison to Related Work

In this chapter, we have investigated the relation between type-based and PDG-based information flow analysis. To this end, we have established a formal connection between the type-based information flow analysis for sequential programs from [HS06] and the PDG-based information flow analysis for sequential programs from [WLS09]. We were able to use this connection in different directions. Firstly, we showed that the type-based information flow analysis has exactly the same precision as the PDG-based information flow analysis. This result is somewhat surprising, in particular, because a motivation for using PDGs in an information flow analysis was their precision (e.g., [HS09]). Secondly, we used the relation to transfer techniques and ideas from one tradition of information flow analysis to the other. In the one direction, we transferred the concept of summary edges from PDG-based analysis to type-based analysis, resulting in a context-sensitive

type-based information flow analysis. In the other direction, we transferred the concept of assumption-guarantee reasoning that was the basis for the type-based analysis of multi-threaded programs in Chapter 4 to PDG-based analysis. This transfer resulted in a provably sound PDG-based information flow analysis for multi-threaded programs that inherits benefits of the type-based analysis over existing PDG-based information flow analyses for multi-threaded programs. This shows that the connection between type-based and PDG-based information flow analysis is suitable to facilitate learning by one research sub-community from the other. For instance, this also gives hope that results on controlling declassification with security type systems can be used to develop semantic foundations for PDG-based analysis that permit declassification. Though there are PDG-based analyses that permit declassification (e.g., [HS09]), they yet lack a soundness result, and, hence, it is unclear which noninterference-like property they certify.

We are aware of only two approaches to PDG-based information flow analysis for multi-threaded programs. Hammer [Ham09] presents an implementation of a PDG-based analysis for concurrent Java programs, where edges between the PDGs of the individual Java threads are computed following the extension of PDGs to concurrent programs from Krinke [Kri03]. However, since this PDG construction does not capture dependencies between nodes in the PDG that are a result of internal timing leaks, the resulting PDG-based information flow analysis does not detect such information leaks.

Giffhorn and Snelting [GS12] present a PDG-based information flow analysis for multi-threaded programs that does not accept programs with internal timing leaks. The analysis enforces an information flow property defined in the tradition of observational determinism [RWW94, ZM03], and, therefore, classifies all programs as insecure that have nondeterministic public output. Hence, the analysis forbids useful nondeterminism, which occurs, for instance, when multiple threads append entries to the same log file (compare also the examples in Sections 3.5 and 4.6). Our novel analysis permits concurrent writes to public variables and, hence, accepts secure programs that are not accepted by the analysis from [GS12]. Moreover, in contrast to the analysis from [GS12] our novel analysis is compositional.

CHAPTER 7

Conclusions and Outlook

7.1 Conclusions

The motivation for this thesis was that existing information flow analyses for multi-threaded programs were not as satisfactory as existing information flow analyses for sequential programs, in particular (a) that existing scheduler-independent properties were overly restrictive, and (b) that existing properties were not suitable for flow-sensitive information flow analyses. Moreover, these deficiencies were already caused by the underlying semantic characterizations of information flow security. These characterizations classify multi-threaded programs as insecure if the programs have certain properties that indicate potential for but do not necessarily result in information flow (like, e.g., nondeterministic public program output or secret control conditions).

In this thesis we have developed novel compositional and scheduler-independent semantic characterizations of information flow security for multi-threaded programs that overcome deficiencies of existing solutions. Moreover, we have exploited the improvements in the semantic characterizations in the development of novel type-based program analysis techniques for information flow security. We have shown that both the novel security properties and the novel type-based analyses can be integrated with each other, resulting in properties and analyses that combine the benefits of the single properties and analyses. Finally, we have developed a bridge between type-based and PDG-based information flow analysis that we have exploited for illustrating that the novel properties are also suitable for developing sound PDG-based information flow analysis for multi-threaded programs.

The benefit of the novel property FSI-security from Chapter 3 is that FSI-security (a) does not unconditionally classify programs with nondeterministic public output as insecure and (b) does not unconditionally classify programs with secret control conditions as insecure. That such a compositional and scheduler-independent information flow property exists was rather surprising in the light of Sabelfeld's result that a more restrictive information flow property, strong security, is the least restrictive compositional information flow property that is scheduler-independent for a natural class of schedulers [Sab03]. The key insight for developing the less restrictive property FSI-security was the characterization

of another natural but smaller class of schedulers, the robust schedulers, that excludes some non-relevant schedulers. We defined FSI-security using the Per-approach, which enables an integration with other information flow properties defined using that approach. We believe that this approach simplifies the integration of FSI-security with information flow properties resulting from research efforts in other directions that are also based on the Per-approach, like, e.g., properties that support controlled declassification. Such properties have been shown to be scheduler independent, but they share deficiencies of scheduler-independent properties predating FSI-security (e.g., [LMP12]).

SIFUM-security (Chapter 4) is the basis for a flow-sensitive information flow analysis for multi-threaded programs. The key for developing SIFUM-security and the corresponding flow-sensitive security type system was the adoption of assumption-guarantee based reasoning in the information flow analysis. Assumption-guarantee based reasoning has been used widely in other application domains (see, e.g., [Sta85, Var95, HQR98, PDH99, Din03]), but was previously not used for language-based information flow security. We have shown with a realistic example program that flow-sensitive information flow analysis can significantly improve the precision of the analysis compared to flow-insensitive information flow analysis. Moreover, we have shown that SIFUM-security and the flow-sensitive security type system can be integrated with the improvements regarding scheduler-independence from Chapter 3 (in Chapter 5), and we believe that the resulting property can be further integrated with properties that support controlled declassification. We hope that our contributions initiate the adoption of both flow-sensitivity as well as assumption-guarantee based reasoning in information flow analysis and lead to more widely and better applicable information flow analyses for multi-threaded programs.

SIFUM-security is one of the two pillars on which we base the development of a sound, compositional, and scheduler-independent PDG-based information flow analysis for multi-threaded programs in Chapter 6. The second pillar is a formal bridge between existing type-based and PDG-based information flow analyses for sequential programs. Besides enabling the development of a PDG-based information flow analysis for multi-threaded programs, with this bridge we could also show that the flow-sensitivity and the context-sensitivity of PDGs are no compelling reasons for favoring PDG-based over type-based information flow analyses. We believe that our results on the relation between PDG-based and type-based information flow analysis can be further used as basis for the communication between the sub-communities advancing the respective analysis techniques, to help learning from each other, and possibly even to jointly advance the state of the art on language-based information flow analysis.

7.2 Outlook

There are multiple promising avenues for building on the results developed in this thesis. Some of them arise from the possibility to further combine results from this thesis, and some of them arise as natural continuations of the presented work. In the remainder of this chapter we briefly discuss possibilities that we believe are most promising.

Exploiting Concrete Synchronization Primitives. To ensure that assumptions and guarantees that are used in the security analyses from Chapters 4, 5, and 6 are valid for the analyzed program, usually threads have to be properly synchronized. While we defined the corresponding security analyses such that they are independent of concrete synchronization primitives, the properties provide the basis for exploiting synchronization

primitives in the analysis. In particular, assumptions and guarantees could be derived directly from the synchronization primitives in the program. For instance, using critical regions that ensure exclusive access for a variable could be exploited by making no-read and no-write assumptions while in the critical region, as well as providing no-read and no-write guarantees when outside the critical region. The benefit of this approach is that assumptions and guarantees would no longer be needed to be specified manually.

Extending Assumptions and Guarantees. In this thesis, we considered no-read and no-write assumptions for shared variables. Beyond this, assumptions and guarantees could also characterize other aspects of program behavior. For instance, another type of assumptions and guarantees could be used to further increase the precision of FSI-security from Chapter 3: FSI-security requires that threads do not write low variables if the number of execution steps before such a write depends on a secret. The corresponding security type system ensures this by forbidding writes to low variables after a high control condition. This requirement prevents internal timing channels that might occur if the low variable is also written by other threads. Under the assumption that the written low variables will not be written by other threads, however, the requirement could be relaxed. For instance, the thread $\mathsf{while}\ (\mathsf{secret} > 0)\ \mathsf{do}\ \mathsf{secret}{:=}\mathsf{secret} - 1\ \mathsf{od}; l{:=}0$ could be classified as secure under the assumption that after the execution of the loop no other thread writes l.

Further Exploiting the Bridge between PDG-based and Type-based Information Flow Analysis. We have used the bridge between PDG-based and type-based information flow analysis from Chapter 6 for comparing the precision of analyses and for transferring concepts from type-based to PDG-based analysis and vice versa. We believe that the bridge is also useful for further transfers of this kind. For instance, the ideas exploited in the definition of FSI-security from Chapter 3 could also be transferred to PDGs. Intuitively, it suffices to remove edges in the PDG for multi-threaded programs from Section 6.6 that represent dependencies of the final values of low variables on control conditions if no low assignments occur after the corresponding conditional or loop. Beyond this, it would be desirable to use results on controlling declassification with security type systems to develop semantic foundations for PDG-based analysis that permit declassification. Moreover, it would also be desirable to transfer further concepts from PDG-based to type-based information flow analysis. For instance, object-sensitive information flow analyses have been investigated in detail for PDGs (e.g., [HS09]), but not for type-based analysis.

Controlled Declassification. We believe that information flow properties and analyses developed in this thesis can be integrated with existing properties and analyses that support controlled declassification (e.g., [MR07, LM09]). These properties are also defined using the Per-approach, and, hence, we expect them to be compatible with the properties developed in this thesis.

APPENDIX A

Detailed Proofs

A.1 Properties of Low Bisimulation Modulo Low Matching

This appendix proves several lemmas that establish that certain relations are low bisimulations modulo low matching.

The proofs in this appendix make frequent use of the following property of low matches:

Lemma A.1. Let thr_1, thr_2, and thr_3 be thread pools with $i_2 = \mathit{l\text{-}match}_{thr_1,thr_2}(i_1)$ and $i_3 = \mathit{l\text{-}match}_{thr_2,thr_3}(i_2)$. Then $i_3 = \mathit{l\text{-}match}_{thr_1,thr_3}(i_1)$. ◊

Proof. The lemma follows immediately from the fact that low matchings are order-preserving bijections. □

Lemma A.2. Define the relation $\mathcal{R}$ on thread pools by $thr_1 \mathcal{R} thr_2$ if and only if there exists thr' such that $thr_1 \sim_{lm} thr'$ and $thr' \sim_{lm} thr_2$. Then $\mathcal{R}$ is a low bisimulation modulo low matching. ◊

Proof. Assume that $thr_1 \mathcal{R} thr_2$, $mem_1 =_L mem_2$, and $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$ where $\alpha_1 = \mathit{new}(thr_1)$. By the definition of $\mathcal{R}$ there exists thr' such that $thr_1 \sim_{lm} thr'$ and $thr' \sim_{lm} thr_2$.

Assume that $thr_1[k_1] \in \mathit{LCom}$. Since $thr_1 \sim_{lm} thr'$ there exist c_2', mem_2', and $\alpha_2 = \mathit{new}(thr_2')$ such that the following conditions are satisfied:

- $\langle thr'[k'], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$ where $k' = \mathit{l\text{-}match}_{thr_1,thr'}(k_1)$,
- $mem_1' =_L mem_2'$,
- $\big|\{i \in \{1,\ldots,\sharp(thr_1')\} \mid thr_1'[i] \in \mathit{LCom}\}\big| = \big|\{i \in \{1,\ldots,\sharp(thr_2')\} \mid thr_2'[i] \in \mathit{LCom}\}\big|$, and
- $\mathit{update}_{k_1}(thr_1, c_1', \alpha_1) \sim_{lm} \mathit{update}_{k'}(thr', c_2', \alpha_2)$.

With $thr' \sim_{lm} thr_2$ it follows that the following conditions are satisfied:

- $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$ where $k_2 = \mathit{l\text{-}match}_{thr',thr_2}(k')$ and
- $\mathit{update}_{k'}(thr', c_1', \alpha_1) \sim_{lm} \mathit{update}_{k_2}(thr_2, c_2', \alpha_2)$.

Hence, $update_{k_1}(thr_1, c_1', \alpha_1) \mathrel{\mathcal{R}} update_{k_2}(thr_2, c_2', \alpha_2)$. Moreover, since low matchings are order-preserving bijections it follows from the equalities $k' = l\text{-}match_{thr_1, thr'}(k_1)$ and $k_2 = l\text{-}match_{thr', thr_2}(k')$ that $k_2 = l\text{-}match_{thr_1, thr_2}(k_1)$.

Assume now that $thr_1[k_1] \in HCom$. Then $update_{k_1}(thr_1, c_1', \alpha_1) \sim_{lm} thr'$ because $thr_1 \sim_{lm} thr'$. Since also $thr' \sim_{lm} thr_2$ it follows that $update_{k_1}(thr_1, c_1', \alpha_1) \mathrel{\mathcal{R}} thr_2$. □

Lemma A.3. Define $thr_1 \mathrel{\mathcal{R}} thr_2$ if and only if there exist $thr_{1,1}$, $thr_{1,2}$, $thr_{2,1}$, and $thr_{2,2}$ such that $thr_1 = thr_{1,1} :: thr_{1,2}$, $thr_2 = thr_{2,1} :: thr_{2,2}$, $thr_{1,1} \sim_{lm} thr_{2,1}$, and $thr_{1,2} \sim_{lm} thr_{2,2}$. Then the relation $\mathcal{R}$ is a low bisimulation modulo low matching. ◊

Proof. Assume that $thr_1 \mathrel{\mathcal{R}} thr_2$, $mem_1 =_L mem_2$, and $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$. As $thr_1 \mathrel{\mathcal{R}} thr_2$ there exist $thr_{1,1}$, $thr_{1,2}$, $thr_{2,1}$, and $thr_{2,2}$ such that $thr_1 = thr_{1,1} :: thr_{1,2}$, $thr_2 = thr_{2,1} :: thr_{2,2}$, $thr_{1,1} \sim_{lm} thr_{2,1}$, and $thr_{1,2} \sim_{lm} thr_{2,2}$. We distinguish between the cases $thr_1[k_1] \in LCom$ and $thr_1[k_1] \in HCom$, as well as between the cases $k_1 \in \{1, \ldots, \sharp(thr_{1,1})\}$ and $k_1 \in \{\sharp(thr_{1,1}) + 1, \ldots, \sharp(thr_1)\}$, resulting in four different cases.

- **Case 1:** $thr_1[k_1] \in LCom$ and $k_1 \in \{1, \ldots, \sharp(thr_{1,1})\}$.

  In this case, $thr_{1,1}[k_1] \in LCom$ and $\langle thr_{1,1}[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$. Hence, as $thr_{1,1} \sim_{lm} thr_{2,1}$ there exist c_2', mem_2', and α_2 such that

  1. $\langle thr_{2,1}[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$,
  2. $mem_1' =_L mem_2'$, and
  3. $update_{k_1}(thr_{1,1}, c_1', \alpha_1) \sim_{lm} update_{k_2}(thr_{2,1}, c_2', \alpha_2)$,

  where $k_2 = l\text{-}match_{thr_{1,1}, thr_{2,1}}(k_1)$.

  The thread pools $thr_{1,1}$ and $thr_{2,1}$ contain the same number of low threads as $thr_{1,1} \sim_{lm} thr_{2,1}$. Hence, $l\text{-}match_{thr_{1,1}, thr_{2,1}}(k_1) = l\text{-}match_{thr_1, thr_2}(k_1)$. With Items (1)–(3) above, $thr_1 = thr_{1,1} :: thr_{1,2}$, and $thr_2 = thr_{2,1} :: thr_{2,2}$ it follows that

  * $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$, and
  * $update_{k_1}(thr_1, c_1', \alpha_1) \mathrel{\mathcal{R}} update_{k_2}(thr_2, c_2', \alpha_2)$,

  where $k_2 = l\text{-}match_{thr_1, thr_2}(k_1)$. This concludes the proof for this case.

- **Case 2:** $thr_1[k_1] \in LCom$ and $k_1 \in \{\sharp(thr_{1,1}) + 1, \ldots, \sharp(thr_1)\}$.

  In this case, $thr_{1,2}[k_1 - \sharp(thr_{1,1})] \in LCom$ and $\langle thr_{1,2}[k_1 - \sharp(thr_{1,1})], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$. The proof is as in the previous case, using that $thr_{1,2} \sim_{lm} thr_{2,2}$ and that $l\text{-}match_{thr_{1,2}, thr_{2,2}}(k_1 - \sharp(thr_{1,1})) = l\text{-}match_{thr_1, thr_2}(k_1) - \sharp(thr_{2,1})$.

- **Case 3:** $thr_1[k_1] \in HCom$ and $k_1 \in \{1, \ldots, \sharp(thr_{1,1})\}$.

  In this case, $thr_{1,1}[k_1] \in HCom$ and $\langle thr_{1,1}[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$. Hence, it follows with $thr_{1,1} \sim_{lm} thr_{2,1}$ that $update_{k_1}(thr_{1,1}, c_1', \alpha_1) \sim_{lm} thr_{2,1}$. Since $thr_1 = thr_{1,1} :: thr_{1,2}$, $thr_2 = thr_{2,1} :: thr_{2,2}$, and $thr_{1,2} \mathrel{\mathcal{R}} thr_{2,2}$, it follows that $update_{k_1}(thr_1, c_1', \alpha_1) \mathrel{\mathcal{R}} thr_2$. This concludes the proof for this case.

- **Case 4:** $thr_1[k_1] \in HCom$ and $k_1 \in \{\sharp(thr_{1,1}) + 1, \ldots, \sharp(thr_1)\}$.

  In this case, $thr_{1,2}[k_1 - \sharp(thr_{1,1})] \in HCom$ and $\langle thr_{1,2}[k_1 - \sharp(thr_{1,1})], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$. The proof is as in the previous case, using that $thr_{1,2} \sim_{lm} thr_{2,2}$. □

Lemma A.4. Define the relation $\mathcal{R}'$ on thread pools by $thr_1\ \mathcal{R}'\ thr_2$ if and only if there exists thr_2^p such that $thr_2|_{LCom} = thr_2^p|_{LCom}$ and $thr_1 \sim_{lm} thr_2^p$. Moreover, define the relation $\mathcal{R}$ as the symmetric closure of $\mathcal{R}'$. Then relation $\mathcal{R}$ is a low bisimulation modulo low matching. ◊

Proof. Assume that $thr_1\ \mathcal{R}\ thr_2$, $mem_1 =_L mem_2$, and $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$ where $\alpha_1 = new(thr_1)$. By the definition of $\mathcal{R}$ there are the two cases that $thr_1\ \mathcal{R}'\ thr_2$ and that $thr_2\ \mathcal{R}'\ thr_1$. We distinguish between these two cases in the following.

Assume that $thr_1\ \mathcal{R}'\ thr_2$. We distinguish between the cases that $thr_1[k_1] \in LCom$ and that $thr_1[k_1] \in HCom$.

– **Case 1:** Assume that $thr_1[k_1] \in LCom$.

Since $thr_1\ \mathcal{R}'\ thr_2$ there exists thr_2^p with $thr_2|_{LCom} = thr_2^p|_{LCom}$ and $thr_1 \sim_{lm} thr_2^p$. Since $thr_1 \sim_{lm} thr_2^p$ there exist c_2', mem_2', and $\alpha_2 = new(thr_2')$ such that the following conditions are satisfied for $k_2 = l\text{-}match_{thr_1, thr_2^p}(k_1)$:

* $\langle thr_2^p[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$,
* $mem_1' =_L mem_2'$,
* $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, and
* $update_{k_1}(thr_1, c_1', \alpha_1) \sim_{lm} update_{k_2}(thr_2^p, c_2', \alpha_2)$.

With $thr_2|_{LCom} = thr_2^p|_{LCom}$ it follows that the following conditions are satisfied:

* $\langle thr_2[k_3], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$ where $k_3 = l\text{-}match_{thr_2^p, thr_2}(k_2)$, and
* $(update_{k_3}(thr_2, c_2', \alpha_2))|_{LCom} = (update_{k_2}(thr_2^p, c_2', \alpha_2))|_{LCom}$

Hence, $update_{k_1}(thr_1, c_1', \alpha_1)\ \mathcal{R}'\ update_{k_2}(thr_2, c_2', \alpha_2)$ and hence, by the definition of $\mathcal{R}$, $update_{k_1}(thr_1, c_1', \alpha_1)\ \mathcal{R}\ update_{k_2}(thr_2, c_2', \alpha_2)$. Moreover, it follows from $k_2 = l\text{-}match_{thr_1, thr_2^p}(k_1)$ and $k_3 = l\text{-}match_{thr_2^p, thr_2}(k_2)$ that $k_3 = l\text{-}match_{thr_1, thr_2}(k_1)$.

– **Case 2:** Assume that $thr_1[k_1] \in HCom$.

Since $thr_1\ \mathcal{R}'\ thr_2$ there exists thr_2^p with $thr_2|_{LCom} = thr_2^p|_{LCom}$ and $thr_1 \sim_{lm} thr_2^p$. Since $thr_1 \sim_{lm} thr_2^p$ it follows from the definition of low bisimulations modulo low matching that $update_{k_1}(thr_1, c_1', \alpha_1) \sim_{lm} thr_2^p$, i.e., $update_{k_1}(thr_1, c_1', \alpha_1)\ \mathcal{R}'\ thr_2$. Hence, by the definition of $\mathcal{R}$, $update_{k_1}(thr_1, c_1', \alpha_1)\ \mathcal{R}\ thr_2$.

Assume now that $thr_2\ \mathcal{R}'\ thr_1$. We distinguish between the cases that $thr_2[k_1] \in LCom$ and that $thr_2[k_1] \in HCom$.

– **Case 1:** Assume that $thr_1[k_1] \in LCom$.

Since $thr_2\ \mathcal{R}'\ thr_1$ there exists thr_1^p with $thr_1|_{LCom} = thr_1^p|_{LCom}$ and $thr_2 \sim_{lm} thr_1^p$. By symmetry of $\sim_{lm}$ it follows that $thr_1^p \sim_{lm} thr_2$.

From $thr_1|_{LCom} = thr_1^p|_{LCom}$ and $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$ it follows that $\langle thr_1^p[k_2], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$, where $k_2 = l\text{-}match_{thr_1, thr_1^p}(k_1)$. Hence, it follows with $thr_1^p \sim_{lm} thr_2$ that there exist c_2', mem_2', and $\alpha_2 = new(thr_2')$ such that the following conditions are satisfied for $k_3 = l\text{-}match_{thr_1^p, thr_2}(k_2)$:

* $\langle thr_2[k_3], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$,
* $mem_1' =_L mem_2'$,
* $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, and
* $update_{k_2}(thr_1^p, c_1', \alpha_1) \sim_{lm} update_{k_3}(thr_2, c_2', \alpha_2)$.

Then $update_{k_3}(thr_2, c_2', \alpha_2) \sim_{lm} update_{k_2}(thr_1^p, c_1', \alpha_1)$ by symmetry of $\sim_{lm}$. Moreover, $(update_{k_1}(thr_1, c_1', \alpha_1))|_{LCom} = (update_{k_2}(thr_1^p, c_1', \alpha_1))|_{LCom}$ follows from $k_2 = l\text{-}match_{thr_1, thr_1^p}(k_1)$ and $thr_1|_{LCom} = thr_1^p|_{LCom}$. In consequence, $update_{k_3}(thr_2, c_2', \alpha_2)\ \mathcal{R}'\ update_{k_1}(thr_1, c_1', \alpha_1)$ (set $thr_2^p = update_{k_2}(thr_1^p, c_1', \alpha_1)$ in the definition of $\mathcal{R}'$), and, hence, $update_{k_1}(thr_1, c_1', \alpha_1)\ \mathcal{R}\ update_{k_3}(thr_2, c_2', \alpha_2)$.

Moreover, $k_3 = l\text{-}match_{thr_1, thr_2}(k_1)$, because $k_2 = l\text{-}match_{thr_1, thr_1^p}(k_1)$ and $k_3 = l\text{-}match_{thr_1^p, thr_2}(k_2)$.

– **Case 2:** Assume that $thr_1[k_1] \in HCom$.

Since $thr_2\ \mathcal{R}'\ thr_1$ there exists thr_1^p with $thr_1|_{LCom} = thr_1^p|_{LCom}$ and $thr_2 \sim_{lm} thr_1^p$. Moreover, $(update_{k_1}(thr_1, c_1', \alpha_1))|_{LCom} = thr_1^p|_{LCom}$ because high threads only spawn high threads. In consequence, $thr_2\ \mathcal{R}'\ (update_{k_1}(thr_1, c_1', \alpha_1))|_{LCom}$ and, hence, $(update_{k_1}(thr_1, c_1', \alpha_1))|_{LCom}\ \mathcal{R}'\ thr_2$. □

Lemma A.5. If $thr_1 \sim_{lm} thr_2$ then $thr_1 \sim_{lm} thr_2|_{LCom}$. ◊

Proof. We define the relation $\mathcal{R}'$ on thread pools by $thr_1\ \mathcal{R}'\ thr_2$ if and only if there exists thr_2^p such that $thr_2|_{LCom} = thr_2^p|_{LCom}$ and $thr_1 \sim_{lm} thr_2^p$. Moreover, we define the relation $\mathcal{R}$ as the symmetric closure of $\mathcal{R}'$. Then $\mathcal{R}$ is a low bisimulation modulo low matching by Lemma A.4.

Assume that $thr_1 \sim_{lm} thr_2$. Then $thr_1\ \mathcal{R}\ thr_2|_{LCom}$. Since $\mathcal{R}$ is a low bisimulation modulo low matching it follows from the definition of $\sim_{lm}$ that $thr_1 \sim_{lm} thr_2|_{LCom}$. □

A.2 Scheduler Simulations

This appendix contains the proofs that the simulation relations used in the proofs of Theorems 3.10–3.12 that establish the robustness of Round-Robin, uniform, and lottery schedulers are actually $\mathcal{S}$-simulations for the respective schedulers.

Lemma A.6. We define relation $\prec$ by $\langle thr_1, mem_1, sst_1\rangle \prec \langle thr_2, mem_2, sst_2\rangle$ if and only if the following conditions are satisfied:

1. $thr_1 \sim_{lm} thr_2$,
2. $mem_1 =_L mem_2$,
3. $thr_1|_{LCom} = thr_2$, and
4. if $k_1 = [(sst_1(choice) + (\sharp(thr_1) - sst_1(size)) \mod \sharp(thr_1)] + 1$ and $thr_1[k_1] \in LCom$ then $l\text{-}match_{thr_1, thr_2}(k_1) = [(sst_2(choice) + (\sharp(thr_2) - sst_2(size)) \mod \sharp(thr_2)] + 1$.

Then the relation $\prec$ is an RR-simulation. ◊

Proof. Let $sc_1 = \langle thr_1, mem_1, sst_1\rangle \prec sc_2 = \langle thr_2, mem_2, sst_2\rangle$.

Then $\sharp(thr_1|_{LCom}) = \sharp(thr_2)$ holds due to Item 3 of the definition of $\prec$ and the fact that $(thr|_{LCom})|_{LCom} = thr|_{LCom}$ for all thread pools thr.

Let k_1, p_1, and $sc_1' = \langle thr_1', mem_1', sst_1'\rangle$ such that $sc_1 \xrightarrow{k_1, p_1}_{RR} sc_1'$. Then, by the derivation rule for transitions of system configurations there exist c_1' and α_1 such that the following conditions are satisfied:

(a) $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mem_1'\rangle$ and

(b) $thr_1' = update_{k_1}(thr_1, c_1', \alpha_1)$

Moreover, it follows from the derivation rule for transitions of system configurations that $(sst_1, \sharp(thr_1)) \overset{k_1,p_1}{\rightsquigarrow}_{RR} sst'_1$. Hence, by the definition of RR, the following conditions are satisfied:
(c) $k_1 = [(sst_1(choice) + (\sharp(thr_1) - sst(size)))\mod \sharp(thr_1)] + 1$ and
(d) $p_1 = 1$

We distinguish between the cases that $thr_1[k_1] \in LCom$ and $thr_1[k_1] \in HCom$.

Assume that $thr_1[k_1] \in LCom$. Then, due to Items 1 and 2 in the definition of $\prec$, there exist c'_2, mem'_2, and α_2 such that the following conditions are satisfied:
(e) $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c'_2, mem'_2\rangle$ where $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$,
(f) $mem'_1 =_L mem'_2$, and
(g) $update_{k_1}(thr_1, c'_1, \alpha_1) \sim_{lm} update_{k_2}(thr_2, c'_2, \alpha_2)$

From (c), $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$ (in (e)), and Item 4 in the definition of $\prec$ it follows that $k_2 = [(sst_2(choice)+(\sharp(thr_2)-sst_2(size)))\mod \sharp(thr_2)]+1$. Hence, by the definition of RR there exists sst'_2 such that $(sst_2, \sharp(thr_2)) \overset{k_2,1}{\rightsquigarrow}_{RR} sst'_2$. Hence, by the derivation rule for transitions of system configurations $sc_2 \overset{k_2,p_2}{\Longrightarrow}_{RR} sc'_2$ for $sc'_2 = \langle thr'_2, mem'_2, sst'_2\rangle$ and $thr'_2 = update_{k_2}(thr_2, c'_2, \alpha_2)$, and $p_2 = 1$. From $p_1 = 1$ it follows that $l\text{-}\rho_{RR}(sc_1) = 1$, and, hence, $p_2 = p_1/l\text{-}\rho_{RR}(sc_1)$.

It remains to show that $sc'_1 \prec sc'_2|_{LCom}$.

From (g) and Lemma A.5 it follows that $thr'_1 \sim_{lm} thr'_2|_{LCom}$, i.e., Item 1 of the definition of $\prec$ holds for sc'_1 and $sc'_2|_{LCom}$. Moreover, Item 2 holds due to (f) and Item 3 holds because $(thr|_{LCom})|_{LCom} = thr|_{LCom}$ for all thread pools.

To show that Item 4 is satisfied, let $k'_1 = [(sst'_1(choice) + (\sharp(thr'_1) - sst'_1(size))\mod \sharp(thr'_1)] + 1$ and assume that $thr'_1[k'_1] \in LCom$. By the definition of RR, it follows that $k'_1 =[(k_1 + (\sharp(thr'_1) - \sharp(thr_1)))\mod \sharp(thr'_1)] + 1$. We distinguish between the cases that $k_1 = \sharp(thr_1)$ and that $k_1 < \sharp(thr_1)$, and write thr''_2 for $thr'_2|_{LCom}$ in the following.

Assume that $k_1 = \sharp(thr_1)$. Then $k'_1 = 1$. Hence, $l\text{-}match_{thr'_1,thr''_2}(k'_1) = 1$ because thr''_2 does not contain high threads. Moreover, $k_2 = \sharp(thr_2)$ because $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$ and thr_2 does not contain any high threads. Hence, $k'_2 =[(k_2 + (\sharp(thr''_2) - \sharp(thr_2)))\mod \sharp(thr''_2)] + 1 = 1$. Thus, $[(sst'_2(choice) + (\sharp(thr''_2) - sst'_2(size))\mod \sharp(thr''_2)] + 1 = 1$ by the definition of RR. I.e., $l\text{-}match_{thr'_1,thr'_2|_{LCom}}(k'_1) = [(sst'_2(choice) + (\sharp(thr'_2|_{LCom}) - sst'_2(size))\mod \sharp(thr'_2|_{LCom})] + 1$.

Assume that $k_1 < \sharp(thr_1)$. Then $k_1 + (\sharp(thr'_1) - \sharp(thr_1)) < \sharp(thr'_1)$, and, hence, $k'_1 = k_1 + (\sharp(thr'_1) - \sharp(thr_1)) + 1$. Moreover, it follows from $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$, $thr'_1[k'_1] \in LCom$, and $k_1 < \sharp(thr_1)$ that $k_2 < \sharp(thr_2)$ (if $k_2 = \sharp(thr_2)$ would hold then k_1 would be the largest index of a low thread in thr_1, and, hence, k'_1 would be the index of a high thread in thr'_1). Hence, $k_2 + (\sharp(thr''_2) - \sharp(thr_2)) < \sharp(thr''_2)$, and, in consequence, $[(k_2 + (\sharp(thr''_2) - \sharp(thr'_1)))\mod \sharp(thr''_2)] + 1 = k_2 + (\sharp(thr''_2) - \sharp(thr'_1)) + 1$. It follows from $l\text{-}match_{thr_1,thr_2}(k_1) = k_2$ that $l\text{-}match_{thr'_1,thr''_2}(k'_1) = k_2 + (\sharp(thr''_2) - \sharp(thr'_1)) + 1$ (k'_1 is obtained by increasing k_1 by 1 plus the number of new threads, and k_2 is increased by 1 plus the number of new low threads) . Hence, it follows with the definition of RR that $l\text{-}match_{thr'_1,thr''_2}(k'_1) = (sst'_2(choice) + (\sharp(thr''_2) - sst'_2(size))\mod \sharp(thr''_2)] + 1$. I.e., we have shown $l\text{-}match_{thr'_1,thr'_2|_{LCom}}(k'_1) = [(sst'_2(choice)+(\sharp(thr'_2|_{LCom}) - sst'_2(size))\mod \sharp(thr'_2|_{LCom})] + 1$.

Assume now that $thr_1[k_1] \in HCom$. We have to show that $sc'_1 \prec sc_2$. From Items 1 and 2 in the definition of $\prec$ and the definition of $\sim_{lm}$ it follows that $thr'_1 \sim_{lm} thr_2$ (i.e., Item 1 of the definition of $\prec$ holds for sc'_1 and sc_2). That Item 2 holds follows from

$thr_1[k_1] \in HCom$. Items 3 and 4 follow from the fact that an execution step of a high thread does not change the order of low threads in the thread pool, and, in consequence, $thr_1|_{LCom} = thr'_1|_{LCom}$. □

Lemma A.7. We define relation $\prec$ by $\langle thr_1, mem_1, sst_1\rangle \prec \langle thr_2, mem_2, sst_2\rangle$ if and only if the following conditions are satisfied:

1. $thr_1 \sim_{lm} thr_2$,
2. $mem_1 =_L mem_2$, and
3. $thr_1|_{LCom} = thr_2$.

Then the relation $\prec$ is a *Uni*-simulation. ◊

Proof. Let $sc_1 = \langle thr_1, mem_1, sst_1\rangle \prec sc_2 = \langle thr_2, mem_2, sst_2\rangle$.

Then $\sharp(thr_1|_{LCom}) = \sharp(thr_2)$ holds due to Item 3 of the definition of $\prec$ and the fact that $(thr|_{LCom})|_{LCom} = thr|_{LCom}$ for all thr.

Let k_1, p_1, and $sc'_1 = \langle thr'_1, mem'_1, sst'_1\rangle$ such that $sc_1 \xrightarrow{k_1,p_1}_{Uni} sc'_1$. Then by the derivation rule for transitions of system configurations there exist c'_1 and α_1 such that the following conditions are satisfied:
(a) $\langle thr_1[k_1], mem_1\rangle \xrightarrow{\alpha_1} \langle c'_1, mem'_1\rangle$ and
(b) $thr'_1 = update_{k_1}(thr_1, c'_1, \alpha_1)$.

Moreover, it follows from the derivation rule for transitions of system configurations that $(sst_1, \sharp(thr_1)) \overset{k_1,p_1}{\rightsquigarrow}_{Uni} sst'_1$. It follows by the definition of *Uni* that $p_1 = 1/\sharp(thr_1)$.

We distinguish between the cases that $thr_1[k_1] \in LCom$ and that $thr_1[k_1] \in HCom$.

Assume that $thr_1[k_1] \in LCom$. Then, due to Items 1 and 2 in the definition of $\prec$, there exist c'_2, mem'_2, and α_2 such that the following conditions are satisfied:
(c) $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c'_2, mem'_2\rangle$ where $k_2 = l\text{-}match_{thr_1,thr_2}(k_1)$,
(d) $mem'_1 =_L mem'_2$, and
(e) $update_{k_1}(thr_1, c'_1, \alpha_1) \sim_{lm} update_{k_2}(thr_2, c'_2, \alpha_2)$.

From the definition of *Uni* it follows that there are p_2 and sst'_2 with $(sst_2, \sharp(thr_2)) \overset{k_2,p_2}{\rightsquigarrow}_{Uni} sst'_2$, where $p_2 = 1/\sharp(thr_2)$. Hence, by the derivation rule for transitions of system configurations we have $sc_2 \xrightarrow{k_2,p_2}_{Uni} sc'_2$ where $sc'_2 = \langle update_{k_2}(thr_2, c'_2, \alpha_2), mem'_2, sst'_2\rangle$.

From $p_2 = 1/\sharp(thr_2)$ and Item 3 in the definition of $\prec$ it follows $p_2 = 1/\sharp(thr_1|_{LCom})$. Hence, by the definition of *Uni* it follows that $l\text{-}\rho_{Uni}(sc_1) = \sharp(thr_1|_{LCom})/\sharp(thr_1)$. In consequence, $p_1/l\text{-}\rho_{Uni}(sc_1) = (1/\sharp(thr_1))/(\sharp(thr_1|_{LCom})/\sharp(thr_1)) = p_2$.

It remains to show that $sc'_1 \prec sc'_2|_{LCom}$. From (e) and Lemma A.5 it follows that

$$update_{k_1}(thr_1, c'_1, \alpha_1) \sim_{lm} (update_{k_2}(thr_2, c'_2, \alpha_2))|_{LCom},$$

i.e., Item 1 of the definition of $\prec$ holds for sc'_1 and sc'_2. Moreover, Item 2 holds due to (d) and Item 3 holds because $(thr|_{LCom})|_{LCom} = thr|_{LCom}$ for all thr. □

Lemma A.8. We define relation $\prec$ by $\langle thr_1, mem_1, sst_1\rangle \prec \langle thr_2, mem_2, sst_2\rangle$ if and only if the following conditions are satisfied:

1. $thr_1 \sim_{lm} thr_2$,
2. $mem_1 =_L mem_2$, and
3. $thr_1|_{LCom} = thr_2$.

Then the relation $\prec$ is a *Lot*-simulation. ◊

Proof. The proof is in large parts the same as the proof of Lemma A.7. The only difference is in the argument that $p_1/l\text{-}\rho_{Uni}(sc_1) = p_2$ in the case that $thr_1[k_1] \in LCom$.

For the lottery scheduler, $p_1 = tickets_{k_1}(sin_1)/\big[\sum_{i=1}^{\sharp(thr_1)} tickets_i(sin_1)\big]$ and $p_2 = tickets_{k_2}(sin_2)/\big[\sum_{i=1}^{\sharp(thr_2)} tickets_i(sin_2)\big]$, where $sin_1 = obs(thr_1, mem_1)$ and $sin_2 = obs(thr_2, mem_2)$. Moreover, by the definition of *Lot* it follows that

$$l\text{-}\rho_{Lot}(sc_1) = \big[\sum_{i\in\{1,\ldots,\sharp(thr_1)|thr_1[i]\in LCom\}} tickets_i(sin_1)\big]/\big[\sum_{i=1}^{\sharp(thr_1)} tickets_i(sin_1)\big].$$

Hence, $p_1/l\text{-}\rho_{Lot}(sc_1) = tickets_{k_1}(sin_1)/\big[\sum_{i\in\{1,\ldots,\sharp(thr_1)|thr_1[i]\in LCom\}} tickets_i(sin_1)\big]$. As the observation function is confined the number of tickets is stored in low variables. Hence, it follows from $mem_1 =_L mem_2$ that $tickets_i(sin_1) = tickets_{l\text{-}match_{thr_1,thr_2}}(sin_2)$. In consequence, $p_1/l\text{-}\rho_{Uni}(sc_1) = p_2$. □

A.3 Scheduler-independence of FSI-security

This section contains the proof of Theorem 3.14 from Section 3.3.4. Theorem 3.14 is the key for proving the scheduler independence of FSI-security.

Before proving Theorem 3.14, we prove two lemmas that relate the probability that a run starting in some system configuration terminates with a given memory to the probabilities that a run starting in a system configuration reached from that system configuration in one step terminates in that memory.

Lemma A.9. Let sc be a system configuration that is not final and let $J \subset \mathbb{N}$ be the set of $j \in \mathbb{N}$ for which there exist $sc' \in SysConf$ and $p > 0$ with $sc \stackrel{j,p}{\Longrightarrow}_{\mathcal{S}} sc'$.

By the definition of schedulers and the semantics for execution steps of system configurations, the system configuration sc' is uniquely determined by sc and j, and we denote that system configuration for $j \in J$ with sc'_j.

Then for every scheduler $\mathcal{S}$, every memory mem, and every $k \in \mathbb{N}$ the following equation is satisfied:

$$\rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem} \mid \sharp(str) = k+1\}) = \sum_{j\in J} \rho^{\mathcal{S}}_{sc}(j) * \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc'_j)_{\Downarrow mem} \mid \sharp(str) = k\}) \quad \Diamond$$

Proof. In the proof, we denote the set $\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem} \mid \sharp(str) = k\}$ with $\mathcal{T}_{\mathcal{S}}(sc)^k_{\Downarrow mem}$ for every $k \in \mathbb{N}$.

Since sc is not final, $J \neq \{\}$.

By the definition of $\rho_{\mathcal{S}}$ (cf. Definition 2.15) and the definition of induced probability measures (cf. Definition 2.4) we have

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)^k_{\Downarrow mem}) = \sum_{tr\in\mathcal{T}_{\mathcal{S}}(sc)^k_{\Downarrow mem}} \rho(tr)$$

for every $k \in \mathbb{N}$, where $\rho(tr)$ is the trace probability defined in Definition 2.12. Since $J \neq \{\}$, we can rewrite this sum as

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)^{k}_{\Downarrow mem}) = \sum_{j \in J} \left(\sum_{\substack{(str,dtr) \in \mathcal{T}_{\mathcal{S}}(sc)^{k}_{\Downarrow mem} \\ str[2] = sc'_j}} \rho(tr) \right). \tag{A.1}$$

Let $tr = (str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)^{k}_{\Downarrow mem}$. Then, by Definition 2.12,

$$\rho(tr) = \prod_{i=1}^{\sharp(dtr)} p_{i,dtr(i)}$$

where $p_{i,k}$ is the probability p for which $str[i] \overset{k,p}{\Longrightarrow}_{\mathcal{S}} str[i+1]$ is derivable. Hence, if $str[2] = sc'_j$ then

$$\rho(tr) = \rho^{\mathcal{S}}_{sc}(j) * \prod_{i=2}^{\sharp(dtr)} p_{i,dtr(i)}. \tag{A.2}$$

Moreover, for every $k \in \mathbb{N}$ and every $j \in J$, one obtains the set $\mathcal{T}_{\mathcal{S}}(sc'_j)^{k}_{\Downarrow mem}$ by taking each element (str, dtr) from the set $\mathcal{T}_{\mathcal{S}}(sc)^{k+1}_{\Downarrow mem}$ and removing the first element of str and the first element of dtr. Hence, with Definition 2.12 we have for every $j \in J$ that

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc'_j)^{k}_{\Downarrow mem}) = \sum_{\substack{(str,dtr) \in \mathcal{T}_{\mathcal{S}}(sc)^{k+1}_{\Downarrow mem} \\ str[2] = sc'_j}} \left(\prod_{i=2}^{\sharp(dtr)} p_{i,dtr(i)} \right) \tag{A.3}$$

where $p_{i,k}$ is the probability p for which $str[i] \overset{k,p}{\Longrightarrow}_{\mathcal{S}} str[i+1]$ is derivable.

From Equations (A.1), (A.2), and (A.3) we obtain that

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)^{k+1}_{\Downarrow mem}) = \sum_{j \in J} \rho^{\mathcal{S}}_{sc}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc'_j)^{k}_{\Downarrow mem})$$

□

Lemma A.10. Let sc be a system configuration that is not final, and let $J \subset \mathbb{N}$ be the set of $j \in \mathbb{N}$ for which there exist $sc' \in SysConf$ and $p > 0$ with $sc \overset{j,p}{\Longrightarrow}_{\mathcal{S}} sc'$.

By the definition of schedulers and the semantics for execution steps of system configurations, the system configuration sc' is uniquely determined by sc and j, and we denote that system configuration for $j \in J$ with sc'_j.

Then for every scheduler $\mathcal{S}$ and every memory mem the following equation is satisfied:

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem}) = \sum_{j \in J} \rho^{\mathcal{S}}_{sc}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc'_j)_{\Downarrow mem})$$

Proof. It follows from Lemma A.9 that

$$\sum_{k \in \mathbb{N}} \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem} \mid \sharp(str) = k+1\}) =$$
$$\sum_{k \in \mathbb{N}} \sum_{j \in J} \rho^{\mathcal{S}}_{sc}(j) * \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc'_j)_{\Downarrow mem} \mid \sharp(str) = k\}).$$

Since sc is not final, there is no $(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem}$ with $\sharp(str) = 1$. Hence, we can rewrite the equation as follows, replacing $(k+1)$ with k on the left hand side, and reordering the sums on the right hand side:

$$\sum_{k\in\mathbb{N}} \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem} \mid \sharp(str) = k\}) = \sum_{j\in J} \rho^{\mathcal{S}}_{sc}(j) * \left(\sum_{k\in\mathbb{N}} \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc'_j)_{\Downarrow mem} \mid \sharp(str) = k\}) \right)$$

Since

$$\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem} = \bigcup_{k\in\mathbb{N}} \{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem} \mid \sharp(str) = k\}$$

for every system configuration sc, every scheduler $\mathcal{S}$, and every memory mem, it follows with the properties of the probability measure $\rho^{\mathcal{S}}_{sc}(j)$ that

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem}) = \sum_{j\in J} \rho^{\mathcal{S}}_{sc}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc'_j)_{\Downarrow mem}).$$

□

Now we prove Theorem 3.14.

Proof of Theorem 3.14. Let $mem \in Mem$. We must show that

$$\sum_{mem'\in[mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem'\in[mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'}).$$

In the remainder of the proof, we write $P(sc)$ for $\sum_{mem'\in[mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'})$. I.e., we must show $P(sc_1) = P(sc_2)$.

Let $\sharp(T) = \sum_{tr\in T} \sharp(tr)$ be the sum of the lengths of all traces in a finite set of terminating traces T. Our proof is by induction on $n = \sharp(\mathcal{T}_{\mathcal{S}}(sc_1)) + \sharp(\mathcal{T}_{\mathcal{S}}(sc_2))$. (Both sets $\sharp(\mathcal{T}_{\mathcal{S}}(sc_1))$ and $\sharp(\mathcal{T}_{\mathcal{S}}(sc_2))$ contain only terminating traces because sc_1 and sc_2 are terminating. Moreover, both sets are finite because the set of terminating traces for a given configuration is finite.)

Induction basis ($n = 2$):
Note that $n < 2$ is not possible because for an arbitrary system configuration sc the set $\mathcal{T}_{\mathcal{S}}(sc)$ contains at least one trace of length 1, namely the trace $(\langle sc\rangle, \langle\rangle)$ with system trace $\langle sc\rangle$ and decision trace $\langle\rangle$.

If $n = 2$ then both $\mathcal{T}_{\mathcal{S}}(sc_1)$ and $\mathcal{T}_{\mathcal{S}}(sc_1)$ contain exactly one trace of length 1. In consequence, both sc_1 and sc_2 are final configurations, and $\mathcal{T}_{\mathcal{S}}(sc_1)$ and $\mathcal{T}_{\mathcal{S}}(sc_1)$ contain only the trace $(\langle sc_1\rangle, \langle\rangle)$ and $(\langle sc_2\rangle, \langle\rangle)$, respectively.

In consequence, $P(sc_1) = 1$ if and only if $mem_1 =_L mem$ and $P(sc_1) = 0$ otherwise, and $P(sc_2) = 1$ if and only if $mem_2 =_L mem$ and $P(sc_2) = 0$ otherwise. Since $mem_1 =_L mem_2$, it follows that $P(sc_1) = P(sc_2)$.

Induction step ($n > 2$):
Let us at first briefly elaborate on the idea of the proof of the induction step: We consider transitions of low and high threads in sc_1 and sc_2 separately. If sc_1 makes a transition in a high thread to sc'_1, we prove that sc'_1, sc_2, $sc_{1,p}$, and $sc_{2,p}$ satisfy all the preconditions of the induction hypothesis, and, hence, $P(sc'_1) = P(sc_2)$. We proceed likewise for transitions of low threads, showing that $P(sc'_1) = P(sc'_2)$, where sc'_1 and sc'_2 are reached

from sc_1 and sc_2 by transitions of low threads linked by $l\text{-}match_{thr_1,thr_2}$. Summing over the probabilities of the possible execution steps in sc_1 and sc_2 yields a system of two linear equations with unknowns $P(sc_1)$ and $P(sc_2)$. We solve the equation system using the relation on probabilities given by $\mathcal{S}$-simulations for robust schedulers, resulting in $P(sc_1) = P(sc_2)$, which concludes the proof.

We now prove the induction step.

1. For each system configuration sc, we denote with $enabled(sc) \subset \mathbb{N}$ the set of $j \in \mathbb{N}$ for which there exist $sc' \in SysConf$ and $p > 0$ with $sc \overset{j,p}{\Longrightarrow}_{\mathcal{S}} sc'$.
   By the definition of schedulers and the semantics for system configurations, sc' is uniquely determined by sc and j. We denote this system configuration with $succ_j(sc)$.
2. By Lemma A.10 the following equation holds for every memory mem:

$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem}) = \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))_{\Downarrow mem})$$

   Hence, the following holds:

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem}) = \sum_{mem' \in [mem]_{LO}} \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))_{\Downarrow mem})$$

   Switching the sums on the right hand side of the equation, we obtain

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem}) = \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))_{\Downarrow mem}).$$

   Using the notation introduced for this proof, we write this equation as follows:

$$P(sc_1) = \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1)).$$

3. We rewrite this sum as $P(sc_1) = P(sc_1, high) + P(sc_1, low)$, where

$$P(sc_1, high) = \sum_{\substack{j \in enabled(sc_1) \\ \wedge thr_1[j] \in HCom}} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1))$$

   and

$$P(sc_1, low) = \sum_{\substack{j \in enabled(sc_1) \\ \wedge thr_1[j] \in LCom}} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1)).$$

4. Let $j \in enabled(sc_1)$. Then there exist c_1', mem_1', α, p, and sst_1' such that $\langle thr_1[j], mem_1\rangle \xrightarrow{\alpha} \langle c_1', mem_1'\rangle$ and $(sst_1, obs(thr_1, mem_1)) \overset{j,p}{\rightsquigarrow}_{\mathcal{S}} sst_1'$. Moreover, $succ_j(sc_1) = \langle update_j(thr_1, c_1', \alpha), mem_1', sst_1'\rangle$.

5. Assume firstly that $thr_1[j] \in HCom$. We show that this implies $P(succ_j(sc_1)) = P(sc_2)$.
   a) Since $thr_1 \sim_{lm} thr_2$, by the definition of low bisimulations modulo low matching $getThr(succ_j(sc_1)) \sim_{lm} thr_2$ holds.
   b) From $thr_1[j] \in HCom$ it follows that $getMem(succ_j(sc_1)) =_L mem_2$.
   c) From $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and the definition of $\mathcal{S}$-simulations it follows that

$$succ_j(sc_1) \prec_{\mathcal{S}} sc_{1,p}.$$

d) From $sc_1 \sim_{lm} sc_{1,p}$ and the definition of $\sim_{lm}$ it follows that

$$succ_j(sc_1) \sim_{lm} sc_{1,p}.$$

e) Since sc_1 makes a transition to $succ_j(sc_1)$ it follows that

$$\sharp(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))) + \sharp(\mathcal{T}_{\mathcal{S}}(sc_2)) < n,$$

and that $succ_j(sc_1)$ is also terminating.

f) From (a), (b), (c), (d), (e), and the induction hypothesis for the configurations $succ_j(sc_1)$, sc_2, $sc_{1,p}$, and $sc_{2,p}$ it follows that

$$P(succ_j(sc_1)) = P(sc_2).$$

6. Using (5.), we rewrite $P(sc_1, high)$ as follows:

$$P(sc_1, high) = P(sc_2) * \sum_{\substack{j \in enabled(sc_1) \\ \wedge thr_1[j] \in HCom}} \rho^{\mathcal{S}}_{sc_1}(j).$$

By the definition of $l\text{-}\rho_{\mathcal{S}}$, it follows that

$$1 - l\text{-}\rho_{\mathcal{S}}(sc_1) = \sum_{\substack{j \in enabled(sc_1) \\ \wedge thr_1[j] \in HCom}} \rho^{\mathcal{S}}_{sc_1}(j).$$

Hence, $P(sc_1, high) = P(sc_2) * (1 - l\text{-}\rho_{\mathcal{S}}(sc_1))$.
Analogously, it follows that $P(sc_2, high) = P(sc_1) * (1 - l\text{-}\rho_{\mathcal{S}}(sc_2))$.

7. Assume now that $thr_1[j] \in LCom$ and that $k = l\text{-}match_{thr_1, thr_2}(j)$.

8. We show that $\rho^{\mathcal{S}}_{sc_1}(j)/l\text{-}\rho_{\mathcal{S}}(sc_1) = \rho^{\mathcal{S}}_{sc_2}(k)/l\text{-}\rho_{\mathcal{S}}(sc_2)$.
   a) Since $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ it follows from the definition of $\mathcal{S}$-simulations that

$$\rho^{\mathcal{S}}_{sc_1}(j)/l\text{-}\rho_{\mathcal{S}}(sc_1) = \rho^{\mathcal{S}}_{sc_{1,p}}(j'),$$

   where $j' = l\text{-}match_{thr_1, thr_{1,p}}(j)$.
   b) Analogously, it follows that

$$\rho^{\mathcal{S}}_{sc_2}(k)/l\text{-}\rho_{\mathcal{S}}(sc_2) = \rho^{\mathcal{S}}_{sc_{2,p}}(k'),$$

   where $k' = l\text{-}match_{thr_2, thr_{2,p}}(k)$.
   c) Let m be such that $thr_1[j]$ is the mth low thread in thr_1. Then $thr_2[k]$ is the mth low thread in thr_2 because $k = l\text{-}match_{thr_1, thr_2}(j)$. Since $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and $sc_2 \prec_{\mathcal{S}} sc_{2,p}$, $sc_{1,p}$ and $sc_{2,p}$ do not contain low threads. It follows that $m = j' = k'$.
   d) It follows from $sc_1 \sim_{lm} sc_2$ that $\sharp(thr_1|_{LCom}) = \sharp(thr_2|_{LCom})$. Hence, because $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and $sc_2 \prec_{\mathcal{S}} sc_{2,p}$, it follows that $\sharp(thr_{1,p}) = \sharp(thr_{2,p})$. With $mem_{1,p} =_L mem_{2,p}$ and the fact that obs is confined to L it follows that

$$obs(thr_{1,p}, mem_{1,p}) = obs(thr_{2,p}, mem_{2,p}).$$

   In consequence, $\rho^{\mathcal{S}}_{sc_{1,p}}(m) = \rho^{\mathcal{S}}_{sc_{2,p}}(m)$.

e) Combining the equalities from (a), (b), and (d), and the fact that $m = j' = k'$ (from (c)), it follows that

$$\rho^{\mathcal{S}}_{sc_1}(j)/l\text{-}\rho_{\mathcal{S}}(sc_1) = \rho^{\mathcal{S}}_{sc_2}(k)/l\text{-}\rho_{\mathcal{S}}(sc_2).$$

9. The inequality $\rho^{\mathcal{S}}_{sc_1}(j) > 0$ holds because $j \in \mathit{enabled}(sc_1)$. Hence, by (8.), the inequality $\rho^{\mathcal{S}}_{sc_2}(k) > 0$ holds. Moreover, since $sc_1 \sim_{lm} sc_2$, $j \in \mathit{enabled}(sc_1)$, $\mathit{thr}_1[j] \in \mathit{LCom}$, and $k = l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_2}(j)$, by the definition of low bisimulations modulo low matching there exists c', mem', and α' such that $\langle \mathit{thr}_2[k], \mathit{mem}_2\rangle \xrightarrow{\alpha'} \langle c', \mathit{mem}'\rangle$. In consequence, $k \in \mathit{enabled}(sc_2)$.

10. We now show that $P(\mathit{succ}_j(sc_1)) = P(\mathit{succ}_k(sc_2))$.
    a) Since $\mathit{thr}_1 \sim_{lm} \mathit{thr}_2$ and $\mathit{mem}_1 =_L \mathit{mem}_2$ it follows from the definition of low bisimulations modulo low matching that the following conditions are satisfied:
       – $\mathit{getThr}(\mathit{succ}_j(sc_1)) \sim_{lm} \mathit{getThr}(\mathit{succ}_k(sc_2))$ and
       – $\mathit{getMem}(\mathit{succ}_j(sc_1)) =_L \mathit{getMem}(\mathit{succ}_k(sc_2))$.
    b) From $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and $\mathit{thr}_1 \sim_{lm} \mathit{thr}_{1,p}$ it follows that
       – $\mathit{succ}_j(sc_1) \prec_{\mathcal{S}} (\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_{1,p}}(j)}(sc_{1,p}))|_{\mathit{LCom}}$,
       – $\mathit{getThr}(\mathit{succ}_j(sc_1)) \sim_{lm} \mathit{getThr}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_{1,p}}(j)}(sc_{1,p}))$, and
       – $\mathit{getMem}(\mathit{succ}_j(sc_1)) =_L \mathit{getMem}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_{1,p}}(j)}(sc_{1,p}))$.

       By Lemma A.5 it follows that
       – $\mathit{getThr}(\mathit{succ}_j(sc_1)) \sim_{lm} (\mathit{getThr}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_{1,p}}(j)}(sc_{1,p})))|_{\mathit{LCom}}$.
    c) Moreover, from $sc_2 \prec_{\mathcal{S}} sc_{2,p}$ and $\mathit{thr}_2 \sim_{lm} \mathit{thr}_{2,p}$ it follows that
       – $\mathit{succ}_j(sc_2) \prec_{\mathcal{S}} (\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_2,\mathit{thr}_{2,p}}(k)}(sc_{2,p}))|_{\mathit{LCom}}$,
       – $\mathit{getThr}(\mathit{succ}_j(sc_2)) \sim_{lm} \mathit{getThr}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_2,\mathit{thr}_{2,p}}(k)}(sc_{2,p}))$, and
       – $\mathit{getMem}(\mathit{succ}_j(sc_2)) =_L \mathit{getMem}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_2,\mathit{thr}_{2,p}}(k)}(sc_{2,p}))$.

       By Lemma A.5 it follows that
       – $\mathit{getThr}(\mathit{succ}_k(sc_2)) \sim_{lm} (\mathit{getThr}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_2,\mathit{thr}_{2,p}}(k)}(sc_{2,p})))|_{\mathit{LCom}}$.

       Since obs is confined to L, it follows from $\sharp(\mathit{getThr}(sc_{1,p})) = \sharp(\mathit{getThr}(sc_{2,p}))$ and $\mathit{mem}_{1,p} =_L \mathit{mem}_{2,p}$ that

$$\mathit{getSst}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_{1,p}}(j)}(sc_{1,p})) = \mathit{getSst}(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_2,\mathit{thr}_{2,p}}(k)}(sc_{2,p})).$$

    d) Since sc_1 and sc_2 are both terminating, $\mathit{succ}_j(sc_1)$ and $\mathit{succ}_k(sc_2)$ are also terminating. Moreover, $\sharp(\mathcal{T}_{\mathcal{S}}(\mathit{succ}_j(sc_1))) + \sharp(\mathcal{T}_{\mathcal{S}}(\mathit{succ}_k(sc_2))) < n$.
    e) With (a), (b), (c), (d), and the induction hypothesis for $\mathit{succ}_j(sc_1)$, $\mathit{succ}_k(sc_2)$, $(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_1,\mathit{thr}_{1,p}}(j)}(sc_{1,p}))|_{\mathit{LCom}}$, and $(\mathit{succ}_{l\text{-}\mathit{match}_{\mathit{thr}_2,\mathit{thr}_{2,p}}(k)}(sc_{2,p}))|_{\mathit{LCom}}$ it follows that

$$P(\mathit{succ}_j(sc_1)) = P(\mathit{succ}_k(sc_2)).$$

11. Expressing $P(sc_1)$ and $P(sc_2)$ using the equations from (6.) results in the following equation system in $X = P(sc_1)$ and $Y = P(sc_2)$:

$$X = Y * (1 - l\text{-}\rho_{\mathcal{S}}(sc_1)) + \sum_{\substack{j \in \mathit{enabled}(sc_1) \\ \wedge\, \mathit{thr}_1[j] \in \mathit{LCom}}} \rho^{\mathcal{S}}_{sc_1}(j) * P(\mathit{succ}_j(sc_1))$$

$$Y = X * (1 - l\text{-}\rho_{\mathcal{S}}(sc_2)) + \sum_{\substack{k \in \mathit{enabled}(sc_2) \\ \wedge\, \mathit{thr}_2[k] \in \mathit{LCom}}} \rho^{\mathcal{S}}_{sc_2}(k) * P(\mathit{succ}_k(sc_2))$$

To complete the induction step it remains to show that each solution (x, y) of this equation system satisfies $x = y$.

12. For solving the equation system, we introduce abbreviations to make the formulas more readable. We define $S_i = P(sc_i, low)$ and $\rho_i = 1 - l\text{-}\rho_{\mathcal{S}}(sc_i)$ for $i \in \{1, 2\}$. Using these notations, the equation system reads as follows:

$$X = Y * \rho_1 + S_1$$
$$Y = X * \rho_2 + S_2$$

13. Solving the above equation system for X and Y yields

$$X = \frac{\rho_1 * S_2 + S_1}{1 - \rho_1 * \rho_2} \text{ and } Y = \frac{\rho_2 * S_1 + S_2}{1 - \rho_1 * \rho_2}.$$

14. To show that $X = Y$ it hence suffices to show that $\rho_1 * S_2 + S_1 = \rho_2 * S_1 + S_2$. This is equivalent to showing that $(1 - \rho_2) * S_1 = (1 - \rho_1) * S_2$. Expanding the definitions of ρ_1 and ρ_2 this equation reads $l\text{-}\rho_{\mathcal{S}}(sc_2) * S_1 = l\text{-}\rho_{\mathcal{S}}(sc_1) * S_2$.
After expanding S_1 and S_2 the equation reads as

$$l\text{-}\rho_{\mathcal{S}}(sc_2) * \sum_{\substack{j \in enabled(sc_1) \\ \wedge thr_1[j] \in LCom}} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1))$$
$$= l\text{-}\rho_{\mathcal{S}}(sc_1) * \sum_{\substack{k \in enabled(sc_2) \\ \wedge thr_2[k] \in LCom}} \rho^{\mathcal{S}}_{sc_2}(k) * P(succ_k(sc_2)).$$

That this equation is satisfied follows directly from the equalities derived in (8.) and in (10.), viz

$$\rho^{\mathcal{S}}_{sc_1}(j)/l\text{-}\rho_{\mathcal{S}}(sc_1) = \rho^{\mathcal{S}}_{sc_2}(k)/l\text{-}\rho_{\mathcal{S}}(sc_2),$$

and

$$P(succ_j(sc_1)) = P(succ_k(sc_2)),$$

where $k = l\text{-}match_{thr_1, thr_2}(j)$. This shows that $X = Y$, i.e., $P(sc_1) = P(sc_2)$. □

A.4 Security Type System for FSI-Security

This appendix contains the soundness proof of the security type system for FSI-security. The proof uses Lemmas 3.1, 3.2, and 3.3 from Section 3.4, and we start by proving these lemmas.

Proof of Lemma 3.1. By assumption, c is typable, i.e., $\vdash c : (mdf, stp)$ is derivable for some $mdf, stp \in \{low, high\}$. We proceed by induction over the derivation of this judgment.

– [STOP]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [STOP]. Then $c = stop$. Since *stop* cannot perform a transition, nothing has to be shown.

– [SKIP]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SKIP]. Then $c = \mathsf{skip}$, and, hence, $c' = stop$ and $thr = \langle\rangle$. Moreover, $mdf = high$ and $stp = low$. Hence, c' is typable with (mdf, stp) by rule [STOP].

– [ASS]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [ASS]. Then $c = x{:=}e$, and, hence, $c' = stop$ and $thr = \langle\rangle$. Moreover, $mdf = dma(x)$ and $stp = low$. Hence, c' is typable with type (mdf', stp) where $mdf \sqsubseteq mdf'$ (namely $mdf' = high$) by rule [STOP].

– [IF]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [IF]. In this case, $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ and, hence, $c' \in \{c_1, c_2\}$ and $thr = \langle\rangle$. Moreover, both $\vdash c_1 : (mdf, stp')$ and $\vdash c_2 : (mdf, stp')$ are derivable and $stp = stp' \sqcup d$ for some security domain d. I.e., both c_1 and c_2 are typable with (mdf, stp') with $stp' \sqsubseteq stp$.

– [WHILE]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [WHILE]. Then $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$ and, hence, $c' = \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}$ and $thr = \langle\rangle$. Moreover, $\vdash c_1 : (mdf, stp')$ is derivable with $stp = stp' \sqcup d$ and $stp' \sqcup d \sqsubseteq mdf$ for some security domain d.

  Since $stp' \sqsubseteq mdf$, c_1 is typable with type (mdf, stp'), and c is typable with type (mdf, stp), it follows that $c_1; c$ is typable with type (mdf, stp) by rule [SEQ]. Moreover, $stop$ is typable with (mdf, stp) by rule [STOP] and rule [SUB]. Hence, since $d \sqsubseteq mdf$, c' is typable with type (mdf, stp) by rule [IF].

– [SPAWN]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SPAWN]. Then $c = \mathsf{spawn}(c_1, \ldots, c_n)$. Hence, $c' = stop$ and $thr = \langle c_1, \ldots, c_n\rangle$. Moreover, c_i is typable for each $i \in \{1, \ldots, n\}$ due to the conditions of rule [SPAWN].

– [SEQ]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SEQ]. Then $c = c_1; c_2$, and c_1 and c_2 are typable with types (mdf_1, stp_1) and (mdf_2, stp_2), respectively, where $stp_1 \sqsubseteq mdf_2$, $mdf = mdf_1 \sqcap mdf_2$, and $stp = stp_1 \sqcup stp_2$.

  We distinguish between $c_1 = stop$ and $c_1 \neq stop$.

  If $c_1 = stop$ then $c' = c_2$, and, hence, c' is typable with (mdf_2, stp_2). Moreover,$mdf \sqsubseteq mdf_2$ and $stp_2 \sqsubseteq stp$ follow from $mdf = mdf_1 \sqcap mdf_2$ and $stp = stp_1 \sqcup stp_2$.

  If $c_1 \neq stop$ then $\langle c_1, mem\rangle \xrightarrow{\alpha}_{\triangleright} \langle c_1', mem'\rangle$ and $c' = c_1'; c_2$. Then, by the induction hypothesis for c_1, c_1' is typable with type (mdf_1', stp_1') where $mdf_1 \sqsubseteq mdf_1'$ and $stp_1' \sqsubseteq stp_1$. Hence, $c_1'; c_2$ is typable with type $(mdf_1' \sqcap mdf_2, stp_1' \sqcup stp_2)$ by rule [SEQ], and $mdf \sqsubseteq mdf_1' \sqcap mdf_2$ as well as $stp_1' \sqcup stp_2 \sqsubseteq stp$ follows from $mdf = mdf_1 \sqcap mdf_2$, and $stp = stp_1 \sqcup stp_2$.

– [SUB]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SUB]. In this case, c is typable with type (mdf', stp') where $mdf \sqsubseteq mdf'$ and $stp' \sqsubseteq stp$. Hence, by the induction hypothesis, c' is typable with (mdf'', stp'') where $mdf' \sqsubseteq mdf''$ and $stp'' \sqsubseteq stp'$. Moreover, the inequations $mdf \sqsubseteq mdf''$ and $stp'' \sqsubseteq stp$ follows from these inequations. □

Proof of Lemma 3.2. By assumption, $\vdash c : (high, stp)$ is derivable for some $stp \in \{low, high\}$. We prove the lemma by induction over the derivation of this judgment.

– [STOP]. Assume that $\vdash c : (high, stp)$ is derived with rule [STOP]. Then $c = stop$, and $stop \in HCom$ holds trivially.

– [SKIP]. Assume that $\vdash c : (high, stp)$ is derived with rule [SKIP]. Then $c = \mathsf{skip}$, and $\mathsf{skip} \in HCom$ holds trivially.

– [ASS]. Assume that $\vdash c : (high, stp)$ is derived with rule [ASS]. Then $c = x{:=}e$ and $dma(x) = high$. Hence, $c \in HCom$ by the definition of $HCom$ and the operational semantics of the programming language.

– [IF]. Assume that $\vdash c : (high, stp)$ is derived with rule [IF]. In this case, $c =$ if e then c_1 else c_2 fi and both c_1 and c_2 are typable with $(high, stp')$ for some stp'. Hence, by the induction hypothesis, both $c_1 \in HCom$ and $c_2 \in HCom$. Since one execution step of c either results in c_1 or in c_2 and does not modify the memory, $c \in HCom$ by the definition of high commands.

– [WHILE]. Assume that $\vdash c : (high, stp)$ is derived with rule [WHILE]. Then $c =$ while e do c_1 od and c_1 is typable with $(high, stp')$ for some $stp' \in \{low, high\}$. Hence, by the induction hypothesis, $c_1 \in HCom$. But then $c \in HCom$ by the operational semantics for while loops.

– [SPAWN]. Assume that $\vdash c : (high, stp)$ is derived with rule [SPAWN]. Then $c = \mathsf{spawn}(c_1, \ldots, c_k)$, and $\vdash c_i : (high, stp_i)$ is derivable for some stp_i for all $i \in \{1, \ldots, k\}$. Hence, all c_i are high commands by the induction hypothesis. Moreover, $\mathsf{spawn}(c_1, \ldots, c_k)$ makes a transition to $stop$. Hence, $\mathsf{spawn}(c_1, \ldots, c_k) \in HCom$.

– [SEQ]. Assume that $\vdash c : (high, stp)$ is derived with rule [SEQ]. Then $c = c_1; c_2$, and c_1 and c_2 are both typable with $(high, stp')$ for some stp'. Hence, by the induction hypothesis, both $c_1 \in HCom$ and $c_2 \in HCom$. By the operational semantics for sequential composition, the sequential composition of high commands is a high command, and, hence $c_1; c_2 \in HCom$.

– [SUB]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SUB]. Then $\vdash c : (mdf', stp')$ is derivable where $high \sqsubseteq mdf'$. In consequence, $mdf' = high$, and, hence, $c \in HCom$ by the induction hypothesis. □

Proof of Lemma 3.3. We start by proving the first statement of the lemma. Assume that $stp = low$. Then, by assumption, $\vdash c : (mdf, low)$ is typable for some $mdf \in \{low, high\}$. We prove the statement by induction over the derivation of this judgment.

– [STOP]. Assume that $\vdash c : (mdf, low)$ is derived with rule [STOP]. Then $c = stop$, for which the statement is trivially satisfied.

– [SKIP]. Assume that $\vdash c : (mdf, low)$ is derived with rule [SKIP]. Then $c =$ skip, for which the statement is trivially satisfied.

– [ASS]. Assume that $\vdash c : (mdf, low)$ is derived with rule [ASS]. Then $c = x{:=}e$ and $dma(y) \sqsubseteq dma(x)$ for each $y \in vars(e)$. In consequence, if $dma(x) = low$ then $eval(e, mem_1) = eval(e, mem_2)$ whenever $mem_1 =_L mem_2$. Hence, $mem_1' =_L mem_2'$ because c only modifies the value of x. The remaining statements of the lemma follow trivially.

– [IF]. Assume that $\vdash c : (mdf, low)$ is derived with rule [IF]. In this case, $c =$ if e then c_1 else c_2 fi and $\vdash e : low$. As the first execution step of a conditional does not modify the memory we have $mem_1' =_L mem_2'$. Moreover, as $mem_1 =_L mem_2$ and $\vdash e : low$ the expression e evaluates to the same value in mem_1 and mem_2. Hence, $c_1' = c_2'$.

– [WHILE]. Assume that $\vdash c : (mdf, low)$ is derived with rule [WHILE]. Then $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$ and $\vdash e : low$. As the first execution step of a while loop does not modify the memory we have $mem'_1 =_L mem'_2$. Moreover, as $mem_1 =_L mem_2$ and $\vdash e : low$ the expression e evaluates to the same value in mem_1 and mem_2. Hence, $c'_1 = c'_2$.

– [SPAWN]. Assume that $\vdash c : (mdf, low)$ is derived with rule [SPAWN]. Then $c = \mathsf{spawn}(c_1, \ldots, c_n)$. For spawn-commands, the statements from the lemma are trivial.

– [SEQ]. Assume that $\vdash c : (mdf, low)$ is derived with rule [SEQ]. Then $c = c_1; c_2$, and both c_1 and c_2 are typable with (mdf', low) for some mdf'. Hence, the statement for the lemma follow using the induction hypothesis for c_1 and the operational semantics for sequential composition.

– [SUB]. Assume that $\vdash c : (mdf, low)$ is derived with rule [SUB]. Then $\vdash c : (mdf', stp')$ is derivable where $stp' \sqsubseteq low$, i.e., $stp' = low$. Thus, we conclude using the induction hypothesis.

We now prove the second statement of the lemma by induction over the derivation of the typing of c.

– [STOP]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [STOP]. Then $c = stop$. Hence, $\langle c, mem_1\rangle \xrightarrow{\alpha_1} \langle c'_1, mem'_1\rangle$ is not derivable for any mem_1, α_1, c'_1, and mem'_1, which violates the assumptions of the lemma.

– [SKIP]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SKIP]. Then $c = \mathsf{skip}$. Hence, $c'_1 = c'_2 = stop$, i.e., the assumption of the statement, inequality of c'_1 and c'_2, is false.

– [ASS]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [ASS]. Then $c = x{:=}e$. Hence, $c'_1 = c'_2 = stop$, i.e., the assumption of the statement, inequality of c'_1 and c'_2, is false.

– [IF]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [IF]. In this case, $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ and $\vdash e : d$ for some d with $d \sqsubseteq stp$. Assume that $c'_1 \neq c'_2$. Then $eval(e, mem_1) \neq eval(e, mem_2)$ due to the semantics of conditionals. Hence, since $mem_1 =_L mem_2$, there must be some high variable on which the value of e depends, i.e., there exists $x \in vars(e) \cap H$. In consequence, $d = high$ since $\vdash e : d$ is derived using typing rule [EXP]. Hence, $stp = high$ since $d \sqsubseteq stp$.

– [WHILE]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [WHILE]. Then $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$ and $\vdash e : low$. Hence, $c'_1 = c'_2 = \mathsf{if}\ e\ \mathsf{then}\ c_1; \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}\ \mathsf{else}\ stop\ \mathsf{fi}$, i.e., the assumption of the statement, inequality of c'_1 and c'_2, is false.

– [SPAWN]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SPAWN]. Then $c = \mathsf{spawn}(c_1, \ldots, c_n)$. In consequence, $c'_1 = c'_2 = stop$. I.e., the assumption of the statement, inequality of c'_1 and c'_2, is false.

– [SEQ]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SEQ]. Then $c = c_1; c_2$, and c_1 is typable with (mdf_1, stp_1) for some mdf_1 and stp_1, where $stp_1 \sqsubseteq stp$. By the semantics of sequential composition, $c'_1 = c''_1; c_2$ and $c'_2 = c''_2; c_2$ where $\langle c_1, mem_1\rangle \xrightarrow{\alpha_1} \langle c''_1, mem'_1\rangle$ and $\langle c_1, mem_2\rangle \xrightarrow{\alpha_2} \langle c''_2, mem'_2\rangle$. From $c'_1 \neq c'_2$ it follows

that $c_1'' \neq c_2''$. Hence, by the induction hypothesis, $stp_1 = high$. It follows with $stp_1 \sqsubseteq stp$ that $stp = high$.

– [SUB]. Assume that $\vdash c : (mdf, stp)$ is derived with rule [SUB]. Then $\vdash c : (mdf', stp')$ is derivable where $stp' \sqsubseteq stp$. By the induction hypothesis, $stp' = high$. Hence, $stp = high$. □

Proof of Theorem 3.15. We must show that every typable command is FSI-secure. To this end, we define a relation $\mathcal{R}$ on thread pools such that $\langle c \rangle \mathcal{R} \langle c \rangle$ for every typable command c, and we show that $\mathcal{R}$ is a low bisimulation modulo low matching.

We define $thr_1 \mathcal{R} thr_2$ if and only if $\sharp(thr_1|_{LCom}) = \sharp(thr_2|_{LCom})$, $thr_1[i] = thr_2[j]$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$ with $thr_1[i] \in LCom$ and $j = l\text{-}match_{thr_1, thr_2}(i)$, and $thr_1[i]$ is typable for all $i \in \{1, \ldots, \sharp(thr_1)\}$ with $thr_1[i] \in LCom$.

By definition, $\mathcal{R}$ is symmetric and relates only thread pools with equally many low threads. Hence, to show that $\mathcal{R}$ is a low bisimulation modulo low matching it remains to show the following statement:

If $thr_1 \mathcal{R} thr_2$, $mem_1 =_L mem_2$, $k_1 \in \{1, \ldots, \sharp(thr_1)\}$, $\alpha_1 = new(thr_1') \in Lab$, $c_1' \in Com$, and $\langle thr_1[k_1], mem_1 \rangle \xrightarrow{\alpha_1} \langle c_1', mem_1' \rangle$, then the following conditions are satisfied:

1. if $thr_1[k_1] \in LCom$ then there exist $c_2' \in Com$, $mem_2' \in Mem$, and $\alpha_2 = new(thr_2') \in Lab$ with
   a) $\langle thr_2[k_2], mem_2 \rangle \xrightarrow{\alpha_2} \langle c_2', mem_2' \rangle$,
   b) $mem_1' =_L mem_2'$,
   c) $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, and
   d) $update_{k_1}(thr_1, c_1', \alpha_1) \mathcal{R} update_{k_2}(thr_2, c_2', \alpha_2)$

   where $k_2 = l\text{-}match_{thr_1, thr_2}(k_1)$; and
2. if $thr_1[k_1] \in HCom$, then $update_{k_1}(thr_1, c_1', \alpha_1) \mathcal{R} thr_2$.

If $thr_1[k_1] \in HCom$ then this statement is satisfied because execution steps of high threads do not modify the values of low variables, result in a high thread, and spawn only high threads (Theorem 3.2).

In the case that $thr_1[k_1] \in LCom$ it follows from $thr_1 \mathcal{R} thr_2$ that $thr_1[k_1]$ is typable, and we prove the statement by induction over the derivation of the corresponding judgment $\vdash thr_1[k_1] : (mdf, stp)$. Note that $mdf = low$ because otherwise $thr_1[k_1] \in HCom$ would hold by Lemma 3.2.

– [STOP]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [STOP]. Then $thr_1[k_1] = thr_2[k_2] = stop$. Hence, $\langle thr_1[k_1], mem_1 \rangle \xrightarrow{\alpha_1} \langle c_1', mem_1' \rangle$ cannot hold, and, hence, nothing needs to be shown.

– [SKIP]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [SKIP]. Then $thr_1[k_1] = thr_2[k_2] = \mathsf{skip}$. Hence, $c_1' = stop$, $thr_1' = \langle\rangle$ and $mem_1' = mem_1$. We define $c_2' = stop$, $thr_2' = \langle\rangle$, and $mem_2' = mem_2$. Then all the conditions we have to prove are satisfied.

– [ASS]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [ASS]. Then $thr_1[k_1] = thr_2[k_2] = x{:=}e$, $dma(x) = low$, and $dma(y) = low$ for all $y \in vars(e)$.

  Hence, $thr_1' = \langle\rangle$, $c_1' = stop$, and $mem_1' = mem_1[x \mapsto v_1]$ where $v_1 = eval(e, mem_1)$. Define $thr_2' = \langle\rangle$, $c_2' = stop$, and $mem_2' = mem_2[x \mapsto v_2]$ where $v_2 = eval(e, mem_2)$.

Then $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$. Moreover, since $dma(y) = low$ for all $y \in vars(e)$, $v_1 = v_2$. In consequence, $mem_1' =_L mem_2'$. Conditions (c) and (d) are evidently also satisfied.

– [IF]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [IF]. Then $thr_1[k_1] = thr_2[k_2] = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$, both $\vdash c_1 : (low, stp)$ and $\vdash c_2 : (low, stp)$ are derivable, and $\vdash e : low$.

Let $mem_1 =_L mem_2$. Because $\vdash e : low$ the expression e evaluates to the same value in mem_1 and mem_2, i.e., $eval(e, mem_1) = eval(e, mem_2)$. Let $v \in Val$ be this value.

Assume firstly that $v = \mathsf{true}$. Then $c_1' = c_1$, $thr_1' = \langle\rangle$, and $mem_1' = mem_1$. Define $c_2' = c_1$, $thr_2' = \langle\rangle$, and $mem_2' = mem_2$. Then $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$. Moreover, $mem_1' =_L mem_2'$ holds because $mem_1 =_L mem_2$. Condition (c) is trivially satisfied. Condition (d) is satisfied due to the derivability of $\vdash c_1 : (low, stp)$.

Assume now that $v = \mathsf{false}$. Then we conclude as in the case for $v = \mathsf{true}$, exploiting the derivability of $\vdash c_2 : (low, stp)$.

– [SPAWN]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [SPAWN]. Then $thr_1[k_1] = thr_2[k_2] = \mathsf{spawn}(c_1, \ldots, c_n)$, and for all $i \in \{1, \ldots, n\}$ there exists stp_i such that $\vdash c_i : (low, stp_i)$ is derivable.

Then $thr_1' = \langle c_1, \ldots, c_n\rangle$, $c_1' = stop$, and $mem_1' = mem_1$. Define $thr_2' = \langle c_1, \ldots, c_n\rangle$, $c_2' = stop$, and $mem_2' = mem_2$. Hence, $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$, and $mem_1' =_L mem_2'$ follows from $mem_1 =_L mem_2$. Condition (c) is trivially satisfied, and Condition (d) is satisfied because $\vdash c_i : (low, stp_i)$ is derivable for all i.

– [SEQ]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [SEQ]. Then $thr_1[k_1] = thr_2[k_2] = c_1; c_2$, and there are $mdf_1, mdf_2, stp_1, stp_2$ such that $low = mdf_1 \sqcap mdf_2$, $stp = stp_1 \sqcup stp_2$, $stp_1 \sqsubseteq mdf_2$, and both $\vdash c_1 : (mdf_1, stp_1)$ and $\vdash c_2 : (mdf_2, stp_2)$ are derivable.

We distinguish between the two cases $c_1 = stop$ and $c_1 \neq stop$.

If $c_1 = stop$ then $c_1' = c_2$, $thr_1' = \langle\rangle$, and $mem_1' = mem_1$. Define $c_2' = c_2$, $thr_2' = \langle\rangle$, and $mem_2' = mem_2$. Then Conditions (a), (b), and (c) are immediately satisfied. Moreover, Condition (d) is satisfied because $\vdash c_2 : (mdf_2, stp_2)$ is derivable.

Assume now that $c_1 \neq stop$. Then, by the operational semantics for sequential composition, there exists a command c_1'' and a thread pool thr_1' such that $\langle c_1, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1'', mem_1'\rangle$ with $\alpha_1 = new(thr_1')$ and such that $c_1' = c_1''; c_2$. Then, by the induction hypothesis for the derivation of $\vdash c_1 : (mdf_1, stp_1)$, there exists c_2'' and $\alpha_2 = new(thr_2')$ such that $\langle c_1, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2'', mem_2'\rangle$, $mem_1' =_L mem_2'$, $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, and $update_1(\langle c_1\rangle, c_1'', \alpha_1)\ \mathcal{R}\ update_1(\langle c_1\rangle, c_2'', \alpha_2)$ (i.e., $\langle c_1''\rangle :: thr_1'\ \mathcal{R}\ \langle c_2''\rangle :: thr_2'$).

Define $c_2' = c_2''; c_2$. Then $\langle thr_2[k_2], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$. Hence, Conditions (a)–(c) are satisfied.

It remains to show that Condition (d) is satisfied. Since $\langle c_1''\rangle :: thr_1'\ \mathcal{R}\ \langle c_2''\rangle :: thr_2'$ and $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$, by the definition of $\mathcal{R}$ this means that we must show that c_1' and c_2' are either both high commands, or that c_1' and c_2' are equal typable low commands.

Since c_1 is typable with (mdf_1, stp_1) and $\langle c_1, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1'', mem_1'\rangle$ it follows from Lemma 3.1 that c_1'' is typable with (mdf_1'', stp_1'') for some mdf_1'' and stp_1'' where

$stp_1'' \sqsubseteq stp_1$. Moreover, since $\langle c_1, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2'', mem_2'\rangle$ it follows from Lemma 3.1 that c_2'' is typable with (mdf_2'', stp_2'') for some mdf_2'' and stp_2'' where $stp_2'' \sqsubseteq stp_1$. Since $stp_1 \sqsubseteq mdf_2$ it follows that $stp_1'' \sqsubseteq mdf_2$ and $stp_2'' \sqsubseteq mdf_2$. Hence, both $c_1' = c_1''; c_2$ and $c_2' = c_2''; c_2$ are both typable by rule [SEQ].

Moreover, from $\langle c_1''\rangle :: thr_1' \mathrel{\mathcal{R}} \langle c_2''\rangle :: thr_2'$ and $\sharp(thr_1'|_{LCom}) = \sharp(thr_2'|_{LCom})$ it follows with the definition of $\mathcal{R}$ that either c_1'' and c_2'' are both high commands, or that they are equal typable low commands.

Consider firstly the case that c_1'' and c_2'' are both high commands. If $c_1'' = c_2''$ then $c_1' = c_2'$ by the definition of c_1' and c_2'. Hence, c_1' and c_2' are either both high or both low commands; i.e., c_1' and c_2' are either both high commands, or they are equal typable low commands. If $c_1'' \neq c_2''$ then, by Lemma 3.3, $stp_1 = high$. Hence, $mdf_2 = high$ since $stp_1 \sqsubseteq mdf_2$. But then c_2 is a high command by Lemma 3.2. In consequence, c_1' and c_2' are both high commands.

Consider now the case that c_1'' and c_2'' are equal typable low commands. Since they are equal, it follows that $c_1' = c_2'$, i.e., c_1' and c_2' are equal typable low commands.

– [WHILE]. Assume that $\vdash thr_1[k_1] : (low, stp)$ is derived with rule [WHILE]. Then $thr_1[k_1] = thr_2[k_2] = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$, $\vdash c_1 : (low, stp)$ is derivable, $\vdash e : d$, and $stp \sqcup d \sqsubseteq low$.

It follows that $stp = d = low$.

Moreover, it follows that $c_1' = \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}$, $thr_1' = \langle\rangle$, and $mem_1' = mem_1$. Define $c_2' = \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}$, $thr_2' = \langle\rangle$, and $mem_2' = mem_2$. Then $\langle c, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mem_2'\rangle$, and $mem_1' =_L mem_2'$ follows from $mem_1 =_L mem_2$. Condition (c) is trivially satisfied.

To prove Condition (d), it suffices to prove that $\mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}$ is typable. This can be seen as follows: The judgment $\vdash c_1; c : (low, low)$ is typable with rule [SEQ] because the judgments $\vdash c_1 : (low, stp)$ and $\vdash c : (low, stp)$ are derivable and $stp = low$. Hence, the judgment $\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi} : (low, stp)$ is derivable with rule [IF], using rule [STOP] for typing $stop$ and the fact that $\vdash e : d$ is derivable.

– [SUB]. Assume that $\vdash c : (low, stp)$ is derived with rule [SUB]. Then $\vdash c : (mdf', stp')$ is derivable where $low \sqsubseteq mdf'$ and $stp' \sqsubseteq stp$.

Then, by the induction hypothesis for the derivation of $\vdash c : (mdf', stp')$, Conditions (a)–(d) are all satisfied. □

A.5 Properties of Strong Low Bisimulation Modulo Modes

This appendix contains the proofs of some properties of strong low bisimulations modulo modes.

Proof of Theorem 4.1. We must show that $\sim_{mm}$ is symmetric, closed under globally consistent changes, and that whenever $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$ the following conditions are satisfied:

1. $mem_1 =_L^{mds} mem_2$,
2. for all $c_1' \in Com$, $mds' \in Mds$, $mem_1' \in Mem$, and $\alpha_1 = new(thr_1, mdss_1) \in Lab_m$, if $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$ then there exist $c_2' \in Com$, $mem_2' \in Mem$, and $\alpha_2 = new(thr_2, mdss_2) \in Lab_m$ such that the following conditions are satisfied:

a) $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2}\triangleright \langle c_2', mds', mem_2'\rangle$,
b) $\langle c_1', mds', mem_1'\rangle \sim_{mm} \langle c_2', mds', mem_2'\rangle$,
c) $\sharp(thr_1) = \sharp(thr_2)$, and
d) $\langle thr_1[i], mdss_1[i], mem_1'\rangle \sim_{mm} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

The relation $\sim_{mm}$ is symmetric because it is the union of symmetric relations.
Assume that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$. Then by definition of $\sim_{mm}$ there is a strong low bisimulation modulo modes $\mathcal{R}$ such that $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$.

Since $\mathcal{R}$ is closed under globally consistent changes,
- $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \mathcal{R} \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$ holds for all $v_1, v_2 \in \mathit{Val}$ whenever $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ and $dma(x) = \mathit{high}$, and
- $\langle c_1, mds, mem_1[x \mapsto v]\rangle \mathcal{R} \langle c_2, mds, mem_2[x \mapsto v]\rangle$ holds for all $v \in \mathit{Val}$ whenever $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ and $dma(x) = \mathit{low}$.

Hence, by definition of $\sim_{mm}$,
- $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \sim_{mm} \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$ holds for all $v_1, v_2 \in \mathit{Val}$ whenever $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ and $dma(x) = \mathit{high}$, and
- $\langle c_1, mds, mem_1[x \mapsto v]\rangle \sim_{mm} \langle c_2, mds, mem_2[x \mapsto v]\rangle$ holds for all $v \in \mathit{Val}$ whenever $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ and $dma(x) = \mathit{low}$.

Thus, $\sim_{mm}$ is closed under globally consistent changes.

Moreover, since $\mathcal{R}$ is a strong low bisimulation modulo modes, $mem_1 =_L mem_2$.

Assume that $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1}\triangleright \langle c_1', mds', mem_1'\rangle$. Since $\mathcal{R}$ is a strong low bisimulation modulo modes there exist $c_2' \in \mathit{Com}$, $mem_2' \in \mathit{Mem}$, and $\alpha_2 = new(thr_2, mdss_2) \in \mathit{Lab}_m$ such that the following hold:
1. $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2}\triangleright \langle c_2', mds', mem_2'\rangle$,
2. $\langle c_1', mds', mem_1'\rangle \mathcal{R} \langle c_2', mds', mem_2'\rangle$,
3. $\sharp(thr_1) = \sharp(thr_2)$, and
4. $\langle thr_1[i], mdss_1[i], mem_1'\rangle \mathcal{R} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

By the definition of $\sim_{mm}$ it follows that $\langle c_1', mds_1', mem_1'\rangle \sim_{mm} \langle c_2', mds_2', mem_2'\rangle$ and that $\langle thr_1[i], mdss_1[i], mem_1'\rangle \sim_{mm} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$. □

Proof of Theorem 4.2. Symmetry of the relation $\sim_{mm}$ follows directly from the definition of strong low bisimulations modulo modes.

To show transitivity of $\sim_{mm}$, we define a relation $\mathcal{R}$ on thread configurations with modes by $tcm_1 \mathcal{R} tcm_2$ if and only if there exists tcm' such that $tcm_1 \sim_{mm} tcm'$ and $tcm' \sim_{mm} tcm_2$. We show that $\mathcal{R}$ is a strong low bisimulation modulo modes, from which the transitivity of $\sim_{mm}$ follows with the definition of $\sim_{mm}$.

The symmetry of $\mathcal{R}$ follows from the definition of $\mathcal{R}$ and the symmetry of $\sim_{mm}$.

Assume that $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$. Then by definition of $\mathcal{R}$ there exist c' and mem' such that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c', mds, mem'\rangle$ and $\langle c', mds, mem'\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$.

We show that $\mathcal{R}$ is closed under globally consistent changes:
- Let $x \in L$ with $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ and let $v \in \mathit{Val}$. Then

$$\langle c_1, mds, mem_1[x \mapsto v]\rangle \sim_{mm} \langle c', mds, mem'[x \mapsto v]\rangle$$

and

$$\langle c', mds, mem'[x \mapsto v]\rangle \sim_{mm} \langle c_2, mds, mem_2[x \mapsto v]\rangle$$

because $\sim_{mm}$ is closed under globally consistent changes. Hence,

$$\langle c_1, mds, mem_1[x \mapsto v]\rangle \sim_{mm} \langle c_2, mds, mem_2[x \mapsto v]\rangle.$$

– Let $x \in H$ with $x \notin mds(asm\text{-}no\text{-}w)$ and let $v_1, v_2 \in Val$. Then

$$\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \sim_{mm} \langle c', mds, mem'\rangle$$

and

$$\langle c', mds, mem'\rangle \sim_{mm} \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$$

because $\sim_{mm}$ is closed under globally consistent changes. Hence,

$$\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \sim_{mm} \langle c_2, mds, mem_2[x \mapsto v_2]\rangle.$$

We now show that $mem_1 =_L^{mds} mem_2$. From $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c', mds, mem'\rangle$ and $\langle c', mds, mem'\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$ it follows that $mem_1 =_L^{mds} mem'$ and $mem' =_L^{mds} mem_2$, respectively. Hence, $mem_1 =_L^{mds} mem_2$.

Assume now that $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$. From $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c', mds, mem'\rangle$ it follows that there exist c'', mem'', and $\alpha' = new(thr', mdss')$ such that the following conditions are satisfied:

– $\langle c', mds, mem'\rangle \xrightarrow{\alpha'} \langle c'', mds', mem''\rangle$,
– $\langle c_1', mds', mem_1'\rangle \sim_{mm} \langle c'', mds', mem''\rangle$,
– $\sharp(thr_1) = \sharp(thr')$, and
– $\langle thr_1[i], mdss_1[i], mem_1'\rangle \sim_{mm} \langle thr'[i], mdss'[i], mem''\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

It follows with $\langle c', mds, mem'\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$ that there exist c_2', mem_2', and $\alpha_2 = new(thr_2, mdss_2)$ such that the following conditions are satisfied:

– $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
– $\langle c'', mds', mem''\rangle \sim_{mm} \langle c_2', mds', mem_2'\rangle$,
– $\sharp(thr') = \sharp(thr_2)$, and
– $\langle thr'[i], mdss'[i], mem''\rangle \sim_{mm} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr')\}$.

Hence, $\langle c_1', mds', mem_1'\rangle \mathcal{R} \langle c_2', mds', mem_2'\rangle$ and, for all $i \in \{1, \ldots, \sharp(thr_1)\}$, $\langle thr_1[i], mdss_1[i], mem_1'\rangle \sim_{mm} \langle thr_2[i], mdss_2[i], mem_2'\rangle$.

This concludes the proof that $\mathcal{R}$ is a strong low bisimulation modulo modes.

To show that $\sim_{mm}$ is transitive, assume that $tcm_1 \sim_{mm} tcm_2$ and $tcm_2 \sim_{mm} tcm_3$. Then $tcm_1 \mathcal{R} tcm_3$. Since $\mathcal{R}$ is a strong low bisimulation modulo modes, it follows that $tcm_1 \sim_{mm} tcm_3$. □

A.6 Compositionality and Scheduler-independence Result for SIFUM-security

This appendix contains the proofs of the lemmas and theorems that are used for proving compositionality and scheduler independence of SIFUM-security (Theorem 4.3).

We firstly establish a result on the possible transitions of a command that does not read any variable in a given set of variables $X \subseteq Var$. For this result and also in the remainder of this section we use the notation introduced in the following definition.

Definition A.1. Given two partial functions $f, g : X \rightharpoonup Y$ we write $f \leq g$ if and only if $f = g|_{dom(f)}$. Moreover, we write $f[g]$ for the function $f[x \mapsto g(x) \mid x \in dom(g)]$. ◊

Lemma A.11. Let $c, c' \in Com$, $mds, mds' \in Mds$, $mem, mem' \in Mem$, and $\alpha \in Lab_m$ such that $\langle c, mds, mem\rangle \xrightarrow{\alpha} \langle c', mds', mem'\rangle$. Let $X \subseteq Var$ such that c does not read x for all $x \in X$. Then there exists a family of partial functions $(g_f)_{f \in \{f : Var \rightharpoonup Val \mid dom(f) = X\}}$ with the following properties:

1. $g_f \leq f$ for all $f : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(f) = X$,
2. $\mathit{dom}(g_f) = \mathit{dom}(g_{f'})$ for all $f, f' : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(f) = X$, and
3. $\langle c, \mathit{mds}, \mathit{mem}[f]\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'[g_f]\rangle$ for all $f : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(f) = X$. $\Diamond$

Intuitively, the variables in the domain of the functions g_f are those variables in X that are not modified in the execution step, and the remaining variables in X are those that are modified in the execution step.

Proof. The proof is by induction on $\big|X\big|$.

Induction basis: Assume that $\big|X\big| = 0$. Then $X = \{\}$. There is exactly one partial function $f : \{\} \rightharpoonup \mathit{Val}$. This function has the property that $\mathit{mem}[f] = \mathit{mem}$ for all $\mathit{mem} \in \mathit{Mem}$. Define $g_f = f$. Then all properties stated by the lemma are satisfied.

Induction step: Assume that $\big|X\big| > 0$. Then there exists $x \in X$. Define $\overline{X} = X \setminus \{x\}$. Then $\big|\overline{X}\big| < \big|X\big|$. Let $(\overline{g}_{\overline{f}})_{\overline{f} \in \{f : \mathit{Var} \rightharpoonup \mathit{Val} \mid \mathit{dom}(f) = \overline{X}\}}$ be the family of partial functions that exists due to the induction hypothesis for $\overline{X}$.

Based on this family we now define the family $(g_f)_{f \in \{f : \mathit{Var} \rightharpoonup \mathit{Val} \mid \mathit{dom}(f) = X\}}$ and show that all properties stated by the lemma for this family are satisfied.

Let $f : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(f) = X$ and define $\overline{f} = f|_{\overline{X}}$. Then, by the properties of the functions $\overline{g}_{\overline{f}}$, $\langle c, \mathit{mds}, \mathit{mem}[\overline{f}]\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'[\overline{g}_{\overline{f}}]\rangle$. Since $x \in X$, c does not read x, and, hence, one of the following two conditions is satisfied:
(1) $\forall v \in \mathit{Val} : \langle c, \mathit{mem}[\overline{f}][x \mapsto v]\rangle \xrightarrow{\alpha} \langle c', \mathit{mem}'[\overline{g}_{\overline{f}}][x \mapsto v]\rangle$, or
(2) $\forall v \in \mathit{Val} : \langle c, \mathit{mem}[\overline{f}][x \mapsto v]\rangle \xrightarrow{\alpha} \langle c', \mathit{mem}'[\overline{g}_{\overline{f}}]\rangle$.

If Condition (2) is satisfied and Condition (1) is not satisfied then we define $g_f = \overline{g}_{\overline{f}}$. Moreover, if Condition (1) is satisfied we define g_f as the partial function with $\mathit{dom}(g_f) = \mathit{dom}(\overline{g}_{\overline{f}}) \cup \{x\}$, $g_f(\overline{x}) = \overline{g}_{\overline{f}}(\overline{x})$ for $\overline{x} \in \overline{X}$, and $g_f(x) = f(x)$.

Since $\mathit{mem}[f] = \mathit{mem}[\overline{f}][x \mapsto f(x)]$, if Condition (1) is satisfied it follows that $\langle c, \mathit{mem}[f]\rangle \xrightarrow{\alpha} \langle c', \mathit{mem}'[\overline{g}_{\overline{f}}][x \mapsto f(x)]\rangle = \langle c', \mathit{mem}'[g_f]\rangle$. Moreover, if Condition (2) is satisfied it follows that $\langle c, \mathit{mem}[f]\rangle \xrightarrow{\alpha} \langle c', \mathit{mem}'[\overline{g}_{\overline{f}}]\rangle = \langle c', \mathit{mem}'[g_f]\rangle$. Hence, in both cases $\langle c, \mathit{mds}, \mathit{mem}[f]\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'[g_f]\rangle$, i.e., property (3) required by the lemma is satisfied.

From the properties of the functions $\overline{g}_{\overline{f}}$ it follows that $\overline{g}_{\overline{f}} \leq \overline{f}$ for all $\overline{f} : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(\overline{f}) = \overline{X}$, it follows from the construction of the family $(g_f)_{f : X \rightharpoonup \mathit{Val}}$ that $g_f \leq f$ for all $f : X \rightharpoonup \mathit{Val}$ (because either $x \notin \mathit{dom}(g_f)$ or $g_f(x) = f(x)$). I.e., Property (1) required by the lemma is satisfied.

Let $f, f' : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(f) = \mathit{dom}(f') = X$, and assume that Condition (1) is not satisfied for $\overline{f} = f|_{\overline{X}}$ (i.e., Condition (2) is satisfied for $\overline{f}$), and that Condition (1) is satisfied for $\overline{f}' = f'|_{\overline{X}}$. Assume that there are $\mathit{mem}', \mathit{mem}$ with $\mathit{mem}'(x) \neq \mathit{mem}(x)$ and $\langle c, \mathit{mds}, \mathit{mem}\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'\rangle$. By Condition (1) for $\overline{f}'$ it follows that $\langle c, \mathit{mem}[\overline{f}]\rangle \xrightarrow{\alpha} \langle c', \mathit{mem}'[\overline{g}_{\overline{f}}][x \mapsto \mathit{mem}(x)]\rangle$. By Condition (2) for $\overline{f}$ it follows that $\langle c, \mathit{mem}[\overline{f}]\rangle \xrightarrow{\alpha} \langle c', \mathit{mem}'[\overline{g}_{\overline{f}}]\rangle$. But this contradicts that $\langle c, \mathit{mds}, \mathit{mem}[\overline{f}]\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'[\overline{g}_{\overline{f}}]\rangle$ and $\langle c, \mathit{mds}, \mathit{mem}[\overline{f}']\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'[\overline{g}_{\overline{f}'}]\rangle$ by the properties of the functions $\overline{g}_{\overline{f}}$. Hence, if $\langle c, \mathit{mds}, \mathit{mem}\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'\rangle$ then $\mathit{mem}'(x) = \mathit{mem}(x)$. But then Condition (1) is satisfied for $\overline{f}$.

Hence, for $f, f' : \mathit{Var} \rightharpoonup \mathit{Val}$ with $\mathit{dom}(f) = \mathit{dom}(f') = X$ the same case in the construction of g_f applies, and, hence, it follows from the construction of g_f and the

properties of the functions $\overline{g}_{\overline{f}}$ that $dom(g_f) = dom(g_{f'})$ for all $f, f' : Var \rightharpoonup Val$ with $dom(f) = X$. I.e., Property (2) required by the lemma is satisfied. □

Proof of Lemma 4.1. Since the lists of memories $mems_1$ and $mems_2$ make the thread configurations $\langle thr_1, mdss, mem_1\rangle$ and $\langle thr_2, mdss, mem_2\rangle$ compatible with strong low bisimulation modulo modes, Items (1)–(3) from Definition 4.18 hold for $mems_1$, $mems_2$, $\langle thr_1, mdss, mem_1\rangle$, and $\langle thr_2, mdss, mem_2\rangle$.

Let n be the length of thr_1 and thr_2 (i.e., $n = \sharp(thr_1)$, which by assumption equals $\sharp(thr_2)$). If $n = 0$ then there is nothing to prove, since Lemma 4.1 makes a statement about all $i \in \{1, \ldots, \sharp(thr_1)\}$. Assume hence that $n \neq 0$. Let $i \in \{1, \ldots, n\}$ and $x \in X_i$. Since $x \in X_i$, by Item (2) in Definition 4.18 it follows that

(a) $x \in L$ and

(b) $mem_1(x) \neq mem_2(x)$.

Moreover, since $n \neq 0$, by Item (3) in Definition 4.18 there exists $j \in \{1, \ldots, n\}$ such that $x \notin X_j$ (where $j \neq i$ because $x \in X_i$). By Item (1) in Definition 4.18 the following holds for all $f : Var \rightharpoonup Val$ with $dom(f) = X_j$:

(c) $\langle thr_1[j], mdss[j], mems_1[j][x \mapsto f(x) \mid x \in X_j]\rangle \sim_{mm} \langle thr_2[j], mdss[j], mems_2[j][x \mapsto f(x) \mid x \in X_j]\rangle$

I.e., using the more compact notation from Definition A.1:

(d) $\langle thr_1[j], mdss[j], mems_1[j][f]\rangle \sim_{mm} \langle thr_2[j], mdss[j], mems_2[j][f]\rangle$

Let $f : Var \rightharpoonup Val$ be a partial function with $dom(f) = X_j$. It follows from (d) and the definition of strong low bisimulations modulo modes that

(e) $mems_1[j][f] =_L^{mdss[j]} mems_2[j][f]$.

It follows from $x \notin X_j$ and $dom(f) = X_j$ that

(f) $mems_1[j][f](x) = mems_1[j](x)$ and $mems_2[j][f](x) = mems_2[j](x)$.

Assume that $x \notin mdss[j](\textit{asm-no-r})$. Then it follows with $x \notin X_j$, $x \in L$ (see (a)), (e), and (f) that $mems_1[j](x) = mems_2[j](x)$. Since $x \notin X_j$ we have $mems_1[j](x) = mem_1(x) \wedge mems_2[j](x) = mem_2(x)$, and, hence, it follows that $mem_1(x) = mem_2(x)$. But $mem_1(x) \neq mem_2(x)$ (see (b)), and, hence, our assumption that $x \notin mdss[j](\textit{asm-no-r})$ must be wrong. Hence,

(g) $x \in mdss[j](\textit{asm-no-r})$.

The list of mode states $mdss$ has compatible modes (cf. Definition 4.10) because $(thr_1, mdss)$ has sound modes by assumption. Hence,

(h) $x \in mdss[i](\textit{guar-no-r})$

follows from (g). Moreover, the configuration $\langle thr_1[i], mdss[i], mem_1\rangle$ respects guarantees because $\langle thr_1, mdss, mem_1, sst\rangle$ has sound modes. Hence, due to (h), $thr_1[i]$ does not read x. With the same argument it follows that $thr_2[i]$ does not read x because $(thr_2, mdss)$ has sound modes. □

Proof of Lemma 4.2. Let n be the length of thr_1 and thr_2. Define $X_i = \{x \in Var \mid mems_1[i](x) \neq mem_1(x) \vee mems_2[i](x) \neq mem_2(x)\}$ for all $i \in \{1, \ldots, n\}$. Then the following holds by Lemma 4.1:

(a) For all $i \in \{1, \ldots, n\}$, if $x \in X_i$ then $thr_1[i]$ and $thr_2[i]$ do not read x.

We now show that there exist thr_2' with $\sharp(thr_1') = \sharp(thr_2')$ and mem_2' such that $\langle thr_2, mdss, mem_2, sst\rangle \stackrel{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr_2', mdss', mem_2', sst'\rangle$. Since

$$\langle thr_1, mdss, mem_1, sst\rangle \stackrel{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr_1', mdss', mem_1', sst'\rangle$$

there exist $c_1' \in Com$, $mds' \in Mds$, and $\alpha_1 = new(thr_1'', mdss'') \in Lab_m$ with
(b1) $\langle thr_1[k], mdss[k], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$,
(b2) $thr_1' = update_k(thr_1, c_1', thr_1'')$,
(b3) $mdss' = update_k(mdss, mds', mdss'')$, and
(b4) $(sst, sin) \stackrel{k,p}{\rightsquigarrow}_{\mathcal{S}} sst'$ where $sin = obs(thr_1, mem_1)$.

Due to (a) $thr_1[k]$ does not read x if $x \in X_k$. Hence, by (b1) and Lemma A.11 there exists a family of partial functions $(g_f^1)_{f \in \{f: Var \rightharpoonup Val \mid dom(f) = X_k\}}$ such that
(c1) $g_f^1 \leq f$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X_k$,
(c2) $dom(g_f^1) = dom(g_{f'}^1)$ for all $f, f' : Var \rightharpoonup Val$ with $dom(f) = dom(f') = X_k$, and
(c3) $\langle thr_1[k], mdss[k], mem_1[f]\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'[g_f^1]\rangle$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X_k$.

By the definition of X_k, $mems_1[k][f] = mem_1[f]$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X_k$, and, hence, by (c3) $\langle thr_1[k], mdss[k], mems_1[k][f]\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'[g_f^1]\rangle$ for all such f. Moreover, $\langle thr_1[k], mdss[k], mems_1[k][f]\rangle \sim_{mm} \langle thr_2[k], mdss[k], mems_2[k][f]\rangle$ for all such f follows from Item 3 in Definition 4.18. Hence, by the definition of strong low bisimulations modulo modes for all such f there exist c_f, $\alpha_f = new(thr_f, mdss_f)$, and mem_f such that
(d1) $\langle thr_2[k], mdss[k], mems_2[k][f]\rangle \xrightarrow{\alpha_f} \langle c_f, mds', mem_f\rangle$,
(d2) $\langle c_1', mds', mem_1'[g_f^1]\rangle \sim_{mm} \langle c_f, mds', mem_f\rangle$,
(d3) $\sharp(thr_1'') = \sharp(thr_f)$, and
(d4) $\langle thr_1''[j], mdss_2''[j], mem_1'[g_f^1]\rangle \sim_{mm} \langle thr_f[j], mdss_2''[j], mem_f\rangle$ for all $j \in \{1, \ldots, \sharp(thr_1'')\}$.

The set $\{f : Var \rightharpoonup Val \mid dom(f) = X_k\}$ is non-empty, let h be an element of the set. Due to (a) $thr_2[k]$ does not read x if $x \in X_k$. Hence, by (d1) and Lemma A.11 there exists a family of partial functions $(g_f^2)_{f \in \{f: Var \rightharpoonup Val \mid dom(f) = X_k\}}$ such that
(e1) $g_f^2 \leq f$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X_k$,
(e2) $dom(g_f^2) = dom(g_{f'}^2)$ for all $f, f' : Var \rightharpoonup Val$ with $dom(f) = dom(f') = X_k$, and
(e3) $\langle thr_2[k], mdss[k], mems_2[k][h][f]\rangle \xrightarrow{\alpha_h} \langle c_h, mds', mem_h[g_f^2]\rangle$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X_k$.

Since $mem[h][f] = mem[f]$ for all memories mem and all partial functions f and h with $dom(f) = dom(h) = X_k$ it follows from (e3) that
(f) $\langle thr_2[k], mdss[k], mems_2[k][f]\rangle \xrightarrow{\alpha_h} \langle c_h, mds', mem_h[g_f^2]\rangle$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X_k$.

Hence, it follows from (d1), (f), and Requirement 2.2 (determinism of transitions of commands) that $c_h = c_f$ and $\alpha_f = \alpha_h$ for all $f, h : Var \rightharpoonup Val$ with $dom(f) = dom(h) = X_k$. We denote this command and this label with c_2' and $\alpha_2 = new(thr_2'', mdss_2')$ in the following. Moreover, it follows that $mem_f = mem_h[g_f^2]$, and we can hence rewrite (d4) as follows:
(g) $\langle c_1', mds', mem_1'[g_f^1]\rangle \sim_{mm} \langle c_2', mds', mem_h[g_f^2]\rangle$

We define
(h) $mem_2' = mem_h[g^2_{mem_2|_{X_k}}]$.

It follows from the definition of X_k that $mem_2 = mems_2[k][mem_2|_{X_k}]$. Hence, by (f) and (h),
(i) $\langle thr_2[k], mdss[k], mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$.

Since $\langle thr_1, mdss, mem_1, sst\rangle$ has sound modes it follows that $\langle thr_1[k], mdss[k], mem_1\rangle$ is compatible with obs. Hence, since $\mathcal{S}$ is confined to L, it follows from $\sharp(thr_1) = \sharp(thr_2)$ and (a) that $obs(thr_1, mem_1) = obs(thr_2, mem_2)$. Hence, it follows from (b4) that $(sst, sin) \overset{k,p}{\rightsquigarrow}_{\mathcal{S}} sst'$ for $sin = obs(thr_2, mem_2)$. We define $thr_2' = update_k(thr_2, c_2', thr_2'')$. Then $\langle thr_2, mdss, mem_2, sst\rangle \overset{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr_2', mdss', mem_2', sst'\rangle$ by (i) and the definition of mem_2' in (h). Moreover, $\sharp(thr_1') = \sharp(thr_2')$ by (d3), (d2), and the fact that configurations with command *stop* are only related to configurations with command *stop* by $\sim_{mm}$. We denote $\sharp(thr_1')$ by n'. Moreover, we define $X_i' = \{x \in Var \mid mems_1'[i](x) \neq mem_1'(x) \vee mems_2'[i](x) \neq mem_2'(x)\}$ for all $i \in \{1, \ldots, n'\}$.

Note that due to (c2) and (e2) neither the domain of g^1_f nor the domain of g^2_f depends on f. Hence, we may omit the subscript, and we denote these two domains with $dom(g^1)$ and $dom(g^2)$ in the following.

It remains to define $mems_1'$ and $mems_2'$ and to show that they make $\langle thr_1', mds', mem_1'\rangle$ and $\langle thr_2', mds', mem_2'\rangle$ compatible with strong low bisimulation modulo modes. We firstly define the lists of memories and show that Items (1) and (2) in Definition 4.18 are satisfied for $mem_1', mem_2', mems_1'$, and $mems_2'$, distinguishing three cases: (a) that $i = k$ is the index of the thread that performed the execution step and did not terminate after the execution step, (b) that i is the index of a spawned thread, and (c) that i is the index of some other thread. (We show that Item (3) in Definition 4.18 is also satisfied for $mem_1', mem_2', mems_1'$, and $mems_2'$ afterwards.)

We consider firstly the thread index k in case that the thread at this index did not terminate in the considered execution step, i.e., $c_1' \neq stop$.

We define $mems_1'[k]$ and $mems_2'[k]$ as follows:
(j1) If $x \notin X_k$ then $mems_1'[k](x) = mem_1'(x)$ and $mems_2'[k](x) = mem_2'(x)$.
(j2) If $x \in X_k$ and $x \notin dom(g^1) \vee x \notin dom(g^2)$ then $mems_1'[k](x) = mem_1'(x)$ and $mems_2'[k](x) = mem_2'(x)$.
(j3) If $x \in X_k$ and $x \in dom(g^1) \wedge x \in dom(g^2)$ then $mems_1'[k](x) = mems_1[k](x)$ and $mems_2'[k](x) = mems_2[k](x)$.

We now show that Item (2) in Definition 4.18 is satisfied for $i = k$, i.e., that $[mem_1'(x) = mem_2'(x) \vee x \in H] \Rightarrow x \notin X_k'$ for all $x \in Var$.
- If $x \notin X_k$, if $x \notin dom(g^1)$, or if $x \notin dom(g^2)$ then $x \notin X_k'$ by the definition of $mems_1'[k]$ and $mems_2'[k]$ in (j1) and (j2).
- Assume that $x \in X_k$, $x \in dom(g^1)$ and $x \in dom(g^2)$. We show that this leads to a contradiction if $[mem_1'(x) = mem_2'(x) \vee x \in H]$ is satisfied.
  Setting $f = mem_1|_{X_k}$ in (c3) $\langle thr_1[k], mdss[k], mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'[g^1_{mem_1|_{X_k}}]\rangle$ holds. Then, due to (b1) and Requirement 2.2, $mem_1' = mem_1'[g^1_{mem_1|_{X_k}}]$. In conse-

quence, $mem'_1(x) = mem_1(x)$ because $x \in dom(g^1)$.
Setting $f = mem_2|_{X_k}$ in (f) $\langle thr_1[k], mdss[k], mem_2\rangle \xrightarrow{\alpha_2} \langle c'_2, mds', mem'_2[g^2_{mem_2|_{X_k}}]\rangle$ holds. Then, due to (i) and Requirement 2.2, $mem'_2 = mem'_2[g^2_{mem_2|_{X_k}}]$. In consequence, $mem'_2(x) = mem_2(x)$ because $x \in dom(g^2)$.
With $mem'_1(x) = mem'_2(x)$ it follows from $mem'_1(x) = mem_1(x)$ and $mem'_2(x) = mem_2(x)$ that $mem_1(x) = mem_2(x)$. I.e., if $mem'_1(x) = mem'_2(x) \vee x \in H$ then $mem_1(x) = mem_2(x) \vee x \in H$. Hence, it follows from Item (2) in Definition 4.18 for $mem_1, mem_2, mems_1$, and $mems_2$ that $x \notin X_k$, a contradiction to $x \in X_k$.

We now show that Item (1) in Definition 4.18 is satisfied for $i = k$, i.e., we show that $\langle thr'_1[k], mdss'[k], mems'_1[k][f]\rangle \sim_{mm} \langle thr'_2[k], mdss'[k], mems'_2[k][f]\rangle$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X'_k$. Let such a function f be given. We lead the proof by defining a function $f^* : Var \rightharpoonup Val$ with $dom(f^*) = X_k$ such that $mems'_1[k][f] = mem'_1[g^1_{f^*}]$ and $mems'_2[k][f] = mem_h[g^2_{f^*}]$, from which the statement we have to prove follows from (g).

We define the function $f^* : Var \rightharpoonup Val$ with $dom(f^*) = X_k$ as follows, where $v \in Val$ is an arbitrary value:

$$f^*(x) = \begin{cases} f(x) & x \in X'_k \\ mems'_1[k](x) & x \notin X'_k \ \wedge \ x \in dom(g^1) \\ mems'_2[k](x) & x \notin X'_k \ \wedge \ x \in dom(g^2) \\ v & x \in X_k \setminus (X'_k \cup dom(g^1) \cup dom(g^2)) \end{cases}$$

The function f^* is well-defined because if $x \in dom(f^*)$ and $x \notin X'_k$ then x cannot be contained in both $dom(g^1)$ and $dom(g^2)$, and, hence, for each $x \in dom(f^*)$ exactly one of the four cases in the definition of f^* applies. To show this, assume that $x \in dom(f^*)$, $x \notin X'_k$, and $x \in dom(g^1) \cap dom(g^2)$. As shown above (when proving that Item (2) in Definition 4.18 is satisfied for $i = k$), it follows that $mem'_1(x) = mem_1(x)$ and $mem'_2(x) = mem_2(x)$. Moreover, by (j3), $mems'_1[k](x) = mems_1[k](x)$ and $mems'_2[k](x) = mems_2[k](x)$. But then it follows with $x \notin X'_k$ that $x \notin X_k$, which contradicts that $x \in dom(f^*) = X_k$.

We now show that $mems'_1[k][f] = mem'_1[g^1_{f^*}]$. Assume firstly that $x \in X'_k$. Then, since $dom(f) = X'_k$, $mems'_1[k][f](x) = f(x)$, and, hence, $mems'_1[k][f](x) = f^*(x)$ by the definition of f^*. Moreover, by $x \in X'_k$ and (j1)–(j3) it follows that $x \in dom(g^1)$. Hence, $g^1_{f^*}(x) = f^*(x)$ by (c1) and, in consequence, $f^*(x) = mem'_1[g^1_{f^*}](x)$. Thus, $mems'_1[k][f](x) = mem'_1[g^1_{f^*}](x)$. Assume secondly that $x \notin X'_k$. Then, since $dom(f) = X'_k$, $mems'_1[k][f](x) = mems'_1[k](x)$. If $x \in dom(g^1)$ then $mems'_1[k](x) = f^*(x)$ by the definition of f^*, and, hence, $mems'_1[k](x) = g^1_{f^*}(x)$ by (c1). Moreover, if $x \notin dom(g^1)$ then $mems'_1[k](x) = mem'_1(x)$ by (j1) and (j2). Hence, $mems'_1[k](x) = mem'_1[g^1_{f^*}](x)$. In consequence,
(k) $mems'_1[k][f] = mem'_1[g^1_{f^*}]$.

We now show that $mems'_2[k][f] = mem_h[g^2_{f^*}]$. Assume firstly that $x \in X'_k$. Then, since $dom(f) = X'_k$, $mems'_2[k][f](x) = f(x)$, and, hence, $mems'_2[k][f](x) = f^*(x)$ by the definition of f^*. Moreover, by $x \in X'_k$ and (j1)–(j3) it follows that $x \in dom(g^2)$. Hence, $g^2_{f^*}(x) = f^*(x)$ by (e1) and, in consequence, $f^*(x) = mem_h[g^2_{f^*}](x)$. Thus, $mems'_2[k][f] = mem_h[g^2_{f^*}](x)$. Assume secondly that $x \notin X'_k$. Then, since $dom(f) = X'_k$, $mems'_2[k][f](x) = mems'_2[k](x)$. If $x \in dom(g^2)$ then $mems'_2[k](x) = f^*(x)$ by the

definition of f^*, and, hence, $mems'_2[k](x) = g^2_{f^*}(x)$ by (e1). Moreover, if $x \notin dom(g^2)$ then $mems'_2[k](x) = mem'_2(x)$ by (j1) and (j2). Hence, $mems'_2[k](x) = mem'_2[g^2_{f^*}](x)$. By the definition of mem'_2, it follows that $mems'_2[k](x) = mem_h[g^2_{mem_2|_{X_k}}][g^2_{f^*}](x)$, which equals $mem_h[g^2_{f^*}](x)$. In consequence,

(l) $mems'_2[k][f] = mem_h[g^2_{f^*}]$.

That $\langle c'_1, mds', mem'_1[f]\rangle \sim_{mm} \langle c'_2, mds', mem'_2[f]\rangle$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X'_k$ follows now from (g), (k), and (l). Since $thr'_1[k] = c'_1$, $thr'_2[k] = c'_2$, and $mdss'[k] = mds'$, it follows that

$$\langle thr'_1[k], mdss'[k], mem'_1[f]\rangle \sim_{mm} \langle thr'_2[k], mdss'[k], mem'_2[f]\rangle$$

for all $f : Var \rightharpoonup Val$ with $dom(f) = X'_k$.

Now we consider the case that i is the index of a thread that was created in the execution step.

Let $s = \sharp(thr''_1) - r$, where $r = 0$ if $c'_1 \neq stop$ and $r = 1$ if $c'_1 = stop$. That is, s is the number by which the thread index of a thread with an index $i > k$ before the execution steps from thr_1 to thr'_1 and thr_2 to thr'_2, respectively, differs from its new index after the execution step. We consider $i \in \{k + 1 - r, \ldots, k + \sharp(thr''_1) - r\}$, i.e., i is the index of a thread that was spawned in the execution step. We define $mems'_1[i]$ and $mem'_2[i]$ exactly as the memories $mems'_1[k]$ and $mems'_2[k]$ above. That Item (2) in Definition 4.18 is satisfied follows as in the case for $i = k$ above. That Item (1) in Definition 4.18 is satisfied, i.e., that $\langle thr'_1[i], mdss'_2[i], mems'_1[i][f]\rangle \sim_{mm} \langle thr'_2[i], mdss'_2[i], mems'_2[i][f]\rangle$ is satisfied for all $f : Var \rightharpoonup Val$ with $dom(f) = X'_i$ follows with the same argumentation as in the case for $i = k$ above, exploiting (d4) instead of (g) in the argument.

We finally consider i such that $i < k$ or $i > k + s$, i.e., i is the index of a thread that was in the thread pool before the execution step considered above and did not perform that execution step. Without loss of generality, we assume that $i < k$ (for $i > k + s$ the remaining argumentation is the same but for shifts of the index by s).

We define $mems'_1[i]$ and $mems'_2[i]$ as follows:

(m1) If $mem_1(x) \neq mem'_1(x) \vee mem_2(x) \neq mem'_2(x)$ and $mem'_1(x) = mem'_2(x) \vee x \in H$ then $mems'_1[i](x) = mem'_1(x)$ and $mem'_2[i](x) = mem'_2(x)$.

(m2) If $mem_1(x) \neq mem'_1(x) \vee mem_2(x) \neq mem'_2(x)$ and $mem'_1(x) \neq mem'_2(x) \ \wedge \ x \in L$ then $mems'_1[i](x) = mem'_2[i](x) = v$ for some $v \in Val$.

(m3) If $mem_1(x) = mem'_1(x) \ \wedge \ mem_2(x) = mem'_2(x)$ then $mems'_1[i](x) = mems_1[i](x)$ and $mem'_2[i](x) = mems_2[i](x)$.

We show that Item (2) in Definition 4.18 is satisfied. The proof is by contraposition, i.e., assuming $x \in X'_i$ we show that $mem'_1(x) \neq mem'_2(x)$ and $x \in L$. If $x \in X'_i$ the definition of $mems'_1[i](x)$ and $mem'_2[i](x)$ is by (m2) or by (m3), because otherwise $x \notin X'_i$ holds. If the definition is by (m2) then $mem'_1(x) \neq mem'_2(x)$ and $x \in L$. Assume now that the definition is by (m3). From $x \in X'_i$ it follows that $mems'_1[i](x) \neq mem'_1(x) \vee mem'_2[i](x) \neq mem'_2(x)$. If $mems'_1[i](x) \neq mem'_1(x)$ then $mems_1[i](x) \neq mem_1(x)$ because by (m3) $mem_1(x) = mem'_1(x)$ and $mems_1[i](x) = mems'_1[i](x)$. Moreover, if $mems_2[i]'(x) \neq mem'_2(x)$ then $mems_2[i](x) \neq mem_2(x)$ because by (m3) $mem_1(x) = mem'_1(x)$ and $mems_1[i](x) = mems'_1[i](x)$. Hence, $x \in X_i$. Then, because $mems_1$ and $mems_2$ make $\langle thr_1, mdss_1, mem_1\rangle$ and $\langle thr_2, mdss_2, mem_2\rangle$ compatible with strong low bisimulation

modulo modes, $x \in L \ \wedge \ mem_1(x) \neq mem_2(x)$. But then, because by (m3) $mem_1(x) = mem'_1(x) \ \wedge \ mem_2(x) = mem'_2(x)$, it follows that $x \in L \ \wedge \ mem'_1(x) \neq mem'_2(x)$.

We now show that Item (1) in Definition 4.18 is satisfied, i.e.,

$$\langle thr'_1[i], mdss'[i], mems'_1[i][f] \rangle \sim_{mm} \langle thr'_2[i], mdss'[i], mems'_2[i][f] \rangle$$

holds for all $f : Var \rightharpoonup Val$ with $dom(f) = X'_i$. By (b2), (b3), and the definition of thr'_2 this means that we have to show that $\langle thr_1[i], mdss[i], mems'_1[i][f] \rangle \sim_{mm} \langle thr_2[i], mdss[i], mem'_2[i][f] \rangle$ for all $f : Var \rightharpoonup Val$ with $dom(f) = X'_i$. Let such a function f be given.

We define a function $f^* : Var \rightharpoonup Val$ with $dom(f^*) = X_i$ as follows:
(n1) If $x \in X_i \cap X'_i$ then $f^*(x) = f(x)$.
(n2) If $x \in X_i \setminus X'_i$ then $f^*(x) = mem'_1(x)$

We show that if $mems'_1[i][f](x) \neq mems_1[i][f^*](x)$ or if $mem'_2[i][f](x) \neq mems_2[i][f^*](x)$ for $x \in Var$ then $mem'_1(x) \neq mem_1(x)$ or $mem'_2(x) \neq mem_2(x)$. The proof is by contraposition. Assume that $mem'_1(x) = mem_1(x)$ and $mem'_2(x) = mem_2(x)$. Then the definition of $mems'_1[i](x)$ and $mem'_2[i](x)$ is by (m3) because (m1) and (m2) require that $mem_1(x) \neq mem'_1(x) \vee mem_2(x) \neq mem'_2(x)$. I.e., $mems'_1[i](x) = mems_1[i](x)$ and $mem'_2[i](x) = mems_2[i](x)$.

We distinguish the cases $x \in X'_i$ and $x \notin X'_i$. Assume firstly that $x \in X'_i$, i.e., $mems'_1[i](x) \neq mem'_1(x) \vee mem'_2[i](x) \neq mem'_2(x)$. With $mem_1(x) = mem'_1(x)$, $mem_2(x) = mem'_2(x)$, $mems'_1[i](x) = mems_1[i](x)$ and $mem'_2[i](x) = mems_2[i](x)$ it follows that $mems_1[i](x) \neq mem_1(x) \vee mems_2[i](x) \neq mem_2(x)$, i.e., $x \in X_i$. Hence, $mems'_1[i][f](x) = mems_1[i][f^*](x) \ \wedge \ mem'_2[i][f](x) = mems_2[i][f^*](x)$ by (n1).

Assume now that $x \notin X'_i$, i.e., $mems'_1[i](x) = mem'_1(x)$ and $mem'_2[i](x) = mem'_2(x)$. With $mem_1(x) = mem'_1(x)$, $mem_2(x) = mem'_2(x)$, $mems'_1[i](x) = mems_1[i](x)$ and $mem'_2[i](x) = mems_2[i](x)$ it follows that $mems_1[i](x) = mem_1(x) \wedge mems_2[i](x) = mem_2(x)$, i.e., $x \notin X_i$. Hence $mems'_1[i][f](x) = mems_1[i][f^*](x)$ and $mem'_2[i][f](x) = mems_2[i][f^*](x)$ because $mems'_1[i](x) = mems_1[i](x)$ and $mem'_2[i](x) = mems_2[i](x)$.

I.e., we have shown the following implication:
(o) For all $x \in Var$, $mems'_1[i][f](x) \neq mems_1[i][f^*](x) \vee mem'_2[i][f](x) \neq mems_2[i][f^*](x)$ implies $mem'_1(x) \neq mem_1(x)$ or $mem'_2(x) \neq mem_2(x)$.

Let x be such that $mems'_1[i][f](x) \neq mems_1[i][f^*](x)$ or $mem'_2[i][f](x) \neq mems_2[i][f^*](x)$. Then, by (o), (b1), and (i), "$thr_1[k]$ does not write x" and "$thr_2[k]$ does not write x" are not both satisfied. Since $\langle thr_1, mdss, mem_1, sst \rangle$ and $\langle thr_2, mdss, mem_2, sst \rangle$ have sound modes it follows that $x \notin mdss[k](guar\text{-}no\text{-}w)$, and, hence, $x \notin mdss[i](asm\text{-}no\text{-}w)$. I.e., the following holds:
(p) For all $x \in Var$, $mems'_1[i][f](x) \neq mems_1[i][f^*](x) \vee mem'_2[i][f](x) \neq mems_2[i][f^*](x)$ implies $x \notin mdss[i](asm\text{-}no\text{-}w)$.

We now show that if $mems'_1[i][f](x) \neq mems_1[i][f^*](x) \vee mem'_2[i][f](x) \neq mems_2[i][f^*](x)$, $x \in L$, and $x \notin X'_i$ then $mem'_1(x) = mem'_2(x)$. If the definition of $mems'_1[i](x)$ and $mem'_2[i](x)$ is by (m1) then $mem'_1(x) = mem'_2(x)$ holds because $x \in L$. If the definition is by (m2) then $mems'_1[i](x) = mem'_2[i](x)$, and, hence, $mem'_1(x) = mem'_2(x)$ follows with $x \notin X'_i$. Finally, the definition cannot be by (m3) because (m3) requires $mem_1(x) = mem'_1(x) \ \wedge \ mem_2(x) = mem'_2(x)$, which contradicts (p). I.e., we have shown the following:
(q) For all $x \in L$, $mems'_1[i][f](x) \neq mems_1[i][f^*](x) \vee mem'_2[i][f](x) \neq mems_2[i][f^*](x)$ and $x \notin X'_i$ implies that $mem'_1(x) = mem'_2(x)$.

By Item (1) in Definition 4.18 for $mem_1, mem_2, mdss_1$, and $mdss_2$, the following holds:

$$\langle thr_1[i], mdss[i], mems_1[i][f^*]\rangle \sim_{mm} \langle thr_2[i], mdss[i], mems_2[i][f^*]\rangle$$

Exploiting that $\sim_{mm}$ is closed under globally consistent changes, we derive that also

$$\langle thr_1[i], mdss[i], mems'_1[i][f]\rangle \sim_{mm} \langle thr_2[i], mdss[i], mems'_2[i][f]\rangle.$$

To show this, we must show that we can adapt the values of $mems_1[i][f^*]$ and $mems_2[i][f^*]$ with a globally consistent change for all x with $mems'_1[i][f](x) \neq mems_1[i][f^*](x) \vee mem'_2[i][f](x) \neq mems_2[i][f^*](x)$. That $x \notin mdss[i](\textit{asm-no-w})$ follows from (p). I.e., we can adapt the values if $x \in H$ with globally consistent changes. For $x \in L$ distinguish the cases that $x \in X'_i$ and $x \notin X'_i$. If $x \in X'_i$ then, since $dom(f) = X'_i$, $mems'_1[i][f](x) = mems'_2[i][f](x) = f(x)$, i.e., the value of x can be adapted with a globally consistent change. If $x \notin X'_i$ then, since $dom(f) = X'_i$, $mems'_1[i][f](x) = mems'_1[i](x)$ and $mems'_2[i][f](x) = mems'_2[i](x)$. Hence, by the definition of X'_i, $mems'_1[i][f](x) = mem'_1(x)$ and $mems'_2[i][f](x) = mem'_2(x)$. Moreover, by (q), $mem'_1(x) = mem'_2(x)$, and, hence, $mems'_1[i][f](x) = mems'_2[i][f](x)$. I.e., the value of x can be adapted with a globally consistent change.

Finally, we show that Item (3) from Definition 4.18 is satisfied for $mem'_1, mem'_2, mems'_1$, and $mems'_2$, i.e., that either $n' = 0$ and $mem_1 =_L mem_2$, or that for every $x \in \mathit{Var}$ there exists some $i \in \{1, \ldots, n'\}$ such that $x \notin X'_i$. Let $x \in \mathit{Var}$. We distinguish the cases that (a) $mem_1(x) \neq mem'_1(x)$ or $mem_2(x) \neq mem'_2(x)$ and (b) $mem_1(x) = mem'_1(x)$ and $mem_2(x) = mem'_2(x)$.

Assume firstly that $mem_1(x) \neq mem'_1(x)$ or $mem_2(x) \neq mem'_2(x)$. If $c'_1 \neq \mathit{stop}$ then $n' > 0$, and we show that $x \notin X'_k$: If the definition of $mems'_1[k]$ and $mems'_2[k]$ is by (j1) or by (j2) then, by the definition of X'_k, $x \notin X'_k$. If the definition of $mems'_1[k]$ and $mems'_2[k]$ is by (j3) then $x \in dom(g^1)$, $x \in dom(g^2)$, and $x \in X_k$. But then, as established in the second item after (j3), $mem'_1(x) = mem_1(x)$ and $mem'_2(x) = mem_2(x)$, which contradicts our assumption that $mem_1(x) \neq mem'_1(x)$ or $mem_2(x) \neq mem'_2(x)$. If $c'_1 = \mathit{stop}$ then $mem'_1 =_L mem'_2$ because sound modes ensure that threads do not have modes upon termination. Hence, if $n' = 0$ then Item (3) holds. Assume that $n' > 0$. If threads have been spawned in the execution step then by the definition of $mems'_1[i]$ and $mems'_2[i]$ if i is the index of a spawned thread then $x \notin X'_i$ is satisfied. Assume now that no threads were spawned. If $x \in L$ then $mem'_1(x) = mem'_2(x)$. In consequence, the definition of $mems'_1[i](x)$ and $mems'_2[i](x)$ is by (m1) for all $x \in \mathit{Var}$, and, hence, $x \in X_i$ for all $x \in \{1, \ldots, n'\}$.

Assume secondly that $mem_1(x) = mem'_1(x)$ and $mem_2(x) = mem'_2(x)$. By assumption there exists $i \in \{1, \ldots, n\}$ such that $x \notin X_i$. If $i \neq k$ then $x \notin X'_i$ by the definition of $mems'_1[i]$ and $mems'_2[i]$ in (m3). Assume that $i = k$. If $c'_1 \neq \mathit{stop}$ then $x \notin X'_k$ by the definition of $mems'_1[k]$ and $mems'_2[k]$ in (j1)–(j3). If $c'_1 = \mathit{stop}$ we conclude as in the previous case (using that $mem'_1 =_L mem'_2$).

Hence, Item (3) from Definition 4.18 is also satisfied for $mem'_1, mem'_2, mems'_1$, and $mems'_2$, which concludes the proof. □

Proof of Theorem 4.4. The proof is by induction on k. In the remainder of the proof, we write $s_{1,k} = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'} \mid \sharp(str) = k\})$ and $s_{2,k} =$

$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'} \mid \sharp(str) = k\})$. Moreover, we write n for the length of the thread pools thr_1 and thr_2 (by assumption, $\sharp(thr_1) = \sharp(thr_2)$).

Induction basis. Assume that $k = 1$. We distinguish between the case that sc_1 is final and the case that sc_1 is not final.

If sc_1 is final then $n = 0$. Moreover, the set $\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'} \mid \sharp(str) = 1\}$ equals $\{(\langle sc_1 \rangle, \langle\rangle)\}$ if $mem_1 = mem'$ and the set equals $\{\}$ if $mem_1 \neq mem'$. Hence, $s_{1,1} = 1$ if $mem_1 =_{LO} mem$ and $s_{1,1} = 0$ if $mem_1 \neq_{LO} mem$. Moreover, sc_2 is final because $n = 0$. Hence, the set $\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'} \mid \sharp(str) = 1\}$ equals $\{(\langle sc_2 \rangle, \langle\rangle)\}$ if $mem_2 = mem'$ and $\{\}$ if $mem_2 \neq mem'$. Hence, $s_{2,1} = 1$ if $mem_2 =_{LO} mem$ and $s_{2,1} = 0$ if $mem_2 \neq_{LO} mem$. It follows from $n = 0$ and the fact that $mems_1$ and mem_2 make $\langle thr_1, mdss, mem_1 \rangle$ and $\langle thr_2, mdss, mem_2 \rangle$ compatible with strong low bisimulation modulo modes (Item (2) in Definition 4.18) that $mem_1 =_L mem_2$. With $LO \subseteq L$ it hence follows that $mem_1 =_{LO} mem_2$. In consequence, $mem_1 =_{LO} mem$ if and only if $mem_2 =_{LO} mem$, and, hence, $s_{1,1} = s_{2,1}$.

Assume now that sc_1 is not final. Then $\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'} \mid \sharp(str) = 1\} = \{\}$ for all $mem' \in Mem$, and, hence, $s_{1,1} = 0$. Moreover, since sc_1 is not final it follows from Lemma 4.2 that sc_2 is not final (because the possibility of a step of sc_1 implies the possibility of a step of sc_2 by Lemma 4.2). In consequence, $\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'} \mid \sharp(str) = 1\} = \{\}$ for all $mem' \in Mem$, and, hence, $s_{2,1} = 0$. Thus, $s_{1,1} = s_{2,1}$.

Induction step. Assume that $k > 1$.

Define $A_1 = \{(sc, j, p) \mid sc_1 \stackrel{j,p}{\Longrightarrow}_{\mathcal{S}} sc\}$ and $A_2 = \{(sc, j, p) \mid sc_2 \stackrel{j,p}{\Longrightarrow}_{\mathcal{S}} sc\}$. Then the following equality is satisfied due to Lemma A.9:

$$\rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'} \mid \sharp(str) = k\}) = \sum_{(sc,j,p) \in A_1} [p * \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'} \mid \sharp(str) = k - 1\})$$

By summing up this equation for each $mem' \in [mem]_{LO}$ we hence obtain the equation

$$s_{1,k} = \sum_{mem' \in [mem]_{LO}} \sum_{(sc,j,p) \in A_1} [p * \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'} \mid \sharp(str) = k - 1\})].$$

Analogously, for the system configuration sc_2 we obtain the equation

$$s_{2,k} = \sum_{mem' \in [mem]_{LO}} \sum_{(sc,j,p) \in A_2} [p * \rho_{\mathcal{S}}(\{(str, dtr) \in \mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'} \mid \sharp(str) = k - 1\})].$$

By Lemma 4.2, the sets A_1 and A_2 are in one-to-one correspondence, where an element $(sc, j, p) \in A_1$ corresponds to some element (sc', j, p). Moreover, by Lemma 4.2 the assumptions of this theorem are also satisfied for for corresponding sc and sc'. Hence, applying the induction hypothesis yields $s_{1,k} = s_{2,k}$. □

A.7 Security Type System for SIFUM-security

This appendix contains the soundness proofs for the type systems for SIFUM-security from Section 4.5.

In the proofs, we denote the set of mode states that are consistent with the partial environment Λ with $cons(\Lambda)$, and the set of mode states that are consistent with the dependent partial environment Δ with $cons(\Delta)$.

Proof of Theorem 4.6. The proof is by induction on the derivation of $\vdash \Lambda\ \{c\}\ \Lambda'$.

– [ANNO]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [ANNO]. Then $c = /\!/ann/\!/\ c_1$ for some annotation ann and some command c_1 and $\vdash \Lambda''\ \{c_1\}\ \Lambda'$ is derivable where $\Lambda'' = \mathit{adjust\text{-}pe}(\Lambda, ann)$.

Since $\langle c, mds, mem\rangle \xrightarrow{\alpha} \langle c', mds', mem'\rangle$, it follows from the operational semantics that $\langle c_1, \mathit{mds\text{-}update}(mds, ann), mem\rangle \xrightarrow{\alpha} \langle c', mds', mem'\rangle$. Since $\vdash \Lambda''\ \{c_1\}\ \Lambda'$ is derivable by the induction hypothesis there exists Λ''' such that $\vdash \Lambda'''\ \{c'\}\ \Lambda'$ is derivable, and if $\mathit{mds\text{-}update}(mds, ann) \in cons(\Lambda'')$ then $mds' \in cons(\Lambda'')$.

Moreover, $\mathit{mds\text{-}update}(mds, ann) \in cons(\Lambda'')$ follows from $mds \in cons(\Lambda)$ with the definitions of *adjust-pe* and *mds-update* and the equality $\Lambda'' = \mathit{adjust\text{-}pe}(\Lambda, ann)$. Hence, if $mds \in cons(\Lambda)$ then $mds' \in cons(\Lambda''')$.

– [STOP]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [STOP]. Then $c = stop$, and, hence, nothing needs to be shown.

– [SKIP]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [SKIP]. Then $c = \mathsf{skip}$ and $\Lambda = \Lambda'$. Since $c = \mathsf{skip}$ it follows that $c' = stop$ and $mds' = mds$. Hence, $\vdash \Lambda\ \{c'\}\ \Lambda'$ is derivable with rule [STOP] because $\Lambda = \Lambda'$. Moreover, if $mds \in cons(\Lambda)$ then $mds' \in cons(\Lambda)$ because $mds' = mds$.

– [ASS_1] **and** [ASS_2]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with rule [ASS_1] or rule [ASS_2]. Then $c = x{:=}e$, and Λ' is such that $dom(\Lambda) = dom(\Lambda')$. Since $c = x{:=}e$ it follows that $c' = stop$ and $mds' = mds$. By typing rule [STOP] $\vdash \Lambda'\ \{c'\}\ \Lambda'$ is derivable. Moreover, since $dom(\Lambda) = dom(\Lambda')$ and $mds' = mds$ it follows from $mds \in cons(\Lambda)$ that $mds' \in cons(\Lambda')$.

– [IF]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [IF]. Then $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$, $\vdash \Lambda\ \{c_1\}\ \Lambda'$, and $\vdash \Lambda\ \{c_2\}\ \Lambda'$.

Then, by the operational semantics for conditionals, $c' = c_1$ or $c' = c_2$ and $mds' = mds$. In both cases, $\vdash \Lambda\ \{c'\}\ \Lambda'$ is derivable, and $mds \in cons(\Lambda) \Rightarrow mds' \in cons(\Lambda)$ is satisfied as $mds = mds'$.

– [WHILE]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [WHILE]. Then $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$, $\Lambda' = \Lambda$, and $\vdash \Lambda\ \{c_1\}\ \Lambda$.

By the definition of the operational semantics for while loops it follows that $c' = \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}$ and that $mds = mds'$.

Since $\vdash \Lambda\ \{c_1\}\ \Lambda$ and $\vdash \Lambda\ \{c\}\ \Lambda$ are derivable, $\vdash \Lambda\ \{c_1; c\}\ \Lambda$ is derivable by rule [SEQ]. Then, since $\Lambda\langle\cdot\rangle \vdash e : low$ and $\vdash \Lambda\ \{stop\}\ \Lambda$ is derivable, the judgment $\vdash \Lambda\ \{\mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}\}\ \Lambda$ is derivable by rule [IF], i.e., $\vdash \Lambda\ \{c_1'\}\ \Lambda$ is derivable. Moreover, $mds \in cons(\Lambda) \Rightarrow mds' \in cons(\Lambda)$ is satisfied because $mds = mds'$.

– [SEQ]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [SEQ]. Then $c = c_1; c_2$, and there is an environment Λ'' such that $\vdash \Lambda\ \{c_1\}\ \Lambda''$ and $\vdash \Lambda''\ \{c_2\}\ \Lambda'$ are derivable.

We distinguish the cases $c_1 = stop$ and $c_1 \neq stop$.

Assume that $c_1 = stop$. Then, by the operational semantics for sequential composition, $c' = c_2$ and $mds = mds'$, and we conclude as $\vdash \Lambda''\ \{c_2\}\ \Lambda'$ is derivable. Moreover, since $\vdash \Lambda\ \{c_1\}\ \Lambda''$ can only have been derived with rules [SUB] and [STOP], $\Lambda \sqsubseteq \Lambda''$, and, hence, $dom(\Lambda) = dom(\Lambda'')$. In consequence, $mds \in cons(\Lambda) \Rightarrow mds' \in cons(\Lambda'')$ is satisfied as $mds = mds'$.

Assume that $c_1 \neq \mathit{stop}$. Then, by the definition of the operational semantics for sequential composition, there exists c_1' such that $\langle c_1, \mathit{mds}, \mathit{mem}_1\rangle \xrightarrow{\alpha}_{\triangleright} \langle c_1', \mathit{mds}', \mathit{mem}_1'\rangle$ and such that $c' = c_1'; c_2$. By the induction hypothesis and $\vdash \Lambda\ \{c_1\}\ \Lambda''$ there exists Λ''' such that $\vdash \Lambda'''\ \{c_1'\}\ \Lambda''$ is derivable, and if $\mathit{mds} \in \mathit{cons}(\Lambda)$ then $\mathit{mds}' \in \mathit{cons}(\Lambda''')$. Hence, by rule [SEQ] and the derivability of $\vdash \Lambda''\ \{c_2\}\ \Lambda'$, $\vdash \Lambda'''\ \{c\}\ \Lambda'$ is derivable.

– [SUB]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [SUB]. Then there are partial environments Λ_1 and Λ_1' such that $\vdash \Lambda_1\ \{c\}\ \Lambda_1'$, $\Lambda \sqsubseteq \Lambda_1$, and $\Lambda_1' \sqsubseteq \Lambda'$. Hence, by the induction hypothesis, there exists a partial environment Λ'' such that $\vdash \Lambda''\ \{c'\}\ \Lambda_1'$ is derivable and if $\mathit{mds} \in \mathit{cons}(\Lambda)$ then $\mathit{mds}' \in \mathit{cons}(\Lambda'')$. But then, by rule [SUB], $\vdash \Lambda''\ \{c'\}\ \Lambda'$ is derivable because $\Lambda_1' \sqsubseteq \Lambda'$.

– [SPAWN]**:** Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with the rule [SPAWN]. Then $c = \mathsf{spawn}(\mathit{thr})$, $\Lambda' = \Lambda$, $c' = \mathit{stop}$, and $\mathit{mds}' = \mathit{mds}$, and we can conclude as in case for rule [SKIP]. □

Proof of Theorem 4.7.
Outline: We construct a family of relations on thread configurations with modes consisting of a relation for each partial environment Λ, $(\mathcal{R}^{\Lambda})_{\Lambda\colon \mathit{Var}\to\{\mathit{low},\mathit{high}\}}$, such that

$$\langle c, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}^{\Lambda'}\ \langle c, \mathit{mds}, \mathit{mem}_2\rangle$$

and such that $\mathcal{R} = \bigcup_{\Lambda\colon \mathit{Var}\to\{\mathit{low},\mathit{high}\}} \mathcal{R}^{\Lambda}$ is a strong low bisimulation modulo modes. In consequence, $\langle c, \mathit{mds}, \mathit{mem}_1\rangle \sim_{mm} \langle c, \mathit{mds}, \mathit{mem}_2\rangle$ because $\sim_{mm}$ is the union of all strong low bisimulations modulo modes.

We now construct the relation $\mathcal{R}$. For defining $\mathcal{R}$, we write $\mathit{mem}_1 =_{\Lambda} \mathit{mem}_2$ if $\mathit{mem}_1(x) = \mathit{mem}_2(x)$ for all $x \in \mathit{Var}$ with $\Lambda\langle x\rangle = \mathit{low}$.

For each partial environment Λ' we define $\mathcal{R}^{\Lambda'} = \mathcal{R}_1^{\Lambda'} \cup \mathcal{R}_2^{\Lambda'} \cup \mathcal{R}_3^{\Lambda'}$, where

$$\mathcal{R}_1^{\Lambda'} = \{(\langle c, \mathit{mds}, \mathit{mem}_1\rangle\ ,\ \langle c, \mathit{mds}, \mathit{mem}_2\rangle) \mid \exists \Lambda : \vdash \Lambda\ \{c\}\ \Lambda' \wedge \mathit{mds} \in \mathit{cons}(\Lambda) \wedge \mathit{mem}_1 =_{\Lambda} \mathit{mem}_2\},$$

$$\mathcal{R}_2^{\Lambda'} = \{(\langle c_1, \mathit{mds}, \mathit{mem}_1\rangle\ ,\ \langle c_2, \mathit{mds}, \mathit{mem}_2\rangle) \mid \langle c_1, \mathit{mds}, \mathit{mem}_1\rangle \sim_{mm} \langle c_2, \mathit{mds}, \mathit{mem}_2\rangle \wedge [\forall x \in \mathit{dom}(\Lambda') : \Lambda'(x) = \mathit{high}] \wedge [\exists \Lambda_1, \Lambda_2 : \vdash \Lambda_1\ \{c_1\}\ \Lambda' \wedge \vdash \Lambda_2\ \{c_2\}\ \Lambda' \wedge \mathit{mds} \in \mathit{cons}(\Lambda_1) \wedge \mathit{mds} \in \mathit{cons}(\Lambda_2)]\}, \text{ and}$$

$$\mathcal{R}_3^{\Lambda'} = \{(\langle c_1; c, \mathit{mds}, \mathit{mem}_1\rangle\ ,\ \langle c_2; c, \mathit{mds}, \mathit{mem}_2\rangle) \mid \exists \Lambda : \langle c_1, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}_2^{\Lambda}\ \langle c_2, \mathit{mds}, \mathit{mem}_2\rangle \wedge \vdash \Lambda\ \{c\}\ \Lambda'\}.$$

We define $\mathcal{R} = \bigcup_{\Lambda'\colon \mathit{Var}\to\{\mathit{low},\mathit{high}\}} \mathcal{R}^{\Lambda'}$.

We now show that $\langle c, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}\ \langle c, \mathit{mds}, \mathit{mem}_2\rangle$ is satisfied.
It follows from the assumptions of the theorem that $\langle c, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle c, \mathit{mds}, \mathit{mem}_2\rangle$. Hence, $\langle c, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}\ \langle c, \mathit{mds}, \mathit{mem}_2\rangle$ by the definition of $\mathcal{R}$.

We now show that $\mathcal{R}$ is a strong low bisimulation modulo modes.

It follows from the definitions of $\mathcal{R}_1^{\Lambda'}$, $\mathcal{R}_2^{\Lambda'}$, and $\mathcal{R}_3^{\Lambda'}$ that $\mathcal{R}^{\Lambda'}$ is symmetric for each Λ'. In consequence, $\mathcal{R}$ is symmetric.

To show that $\mathcal{R}$ is closed under globally consistent changes it suffices to show that $\mathcal{R}^{\Lambda'}$ is closed under globally consistent changes for all Λ'. To show that $\mathcal{R}^{\Lambda'}$ is closed under globally consistent changes assume that $\langle c_1, mds, mem_1\rangle \mathcal{R}^{\Lambda'} \langle c_2, mds, mem_2\rangle$. We must show that for all $x \in \mathit{Var}$ with $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ the following conditions are satisfied:

(1) $\langle c_1, mds, mem_1[x \mapsto v]\rangle \ \mathcal{R}^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v]\rangle$ for all $x \in L$ and all $v \in \mathit{Val}$
(2) $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \ \mathcal{R}^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$ for all $x \in H$ and all $v_1, v_2 \in \mathit{Val}$

To show that (1) and (2) are satisfied we distinguish between the three cases arising from the definition of $\mathcal{R}^{\Lambda'}$, i.e., that the configurations are related by $\mathcal{R}_1^{\Lambda'}$, by $\mathcal{R}_2^{\Lambda'}$, or by $\mathcal{R}_3^{\Lambda'}$.

– Case 1: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_1^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$. Then $c_1 = c_2$ and there exists Λ such that $\vdash \Lambda \ \{c_1\} \ \Lambda' \wedge mds \in cons(\Lambda) \wedge mem_1 =_\Lambda mem_2$.
For $x \in \mathit{Var}$ and $v \in \mathit{Val}$ it follows from $mem_1 =_\Lambda mem_2$ that $mem_1[x \mapsto v] =_\Lambda mem_2[x \mapsto v]$. In consequence,

$$\langle c_1, mds, mem_1[x \mapsto v]\rangle \ \mathcal{R}_1^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v]\rangle,$$

and, hence,

$$\langle c_1, mds, mem_1[x \mapsto v]\rangle \ \mathcal{R}^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v]\rangle.$$

Assume now that $x \in H$ and let $v_1, v_2 \in \mathit{Val}$. Since $x \notin mds(\mathit{asm\text{-}no\text{-}w})$ and $mds \in cons(\Lambda)$ it follows from $x \in H$ that $x \notin dom(\Lambda)$. Hence, $\Lambda\langle x\rangle = \mathit{high}$, and it follows that $mem_1[x \mapsto v_1] =_\Lambda mem_2[x \mapsto v_2]$. In consequence,

$$\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \ \mathcal{R}_1^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle,$$

and, hence,

$$\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \ \mathcal{R}^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle.$$

– Case 2: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$. By the definition of $\mathcal{R}_2^{\Lambda'}$, $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$, $[\forall x \in dom(\Lambda') : \Lambda'(x) = \mathit{high}]$, and $[\exists \Lambda_1, \Lambda_2 : \vdash \Lambda_1 \ \{c_1\} \ \Lambda' \wedge \vdash \Lambda_2 \ \{c_2\} \ \Lambda' \wedge mds \in cons(\Lambda_1) \wedge mds \in cons(\Lambda_2)]$ are satisfied.
Since $\sim_{mm}$ is a strong low bisimulation modulo modes it is closed under globally consistent changes, and, hence, the following two conditions are satisfied:
  * $\langle c_1, mds, mem_1[x \mapsto v]\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v]\rangle$ for all $x \in L$ and all $v \in \mathit{Val}$
  * $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$ for all $x \in H$ and all $v_1, v_2 \in \mathit{Val}$

  But then the two conditions (1) and (2) that we set out to prove are also satisfied (by the definition of $\mathcal{R}^{\Lambda'}$).

– Case 3: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_3^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$. Then there exist c_1', c_2', and c such that $c_1 = c_1'; c$, $c_2 = c_2'; c$, $\langle c_1', mds, mem_1\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2', mds, mem_2\rangle$, and $\vdash \Lambda \ \{c\} \ \Lambda'$. But then, by the previous case (Case 2) the following two conditions are satisfied:
  * $\langle c_1', mds, mem_1[x \mapsto v]\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2', mds, mem_2[x \mapsto v]\rangle$ for all $x \in L$ and all $v \in \mathit{Val}$
  * $\langle c_1', mds, mem_1[x \mapsto v_1]\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2', mds, mem_2[x \mapsto v_2]\rangle$ for all $x \in H$ and all $v_1, v_2 \in \mathit{Val}$

But then, by the definition of $\mathcal{R}_3^{\Lambda'}$,the following two conditions are satisfied.
* $\langle c_1, mds, mem_1[x \mapsto v]\rangle \ \mathcal{R}_3^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v]\rangle$ for all $x \in L$ and all $v \in Val$
* $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \ \mathcal{R}_3^{\Lambda'} \ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$ for all $x \in H$ and all $v_1, v_2 \in Val$

But then the two conditions (1) and (2) that we set out to prove are also satisfied (by the definition of $\mathcal{R}^{\Lambda'}$).

Now we show that whenever $\langle c_1, mds, mem_1\rangle \ \mathcal{R} \ \langle c_2, mds, mem_2\rangle$ then $mem_1 =_L^{mds} mem_2$. For arbitrary Λ' we distinguish the cases that the two thread configurations with modes are related by $\mathcal{R}_1^{\Lambda'}$, by $\mathcal{R}_2^{\Lambda'}$, and by $\mathcal{R}_3^{\Lambda'}$.
- Case 1: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_1^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$. By the definition of $\mathcal{R}_1^{\Lambda'}$ there exists Λ such that $mds \in cons(\Lambda)$ and $mem_1 =_\Lambda mem_2$. Let $x \in Var$ with $x \in L$ and $x \notin mds(\textit{asm-no-r})$. We must show that $mem_1(x) = mem_2(x)$. Since $mds \in cons(\Lambda)$ it follows from $x \in L$ and $x \notin mds(\textit{asm-no-r})$ that $x \notin dom(\Lambda)$. Then, $mem_1(x) = mem_2(x)$ follows directly from $mem_1 =_\Lambda mem_2$.
- Case 2: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_2^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$. Then by the definition of $\mathcal{R}_2^{\Lambda'}$ it follows that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$. Hence, $mem_1 =_L^{mds} mem_2$ follows directly from the definition of strong low bisimulations modulo modes.
- Case 3: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_3^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$. Then by the definition of $\mathcal{R}_3^{\Lambda'}$ there exist commands c_1' and c_2' such that $\langle c_1', mds, mem_1\rangle \ \mathcal{R}_3^{\Lambda'} \ \langle c_2', mds, mem_2\rangle$. We can hence conclude as in Case 2.

We now show that whenever $\langle c_1, mds, mem_1\rangle \ \mathcal{R} \ \langle c_2, mds, mem_2\rangle$ and $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$ with $\alpha_1 = new(thr_1, mdss_1)$, then there exist c_2', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:
- $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
- $\langle c_1', mds', mem_1'\rangle \ \mathcal{R} \ \langle c_2', mds', mem_2'\rangle$,
- $\sharp(thr_1) = \sharp(thr_2)$, and
- $\langle thr_1[i], mdss_1[i], mem_1'\rangle \ \mathcal{R} \ \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

For arbitrary Λ' we distinguish the three cases that $\langle c_1, mds, mem_1\rangle$ and $\langle c_2, mds, mem_2\rangle$ are related by $\mathcal{R}_1^{\Lambda'}$, $\mathcal{R}_2^{\Lambda'}$, and $\mathcal{R}_3^{\Lambda'}$, respectively.

Case 1: Assume that $\langle c_1, mds, mem_1\rangle \ \mathcal{R}_1^{\Lambda'} \ \langle c_2, mds, mem_2\rangle$.

We prove the following statement, from which the statement we need to prove in this case follows immediately:

For all $c, c_1' \in Com$, all $mds, mds' \in Mds$, all $mem_1, mem_2, mem_1' \in Mem$, all $\alpha_1 = new(thr_1, mdss_1) \in Lab_m$, and all $\Lambda, \Lambda' : Var \rightharpoonup \{low, high\}$ with $\langle c, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$, $\vdash \Lambda \ \{c\} \ \Lambda'$, $mds \in cons(\Lambda)$, and $mem_1 =_\Lambda mem_2$, there exist c_2', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:
- $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
- $\langle c_1', mds', mem_1'\rangle \ \mathcal{R}^{\Lambda'} \ \langle c_2', mds', mem_2'\rangle$,
- $\sharp(thr_1) = \sharp(thr_2)$, and
- $\langle thr_1[i], mdss_1[i], mem_1'\rangle \ \mathcal{R} \ \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

The proof is by induction on the derivation of the judgment $\vdash \Lambda \ \{c\} \ \Lambda'$.

- [ANNO]**:** Assume that $\vdash \Lambda \ \{c\} \ \Lambda'$ is derived with rule [ANNO]. Then $c = /\!\!/ ann /\!\!/ \ c_1$ for some annotation ann and some command c_1. Moreover, $\vdash \Lambda'' \ \{c_1\} \ \Lambda'$ and

$[\forall x \in \mathit{Var} : \Lambda\langle x\rangle \sqsubseteq \Lambda''\langle x\rangle]$ hold for $\Lambda'' = \mathit{adjust\text{-}pe}(\Lambda, \mathit{ann})$.

Since $\langle c, \mathit{mds}, \mathit{mem}_1\rangle \xrightarrow{\alpha_1} \langle c_1', \mathit{mds}', \mathit{mem}_1'\rangle$ is satisfied it follows that $\langle c_1, \mathit{mds\text{-}update}(\mathit{mds}, \mathit{ann}), \mathit{mem}_1\rangle \xrightarrow{\alpha_1} \langle c_1', \mathit{mds}', \mathit{mem}_1'\rangle$. With the definitions of *mds-update* and *adjust-pe* it follows from $\Lambda'' = \mathit{adjust\text{-}pe}(\Lambda, \mathit{ann})$ and $\mathit{mds} \in \mathit{cons}(\Lambda)$ that $\mathit{mds\text{-}update}(\mathit{mds}, \mathit{ann}) \in \mathit{cons}(\Lambda'')$. Since $\Lambda\langle x\rangle \sqsubseteq \Lambda''\langle x\rangle$ for all $x \in \mathit{Var}$ and $\mathit{mem}_1 =_\Lambda \mathit{mem}_2$, it follows that $\mathit{mem}_1 =_{\Lambda''} \mathit{mem}_2$. Hence, by the induction hypothesis for the derivation of the judgment $\vdash \Lambda''\ \{c_1\}\ \Lambda'$ there exist c_2', $\alpha_2 = \mathit{new}(\mathit{thr}_2, \mathit{mdss}_2)$, and mem_2' such that the following properties are satisfied:

* $\langle c_1, \mathit{mds\text{-}update}(\mathit{mds}, \mathit{ann}), \mathit{mem}_2\rangle \xrightarrow{\alpha_2} \langle c_2', \mathit{mds}', \mathit{mem}_2'\rangle$,
* $\langle c_1', \mathit{mds}', \mathit{mem}_1'\rangle\ \mathcal{R}^{\Lambda'}\ \langle c_2', \mathit{mds}', \mathit{mem}_2'\rangle$,
* $\sharp(\mathit{thr}_1) = \sharp(\mathit{thr}_2)$, and
* $\langle \mathit{thr}_1[i], \mathit{mdss}_1[i], \mathit{mem}_1'\rangle\ \mathcal{R}\ \langle \mathit{thr}_2[i], \mathit{mdss}_2[i], \mathit{mem}_2'\rangle$ for all $i \in \{1, \ldots, \sharp(\mathit{thr}_1)\}$.

By the derivation rules for the judgments $\langle c, \mathit{mds}, \mathit{mem}\rangle \xrightarrow{\alpha} \langle c', \mathit{mds}', \mathit{mem}'\rangle$ it follows that $\langle /\!/\mathit{ann}/\!/\ c_1, \mathit{mds}, \mathit{mem}_2\rangle \xrightarrow{\alpha_2} \langle c_2', \mathit{mds}', \mathit{mem}_2'\rangle$. This concludes the proof for rule [ANNO].

– [SKIP]: Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with rule [SKIP]. Then $c = \mathsf{skip}$ and $\Lambda = \Lambda'$. Then $c_1' = \mathit{stop}$, $\alpha_1 = \mathit{new}(\langle\rangle)$, $\mathit{mds}' = \mathit{mds}$, and $\mathit{mem}_1' = \mathit{mem}_1$ by the operational semantics for skip. We define $c_2' = \mathit{stop}$, $\alpha_2 = \alpha_1$, and $\mathit{mem}_2' = \mathit{mem}_2$. Then $\langle c, \mathit{mds}, \mathit{mem}_2\rangle \xrightarrow{\alpha_2} \langle c_2', \mathit{mds}', \mathit{mem}_2'\rangle$.

  Moreover, $\mathit{mem}_1 =_L^{\mathit{mds}} \mathit{mem}_2$ follows from $\mathit{mds} \in \mathit{cons}(\Lambda)$ and $\mathit{mem}_1 =_\Lambda \mathit{mem}_2$. In consequence, $\langle \mathit{stop}, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}_1^\Lambda\ \langle \mathit{stop}, \mathit{mds}, \mathit{mem}_2\rangle$ because $\vdash \Lambda\ \{\mathit{stop}\}\ \Lambda$, $\mathit{mds} \in \mathit{cons}(\Lambda)$, and $\mathit{mem}_1 =_L^{\mathit{mds}} \mathit{mem}_2$. Hence, $\langle \mathit{stop}, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle \mathit{stop}, \mathit{mds}, \mathit{mem}_2\rangle$ because $\Lambda = \Lambda'$, and, in consequence, $\langle \mathit{stop}, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}^{\Lambda'}\ \langle \mathit{stop}, \mathit{mds}, \mathit{mem}_2\rangle$.

– [ASS$_1$]: Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with rule [ASS$_1$]. Then $c = x{:=}e$, $x \notin \mathit{dom}(\Lambda)$, $\Lambda\langle\cdot\rangle \vdash e : d$, $d \sqsubseteq \mathit{dma}(x)$, and $\Lambda = \Lambda'$.

  Let $v_1, v_2 \in \mathit{Val}$ be those values such that $\mathit{eval}(e, \mathit{mem}_1) = v_1$ and $\mathit{eval}(e, \mathit{mem}_2) = v_2$. Then $c_1' = \mathit{stop}$, $\alpha_1 = \mathit{new}(\langle\rangle)$, $\mathit{mds}' = \mathit{mds}$, and $\mathit{mem}_1' = \mathit{mem}_1[x \mapsto v_1]$. We define $c_2' = \mathit{stop}$, $\alpha_2 = \alpha_1$, and $\mathit{mem}_2' = \mathit{mem}_2[x \mapsto v_2]$. Then, by the operational semantics for assignments, $\langle c, \mathit{mds}, \mathit{mem}_2\rangle \xrightarrow{\alpha_2} \langle c'', \mathit{mds}', \mathit{mem}_2'\rangle$.

  We now show that $\mathit{mem}_1[x \mapsto v_1] =_\Lambda \mathit{mem}_2[x \mapsto v_2]$. Since $\mathit{mem}_1 =_\Lambda \mathit{mem}_2$ it remains to show that $v_1 = v_2$ if $\Lambda\langle x\rangle = \mathit{low}$. If $\Lambda\langle x\rangle = \mathit{low}$ then $\mathit{dma}(x) = \mathit{low}$ because $x \notin \mathit{dom}(\Lambda)$. Then $d = \mathit{low}$, and, hence, $\Lambda\langle\cdot\rangle \vdash e : \mathit{low}$. It follows from typing rule [EXP] that $\Lambda\langle y\rangle = \mathit{low}$ for all $y \in \mathit{vars}(e)$. But then it follows from $\mathit{mem}_1 =_\Lambda \mathit{mem}_2$ and our assumptions on expression evaluation in Section 2.3.2 that $v_1 = v_2$.

  Hence, $\langle \mathit{stop}, \mathit{mds}, \mathit{mem}_1[x \mapsto v_1]\rangle\ \mathcal{R}_1^\Lambda\ \langle \mathit{stop}, \mathit{mds}, \mathit{mem}_2[x \mapsto v_2]\rangle$ follows from $\vdash \Lambda\ \{\mathit{stop}\}\ \Lambda$, $\mathit{mds} \in \mathit{cons}(\Lambda)$, and $\mathit{mem}_1[x \mapsto v_1] =_\Lambda \mathit{mem}_2[x \mapsto v_2]$. In consequence, $\langle \mathit{stop}, \mathit{mds}, \mathit{mem}_1[x \mapsto v_1]\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle \mathit{stop}, \mathit{mds}, \mathit{mem}_2[x \mapsto v_2]\rangle$ because $\Lambda = \Lambda'$, and, hence, $\langle \mathit{stop}, \mathit{mds}, \mathit{mem}_1[x \mapsto v_1]\rangle\ \mathcal{R}^{\Lambda'}\ \langle \mathit{stop}, \mathit{mds}, \mathit{mem}_2[x \mapsto v_2]\rangle$.

– [ASS$_2$]: Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with rule [ASS$_2$]. Then $c = x{:=}e$, $x \in \mathit{dom}(\Lambda)$, $\Lambda\langle\cdot\rangle \vdash e : d$, and $\Lambda[x \mapsto d] = \Lambda'$.

  Let $v_1, v_2 \in \mathit{Val}$ be those values such that $\mathit{eval}(e, \mathit{mem}_1) = v_1$ and $\mathit{eval}(e, \mathit{mem}_2) = v_2$. Then $c_1' = \mathit{stop}$, $\alpha_1 = \mathit{new}(\langle\rangle)$, $\mathit{mds}' = \mathit{mds}$, and $\mathit{mem}_1' = \mathit{mem}_1[x \mapsto v_1]$. We define $c_2' = \mathit{stop}$, $\alpha_2 = \alpha_1$, and $\mathit{mem}_2' = \mathit{mem}_2[x \mapsto v_2]$. Then, by the operational semantics for assignments, $\langle c, \mathit{mds}, \mathit{mem}_2\rangle \xrightarrow{\alpha_2} \langle c'', \mathit{mds}', \mathit{mem}_2'\rangle$.

We now show that $mem_1[x \mapsto v_1] =_{\Lambda[x \mapsto d]} mem_2[x \mapsto v_2]$. Since $mem_1 =_\Lambda mem_2$ it remains to show that $v_1 = v_2$ if $\Lambda[x \mapsto d]\langle x \rangle = low$. If $\Lambda[x \mapsto d]\langle x \rangle = low$ then $d = low$, and, hence, $\Lambda\langle\cdot\rangle \vdash e : low$. It follows from typing rule [EXP] that $\Lambda\langle y \rangle = low$ for all $y \in vars(e)$. But then it follows from $mem_1 =_\Lambda mem_2$ and our assumptions on expression evaluation in Section 2.3.2 that $v_1 = v_2$.

Moreover, $mds \in cons(\Lambda[x \mapsto t])$ follows from $dom(\Lambda) = dom(\Lambda[x \mapsto t])$ and $mds \in cons(\Lambda)$.

Hence, $\langle stop, mds, mem_1[x \mapsto v_1]\rangle \; \mathcal{R}_1^{\Lambda'} \; \langle stop, mds, mem_2[x \mapsto v_2]\rangle$ follows from $\vdash \Lambda' \{stop\} \Lambda'$, $mds \in cons(\Lambda')$, and $mem_1[x \mapsto v_1] =_{\Lambda'} mem_2[x \mapsto v_2]$. In consequence, $\langle stop, mds, mem_1[x \mapsto v_1]\rangle \; \mathcal{R}^{\Lambda'} \; \langle stop, mds, mem_2[x \mapsto v_2]\rangle$.

– [IF]**:** Assume that $\vdash \Lambda \{c\} \Lambda'$ is derived with rule [IF]. Then $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$, $\vdash \Lambda \{c_1\} \Lambda'$, $\vdash \Lambda \{c_2\} \Lambda'$, and $\Lambda\langle\cdot\rangle \vdash e : high$ implies that $c_1 \sim_{mm}^{mds} c_2$ and $[\forall x \in dom(\Lambda') : \Lambda'(x) = high]$ are satisfied for all mode states $mds \in cons(\Lambda)$.

Then, by the operational semantics for conditionals, $c_1' = c_1$ or $c_1' = c_2$. Moreover, $\alpha = new(\langle\rangle)$, $mds' = mds$, and $mem_1' = mem_1$.

Let $v_1, v_2 \in Val$ be the values such that $eval(e, mem_1) = v_1$ and $eval(e, mem_2) = v_2$.

We do a case distinction on the cases $\Lambda\langle\cdot\rangle \vdash e : low$ and $\Lambda\langle\cdot\rangle \vdash e : high$.

Assume that $\Lambda\langle\cdot\rangle \vdash e : low$. Then it follows from rule [EXP] that $\Lambda\langle y \rangle = low$ for all $y \in vars(e)$. But then it follows from $mem_1 =_\Lambda mem_2$ and our assumptions on expression evaluation in Section 2.3.2 that $v_1 = v_2$. We define $c_2' = c_1'$, $\alpha_2 = \alpha_1$, and $mem_2' = mem_2$. Then, by the operational semantics for conditionals, $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds, mem_2'\rangle$. Moreover, from the induction hypotheses for the derivations of $\vdash \Lambda \{c_1\} \Lambda'$ and $\vdash \Lambda \{c_2\} \Lambda'$ it follows that $\langle c', mds, mem_1\rangle \; \mathcal{R}^{\Lambda'} \; \langle c', mds, mem_2\rangle$.

Assume now that $\Lambda\langle\cdot\rangle \vdash e : high$. If $v_1 = v_2$ we can proceed as in the case for $\Lambda\langle\cdot\rangle \vdash e : low$. Assume that $v_1 \neq v_2$. Moreover, assume without loss of generality that $c_1' = c_1$ (if $c_1' = c_2$ the remainder of the proof in this case is the same by switching the indices 1 and 2). We define $c_2' = c_2$, $\alpha_2 = \alpha_1$, and $mem_2' = mem_2$. Then $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds, mem_2'\rangle$.

Let $x \in Var$ with $dma(x) = low$ and $x \notin mds(asm\text{-}no\text{-}r)$. Then $x \notin dom(\Lambda)$ because $mds \in cons(\Lambda)$. Hence, $\Lambda\langle x \rangle = low$. Since $mem_1 =_\Lambda mem_2$ it follows that $mem_1(x) = mem_2(x)$. In consequence, $mem_1 =_L^{mds} mem_2$. Since $mds \in cons(\Lambda)$ it follows from the preconditions of rule [IF] that $c_1 \sim_{mm}^{mds} c_2$. With $mem_1 =_L^{mds} mem_2$ it follows that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$. Hence, since $\vdash \Lambda \{c_1\} \Lambda'$, $\vdash \Lambda \{c_2\} \Lambda'$, and $mds \in cons(\Lambda)$ it follows that $\langle c_1, mds, mem_1\rangle \; \mathcal{R}_2^{\Lambda'} \; \langle c_2, mds, mem_2\rangle$. Hence, $\langle c_1, mds, mem_1\rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2, mds, mem_2\rangle$.

– [WHILE]**:** Assume that $\vdash \Lambda \{c\} \Lambda'$ is derived with rule [WHILE]. Then $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$, $\Lambda' = \Lambda$, $\Lambda\langle\cdot\rangle \vdash e : low$, and $\vdash \Lambda \{c_1\} \Lambda$.

By the definition of the operational semantics for while loops it follows that $c_1' = \mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}$. $\alpha = new(\langle\rangle)$, $mds' = mds$, and $mem_1' = mem_1$. We define $c_2' = c_1'$, $\alpha_2 = \alpha_1$, and $mem_2' = mem_2$. Then, by the definition of the operational semantics for while loops, $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds, mem_2'\rangle$.

Since $\vdash \Lambda \{c_1\} \Lambda$ and $\vdash \Lambda \{c\} \Lambda$ are derivable, $\vdash \Lambda \{c_1; c\} \Lambda$ is derivable by rule [SEQ]. Then, since $\Lambda\langle\cdot\rangle \vdash e : low$ and $\vdash \Lambda \{stop\} \Lambda$ are derivable, the judgment $\vdash \Lambda \{\mathsf{if}\ e\ \mathsf{then}\ c_1; c\ \mathsf{else}\ stop\ \mathsf{fi}\} \Lambda$ is derivable by rule [IF], i.e., $\vdash \Lambda \{c_1'\} \Lambda$ is derivable.

Then $\langle c_1', mds', mem_1' \rangle \; \mathcal{R}_1^{\Lambda'} \; \langle c_1', mds', mem_2' \rangle$ because $\vdash \Lambda \; \{c_1'\} \; \Lambda$ is derivable, $c_1' = c_2'$, $mds' = mds$, $mem_1 = mem_1'$, $mem_2 = mem_2'$, $mds \in cons(\Lambda)$, and $mem_1 =_\Lambda mem_2$. Hence, $\langle c_1', mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_1', mds', mem_2' \rangle$.

– [SEQ]**:** Assume that $\vdash \Lambda \; \{c\} \; \Lambda'$ is derived with rule [SEQ]. Then $c = c_1; c_2$, and there is an environment Λ'' such that $\vdash \Lambda \; \{c_1\} \; \Lambda''$ and $\vdash \Lambda'' \; \{c_2\} \; \Lambda'$ are derivable.

We distinguish the cases $c_1 = stop$ and $c_1 \neq stop$.

Assume that $c_1 = stop$. Then, by the operational semantics for sequential composition, $c_1' = c_2$, $\alpha_1 = new(\langle\rangle)$, $mds' = mds$, and $mem_1' = mem_1$. We define $c_2' = c_2$, $\alpha_2 = \alpha_1$, and $mem_2' = mem_2$. Then $\langle c, mds, mem_2 \rangle \xrightarrow{\alpha_2} \langle c_2', mds, mem_2' \rangle$. From $c_1 = stop$, $\vdash \Lambda \; \{c_1\} \; \Lambda''$, and the fact that $\vdash \Lambda \; \{stop\} \; \Lambda''$ can only have been derived by an application of [SUB] and [STOP], it follows that $\Lambda \sqsubseteq \Lambda''$. But then, since $\vdash \Lambda'' \; \{c_2\} \; \Lambda'$, $\vdash \Lambda \; \{c_2\} \; \Lambda'$ by an application of rule [SUB]. Hence, it follows with $mds \in cons(\Lambda)$ and $mem_1 =_\Lambda mem_2$ that $\langle c_2, mds, mem_1 \rangle \; \mathcal{R}_1^{\Lambda'} \; \langle c_2, mds, mem_2 \rangle$, and, thus, $\langle c_2, mds, mem_1 \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2, mds, mem_2 \rangle$.

Assume now that $c_1 \neq stop$. Then, by the definition of the operational semantics for sequential composition, there exists a command c_1'' such that $\langle c_1, mds, mem_1 \rangle \xrightarrow{\alpha_1} \langle c_1'', mds', mem_1' \rangle$ and such that $c_1' = c_1''; c_2$. Hence, by the induction hypothesis for the derivation of $\vdash \Lambda \; \{c_1\} \; \Lambda''$ there exist c_2'', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:

* $\langle c_1, mds, mem_2 \rangle \xrightarrow{\alpha_2} \langle c_2'', mds', mem_2' \rangle$,
* $\langle c_1'', mds', mem_1' \rangle \; \mathcal{R}^{\Lambda''} \; \langle c_2'', mds', mem_2' \rangle$, and
* $\sharp(thr_1) = \sharp(thr_2)$, and
* $\langle thr_1[i], mdss_1[i], mem_1' \rangle \; \mathcal{R} \; \langle thr_2[i], mdss_2[i], mem_2' \rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

We define $c_2' = c_2''; c_2$. Then, by the definition of the operational semantics of sequential composition, $\langle c, mds, mem_2 \rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2' \rangle$.

To show that $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$ we distinguish between the cases that $\langle c_1'', mds', mem_1' \rangle$ and $\langle c_2'', mds', mem_2' \rangle$ are related by $\mathcal{R}_1^{\Lambda''}$, $\mathcal{R}_2^{\Lambda''}$, or $\mathcal{R}_3^{\Lambda''}$.

Assume that $\langle c_1'', mds', mem_1' \rangle \; \mathcal{R}_1^{\Lambda''} \; \langle c_2'', mds', mem_2' \rangle$. Then $c_1'' = c_2''$ and there exists Λ''' such that $\vdash \Lambda''' \; \{c_1''\} \; \Lambda''$, $mds' \in cons(\Lambda''')$ is derivable and $mem_1' =_{\Lambda'''} mem_2'$. But then $\vdash \Lambda''' \; \{c_1''; c_2\} \; \Lambda'$ is derivable by the rule [SEQ] because $\vdash \Lambda'' \; \{c_2\} \; \Lambda'$ is also derivable. Hence, $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}_1^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$, and, in consequence, $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$.

Assume $\langle c_1'', mds', mem_1' \rangle \; \mathcal{R}_2^{\Lambda''} \; \langle c_2'', mds', mem_2' \rangle$. Since $\vdash \Lambda'' \; \{c_2\} \; \Lambda'$ is derivable, it follows that $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}_3^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$, and, in consequence, $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$.

Assume that $\langle c_1'', mds', mem_1' \rangle \; \mathcal{R}_3^{\Lambda''} \; \langle c_2'', mds', mem_2' \rangle$. It follows from the definition of $\mathcal{R}_3^{\Lambda''}$ that $c_1'' = c_1'''; c^*$ and $c_2'' = c_2'''; c^*$ for commands c_1''', c_2''', and c^*, and that there exists a partial environment Λ''' such that $\vdash \Lambda''' \; \{c^*\} \; \Lambda''$ and such that $\langle c_1''', mds', mem_1' \rangle \; \mathcal{R}_2^{\Lambda'''} \; \langle c_2''', mds', mem_2' \rangle$. Since $\vdash \Lambda''' \; \{c^*\} \; \Lambda''$ and $\vdash \Lambda'' \; \{c_2\} \; \Lambda'$ are derivable the judgment $\vdash \Lambda''' \; \{c^*; c_2\} \; \Lambda'$ is derivable with rule [SEQ]. Hence, $\langle c_1'''; c^*; c_2, mds', mem_1' \rangle \; \mathcal{R}_3^{\Lambda'} \; \langle c_2'''; c^*; c_2, mds', mem_2' \rangle$, i.e., with $c_1'' = c_1'''; c^*$ and $c_2'' = c_2'''; c^*$, it follows that $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$.

We have shown in all three cases that $\langle c_1''; c_2, mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2''; c_2, mds', mem_2' \rangle$, i.e., that $\langle c_1', mds', mem_1' \rangle \; \mathcal{R}^{\Lambda'} \; \langle c_2', mds', mem_2' \rangle$ because $c_1' = c_1''; c_2$ and $c_2' = c_2''; c_2$.

– [SUB]: Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with rule [SUB]. Then there are partial environments Λ_1 and Λ_1' such that $\vdash \Lambda_1\ \{c\}\ \Lambda_1'$, $\Lambda \sqsubseteq \Lambda_1$, and $\Lambda_1' \sqsubseteq \Lambda'$.

From $\Lambda \sqsubseteq \Lambda_1$ and $\Lambda_1' \sqsubseteq \Lambda'$ it follows that $dom(\Lambda) \supseteq dom(\Lambda_1)$ and $dom(\Lambda_1') \supseteq dom(\Lambda')$. Hence, $cons(\Lambda) \subseteq cons(\Lambda_1)$ and $cons(\Lambda_1') \subseteq cons(\Lambda')$ by the definition of consistency of mode states with partial environments.

Since $cons(\Lambda) \subseteq cons(\Lambda_1)$ and $mds \in cons(\Lambda)$ it follows that $mds \in cons(\Lambda_1)$. Let $x \in Var$ with $\Lambda_1\langle x\rangle = low$. It follows from $\Lambda \sqsubseteq \Lambda_1$ that $\Lambda\langle x\rangle \sqsubseteq \Lambda_1\langle x\rangle$, and, hence, $\Lambda\langle x\rangle = low$. Hence, since $mem_1 =_\Lambda mem_2$ it follows that $mem_1(x) = mem_2(x)$. In consequence, $mem_1 =_{\Lambda_1} mem_2$.

Since $mds \in cons(\Lambda_1)$ and $mem_1 =_{\Lambda_1} mem_2$, by the induction hypothesis there exist c_2', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:

* $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
* $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}^{\Lambda_1'}\ \langle c_2', mds', mem_2'\rangle$, and
* $\sharp(thr_1) = \sharp(thr_2)$, and
* $\langle thr_1[i], mdss_1[i], mem_1'\rangle\ \mathcal{R}\ \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

This concludes this case if $\mathcal{R}^{\Lambda_1'} \subseteq \mathcal{R}^{\Lambda'}$. We prove this inclusion by showing that $\mathcal{R}_1^{\Lambda_1'} \subseteq \mathcal{R}_1^{\Lambda'}$, $\mathcal{R}_2^{\Lambda_1'} \subseteq \mathcal{R}_2^{\Lambda'}$, and $\mathcal{R}_3^{\Lambda_1'} \subseteq \mathcal{R}_2^{\Lambda'}$.

Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_1^{\Lambda_1'}\ \langle c_2, mds, mem_2\rangle$. Then $c_1 = c_2$ and there exists Λ such that $\vdash \Lambda\ \{c\}\ \Lambda_1'$, $mds \in cons(\Lambda)$, and $mem_1 =_\Lambda mem_2\}$. Since $\vdash \Lambda\ \{c\}\ \Lambda_1'$ and $\Lambda_1' \sqsubseteq \Lambda'$ it follows that $\vdash \Lambda\ \{c\}\ \Lambda'$ by rule [SUB]. Hence, $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle c_2, mds, mem_2\rangle$.

Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda_1'}\ \langle c_2, mds, mem_2\rangle$. Then $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$, $[\forall x \in dom(\Lambda_1') : \Lambda_1'(x) = high]$ is satisfied, and there exist Λ_1 and Λ_2 such that $\vdash \Lambda_1\ \{c_1\}\ \Lambda_1'$, $\vdash \Lambda_2\ \{c_2\}\ \Lambda_1'$, $mds \in cons(\Lambda_1)$, and $mds \in cons(\Lambda_2)$. Since $\Lambda_1' \sqsubseteq \Lambda'$ it follows that $\vdash \Lambda_1\ \{c_1\}\ \Lambda'$ and $\vdash \Lambda_2\ \{c_2\}\ \Lambda'$ by rule [SUB]. Moreover, since $[\forall x \in dom(\Lambda_1') : \Lambda_1'(x) = high]$ is satisfied and $\Lambda_1' \sqsubseteq \Lambda'$ it follows that $[\forall x \in dom(\Lambda') : \Lambda'(x) = high]$. Hence, $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda'}\ \langle c_2, mds, mem_2\rangle$.

Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_3^{\Lambda_1'}\ \langle c_2, mds, mem_2\rangle$. Then there exist commands c_1', c_2', and c and a partial environment Λ such that $c_1 = c_1'; c$, $c_2 = c_2'; c$, $\vdash \Lambda\ \{c\}\ \Lambda_1'$, and $\langle c_1', mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda_1'}\ \langle c_2', mds, mem_2\rangle$. We have already shown above that then $\langle c_1', mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda'}\ \langle c_2', mds, mem_2\rangle$. Moreover, from $\Lambda_1' \sqsubseteq \Lambda'$ it follows that $\vdash \Lambda\ \{c\}\ \Lambda'$ by rule [SUB]. Hence, $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_3^{\Lambda'}\ \langle c_2, mds, mem_2\rangle$.

– [SPAWN]: Assume that $\vdash \Lambda\ \{c\}\ \Lambda'$ is derived with rule [SPAWN]. Then $c = \mathsf{spawn}(thr)$ and there exist $\Lambda_1, \ldots, \Lambda_{\sharp(thr)}$ such that $\Lambda' = \Lambda$ and $\vdash \Lambda_0\ \{thr[i]\}\ \Lambda_i$ is derivable for all $i \in \{1, \ldots, \sharp(thr)\}$. Moreover, $\Lambda\langle x\rangle \sqsubseteq dma(x)$ for all $x \in Var$.

By the operational semantics for spawn-commands, $c_1' = stop$, $\alpha_1 = new(thr)$, $mds' = mds$, and $mem_1' = mem_1$. We define $c_2' = stop$, $\alpha_2 = \alpha_1$, and $mem_2' = mem_2$. Then $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$.

Moreover, $mem_1 =_L^{mds} mem_2$ follows from $mds \in cons(\Lambda)$ and $mem_1 =_\Lambda mem_2$. Hence, $\langle stop, mds, mem_1\rangle\ \mathcal{R}_1^{\Lambda}\ \langle stop, mds, mem_2\rangle$ follows from $\vdash \Lambda\ \{stop\}\ \Lambda$, $mds \in cons(\Lambda)$, and $mem_1 =_L^{mds} mem_2$. Hence, $\langle stop, mds, mem_1\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle stop, mds, mem_2\rangle$ because $\Lambda = \Lambda'$, and, in consequence, $\langle stop, mds, mem_1\rangle\ \mathcal{R}^{\Lambda'}\ \langle stop, mds, mem_2\rangle$.

Let $i \in \{1, \ldots, \sharp(thr)\}$. It follows from the operational semantics for spawn-commands (in particular the definition of the function *spawn-mds*) that $mdss[i] \in cons(\Lambda_0)$.

Moreover, it follows from the fact that $\Lambda\langle x\rangle \sqsubseteq dma(x)$ for all $x \in Var$ and $mem_1 =_\Lambda mem_2$ that $mem_1 =_L mem_2$, and, hence, $mem_1 =_L^{mdss[i]} mem_2$. With the derivability of $\vdash \Lambda_0\ \{thr[i]\}\ \Lambda_i$ it then follows $\langle thr[i], mdss[i], mem_1'\rangle\ \mathcal{R}_1^{\Gamma_i}\ \langle thr[i], mdss[i], mem_2'\rangle$, and, in consequence, $\langle thr[i], mdss[i], mem_1'\rangle\ \mathcal{R}\ \langle thr[i], mdss[i], mem_2'\rangle$.

Case 2: Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda'}\ \langle c_2, mds, mem_2\rangle$. It follows from the definition of $\mathcal{R}_2^{\Lambda'}$ that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$, that $[\forall x \in dom(\Lambda') : \Lambda'(x) = high]$ is satisfied, and that there exist environments Λ_1 and Λ_2 such that $\vdash \Lambda_1\ \{c_1\}\ \Lambda', \vdash \Lambda_2\ \{c_2\}\ \Lambda'$, $mds \in cons(\Lambda_1)$, and $mds \in cons(\Lambda_2)$.

Assume $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$. It follows from $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$, that there exist c_2', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:

- $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
- $\langle c_1', mds', mem_1'\rangle \sim_{mm} \langle c_2', mds', mem_2'\rangle$, and
- $\sharp(thr_1) = \sharp(thr_2)$, and
- $\langle thr_1[i], mdss_1[i], mem_1'\rangle \sim_{mm} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

Since $\vdash \Lambda_1\ \{c_1\}\ \Lambda'$, $\langle c_1, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$ and $mds \in cons(\Lambda_1)$, by Theorem 4.6 there exists Λ_1' such that $\vdash \Lambda_1'\ \{c_1'\}\ \Lambda'$ and $mds' \in cons(\Lambda_1')$. Moreover, since $\vdash \Lambda_2\ \{c_2\}\ \Lambda'$, $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$, and $mds \in cons(\Lambda_2)$, by Theorem 4.6 there exists Λ_2' such that $\vdash \Lambda_2'\ \{c_2'\}\ \Lambda'$ and $mds \in cons(\Lambda_2')$.

Hence, $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}_2^{\Lambda'}\ \langle c_2', mds', mem_2'\rangle$, and, in consequence, $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}^{\Lambda'}\ \langle c_2', mds', mem_2'\rangle$.

Case 3: Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}_3^{\Lambda'}\ \langle c_2, mds, mem_2\rangle$. It follows from the definition of $\mathcal{R}_3^{\Lambda'}$ that there exist commands c_1'', c_2'', and c^* such that $c_1 = c_1''; c^*$ and $c_2 = c_2''; c^*$, and that there exists Λ'' such that $\vdash \Lambda''\ \{c^*\}\ \Lambda'$ and such that $\langle c_1'', mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda''}\ \langle c_2'', mds, mem_2\rangle$.

We distinguish the cases that $c_1'' \neq stop$ and that $c_1'' = stop$.

Assume that $c_1'' \neq stop$. Then it follows from the operational semantics for sequential composition that there exists c_1''' such that $\langle c_1'', mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1''', mds', mem_1'\rangle$ and $c_1' = c_1'''; c^*$. With $\langle c_1'', mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda''}\langle c_2'', mds, mem_2\rangle$ and the proof in the preceding case (Case 2) it follows that there exist c_2''', α_2, and mem_2' such that $\langle c_2'', mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2''', mds', mem_2'\rangle$ and $\langle c_1''', mds', mem_1'\rangle\ \mathcal{R}_2^{\Lambda''}\ \langle c_2''', mds', mem_2'\rangle$. We define $c_2' = c_2'''; c^*$. Then, by the operational semantics for sequential composition, $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$. Moreover, $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}_3^{\Lambda'}\ \langle c_2', mds', mem_2'\rangle$ is satisfied because $\langle c_1'', mds', mem_1'\rangle\ \mathcal{R}_2^{\Lambda''}\ \langle c_2''', mds', mem_2'\rangle$ and $\vdash \Lambda''\ \{c^*\}\ \Lambda'$. In consequence, $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}^{\Lambda'}\ \langle c_2', mds', mem_2'\rangle$.

Assume that $c_1'' = stop$. It follows from the operational semantics for sequential composition that $c_1' = c^*$, $\alpha = new(\langle\rangle)$, $mds' = mds$, and $mem_1' = mem_1$. Moreover, $\langle c_1'', mds, mem_1\rangle \sim_{mm} \langle c_2'', mds, mem_2\rangle$ because $\langle c_1'', mds, mem_1\rangle\ \mathcal{R}_2^{\Lambda''}\ \langle c_2'', mds, mem_2\rangle$, and, hence, $c_2'' = stop$.

We define $c_2' = c^*$, $\alpha_2 = \alpha$, and $mem_2' = mem_2$. Then $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c^*, mds', mem_2'\rangle$ by the operational semantics for sequential composition.

From $\langle c_1'', mds, mem_1\rangle\mathcal{R}_2^{\Lambda''}\langle c_2'', mds, mem_2\rangle$ it follows that $mds \in cons(\Lambda'')$. Moreover, it follows that $\langle c_1'', mds, mem_1\rangle \sim_{mm} \langle c_2'', mds, mem_2\rangle$, and, hence, $mem_1 =_L^{mds} mem_2$. It also follows that $\Lambda''(x) = high$ for all $x \in dom(\Lambda'')$. Let $x \in Var$ such that $\Lambda''\langle x\rangle = low$. Then $x \notin dom(\Lambda'')$ because $\Lambda''(x) = high$ for all $x \in dom(\Lambda'')$. Hence, $dma(x) = low$,

and $x \notin mds(\textit{asm-no-}r)$ because $mds \in cons(\Lambda'')$. Thus, $mem_1(x) = mem_2(x)$ because $mem_1 =_L^{mds} mem_2$. This shows that $mem_1 =_{\Lambda''} mem_2$.

Since $\vdash \Lambda''\ \{c^*\}\ \Lambda'$, $mds \in cons(\Lambda'')$, and $mem_1 =_{\Lambda''} mem_2$, it follows that $\langle c^*, mds, mem_1\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle c^*, mds, mem_2\rangle$. Since $c_1' = c_2' = c^*$, $mds = mds'$, $mem_1 = mem_1'$, and $mem_2 = mem_2'$, it follows with the definition of $\mathcal{R}^{\Lambda'}$ that $\langle c_2', mds', mem_1'\rangle\ \mathcal{R}_1^{\Lambda'}\ \langle c_2', mds', mem_2'\rangle$. □

Proof of Theorem 4.9. The basic idea of the proof is the same as for the proof of Theorem 4.7: We define a family of relations $\mathcal{R}^{\Delta'}$ which is parametric in the set of dependent partial environments. To define this family we adapt the definition of the family $\mathcal{R}^{\Lambda'}$ from the proof of Theorem 4.7. Since we do not consider the semantic side condition for rule [IF] in the type system with dependent partial environments, we only need to consider the first of the three sub-relations in the adaptation. We define the relation $\mathcal{R}^{\Delta'}$ for each dependent partial environment as follows (where X is the domain of the partial memories in $dom(\Delta')$):

$$\begin{aligned}\mathcal{R}^{\Delta'} = \{(\langle c, mds, mem_1\rangle\ ,\ \langle c, mds, mem_2\rangle) \mid \exists \Delta : \exists pm_1, pm_2 \in dom(\Delta) :\\ \vdash \Delta\ \{c\}\ \Delta' \wedge mds \in cons(\Delta) \wedge mem_1 =_{\Delta(pm_1)} mem_2 \wedge mem_1 =_{\Delta(pm_2)} mem_2\\ \wedge\ [\forall x \in dom(pm) : mem_1(x) = pm_1(x) \wedge mem_2(x) = pm_2(x)]\}\end{aligned}$$

We define $\mathcal{R} = \bigcup_{\Delta'} \mathcal{R}^{\Delta'}$. Then $\langle c, mds, mem_1\rangle\ \mathcal{R}\ \langle c, mds, mem_2\rangle$ follows from the assumptions of the theorem. It remains to show that $\mathcal{R}$ is a strong low bisimulation modulo modes.

Symmetry of $\mathcal{R}$ follows directly from the definition of $\mathcal{R}$.

We now show that $\mathcal{R}$ is closed under globally consistent changes. Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}^{\Delta'}\ \langle c_2, mds, mem_2\rangle$. Then $c_1 = c_2$ and there exists Δ and $pm_1, pm_2 \in dom(\Delta)$ such that $\vdash \Delta\ \{c_1\}\ \Delta'$, $mds \in cons(\Delta)$, $mem_1 =_{\Delta(pm_1)} mem_2$, $mem_1 =_{\Delta(pm_2)} mem_2$, and $mem_1(x) = pm_1(x) \wedge mem_2(x) = pm_2(x)$ for all $x \in dom(pm_1)$.

Let $x \in \textit{Var}$ with $x \notin mds(\textit{asm-no-}w)$. It follows from $mds \in cons(\Delta)$ that $x \notin dom(pm_1)$ and $x \notin dom(pm_2)$.

Assume that $x \in L$ and let $v \in \textit{Val}$. It follows from $mem_1 =_{\Delta(pm_i)} mem_2$ that $mem_1[x \mapsto v] =_{\Delta(pm_i)} mem_2[x \mapsto v]$ for $i \in \{1, 2\}$. Moreover, it follows from $x \notin dom(pm_1)$ that $mem_1[x \mapsto v](x) = pm_1(x)$ and $mem_2[x \mapsto v](x) = pm_2(x)$ for all $x \in dom(pm_1)$. In consequence, $\langle c_1, mds, mem_1[x \mapsto v]\rangle\ \mathcal{R}^{\Delta'}\ \langle c_2, mds, mem_2[x \mapsto v]\rangle$.

Assume now that $x \in H$ and let $v_1, v_2 \in \textit{Val}$. Since $x \notin mds(\textit{asm-no-}w)$ and $mds \in cons(\Delta)$ it follows from $x \in H$ that $x \notin dom(\Delta(pm_1))$ and $x \notin dom(\Delta(pm_2))$. Hence, $\Delta(pm_1)\langle x\rangle = \Delta(pm_2)\langle x\rangle = \textit{high}$, and it follows that $mem_1[x \mapsto v_1] =_{\Delta(pm_1)} mem_2[x \mapsto v_2]$ and $mem_1[x \mapsto v_1] =_{\Delta(pm_2)} mem_2[x \mapsto v_2]$. Moreover, it follows from $x \notin dom(pm_1)$ that $mem_1[x \mapsto v_1](x) = pm_1(x)$ and $mem_2[x \mapsto v_2](x) = pm_2(x)$ for all $x \in dom(pm_1)$. In consequence, $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle\ \mathcal{R}^{\Delta'}\ \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$.

Now we show that whenever $\langle c_1, mds, mem_1\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2\rangle$ then $mem_1 =_L^{mds} mem_2$. Assume that $\langle c_1, mds, mem_1\rangle\ \mathcal{R}^{\Delta'}\ \langle c_2, mds, mem_2\rangle$. By the definition of $\mathcal{R}^{\Delta'}$ there exists Δ and $pm \in dom(\Delta)$ such that $mds \in cons(\Delta)$ and $mem_1 =_{\Delta(pm)} mem_2$. Let $x \in \textit{Var}$ with $x \in L$ and $x \notin mds(\textit{asm-no-}r)$. Since $mds \in cons(\Delta)$ it follows from

$x \in L$ and $x \notin mds(asm\text{-}no\text{-}r)$ that $x \notin dom(\Delta(pm))$. Then, $mem_1(x) = mem_2(x)$ follows directly from $mem_1 =_{\Delta(pm)} mem_2$.

Finally, we show that for all $c, c_1' \in Com$, $mds, mds' \in Mds$, $mem_1, mem_2, mem_1' \in Mem$, $\alpha_1 = new(thr_1, mdss_1) \in Lab_m$, dependent partial environments Δ, Δ', and $pm_1, pm_2 \in dom(\Delta)$ with $\langle c, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$, $\vdash \Delta\ \{c\}\ \Delta'$, $mds \in cons(\Delta)$, $mem_1 =_{\Delta(pm_1)} mem_2$, $mem_1 =_{\Delta(pm_2)} mem_2$, and $mem_1(x) = pm_1(x) \wedge mem_2(x) = pm_2(x)$ for all $x \in dom(pm)$, there exist c_2', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:

- $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
- $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}^{\Delta'}\ \langle c_2', mds', mem_2'\rangle$, and
- $\sharp(thr_1) = \sharp(thr_2)$, and
- $\langle thr_1[i], mdss_1[i], mem_1'\rangle\ \mathcal{R}\ \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1, \ldots, \sharp(thr_1)\}$.

The proof is by induction on the derivation of the judgment $\vdash \Delta\ \{c\}\ \Delta'$

- [ANNO], [SKIP], [SEQ], [WHILE], **and** [SPAWN]**:** The proofs for these cases are analogous to the proofs for these cases in the proof of Theorem 4.7.

- [ASS]**:** Assume that $\vdash \Delta\ \{c\}\ \Delta'$ is derived with rule [ASS]. Then $c = x{:=}e$, $\Delta' = update\text{-}dpe(\Delta, x, e)$, and if $pm \in dom(\Delta)$, $x \notin dom(\Delta(pm))$, and $\Delta(pm)\langle\cdot\rangle \vdash e : d_{pm}$ then $d_{pm} \sqsubseteq dma(x)$.

  Let $v_1, v_2 \in Val$ be those values such that $eval(e, mem_1) = v_1$ and $eval(e, mem_2) = v_2$. Then $c_1' = stop$, $\alpha_1 = new(\langle\rangle)$, $mds' = mds$, and $mem_1' = mem_1[x \mapsto v_1]$. We define $c_2' = stop$, $\alpha_2 = \alpha_1$, and $mem_2' = mem_2[x \mapsto v_2]$. Then, by the operational semantics for assignments, $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c'', mds', mem_2'\rangle$.

  By typing rule [STOP], $\vdash \Delta'\ \{stop\}\ \Delta'$. Hence, for showing $\langle stop, mds, mem_1[x \mapsto v_1]\rangle\ \mathcal{R}_{\Delta'}\ \langle stop, mds, mem_2[x \mapsto v_2]\rangle$ it suffices to show that there exist $pm_1', pm_2' \in dom(\Delta')$ such that $mem_1[x \mapsto v_1] =_{\Delta'(pm_1')} mem_2[x \mapsto v_2]$, $mem_1[x \mapsto v_1] =_{\Delta'(pm_2')} mem_2[x \mapsto v_2]$, and $mem_1[x \mapsto v_1](y) = pm_1'(y) \wedge mem_2[x \mapsto v_2](y) = pm_2'(y)$ for all $y \in dom(pm_1')$.

  To show this, assume firstly that $x \notin dom(pm_1)$. In this case, we define $pm_1' = pm_1$ and $pm_2' = pm_2$. Then, by the definition of *update-dpe*, $pm_1', pm_2' \in dom(\Delta')$. Moreover, $mem_1[x \mapsto v_1](y) = pm_1'(y)$ and $mem_2[x \mapsto v_2](y) = pm_2'(y)$ hold for all $y \in dom(pm_1')$. That $mem_1[x \mapsto v_1] =_{\Delta'(pm_1')} mem_2[x \mapsto v_2]$ and $mem_1[x \mapsto v_1] =_{\Delta'(pm_2')} mem_2[x \mapsto v_2]$ are satisfied follows as in the cases for [ASS_1] and [ASS_2] in the proof of Theorem 4.7.

  Assume now that $x \in dom(pm)$. In this case, we define $pm_1' = pm_1[x \mapsto v_1]$ and $pm_2' = pm_2[x \mapsto v_2]$. Then, by the definition of *update-dpe*, $pm_1', pm_2' \in dom(\Delta')$. Moreover, $mem_1[x \mapsto v_1](y) = pm_1'(y)$ and $mem_2[x \mapsto v_2](y) = pm_2'(y)$ hold for all $y \in dom(pm_1')$. Also, that $mem_1[x \mapsto v_1] =_{\Delta'(pm_1')} mem_2[x \mapsto v_2]$ and $mem_1[x \mapsto v_1] =_{\Delta'(pm_2')} mem_2[x \mapsto v_2]$ are satisfied follows as in the cases for [ASS_1] and [ASS_2] in the proof of Theorem 4.7.

- [IF]**:** Assume that $\vdash \Delta\ \{c\}\ \Delta'$ is derived with rule [IF]. Then $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ and $\Delta(pm)\langle\cdot\rangle \vdash e : low$ for all $pm \in dom(\Delta)$. Moreover, let $\Delta_1 = \Delta\,|_{\{pm \mid \mathsf{true} \in eval(e,pm)\}}$ and $\Delta_2 = \Delta\,|_{\{pm \mid \mathsf{false} \in eval(e,pm)\}}$. Then $\vdash \Delta_1\ \{c_1\}\ \Delta'$ if $dom(\Delta_1) \neq \{\}$ and $\vdash \Delta_2\ \{c_2\}\ \Delta'$ if $dom(\Delta_2) \neq \{\}$.

Let $v_1, v_2 \in \mathit{Val}$ be the values such that $\mathit{eval}(e, \mathit{mem}_1) = v_1$ and $\mathit{eval}(e, \mathit{mem}_2) = v_2$. By the operational semantics for conditionals, $c_1' = c_1$ if $v_1 = \mathsf{true}$ and $c_1' = c_2$ if $v_1 = \mathsf{false}$. Moreover, $\alpha = \mathit{new}(\langle\rangle)$, $\mathit{mds}' = \mathit{mds}$, and $\mathit{mem}_1' = \mathit{mem}_1$.

It follows from $\Delta(\mathit{pm}_1)\langle\cdot\rangle \vdash e : \mathit{low}$, $\Delta(\mathit{pm}_2)\langle\cdot\rangle \vdash e : \mathit{low}$, and typing rule [EXP] that $\Delta(\mathit{pm}_1)\langle y\rangle = \Delta(\mathit{pm}_2)\langle y\rangle = \mathit{low}$ for all $y \in \mathit{vars}(e)$. But then it follows from $\mathit{mem}_1 =_{\Delta(\mathit{pm})} \mathit{mem}_2$ and our assumptions on expression evaluation in Section 2.3.2 that $v_1 = v_2$. We define $c_2' = c_1'$, $\alpha_2 = \alpha_1$, and $\mathit{mem}_2' = \mathit{mem}_2$. Then, by the operational semantics for conditionals, $\langle c, \mathit{mds}, \mathit{mem}_2\rangle \xrightarrow{\alpha_2}_\triangleright \langle c_2', \mathit{mds}, \mathit{mem}_2'\rangle$.

Assume that $v_1 = \mathsf{true}$. Then $\mathit{pm}_1 \in \mathit{dom}(\Delta_1)$ because $\mathit{pm}_1(x) = \mathit{mem}_1(x)$ for all $x \in \mathit{dom}(\mathit{pm}_1)$, and, hence, $\mathsf{true} \in \mathit{eval}(e, \mathit{pm}_1)$. Analogously, $\mathit{pm}_2 \in \mathit{dom}(\Delta_1)$. Hence, it follows from the induction hypothesis for the derivation of $\vdash \Delta_1\ \{c_1\}\ \Delta'$ that $\langle c', \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}^{\Delta'}\ \langle c', \mathit{mds}, \mathit{mem}_2\rangle$.

Assume that $v_1 = \mathsf{false}$. Then $\mathit{pm}_1 \in \mathit{dom}(\Delta_2)$ and $\mathit{pm}_2 \in \mathit{dom}(\Delta_2)$ follows as in the case that $v_1 = \mathsf{true}$. Hence, it follows from the induction hypothesis for the derivation of $\vdash \Delta_2\ \{c_2\}\ \Delta'$ that $\langle c', \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}^{\Delta'}\ \langle c', \mathit{mds}, \mathit{mem}_2\rangle$.

– [SUB]**:** Assume that $\vdash \Delta\ \{c\}\ \Delta'$ is derived with rule [SUB]. Then there are partial environments Δ_1 and Δ_1' such that $\vdash \Delta_1\ \{c\}\ \Delta_1'$, $\Delta \sqsubseteq \Delta_1$, and $\Delta_1' \sqsubseteq \Delta'$. Let X, X', X_1, and X_1' be the domains of the partial memories in $\mathit{dom}(\Delta), \mathit{dom}(\Delta'), \mathit{dom}(\Delta_1)$, and $\mathit{dom}(\Delta_1')$, respectively.

From $\Delta \sqsubseteq \Delta_1$ and $\Delta_1' \sqsubseteq \Delta'$ it follows that $X \supseteq X_1$, that $X_1' \supseteq X'$, that $\Delta(\mathit{pm}) \sqsubseteq \Delta_1(\mathit{pm}|_{X_1})$ for all $\mathit{pm} \in \mathit{dom}(\Delta)$, and that $\Delta_1'(\mathit{pm}) \sqsubseteq \Delta'(\mathit{pm}|_{X'})$ for all $\mathit{pm} \in \mathit{dom}(\Delta_1')$. From the last two statements it follows that $\mathit{dom}(\Delta(\mathit{pm})) \supseteq \mathit{dom}(\Delta_1(\mathit{pm}|_{X_1}))$ for all $\mathit{pm} \in \mathit{dom}(\Delta)$ and that $\mathit{dom}(\Delta_1'(\mathit{pm})) \supseteq \mathit{dom}(\Delta'(\mathit{pm}|_{X'}))$ for all $\mathit{pm} \in \mathit{dom}(\Delta_1')$. Hence, $\mathit{cons}(\Delta) \subseteq \mathit{cons}(\Delta_1)$ and $\mathit{cons}(\Delta_1') \subseteq \mathit{cons}(\Delta')$ by the definition of consistency of mode states with dependent partial environments.

Since $\mathit{cons}(\Delta) \subseteq \mathit{cons}(\Delta_1)$ and $\mathit{mds} \in \mathit{cons}(\Delta)$ it follows that $\mathit{mds} \in \mathit{cons}(\Delta_1)$. Let $\mathit{pm}_1 \in \mathit{dom}(\Delta_1)$ and $x \in \mathit{Var}$ with $\Delta_1(\mathit{pm}_1)\langle x\rangle = \mathit{low}$. It follows from $\Delta \sqsubseteq \Delta_1$ that $\Delta(\mathit{pm})\langle x\rangle \sqsubseteq \Delta_1(\mathit{pm}_1)\langle x\rangle$, and, hence, $\Delta(\mathit{pm})\langle x\rangle = \mathit{low}$. Hence, since $\mathit{mem}_1 =_{\Delta(\mathit{pm})} \mathit{mem}_2$ it follows that $\mathit{mem}_1(x) = \mathit{mem}_2(x)$. In consequence, $\mathit{mem}_1 =_{\Delta_1(\mathit{pm}_1)} \mathit{mem}_2$.

Since $\mathit{mds} \in \mathit{cons}(\Delta_1)$ and $\mathit{mem}_1 =_{\Delta_1(\mathit{pm}_1)} \mathit{mem}_2$, by the induction hypothesis there exist c_2', $\alpha_2 = \mathit{new}(\mathit{thr}_2, \mathit{mdss}_2)$, and mem_2' such that the following conditions are satisfied:

* $\langle c, \mathit{mds}, \mathit{mem}_2\rangle \xrightarrow{\alpha_2}_\triangleright \langle c_2', \mathit{mds}', \mathit{mem}_2'\rangle$,
* $\langle c_1', \mathit{mds}', \mathit{mem}_1'\rangle\ \mathcal{R}^{\Delta_1'}\ \langle c_2', \mathit{mds}', \mathit{mem}_2'\rangle$, and
* $\sharp(\mathit{thr}_1) = \sharp(\mathit{thr}_2)$, and
* $\langle \mathit{thr}_1[i], \mathit{mdss}_1[i], \mathit{mem}_1'\rangle\ \mathcal{R}\ \langle \mathit{thr}_2[i], \mathit{mdss}_2[i], \mathit{mem}_2'\rangle$ for all $i \in \{1, \ldots, \sharp(\mathit{thr}_1)\}$.

This concludes this case if $\mathcal{R}^{\Delta_1'} \subseteq \mathcal{R}^{\Delta'}$. Assume $\langle c_1, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}^{\Delta_1'}\ \langle c_2, \mathit{mds}, \mathit{mem}_2\rangle$. Then $c_1 = c_2$ and there exists Δ and $\mathit{pm} \in \mathit{dom}(\Delta)$ such that $\vdash \Delta\ \{c\}\ \Delta_1'$, $\mathit{mds} \in \mathit{cons}(\Delta)$, and $\mathit{mem}_1 =_{\Delta(\mathit{pm})} \mathit{mem}_2\}$, and $\mathit{mem}_1(x) = \mathit{mem}_2(x) = \mathit{pm}(x)$ for all $x \in \mathit{dom}(\mathit{pm})$. Since $\vdash \Delta\ \{c\}\ \Delta_1'$ and $\Delta_1' \sqsubseteq \Delta'$ it follows that $\vdash \Delta\ \{c\}\ \Delta'$ by rule [SUB]. Hence, $\langle c_1, \mathit{mds}, \mathit{mem}_1\rangle\ \mathcal{R}_1^{\Delta'}\ \langle c_2, \mathit{mds}, \mathit{mem}_2\rangle$. □

A.8 Integration of FSI-security and SIFUM-security

This appendix contains the proof of the compositionality and scheduler-independence result for FSIFUM-security, as well as the proofs that FSIFUM-security subsumes FSI-security and SIFUM-security. The proofs are not as detailed as in the previous sections, because they are in large parts analogous to the proofs for FSI-security and SIFUM-security, respectively.

For proving compositionality and scheduler independence of FSIFUM-security, we adopt Definition 4.18 and Lemmas 4.1 and 4.2 (all from Section 4.4.4) to a setting where high threads are treated differently from low threads.

Definition A.2. Let $tcm_1 = \langle thr_1, mdss, mem_1\rangle$ and $tcm_2 = \langle thr_2, mdss, mem_2\rangle$ be thread configurations with modes and let $match : \{1,\ldots,\sharp(thr_1)\} \rightharpoonup \{1,\ldots,\sharp(thr_2)\}$ be an injective partial function such that the indices of low threads of thr_1 are contained in $dom(match)$ and the indices of low threads of thr_2 are contained in $img(match)$. We say that lists of memories $mems_1$ and $mems_2$ with $\sharp(mems_1) = \sharp(thr_1)$ and $\sharp(mems_2) = \sharp(thr_2)$ *make* tcm_1 *and* tcm_2 *compatible with strong low bisimulation modulo low matching and modes for* $match$ if and only if the following conditions are satisfied (where $X_i = \{x \in Var \mid mems_1[i](x) \neq mem_1(x) \vee mems_2[match(i)](x) \neq mem_2(x)\}$ for each $i \in dom(match)$):

1. $\forall i \in dom(match) : \forall f : Var \rightharpoonup Val : dom(f) = X_i \Rightarrow \langle thr_1[i], mdss[i], mems_1[i][x \mapsto f(x) \mid x \in X_i]\rangle \sim_{lmm} \langle thr_2[match(i)], mdss[match(i)], mems_2[match(i)][x \mapsto f(x) \mid x \in X_i]\rangle$,
2. $\forall x \in Var : \forall i \in dom(match) : [mem_1(x) = mem_2(x) \vee x \in H] \Rightarrow x \notin X_i$, and
3. $(dom(match) = \{\} \wedge mem_1 =_L mem_2) \vee (\forall x \in Var : \exists i \in dom(match) : x \notin X_i)$.

Remark A.1. Note that in Definition A.2 not all elements of the lists of memories $mems_1$ and $mems_2$ are actually relevant. Items (1)–(3) in the definition only make statements about the memories $mems_1[i]$ for $i \in dom(match)$ and the memories $mems_1[j]$ for $j \in img(match)$. We choose this definition over a definition in which the irrelevant memories are not included in $mems_1$ and $mems_2$, because it permits us to use the same indices for selecting elements of the thread pools, elements of the lists of mode states, and elements of the lists of memories.

Lemma A.12. Let $\langle thr_1, mdss_1, mem_1, sst\rangle$ and $\langle thr_2, mdss_2, mem_2, sst\rangle$ be system configurations such that $(thr_1, mdss_1)$ and $(thr_2, mdss_2)$ have sound modes, let $match : \{1,\ldots,\sharp(thr_1)\} \rightharpoonup \{1,\ldots,\sharp(thr_2)\}$ be an injective partial function, and let $mems_1$ and $mems_2$ be lists of memories that make $\langle thr_1, mdss, mem_1\rangle$ and $\langle thr_2, mdss, mem_2\rangle$ compatible with strong low bisimulation modulo low matching and modes for $match$.

Then for all $i \in dom(match)$ the commands $thr_1[i]$ and $thr_2[match(i)]$ do not read any variable in $X_i = \{x \in Var \mid mems_1[i](x) \neq mem_1(x) \vee mems_2[match(i)](x) \neq mem_2(x)\}$. ◊

Proof Sketch. The proof is like the proof for Lemma 4.1. □

Lemma A.13. Let $\langle thr_1, mdss_1, mem_1, sst\rangle$ and $\langle thr_2, mdss_2, mem_2, sst\rangle$ be system configurations such that $(thr_1, mdss_1)$ and $(thr_2, mdss_2)$ have sound modes for scheduler $\mathcal{S}$ and observation function obs, and assume that obs is confined to L. Let $match : \{1,\ldots,\sharp(thr_1)\} \rightharpoonup \{1,\ldots,\sharp(thr_2)\}$ be an injective partial function and assume that $mems_1$ and $mems_2$ make $\langle thr_1, mdss, mem_1\rangle$ and $\langle thr_2, mdss, mem_2\rangle$ compatible with

strong low bisimulation modulo low matching and modes for $match$. Moreover, assume $\langle thr_1, mdss_1, mem_1, sst\rangle \overset{k,p}{\Longrightarrow}_{\mathcal{S}} \langle thr_1', mdss_1', mem_1', sst'\rangle$ for $k \in dom(match)$.

Then there exist thr_2', mem_2', $mems_1'$, $mems_2'$, and an injective partial function $match' : \{1,\ldots,\sharp(thr_1')\} \rightharpoonup \{1,\ldots,\sharp(thr_2')\}$ such that the lists of memories $mems_1'$ and $mems_2'$ make $\langle thr_1', mdss_2', mem_1'\rangle$ and $\langle thr_2', mdss_2', mem_2'\rangle$ compatible with strong low bisimulation modulo low matching and modes for $match'$, and $\langle thr_2, mdss_2, mem_2, sst\rangle \overset{match(k),p}{\Longrightarrow}_{\mathcal{S}} \langle thr_2', mdss_2', mem_2', sst'\rangle$. ◊

Proof Sketch. The proof is essentially the same as the proof of Lemma 4.2, with the difference that the index of the two threads performing the execution steps are not necessarily equal but related by the function $match$. The definition of low bisimulations modulo low matching and modes guarantees that such a match exists also after the execution step even if new threads are spawned in any of the two execution steps. □

Theorem A.1. Let $\mathcal{S}$ be a robust scheduler, and assume that the observation function obs is confined to L.

Let $sc_1 = \langle thr_1, mdss_1, mem_1, sst_1\rangle$ and $sc_2 = \langle thr_2, mdss_2, mem_2, sst_2\rangle$ be terminating system configurations with sound modes, $match : \{1,\ldots,\sharp(thr_1)\} \rightharpoonup \{1,\ldots,\sharp(thr_2)\}$ be an injective partial function, and $mems_1$ and $mems_2$ be lists of memories that make sc_1 and sc_2 compatible with strong low bisimulation modulo low matching and modes for $match$.

Let $sc_{1,p} = \langle thr_{1,p}, mdss_{1,p}, mem_{1,p}, sst_p\rangle$ and $sc_{2,p} = \langle thr_{2,p}, mdss_{2,p}, mem_{2,p}, sst_p\rangle$ be system configurations with sound modes, $match_{1,p} : \{1,\ldots,\sharp(thr_1)\} \rightharpoonup \{1,\ldots,\sharp(thr_{1,p})\}$ and $match_{2,p} : \{1,\ldots,\sharp(thr_2)\} \rightharpoonup \{1,\ldots,\sharp(thr_{2,p})\}$ be injective partial functions with $dom(match_1) = dom(match_{1,p})$ and $dom(match_2) = dom(match_{2,p})$, and let $mems_{1,p}$ and $mems_{2,p}$ be lists of memories such that $mems_1$ and $mems_2$ make sc_1 and $sc_{1,p}$ and $mems_2$ and $mems_{2,p}$ make sc_2 and $sc_{2,p}$ compatible with strong low bisimulations modulo low matching and modes, respectively.

Assume finally that $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and $sc_2 \prec_{\mathcal{S}} sc_{2,p}$.

Then the following holds for all $mem \in Mem$:

$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem'}) = \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_2)_{\Downarrow mem'})$$

Proof.

Proof Outline. The general idea and structure of the proof is the same as in the proof of Theorem 3.14. The major difference is to Steps 9 and 10 of the proof of Theorem 3.14, where it is established that the existence of a transition in sc_1 guarantees the existence of a matching transition in sc_2. To establish the same in this proof, we exploit Lemmas A.12 and A.13, exploiting that the system configurations have sound modes.

Like the proof of Theorem 3.14, the proof is by induction on $\sharp(\mathcal{T}_{\mathcal{S}}(sc_1)) + \sharp(\mathcal{T}_{\mathcal{S}}(sc_2))$, where $\sharp(T) = \sum_{tr \in T} \sharp(tr)$, which is possible because the system configurations sc_1 and sc_2 are terminating.

The induction basis is proved as in the proof of Theorem 3.14.

For the induction step, we write $P(sc)$ for the sum $\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc)_{\Downarrow mem'})$.

1. For each system configuration sc, we denote with $enabled(sc) \subset \mathbb{N}$ the set of thread indices such that $j \in enabled(sc)$ if and only if there exist $sc' \in SysConf$ and $p > 0$ with $sc \stackrel{j,p}{\Longrightarrow}_{\mathcal{S}} sc'$.
   By the definition of schedulers and the semantics for system configurations, sc' is uniquely determined by sc and j. We denote sc' with $succ_j(sc)$.
2. By Lemma A.10 the following equation holds for every memory mem:
$$\rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem}) = \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))_{\Downarrow mem})$$
   Hence, the following holds:
$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem}) = \sum_{mem' \in [mem]_{LO}} \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))_{\Downarrow mem})$$
   Switching the sums on the right hand side of the equation, we obtain
$$\sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(sc_1)_{\Downarrow mem}) = \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * \sum_{mem' \in [mem]_{LO}} \rho_{\mathcal{S}}(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))_{\Downarrow mem}).$$
   Using the notation introduced for this proof, we write this equation as follows:
$$P(sc_1) = \sum_{j \in enabled(sc_1)} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1)).$$
3. We rewrite this sum as $P(sc_1) = P(sc_1, high) + P(sc_1, low)$, where
$$P(sc_1, high) = \sum_{\substack{j \in enabled(sc_1) \\ \wedge j \notin dom(match)}} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1))$$
   and
$$P(sc_1, low) = \sum_{\substack{j \in enabled(sc_1) \\ \wedge j \in dom(match)}} \rho^{\mathcal{S}}_{sc_1}(j) * P(succ_j(sc_1)).$$
4. Let $j \in enabled(sc_1)$. Then there exist c_1, mem_1', mds', α, p, and sst_1' such that $\langle thr_1[j], mdss_1[j], mem_1\rangle \xrightarrow{\alpha}_{\triangleright} \langle c_1, mds', mem_1'\rangle$ and $(sst_1, obs(thr_1, mem_1)) \stackrel{j,p}{\rightsquigarrow}_{\mathcal{S}} sst_1'$, where $succ_j(sc_1) = \langle update_j(thr_1, c_1, \alpha), mdss_1', mem_1', sst_1'\rangle$, and $mdss_1'$ is obtained by updating $mdss_1$ with mds'.

5. Assume firstly that $j \notin dom(match)$. Then $thr_1[j] \in HCom$. We show that this implies that $P(succ_j(sc_1)) = P(sc_2)$ holds. Let thr_1' be the thread pool in the system configuration $succ_j(sc_1)$.
   a) Since high threads do not modify the values of low variables, and by the definition of $\sim_{lmm}$, the lists of memories $mems_1$ and $mems_2$ make $succ_j(sc_1)$ and sc_2 compatible with low bisimulations modulo low matching and modes for $match'$ (where $match'$ is obtained from $match$ by taking into account that the high thread at position j might have terminated or spawned new, necessarily high, threads).
   b) The same as in (a) also holds for $succ_j(sc_1)$ and $sc_{1,p}$.
   c) By the definition of termination and of sound modes, $succ_j(sc_1)$ also terminates and has sound modes, because it is the result of an execution step of a system configuration that terminates and has sound modes. Moreover, $\sharp(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))) + \sharp(\mathcal{T}_{\mathcal{S}}(sc_2)) < \sharp(\mathcal{T}_{\mathcal{S}}(sc_1)) + \sharp(\mathcal{T}_{\mathcal{S}}(sc_2))$.

d) From $sc_1 \prec_{\mathcal{S}} sc_{1,p}$ and the definition of $\mathcal{S}$-simulations it follows that

$$succ_j(sc_1) \prec_{\mathcal{S}} sc_{1,p}.$$

e) Hence, all assumptions for the induction hypothesis for the pair of system configurations $succ_j(sc_1)$ and sc_2 are satisfied, and it follows that

$$P(succ_j(sc_1)) = P(sc_2).$$

6. Using (5.), we rewrite $P(sc_1, high)$ as follows:

$$P(sc_1, high) = P(sc_2) * \sum_{\substack{j \in enabled(sc_1) \\ \wedge j \notin dom(match)}} \rho^{\mathcal{S}}_{sc_1}(j).$$

By the definition of $l\text{-}\rho_{\mathcal{S}}$, it follows that

$$P(sc_1, high) = P(sc_2) * (1 - l\text{-}\rho_{\mathcal{S}}(sc_1)).$$

Analogously, it follows that $P(sc_2, high) = P(sc_1) * (1 - l\text{-}\rho_{\mathcal{S}}(sc_2))$.

7. Assume now that $j \in dom(match)$ and that $k = match(j)$.
8. Then $\rho^{\mathcal{S}}_{sc_1}(j)/l\text{-}\rho_{\mathcal{S}}(sc_1) = \rho^{\mathcal{S}}_{sc_2}(k)/l\text{-}\rho_{\mathcal{S}}(sc_2)$. The proof is almost exactly as in the proof of Theorem 3.14. The only difference is in the step where the other proof deduces from $mem_1 =_L mem_2$ that $obs(thr_1|_{LCom}, mem_{1,p}) = obs(thr_2|_{LCom}, mem_{2,p})$. Here we only know that mem_1 and mem_2 agree on low variables for which no thread makes a no-read assumption (and not that $mem_1 =_L mem_2$ as in the other proof). But since it follows from the sound modes of sc_1 and sc_2 that the thread pools in sc_1 and sc_2 have compatible modes with *obs* the equality of the observations follows in this proof as well, because mem_1 and mem_2 differ only on variables for which some thread makes a no-read assumption.

9. The inequality $\rho^{\mathcal{S}}_{sc_1}(j) > 0$ holds because $j \in enabled(sc_1)$. Hence, by (8.), the inequality $\rho^{\mathcal{S}}_{sc_2}(k) > 0$ holds. Moreover, it follows from Lemma A.13 that $k \in enabled(sc_2)$.

10. We now show that $P(succ_j(sc_1)) = P(succ_k(sc_2))$.
    a) Since sc_1 and sc_2 are both terminating and have sound modes, $succ_j(sc_1)$ and $succ_k(sc_2)$ are also terminating and have sound modes. Moreover, $\sharp(\mathcal{T}_{\mathcal{S}}(succ_j(sc_1))) + \sharp(\mathcal{T}_{\mathcal{S}}(succ_k(sc_2))) < \sharp(\mathcal{T}_{\mathcal{S}}(sc_1)) + \sharp(\mathcal{T}_{\mathcal{S}}(sc_2))$.
    b) Since $sc_1 \prec_{\mathcal{S}} \langle thr_1|_L, mem_{1,p}, sst_p\rangle$, the execution step from sc_1 to $succ_j(sc_1)$ can be simulated by an execution step of $\langle thr_1|_L, mem_{1,p}, sst_p\rangle$ by the definition of $\mathcal{S}$-simulations, resulting in the configuration

$$\langle getThr(succ_j(sc_1))|_L, mem'_{1,p}, sst'_p\rangle$$

for some sst'_p. Moreover,

$$succ_j(sc_1) \prec_{\mathcal{S}} \langle getThr(succ_j(sc_1))|_L, mem'_{1,p}, sst'_p\rangle.$$

holds. Finally, $getMem(succ_j(sc_1)) =_L mem'_{1,p}$ holds, because thr_1 is FSIFUM-secure.

c) We repeat (b) with sc_2 instead of sc_1. Note that since *obs* is confined to L, the resulting scheduler state is also sst'_p, because $\sharp(thr_1|_L) = \sharp(thr_1|_L)$ and $mem_{1,p} =_L mem_{2,p}$.
d) It follows from (b) and (c) that the assumptions of the theorem are also satisfied for the pair of system configurations $succ_j(sc_1)$ and $succ_k(sc_2)$.
e) Moreover, by Lemma A.13 there exist memories $mem_{1,i}$ and $mem_{2,i}$ for $i \in \{1, \ldots, n')\}$ (n' being the number of low threads in $succ_j(sc_1)$ and $succ_k(sc_2)$) such that the assumptions of the theorem regarding compatibility with strong low bisimulation modulo low matching and modes are also satisfied for the pair of system configurations $succ_j(sc_1)$ and $succ_k(sc_2)$.
f) Hence, all assumptions of the induction hypothesis are satisfied and it follows that

$$P(succ_j(sc_1)) = P(succ_k(sc_2)). \qquad \square$$

11. The remainder of the proof is exactly as in the proof of Theorem 3.14 (Step (11.) and onwards).

Proof sketch of Theorem 5.3. The theorem follows directly with Theorem A.1. $\square$

Lemma A.14. Assume that $thr_1 \sim_{lm} thr_2$, and that $i \in \{1, \ldots, \sharp(thr_1)\}$ with $thr_1[i] \in LCom$. Then $\langle thr_1[i]\rangle \sim_{lm} \langle thr_2[j]\rangle$ where $j = l\text{-}match_{thr_1,thr_2}(i)$. $\lozenge$

Proof. We define the relation $\mathcal{R}$ as follows: $thr_1 \mathcal{R} thr_2$ if and only if there exist thr'_1, thr''_1, thr'_2, and thr''_2 such that thr'_1 and thr'_2 have the same number of low threads, such that thr''_1 and thr''_2 have the same number of low threads, and such that $thr'_1 :: thr_1 :: thr''_1 \sim_{lm} thr'_2 :: thr_2 :: thr''_2$. One easily sees that $\mathcal{R}$ is a low bisimulation modulo low matching. Moreover, it follows from the assumptions of the lemma that $\langle thr_1[i]\rangle \mathcal{R} \langle thr_2[j]\rangle$. Hence, $\langle thr_1[i]\rangle \sim_{lm} \langle thr_2[j]\rangle$. $\square$

Lemma A.15. Define the relation $\mathcal{R}$ on thread configurations with modes as follows: $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ if and only if $\langle c_1\rangle \sim_{lm} \langle c_2\rangle$, $mem_1 =_L mem_2$, and c_1 and c_2 do not affect the mode state. Then $\mathcal{R}$ is a low bisimulation modulo low matching and modes. $\lozenge$

Proof. Assume that $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$. Then $\langle c_1\rangle \sim_{lm} \langle c_2\rangle$, $mem_1 =_L mem_2$, and both c_1 and c_2 do not affect the mode state.

Let $x \in Var$ and $v, v_1, v_2 \in Val$. If $x \in L$ it follows from $mem_1 =_L mem_2$ that $mem_1[x \mapsto v] =_L mem_2[x \mapsto v]$. Hence, $\langle c_1, mds, mem_1[x \mapsto v]\rangle \mathcal{R} \langle c_2, mds, mem_2[x \mapsto v]\rangle$. If $x \in H$ then $mem_1[x \mapsto v_1] =_L mem_2[x \mapsto v_2]$. In consequence, $\langle c_1, mds, mem_1[x \mapsto v_1]\rangle \mathcal{R} \langle c_2, mds, mem_2[x \mapsto v_2]\rangle$. Hence, $\mathcal{R}$ is closed under globally consistent changes.

Moreover, it follows directly from $mem_1 =_L mem_2$ that $mem_1 =_L^{mds} mem_2$.

Assume that $c_1 \notin HCom$ and that $\langle c_1, mds, mem_1\rangle \xrightarrow{new(thr_1, mdss_1)} \langle c'_1, mds', mem'_1\rangle$. Since c_1 does not affect the mode state it follows that $mds' = mds$ and that c'_1 and the commands in thr_1 do not affect the mode state. Moreover, since $\langle c_1\rangle \sim_{lm} \langle c_2\rangle$ there exist c'_2, mem'_2, and thr_2 such that the following conditions are satisfied:

– $\langle c_2, mem_2\rangle \xrightarrow{new(thr_2)} \langle c'_2, mem'_2\rangle$,
– $mem'_1 =_L mem'_2$,
– $\sharp(thr_1|_L) = \sharp(thr_2|_L)$, and
– $update_1(\langle c_1\rangle, c'_1, thr_1) \sim_{lm} update_1(\langle c_2\rangle, c'_2, thr_2)$.

It follows from $update_1(\langle c_1\rangle, c_1', thr_1) \sim_{lm} update_1(\langle c_2\rangle, c_2', thr_2)$ and Lemma A.14 that $\langle c_1'\rangle \sim_{lm} \langle c_2'\rangle$ and that $\langle thr_1[i]\rangle \sim_{lm} \langle thr_2[j]\rangle$ for all $i \in \{1,\ldots,\sharp(thr_1)\}$ with $thr_1[i] \in LCom$ and $j = l\text{-}match_{thr_1,thr_2}(i)$. Moreover, due to $\langle c_2, mem_2\rangle \xrightarrow{new(thr_2)} \langle c_2', mem_2'\rangle$ and the fact that c_2 does not affect the mode state, $\langle c_2, mds, mem_2\rangle \xrightarrow{new(thr_2, mdss_2)} \langle c_2', mds, mem_2'\rangle$ and c_2 and any threads in thr_2 do not affect the mode state. In consequence, $\langle c_1', mds, mem_1'\rangle \mathcal{R} \langle c_2, mds, mem_2'\rangle$ and $\langle thr_1[i], mdss[i], mem_1'\rangle \mathcal{R} \langle thr_2[j], mdss[j], mem_2'\rangle$ for all $i \in \{1,\ldots,\sharp(thr_1)\}$ with $thr_1[i] \in LCom$ and $j = l\text{-}match_{thr_1,thr_2}(i)$.

Hence, Items (a) and (b) in the definition of low bisimulations modulo low matching and modes are satisfied, and Item (c) is satisfied for $f = l\text{-}match_{thr_1,thr_2}$. □

Lemma A.16. The relation $\sim_{mm}$ is a low bisimulation modulo low matching and modes.◊

Proof. The relation $\sim_{mm}$ is closed under globally consistent changes because it is a strong low bisimulation modulo modes.

Assume that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$. Then $mem_1 =_L^{mds} mem_2$ because $\sim_{mm}$ is a strong low bisimulation modulo modes.

Assume that $\langle c_1, mds, mem_1\rangle \sim_{mm} \langle c_2, mds, mem_2\rangle$, $\langle c_1, mds, mem_1\rangle \xrightarrow{new(thr_1, mdss_1)} \langle c_1', mds', mem_1'\rangle$, and $c_1 \notin HCom$. Since $\sim_{mm}$ is a strong low bisimulation modulo modes it follows that there exist c_2', mem_2', and $\alpha_2 = new(thr_2, mdss_2)$ such that the following conditions are satisfied:

- $\langle c_2, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
- $\langle c_1', mds', mem_1'\rangle \sim_{mm} \langle c_2', mds', mem_2'\rangle$,
- $\sharp(thr_1) = \sharp(thr_2)$, and
- $\langle thr_1[i], mdss_1[i], mem_1'\rangle \sim_{mm} \langle thr_2[i], mdss_2[i], mem_2'\rangle$ for all $i \in \{1,\ldots,\sharp(thr_1)\}$.

Hence, Items (a) and (b) of the definition of low bisimulations modulo low matching and modes are satisfied, and Item (c) is satisfied by defining f as the function with domain $\{1,\ldots,\sharp(thr_1)\}$ and $f(i) = i$ for all $i \in dom(f)$. □

A.9 Security Type System for FSIFUM-security

Like in the case of FSI-security, high commands are FSIFUM-secure.

Lemma A.17. If $c \in HCom$ then c is FSIFUM-secure for any initial mode state. ◊

Proof. Define the relation $\mathcal{R}$ such that $\langle c_1, mds, mem_1\rangle \mathcal{R} \langle c_2, mds, mem_2\rangle$ if and only if $c_1 = c_2$, $c_1 \in HCom$, and $mem_1 =_L^{mds} mem_2$. Then it follows immediately that $\mathcal{R}$ is a strong low bisimulation modulo low matching and modes. In consequence, $c \sim_{lmm}^{mds} c$ for all mode states mds, and, hence, c is FSIFUM-secure for all mds. □

Moreover, like for the type system for FSI-security, commands typable with $mdf = high$ are high.

Lemma A.18. Let $c \in Com$. Assume that c is typable with type $(high, stp)$ for some $stp \in \{low, high\}$ in the security type system for FSIFUM-security. Then $c \in HCom$. ◊

Proof. The proof is exactly like the proof of Lemma 3.2, the analogous lemma for the type system for FSI-security (the typing rules of the type system for FSIFUM-security impose exactly the same requirements as the typing rules of the type system for FSI-security regarding *mdf* and *stp*). □

Moreover, we have a subject reduction result for the type system for FSIFUM-security that is a combination of the results for the type systems for FSI-security and SIFUM-security.

Lemma A.19. Let $c \in Com$. Assume that $\vdash \Lambda\ \{c\}\ \Lambda' : (mdf, stp)$ is derivable, and that $\langle c, mds, mem\rangle \xrightarrow{new(thr,mdss)} \langle c', mdss', mem'\rangle$. Then there exist Λ'', mdf', and stp' such that $\vdash \Lambda''\ \{c\}\ \Lambda' : (mdf', stp')$ is derivable where $mdf \sqsubseteq mdf'$ and $stp' \sqsubseteq stp$. Moreover, if mds is consistent with Λ then mds' is consistent with Λ''. ◊

Proof. The proof is by induction on the derivation of $\vdash \Lambda\ \{c\}\ \Lambda' : (mdf, stp)$. The argumentation is exactly as in the proofs of Lemma 3.1 and Theorem 4.6. □

Proof of Theorem 5.6. The proof follows the same approach as the proof of Theorem 4.7, where we adapt the constructed bisimulation relation to encode also information about the pair (mdf, stp) in the typing judgments, and handle high commands separately, as in the proof of Theorem 3.15.

For each partial environment Λ' we define $\mathcal{R}^{\Lambda'}$ as follows:

$$\mathcal{R}^{\Lambda'} = \{(\langle c, mds, mem_1\rangle\ ,\ \langle c, mds, mem_2\rangle) \mid \exists \Lambda, stp : \\ \vdash \Lambda\ \{c\}\ \Lambda' : (low, stp) \wedge mds \in cons(\Lambda) \wedge mem_1 =_\Lambda mem_2\}$$

We define $\mathcal{R} = \sim_{lmm}\ \cup \left(\bigcup_{\Lambda' : Var \rightharpoonup \{low,high\}} \mathcal{R}^{\Lambda'}\right)$ and show that $\mathcal{R}$ is a low bisimulation modulo low matching and modes.

Note that all properties required by low bisimulations modulo low matching and modes are satisfied for thread configurations that are related by $\sim_{lmm}$, and, hence, it suffices to show the properties for those thread configurations that are related by $\mathcal{R}^{\Lambda'}$ for some Λ'.

The symmetry of $\mathcal{R}$ follows immediately from the definition. and the closure under globally consistent changes is proved exactly as in the proof of Theorem 4.7. Also, proving that $mem_1 =_L^{mds} mem_2$ whenever $\langle c_1, mds, mem_1\rangle\ \mathcal{R}\ \langle c_2, mds, mem_2\rangle$ is exactly as in the proof of Theorem 4.7.

It remains to show that for all $c, c_1' \in Com$, all $mds, mds' \in Mds$, all mem_1, mem_2, and $mem_1' \in Mem$, all $\alpha_1 = new(thr_1, mdss_1) \in Lab_m$, all $\Lambda, \Lambda' : Var \rightharpoonup \{low, high\}$, and all $stp \in \{low, high\}$ with $c \notin HCom$, $\langle c, mds, mem_1\rangle \xrightarrow{\alpha_1} \langle c_1', mds', mem_1'\rangle$, $\vdash \Lambda\ \{c\}\ \Lambda'$, $mds \in cons(\Lambda)$, and $mem_1 =_\Lambda mem_2$, there exist c_2', $\alpha_2 = new(thr_2, mdss_2)$, and mem_2' such that the following conditions are satisfied:

- $\langle c, mds, mem_2\rangle \xrightarrow{\alpha_2} \langle c_2', mds', mem_2'\rangle$,
- $\langle c_1', mds', mem_1'\rangle\ \mathcal{R}\ \langle c_2', mds', mem_2'\rangle$, and
- there exist an injective partial function $match : \{1, \ldots, \sharp(thr_1)\} \rightharpoonup \{1, \ldots, \sharp(thr_2)\}$ such that $\langle thr_1[i], mdss_1[i], mem_1'\rangle\ \mathcal{R}\ \langle thr_2[j], mdss_2[j], mem_2'\rangle$ for all $i \in dom(match)$ and $j = match(i)$, $thr_1[i] \in HCom$ for all $i \notin dom(match)$, and $thr_2[i] \in HCom$ for all $i \notin img(match)$.

The proof is by induction on the derivation of the judgment $\vdash \Lambda\ \{c\}\ \Lambda' : (low, stp)$.

The cases for the derivations via rules [ANNO], [SKIP], [ASS$_1$], [ASS$_2$], [SPAWN], and [SUB] are derived directly from the corresponding cases in the proof of Theorem 4.7, and we do not repeat them here.

The cases for the derivations via rules [IF] and [WHILE] are also like in the proof of Theorem 4.7, taking into account that $mdf = low$ in the typing judgment and, because $d \sqsubseteq mdf$ in both rule [IF] and [WHILE] it follows that $d = low$.

The case for the derivation via rule [SEQ] is like in Theorem 4.7, in which in addition a case distinction on the cases $stp_1 = low$ and $stp_1 = high$ is performed, proceeding as in the proof of Theorem 3.15. □

Proof of Theorem 5.7. Since $\vdash thr$ is derivable, by the typing rule [$\mathsf{PAR}_{\mathcal{S}}$] there exist a list of mode states $mdss$, partial environments $\Lambda_1, \ldots, \Lambda_{\sharp(thr)}$ and $\Lambda'_1, \ldots, \Lambda'_{\sharp(thr)}$, and security domains $mdf_1, \ldots, mdf_{\sharp(thr)}$ and $stp_1, \ldots, stp_{\sharp(thr)}$ such that $\vdash \Lambda_i \ \{thr[i]\} \ \Lambda'_i : (mdf_i, stp_i)$ is derivable, $dma(x) \sqsubseteq \Lambda_i\langle x\rangle$ for all $x \in Var$, and $mdss[i]$ is consistent with Λ_i for all $i \in \{1, \ldots, \sharp(thr)\}$, and such that $(thr, mdss)$ has sound modes for $\mathcal{S}$. Hence, by Theorem 5.6, it holds for all $i \in \{1, \ldots, \sharp(thr)\}$ that $\langle thr[i], mdss[i], mem_1\rangle \sim_{lmm} \langle thr[i], mdss[i], mem_2\rangle$ for all $mem_1, mem_2 \in Mem$ that satisfy $mem_1(x) = mem_2(x)$ for all $x \in Var$ with $\Lambda_i\langle x\rangle = low$. Since $mdss[i] \in cons(\Lambda_i)$ and $dma(x) \sqsubseteq \Lambda_i\langle x\rangle$ for all $x \in Var$ it follows that $\langle thr[i], mdss[i], mem_1\rangle \sim_{lmm} \langle thr[i], mdss[i], mem_2\rangle$ for all $mem_1 =_L mem_2$. I.e., $thr[i]$ is FSIFUM-secure for $mdss[i]$ for all $i \in \{1, \ldots, \sharp(thr)\}$. As modes are sound for $\mathcal{S}$, with Theorem 5.3 it follows that thr is $\mathcal{S}$-noninterferent. □

A.10 Relation between Type-based and PDG-based Information Flow Analysis

This appendix contains the proofs for the relation between type-based and PDG-based information flow analysis for sequential programs from Chapter 6. Moreover, it contains the soundness proof for the PDG-based information flow analysis for multi-threaded programs from the same chapter.

Before proving Theorem 6.3 we prove several lemmas that relate paths in the graph $PDG(CFG_c^{I,O})$ where c is of the form $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$, $\mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$, or $c_1; c_2$ to paths in the graphs $PDG(CFG_{c_1}^{I,O})$ and (if applicable) $PDG(CFG_{c_2}^{I,O})$. In the proofs, we write $p + k$ for the path that is obtained from p by adding k to each node on p that is a natural number, and leaving *start* and *stop* unchanged. Moreover, we write $p - k$ for the path that is obtained from p by subtracting k from each node on p that is a natural number, and leaving *start* and *stop* unchanged.

Lemma A.20. Let $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ and $x, y \in Var$. Then the following hold:

1. There is a path from *in* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if one of the following conditions are satisfied:
   a) there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{x\},\{y\}})$,
   b) there is a path from *in* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$,
   c) $x \in vars(e)$ and there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$, or
   d) $x \in vars(e)$ and there is a path from *start* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$.
2. There is a path from *start* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$ if and only if there is such a path in the graph $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in the graph $PDG(CFG_{c_2}^{\{x\},\{y\}})$. ◊

Proof. We firstly prove Statement (1) of the lemma.

1. By the construction of the CFG for commands the following holds:
   a) Let $n, n' \notin \{1, start, stop\}$ be nodes of CFG_c. Then n' is data (control) dependent on n for CFG_c if and only if $n' \ominus 1$ is data (control) dependent on $n \ominus 1$ for CFG_{c_1} or $n' \ominus 1 \ominus \sharp(c_1)$ is data (control) dependent on $n \ominus 1 \ominus \sharp(c_1)$ for CFG_{c_2}. Moreover, n is not data dependent on 1 and 1 is not data dependent on n. Moreover, n is control dependent on 1 if and only if $n \ominus 1$ is control dependent on *start* for CFG_{c_1} or $n \ominus 1 \ominus \sharp(c_1)$ is control dependent on *start* for CFG_{c_2}.
2. By the construction of the PDG and the construction of the CFG for commands we obtain the following:
   a) Let $n \notin \{1, start, stop\}$ be a node of CFG_c. Then there is an edge (in, n) in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there is an edge $(in, n \ominus 1)$ in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or an edge $(in, n \ominus 1 \ominus \sharp(c_1))$ in $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
   Moreover, there is an edge (n, out) in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there is an edge $(n \ominus 1, out)$ in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or an edge $(n \ominus 1 \ominus \sharp(c_1), out)$ in $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
   b) There is an edge $(in, 1)$ in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if $x \in vars(e)$. Moreover, there is no edge $(1, out)$ in $PDG(CFG_c^{\{x\},\{y\}})$.
   c) There is an edge (in, out) in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there is an edge (in, out) in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
3. Assume that there is a path p from *in* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$.
   a) If $p = \langle in, out\rangle$, then by (2c) p is also a path in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
   b) If $p = \langle in, 1\rangle :: p'$ for some p', then, by (2b), $x \in vars(e)$ and $p' \neq \langle out\rangle$. Moreover, by (1a) the first node in p' is control dependent on *start* in CFG_c, and, since due to (1a) all edges in p' derive from edges in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or $PDG(CFG_{c_1}^{\{x\},\{y\}})$, $\langle start\rangle :: (p' \ominus 1)$ is a path from *start* to *out* in the graph $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in the graph $PDG(CFG_{c_1}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$ (namely the first node of p').
   c) If $p \neq \langle in, out\rangle$ and $p \neq \langle in, 1\rangle :: p'$ for some p', then, using (1a) and (2a), $p - 1$ is a path in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in $PDG(CFG_{c_1}^{\{x\},\{y\}})$.
4. Assume that there is a path p from *in* to *out* in $PDG(CFG_{c_1}^{\{x\},\{y\}})$. Then, by (1a), (2a), and (2c), $p + 1$ is a path in $PDG(CFG_c^{\{x\},\{y\}})$.
5. Assume that there is a path from *in* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$. Then, by (1a), (2a), and (2c), $p + (1 + \sharp(c_1))$ is a path in $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
6. Assume that $x \in vars(e)$ and that there is a path from *start* to *out* in the graph $PDG(CFG_{c_1}^{\{x\},\{y\}})$ that contains $n \notin \{start, out\}$, i.e., of the form $\langle start\rangle :: p'$. Then, by (1a), (2a), and (2b), the sequence $\langle in, 1\rangle :: (p' + 1)$ is a path in $PDG(CFG_c^{\{x\},\{y\}})$.
7. Assume finally that $x \in vars(e)$ and that there is a path from *start* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$, i.e., of the form $\langle start\rangle :: p'$. Then, by (1a), (2a), and (2b), the sequence $\langle in, 1\rangle :: (p' + (1 + \sharp(c_1)))$ is a path in $PDG(CFG_c^{\{x\},\{y\}})$.

Statement (2) of the lemma is seen as follows: Using (1) and (2) of the proof of the first statement of the lemma, paths from *start* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ that contain a node $n \notin \{start, out\}$ are exactly the paths obtained from paths from *start* to *out* in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ and $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contain a node $n \notin \{start, out\}$, respec-

tively, into which the node representing the guard of the conditional (Node 1) is inserted directly after node *start*, and 1 respectively $1 + \sharp(c_1)$ is added to each node. □

Lemma A.21. Let $c = c_1; c_2$ and $x, y \in Var$. Then the following hold:

1. There is a path from *in* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there exists $z \in Var$ such that there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and a path from *in* to *out* in $PDG(CFG_{c_2}^{\{z\},\{y\}})$.
2. There is a path from *start* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$ if and only if one of the following conditions is satisfied:
   a) there exists $z \in Var$ such that there is a path from *start* to *out* that contains a node $n \notin \{start, out\}$ in the graph $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and such that there is a path from *in* to *out* in $PDG(CFG_{c_2}^{\{z\},\{y\}})$, or
   b) there is a path from *start* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$. ◊

Proof. We firstly prove Statement (1).

1. By the construction of the CFG for commands, the following hold:
   a) Let $n, n' \notin \{start, stop\}$ be nodes of CFG_c. Then n' is control dependent on n for CFG_c, if and only if n' is control dependent on n for CFG_{c_1}, or if $n \ominus \sharp(c_1)$ is control dependent on $n \ominus \sharp(c_1)$ for CFG_{c_2}.
   b) Let $n, n' \notin \{start, stop\}$ be nodes of CFG_c. Then n' is data dependent on n for CFG_c if and only if one of the following conditions is satisfied:
      i. n' is data dependent on n for CFG_{c_1}
      ii. $n' \ominus \sharp(c_1)$ is data dependent on $n \ominus \sharp(c_1)$ for CFG_{c_2}
      iii. There exists $z \in Var$ such that $z \in def_{c_1}(n)$, $z \in use_{c_2}(n' \ominus \sharp(c_1))$, a definition of z at n reaches *stop* in CFG_{c_1}, and a definition of z at *start* reaches $n' \ominus \sharp(c_1)$ in CFG_{c_2}.

   Note that condition (iii) is equivalent to the existence of $z \in Var$ such that there is an edge from n to *out* in $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and an edge from *in* to $(n' \ominus \sharp(c_1))$ in $PDG(CFG_{c_2}^{\{z\},\{y\}})$.
2. By the definition of PDGs and the definition of CFGs of commands there is an edge (in, out) in $PDG(CFG_c^{\{z\},\{y\}})$ if and only if $x = y$ and (in, out) is an edge both in $PDG(CFG_{c_1}^{\{z\},\{y\}})$ and in $PDG(CFG_{c_2}^{\{z\},\{y\}})$.
3. Assume that there is a path p from *in* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$.
   a) If $p = \langle in, out \rangle$, then, by (2), we conclude (setting $z = x = y$).
   b) If p contains, besides *in* and *out*, only nodes in the set $\{1, \ldots, \sharp(c_1)\}$, then a definition of y at the one but last node of p reaches *stop* in CFG_c, and, hence, a definition of y at *start* reaches *stop* in CFG_{c_2}. Hence, p is a path in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ and $\langle in, out \rangle$ is a path in $PDG(CFG_{c_2}^{\{y\},\{y\}})$. Thus, we conclude setting $z = y$.
   c) If p contains, besides *in* and *out*, only nodes in the set $\{\sharp(c_1) + 1, \ldots, \sharp(c_1; c_2)\}$, we argue as in the previous case with roles of c_1 and c_2 switched, concluding by setting $z = x$.
   d) If p contains nodes in both $\{1, \ldots, \sharp(c_1)\}$ and $\{\sharp(c_1)+1, \ldots, \sharp(c_1; c_2)\}$, then, by the definition of the CFG, $p = p_1 :: p_2$ where p_1 contains only nodes in $\{1, \ldots, \sharp(c_1)\}$ and p_2 contains only nodes in $\{\sharp(c_1) + 1, \ldots, \sharp(c_1; c_2)\}$. With (1b.iii) it follows that there is z such that $p_1 :: \langle out \rangle$ is a path in $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and $\langle in \rangle :: p_2$ is a path in $PDG(CFG_{c_2}^{\{z\},\{y\}})$.

4. Now assume that there exists $z \in \mathit{Var}$ and paths p_1 and p_2 from *in* to *out* in $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and in $PDG(CFG_{c_2}^{\{z\},\{y\}})$, respectively. With (1b.iii) it follows that there exists a path from *in* to *out* in $PDG(CFG_{c}^{\{x\},\{z\}})$ (by joining these two paths).

We now prove Statement (2).

1. By the construction of the CFG for $c_1; c_2$ and the definition of postdominance, a node n is control dependent on *start* in CFG_c if and only if n is control dependent on *start* in CFG_{c_1} or $n \ominus \sharp(c_1)$ is control dependent on *start* in CFG_{c_2}.
2. Assume that p is a path from *start* to *out* in $PDG(CFG_{c}^{\{x\},\{y\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$.
   a) If p contains, besides *start* and *out*, only nodes in the set $\{\sharp(c_1)+1, \ldots, \sharp(c_1; c_2)\}$, then $p - \sharp(c_1)$ is a path from *start* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$ (by (1), (1a), (1b), and the proof of the first statement of the lemma).
   b) Otherwise, using (1) and arguing analogously to the proof of the first part of the lemma, there exists z such that there is a path from *start* to *out* in the graph $PDG(CFG_{c_1}^{\{x\},\{z\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$ and a path from *in* to *out* in the graph $PDG(CFG_{c_1}^{\{z\},\{y\}})$.
3. The backwards direction (assuming paths in the graphs $PDG(CFG_{c_1}^{\{x\},\{y\}})$ and $PDG(CFG_{c_2}^{\{x\},\{y\}})$) is analogous to the previous cases. □

Lemma A.22. Let $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$. Then the following hold:

1. There is a path from *in* to *out* in $PDG(CFG_{c}^{\{x\},\{y\}})$ if and only if there exist $z_1, \ldots, z_k$ (for some $k > 1$) with $z_1 = x$ and $z_k = y$ such that for each $i \in \{1, \ldots, k-1\}$ one of the following conditions is satisfied:
   a) there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$, or
   b) $z_i \in \mathit{vars}(e)$ and there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$.
2. There is a path from *start* to *out* in $PDG(CFG_{c}^{\{x\},\{y\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$ if and only if there exist $z_1, \ldots, z_k$ (for some $k > 0$) such that $z_k = y$, there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{x\},\{z_1\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$, and for each $i \in \{1, \ldots, k-1\}$ one of the following conditions is satisfied:
   a) there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$, or
   b) $z_i \in \mathit{vars}(e)$ and there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$. ◊

Proof. We start proving Statement (1) of the lemma.

1. By the construction of the CFG for commands the following hold:
   a) Let $n, n' \notin \{1, \mathit{start}, \mathit{stop}\}$ be nodes of CFG_c. Then n' is data dependent on n for CFG_c if and only if one of the following holds:
      i. $n' \ominus 1$ is data dependent on $n \ominus 1$ for CFG_{c_1} or
      ii. there exists $z \in \mathit{Var}$ such that $(n \ominus 1, \mathit{out})$ is an edge in $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and $(\mathit{in}, n' \ominus 1)$ is an edge in $PDG(CFG_{c_1}^{\{z\},\{y\}})$.

      Moreover, Node 1 is data dependent on n if and only if there exists $z \in \mathit{vars}(e)$ such that (n, out) is an edge in $PDG(CFG_{c_1}^{\{x\},\{z\}})$.

   b) Let $n, n' \notin \{1, start, stop\}$ be nodes of CFG_c. Then n' is control dependent on n for CFG_c if and only if $n' \ominus 1$ is control dependent on $n \ominus 1$ for CFG_{c_1}. Moreover, n is control dependent on 1 if and only if $n \ominus 1$ is control dependent on *start* for CFG_{c_1}.
2. By the construction of the PDG and the construction of the CFG for commands the following hold:
   a) Let $n \notin \{1, start, stop\}$ be a node of CFG_c. Then there is an edge (in, n) in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there is an edge $(in, n \ominus 1)$ in $PDG(CFG_{c_1}^{\{x\},\{y\}})$. Moreover, there is an edge (n, out) in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there is an edge $(n \ominus 1, out)$ in $PDG(CFG_{c_1}^{\{x\},\{y\}})$.
   b) There is an edge $(in, 1)$ in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if $x \in vars(e)$. Moreover, there is no edge $(1, out)$ in $PDG(CFG_c^{\{x\},\{y\}})$.
   c) There is an edge (in, out) in $PDG(CFG_c^{\{x\},\{y\}})$ if and only if there is an edge (in, out) in $PDG(CFG_{c_1}^{\{x\},\{y\}})$.
3. Assume that there is a path p from *in* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$.
   a) If $p = \langle in, out \rangle$, then by (2c) it is also a path in $PDG(CFG_{c_1}^{\{x\},\{y\}})$.
   b) If p consists of more than 2 nodes, then by (1) and (2) p consists of segments that correspond to paths in $PDG(CFG_{c_1}^{\{x\},\{y\}})$, potentially separated by Node 1 (representing the guard of the loop), where the edges (n, n') between these segments correspond to edges (n, out) in $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$ and (in, n') respectively $(start, n')$ in $PDG(CFG_{c_1}^{\{z_{i+1}\},\{z_{i+2}\}})$ for a sequence of z_i.
   c) Hence, there exist $z_1, \ldots, z_k$ and paths from *in* to *out* respectively from *start* to *out* (containing a node $n \notin \{start, out\}$) in the PDGs $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$, where $z_1 = x$ and $z_k = y$.
4. Assume now that there exist $z_1, \ldots, z_k$ with $z_1 = x$ and $z_k = y$ and paths from *in* to *out* respectively from *start* to *out* (containing a node $n \notin \{start, out\}$) in the PDGs $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$.
   a) Using (1) and (2), these paths can be concatenated using the same arguments as in Lemmas A.20 and A.21, where Node *start* is replaced by Node 1.

Statement (2) of the lemma is proven analogously to Statement (1), with the difference being that the first segment of the path begins at Node *start*, not at Node *in*. □

We prove a generalization of Theorem 6.3 that permits arbitrary security domains for the program counter in the typing judgment. The generalization permits an inductive proof over the construction of the command c.

Lemma A.23. Let $c \in Com$, $\Gamma : Var \to \mathcal{D}$, $pc \in \mathcal{D}$, and $y \in Var$. Let Γ' be the environment with $pc \vdash \Gamma\,\{c\}\,\Gamma'$. Let $X \subseteq Var$ be such that $x \in X$ if and only if there is a path $\langle in, \ldots, out \rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. Then one of the following two conditions is satisfied:

1. $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$, and there is a path from *start* to *out* in $PDG(CFG_c^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$ for some $x \in Var$.
2. $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$, and there is no such path in $PDG(CFG_c^{\{x\},\{y\}})$ for any $x \in Var$.

◊

Proof. The proof is by induction on the structure of the command c.

Assume that $c = \mathsf{skip}$:

1. By the typing rule [SKIP], $\Gamma = \Gamma'$. Hence, $\Gamma(y) = \Gamma'(y)$.
2. The node *start* is the only node in CFG_{skip} with two outgoing edges, and, hence, all control dependency edges in $PDG(CFG_{\mathsf{skip}}^{\{x\},\{y\}})$ originate at *start*.
3. Node 1 (representing the skip-statement) has empty def and use sets by Definition 6.8. Hence, by Definition 6.11 $PDG(CFG_{\mathsf{skip}}^{\{x\},\{y\}})$ contains no data dependency edges from or to Node 1, and it contains an edge (in, out) if and only if $x = y$.
4. By (2) and (3), there is a path from *in* to *out* in $PDG(CFG_{\mathsf{skip}}^{\{x\},\{y\}})$ if and only if $x = y$, i.e., $X = \{y\}$.
5. Moreover, by (2) and (3) there is no path from *start* to *out* in $PDG(CFG_{\mathsf{skip}}^{\{x\},\{y\}})$ but the path $\langle start, out\rangle$.
6. Hence, by (4) and (5), it suffices to show that $\Gamma'(y) = \Gamma(y)$ to conclude this case, and $\Gamma'(y) = \Gamma(y)$ holds by (1).

Assume that $c = z{:=}e$:

1. By the same argument as in the case for skip, all control dependency edges in $PDG(CFG_{z:=e}^{\{x\},\{y\}})$ originate at *start*.
2. By Definition 6.8, $def_c(1) = \{z\}$ and $use_c(1) = vars(e)$ (where 1 is the node representing the assignment). Hence, by Definition 6.11 there is a data dependency edge $(in, 1)$ in $PDG(CFG_{z:=e}^{\{x\},\{y\}})$ if and only if $x \in vars(e)$ and there is a data dependency edge $(1, out)$ if and only if $y = z$. Moreover, there is an edge (in, out) if and only if $x = y$ and $x \neq z$.
3. Assume that $y = z$.
   a) By (1) and (2), there is a path from *in* to *out* if and only if $x \in vars(e)$ (the path $\langle in, 1, out\rangle$). Hence, $X = vars(e)$.
   b) Since Node 1 is control dependent on *start* there is a path from *start* to *out* that contains a node $n \notin \{start, out\}$ (the path $\langle start, 1, out\rangle$).
   c) Hence, we must show that $\Gamma'(y) = pc \sqcup \left(\bigsqcup_{x \in vars(e)} \Gamma(x)\right)$. This equality holds due to the typing rules [ASS] and [EXP].
4. Assume now that $y \neq z$.
   a) By (1) and (2), there is a path from *in* to *out* if and only if $x = y$ and $x \neq z$ (the path $\langle in, out\rangle$). Hence, $X = \{y\}$.
   b) Moreover, by (1) and (2) there is no path from *start* to *out* but the path $\langle start, out\rangle$.
   c) Hence, we must show that $\Gamma'(y) = \Gamma(y)$. This holds due to the typing rule [ASS].

Assume that $c = c_1; c_2$:

1. By typing rule [SEQ] there is an environment Γ'' such that $pc \vdash \ \Gamma \ \{c_1\} \ \Gamma''$ and $pc \vdash \ \Gamma'' \ \{c_2\} \ \Gamma'$.
2. Let X_2 be the set of variables such that $x \in X_2$ if and only if there is a path from *in* to *out* in $PDG(CFG_{c_2}^{\{x\},\{y\}})$. By the induction hypothesis for k and c_2 (instantiating the environment with Γ'') one of the following conditions is satisfied:
   a) $\Gamma'(y) = pc \sqcup \left(\bigsqcup_{x \in X_2} \Gamma''(x)\right)$ and there is a path from *start* to *out* in the graph $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$, or
   b) $\Gamma'(y) = \bigsqcup_{x \in X_2} \Gamma''(x)$, and there is no such path in $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
3. For each $z \in X_2$, let X_z be the set of variables such that $x \in X_z$ if and only if there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{x\},\{z\}})$. For each $z \in X_2$, by the induction hypothesis for c_1 (instantiating the environment with Γ and the variable with z) one of the following conditions is satisfied:
   a) $\Gamma''(z) = pc \sqcup \left(\bigsqcup_{x \in X_z} \Gamma(x)\right)$ and there is a path from *start* to *out* in the graph

$PDG(CFG_{c_1}^{\{x\},\{z\}})$ that contains a node $n \notin \{start, out\}$, or
   b) $\Gamma''(z) = \bigsqcup_{x \in X_z} \Gamma(x)$, and there is no such path in $PDG(CFG_{c_1}^{\{x\},\{z\}})$.
4. By Lemma A.21, there is a path from *in* to *out* in $PDG(CFG_{c_1;c_2}^{\{x\},\{y\}})$ if and only if there exists $z \in Var$ such that there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{x\},\{z\}})$ and a path from *in* to *out* in $PDG(CFG_{c_2}^{\{z\},\{y\}})$. Hence, it follows from the definitions of X_2 and X_z that $X = \bigcup_{z \in X_2} X_z$.
5. We distinguish the cases (2a) and (2b). Assume firstly that (2a) holds.
   a) Due to (2a) and Lemma A.21, for all $z \in Var$ there is a path from *start* to *out* in $PDG(CFG_{c_1;c_2}^{\{z\},\{y\}})$ that contains a node $n \notin \{start, out\}$. Hence, we must show that $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$.
   b) Due to (2a), $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{z \in X_2} \Gamma''(z)\big)$.
   c) Due to (3), for each $z \in X_2$ either $\Gamma''(z) = pc \sqcup \big(\bigsqcup_{x \in X_z} \Gamma(x)\big)$ or $\Gamma''(z) = \big(\bigsqcup_{x \in X_z} \Gamma(x)\big)$ holds.
   d) It follows from (b) and (c) that $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{z \in X_2} \bigsqcup_{x \in X_z} \Gamma(x)\big)$. Hence, by (4), $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$.
6. Assume now that (2b) holds.
   a) Due to (2b), $\Gamma'(y) = \bigsqcup_{z \in X_2} \Gamma''(z)$.
   b) For each $z \in X_2$, either (3a) or (3b) holds. We distinguish the cases that (3b) holds for all $z \in X_2$ and that (3b) does not hold for some $z \in X_2$.
   c) Assume firstly that (3b) holds for all $z \in X_2$.
      i. By Lemma A.21, $PDG(CFG_{c_1;c_2}^{\{x\},\{y\}})$ does not contain a path from *start* to *out* but for the path $\langle start, out \rangle$.
      ii. For all $z \in X_2$, it follows from (3b) that $\Gamma''(z) = \big(\bigsqcup_{x \in X_z} \Gamma(x)\big)$.
      iii. From (a) and (ii) it follows that $\Gamma'(y) = \bigsqcup_{z \in X_2} \bigsqcup_{x \in X_z} \Gamma(x)$. Hence, by (4), $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$.
   d) Assume now that there exists $z \in X_2$ such that (3a) holds for z.
      i. Hence, $\Gamma''(z) = pc \sqcup \big(\bigsqcup_{x \in X_z} \Gamma(x)\big)$.
      ii. Moreover, $PDG(CFG_{c_1}^{\{x\},\{z\}})$ contains a path from *start* to *out* that contains a node $n \notin \{start, out\}$.
      iii. Since $z \in X_2$ there is a path from *in* to *out* in $PDG(CFG_{c_2}^{\{z\},\{y\}})$.
      iv. By Lemma A.21, (ii), and (iii), $PDG(CFG_{c_1;c_2}^{\{x\},\{y\}})$ contains a path from *start* to *out* that contains a node $n \notin \{start, out\}$.
      v. Due to (3), for each $z \in X_2$ either $\Gamma''(z) = pc \sqcup \big(\bigsqcup_{x \in X_z} \Gamma(x)\big)$ or $\Gamma''(z) = \big(\bigsqcup_{x \in X_z} \Gamma(x)\big)$ holds.
      vi. From (a), (i), and (v), it follows that $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{z \in X_2} \bigsqcup_{x \in X_z} \Gamma(x)\big)$. Hence, by (4), $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$.

Assume that $c =$ if e then c_1 else c_2 fi:

1. Let $\{z_1, \ldots, z_l\} = vars(e)$, and $d = dma(z_1) \sqcup \ldots \sqcup dma(z_l)$.
2. By the typing rule [IF], there are environments Γ_1' and Γ_2' such that $\Gamma' = \Gamma_1' \sqcup \Gamma_2'$, $pc \sqcup d \vdash \Gamma \{c_1\} \Gamma_1'$, and $pc \sqcup d \vdash \Gamma \{c_2\} \Gamma_2'$.
3. For $i \in \{1, 2\}$, let X_i be the set of variables such that $x \in X_i$ if and only if there is a path from *in* to *out* in $PDG(CFG_{c_i}^{\{x\},\{y\}})$.
4. Applying the induction hypothesis for both k and c_1 and k and c_2 (instantiating the program counter security level with $pc \sqcup d$), it follows that for $i \in \{1, 2\}$ one of the following two conditions is satisfied:
   a) $\Gamma_i'(y) = pc \sqcup d \sqcup \big(\bigsqcup_{x \in X_i} \Gamma(x)\big)$ and there is a path from *start* to *out* in the graph

$PDG(CFG_{c_i}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$, or
   b) $\Gamma'_i(y) = \bigsqcup_{x \in X_i} \Gamma(x)$ and there is no such path in $PDG(CFG_{c_i}^{\{x\},\{y\}})$.
5. It follows from Lemma A.20 that there exists a path from *in* to *out* in the graph $PDG(CFG^{\{x\},\{y\}}_{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}})$ if and only if there is such a path in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in $PDG(CFG_{c_2}^{\{x\},\{y\}})$ (i.e., $x \in X_1 \cup X_2$), or if $x \in vars(e)$ and there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in $PDG(CFG_{c_2}^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$.
6. We do a case distinction on whether there exists a path from *start* to *out* in the graph $PDG(CFG^{\{x\},\{y\}}_{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}})$ that contains a node $n \notin \{start, out\}$. Assume firstly that there is such a path.
   a) Hence, by Lemma A.20, there is such a path in the graph $PDG(CFG_{c_1}^{\{x\},\{y\}})$ or in the graph $PDG(CFG_{c_2}^{\{x\},\{y\}})$. Assume without loss of generality that there is such a path in $PDG(CFG_{c_1}^{\{x\},\{y\}})$.
   b) Then, by (5), $X = X_1 \cup X_2 \cup vars(e)$.
   c) Moreover, by (4), $\Gamma'_1(y) = pc \sqcup d \sqcup \big(\bigsqcup_{x \in X_1} \Gamma(x)\big)$.
   d) Moreover, by (4), either $\Gamma'_2(y) = pc \sqcup d \sqcup \big(\bigsqcup_{x \in X_2} \Gamma(x)\big)$ or $\Gamma'_2(y) = \bigsqcup_{x \in X_2} \Gamma(x)$.
   e) Hence, since $\Gamma' = \Gamma'_1 \sqcup \Gamma'_2$, it follows from (1), (c), (d), and (e) that $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$.
7. Assume now that there is no path $\langle start, \ldots, out\rangle$ in $PDG(CFG^{\{x\},\{y\}}_{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}})$ that contains a node $n \notin \{start, out\}$.
   a) Hence, by Lemma A.20, there is no such path in the graph $PDG(CFG_{c_1}^{\{x\},\{y\}})$ and no such path in the graph $PDG(CFG_{c_2}^{\{x\},\{y\}})$.
   b) In consequence, by (5), $X = X_1 \cup X_2$.
   c) Moreover, by (4), $\Gamma'_i(y) = \bigsqcup_{x \in X_i} \Gamma(x)$ for $i \in \{1, 2\}$.
   d) Hence, since $\Gamma' = \Gamma'_1 \sqcup \Gamma'_2$, it follows from (b) and (c) that $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$.

Assume that $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$

1. Since $pc \vdash \Gamma\ \{\mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}\}\ \Gamma'$ is derivable it follows from typing rule [WHILE] that there exist $k \in \mathbb{N}$ and sequences $\Gamma'_0, \ldots, \Gamma'_{k+1}$, $\Gamma''_0, \ldots, \Gamma''_k$, and $d_0, \ldots, d_k$ such that the following hold (for $0 \leq i \leq k$):
   a) $\Gamma'_0 = \Gamma$,
   b) $\Gamma'_{k+1} = \Gamma'_k = \Gamma'$,
   c) $\Gamma'_i \vdash e : d_i$,
   d) $pc \sqcup t_i \vdash \Gamma'_i\ \{c_1\}\ \Gamma''_i$, and
   e) $\Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma$.
2. We say that a loop run of the loop $\mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$ induces a dependency of z' on z if one of the following two conditions is satisfied:
   a) there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{z\},\{z'\}})$ or
   b) $z \in vars(e)$ and there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{z\},\{z'\}})$ that contains a node $n \notin \{start, out\}$.
3. Hence, by Lemma A.22, $x \in X$ if and only if there is a sequence of distinct variables $z_1, \ldots, z_l$ such that $x = z_1$, $z_l = y$, and the loop c induces a dependency of z_{i+1} on z_i for all $i \in \{1, \ldots, l-1\}$.
4. By the induction hypothesis and (1d) it follows that, for all $z' \in Var$ and all $i \in \mathbb{N}$, one of the following holds, where Z is the set of all z such that there is a path from *in* to *out* in $PDG(CFG_{c_1}^{\{z\},\{z'\}})$:
   a) $\Gamma''_i(z') = \big(\bigsqcup_{z \in Z} \Gamma'_i(z)\big) \sqcup pc \sqcup d_i$ if there is a path from *start* to *out* in the graph $PDG(CFG_{c_1}^{\{z\},\{z'\}})$ that contains a node $n \notin \{start, out\}$, and

   b) $\Gamma''_i(z') = \bigsqcup_{z \in Z} \Gamma'_i(z)$ if there is no such path.
5. From (1c) and typing rule [EXP] it follows that $d_i = \bigsqcup_{x \in vars(e)} \Gamma'_i(x)$.
6. From the definition in (2) and from (4) and (5) it follows that for all $z' \in Var$ one of the following holds, where Z' is the set of all z such that the loop c induces a dependency of z' on z:
   a) $\Gamma''_i(z') = \left(\bigsqcup_{z \in Z'} \Gamma'_i(z)\right) \sqcup pc$ if there is a path from *start* to *out* in the graph $PDG(CFG_{c_1}^{\{x\},\{z'\}})$ that contains a node $n \notin \{start, out\}$, and
   b) $\Gamma''_i(z') = \bigsqcup_{z \in Z'} \Gamma'_i(z)$ if there is no such path.
7. Hence, by (1e), it follows that for all $z \in Var$ one of the following holds:
   a) $\Gamma'_{i+1}(z') = \left(\bigsqcup_{z \in Z'} \Gamma'_i(z)\right) \sqcup pc \sqcup \Gamma(z')$ if there is a path from *start* to *out* in $PDG(CFG_{c_1}^{\{z\},\{z'\}})$ that contains a node $n \notin \{start, out\}$, and
   b) $\Gamma'_{i+1}(z') = \left(\bigsqcup_{z \in Z'} \Gamma'_i(z)\right) \sqcup \Gamma(z')$ if there is no such path.
8. Using (7) and starting with $\Gamma'(y) = \Gamma'_k(y)$, we unfold the equation further and further, thereby collecting the term $\Gamma(x)$ for all $x \in X$ on the right hand side of the equation. As a result, the following holds:
   a) $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x) \sqcup pc$, if, for any $x \in Var$, there exists $z_1, \ldots, z_k \in Var$ and a path from *start* to *out* in $PDG(CFG_{c_1}^{\{x\},\{z_1\}})$ (containing a node $n \notin \{start, out\}$) and paths from *in* to *out* in the graphs $PDG(CFG_{c_1}^{\{z_i\},\{z_{i+1}\}})$, and
   b) $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$ if such paths do not exist for any $x \in Var$.
9. But then, with Statement (2) of Lemma A.22, it follows that
   a) $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x) \sqcup pc$, if, there exists a path from *start* to *out* in the graph $PDG(CFG_{c}^{\{z_x\},\{y\}})$ that contains a node $n \notin \{start, out\}$, and
   b) $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$ if such a path does not exist.

   This concludes the proof. □

Proof of Theorem 6.3. Theorem 6.3 follows from Lemma A.23 for $pc = low$. □

To prove Theorem 6.5, we extend the proof of Theorem 6.3 to commands containing procedure calls. For relating summary edges to the functions in the tuple Ψ, we introduce an additional induction over the depth of the procedure call chains that are considered for computing summary edges respectively Ψ. For the tuple Ψ, the variants that reflect only procedure call chains up to depth k are already explicit in the definition of Ψ^*, namely the functions Ψ_k (compare Definition 6.21). For summary edges, we introduce the notion of k-realizable paths in an SDG, where k-realizable paths represent dependencies that are due to procedure call chains of at most depth k.

Definition A.3. A path in the SDG is k-realizable if its projection on the set containing all nodes of the form $(a\text{-}in_{n,k,\mathsf{p}'}, \mathsf{p})$ and $(a\text{-}out_{n,k,\mathsf{p}'}, \mathsf{p})$ is generated by the grammar for R_k, which is recursively defined as follows:

$$\begin{array}{rcl} R_0 & ::= & \varepsilon \\ R_{k+1} & ::= & \varepsilon \mid R_{k+1}\ (a\text{-}in_{n,k_1,\mathsf{p}'}, \mathsf{p})\ R_k\ (a\text{-}out_{n,k_2,\mathsf{p}'}, \mathsf{p}) \end{array}$$

We denote the corresponding PDGs with summary edges defined through k-realizable path with PDG^k. ◊

We denote with $PDG^*(CFG_c^{\{x\},\{y\}})$ the PDG with inputs $\{x\}$ and outputs $\{y\}$ for the command c, where summary edges are added between actual in and out parameter

nodes according to the computation of summary edges. With $PDG^k(CFG_c^{\{x\},\{y\}})$ we denote this PDG where summary edges are determined via k-realizable paths.

It is straightforward to show that Lemmas A.20 to A.22 extend to procedure dependence graphs with summary edges for commands with procedure calls.

We extend Lemma A.23 to the interprocedural analysis as follows:

Lemma A.24. In this lemma, we consider the security type system instantiated for the universal security lattice.

Let $k \in \mathbb{N}$, $c \in \mathit{Com}$, $y \in \mathit{Var}$ and Γ be an environment. Let Γ' be such that $pc \vdash_{\Psi_k} \Gamma\ \{c\}\ \Gamma'$ is derivable in the security type system, and let X be the set of all $x \in \mathit{Var}$ such that there exists a path from *in* to *out* in $PDG^k(CFG_c^{\{x\},\{y\}})$. Then either

1. $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x\in X} \Gamma(x)\big)$ and there is a path from *start* to *out* in $PDG^k(CFG_c^{\{x\},\{y\}})$ that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$ for some $x \in \mathit{Var}$, or
2. $\Gamma'(y) = \bigsqcup_{x\in X} \Gamma(x)$ and there is no such path in $PDG^k(CFG_c^{\{x\},\{y\}})$ for any $x \in \mathit{Var}$.

$\Diamond$

Proof. The proof is by induction on the pair (k, c), using as well-founded ordering $(k, c) < (k', c')$ the lexicographic ordering (i.e., $(k, c) < (k', c')$ if and only if $k < k'$ or $k = k'$ and c is a subcommand of c'). The base cases of the induction correspond to the minimal elements of this order, i.e., $k = 0$ and c is skip, $x{:=}e$, or a procedure call $\mathsf{p}(\ldots)$.

Assume that $k \in \mathbb{N}$ and $c = \mathsf{skip}$: The proof is the same as in the case for skip in Lemma A.23 where the PDG is replaced by the PDG with summary edges for k-realizable paths.

Assume that $k \in \mathbb{N}$ and $c = z{:=}e$: The proof is the same as in the case for assignments in Lemma A.23 where the PDG is replaced by the PDG with summary edges for k-realizable paths.

Assume that $k = 0$ and $c = \mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})$:

1. There is an edge from *start* to 1, and all edges from 1 go to the actual in and out parameter nodes.
2. Since there are no 0-realizable paths, there are no summary edges. Hence, there are no edges starting at actual in parameter nodes and no edges ending at actual out parameter nodes.
3. Moreover, there is an edge from *start* to $a\text{-}in_{1,i,\mathsf{p}}$ if and only if $x \in \mathit{vars}(e_i)$.
4. Moreover, there is an edge from $a\text{-}out_{1,i,\mathsf{p}}$ to *out* if and only if $y = z_i$.
5. Moreover, there is an edge from *in* to *out* if and only if $x = y$ and $y \notin \{z_1, \ldots, z_{m_\mathsf{p}}\}$.
6. We do a case distinction on $y \in \{z_1, \ldots, z_{m_\mathsf{p}}\}$. Assume firstly that $y \notin \{z_1, \ldots, z_{m_\mathsf{p}}\}$.
   a) By (4) and (5), there is a path from *in* to *out* if and only if $x = y$, i.e., $X = \{y\}$.
   b) Moreover, by (4) there is no path from *start* to *out* but the path $\langle \mathit{start}, \mathit{out}\rangle$, and we must show (due to (b)) that $\Gamma'(y) = \Gamma(y)$.
   c) It follows from the typing rule for procedure calls and the assumption that $pc \vdash_{\Psi_0} \Gamma\ \{c\}\ \Gamma'$ that $\Gamma'(y) = \Gamma(y)$.
7. Now assume that $y = z_i$ for some i.
   a) Then, by (1) and (4) there is a path from *start* to *out* that contains a node $n \notin \{\mathit{start}, \mathit{out}\}$. Hence, we must show that $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x\in X} \Gamma(x)\big)$.
   b) By (2) and (5) there is no path from *in* to *out*, i.e., $X = \{\}$. Hence, we must show that $\Gamma'(y) = pc$.

c) But $\Gamma'(y) = pc$ is satisfied due to the typing rule for procedure calls and the assumption that $pc \vdash_{\Psi_0} \Gamma\,\{c\}\,\Gamma'$ (because $(\psi_0)_\mathsf{p}(y) = \{\}$ for all p and y).

Assume that $k > 0$ and $c = \mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})$:

1. There is an edge from *start* to 1, and all edges from 1 go to the actual in and out parameter nodes.
2. All edges starting at actual in parameter nodes are edges ending at actual out parameter nodes, and all edges ending at actual out parameter nodes are edges starting at actual in parameter nodes.
3. Moreover, there is an edge from *start* to $a\text{-}in_{1,i,\mathsf{p}}$ if and only if $x \in vars(e_i)$.
4. Moreover, there is an edge from $a\text{-}out_{1,i,\mathsf{p}}$ to *out* if and only if $y = z_i$.
5. Moreover, there is an edge from *in* to *out* if and only if $x = y$ and $y \notin \{z_1, \ldots, z_{m_\mathsf{p}}\}$.
6. We do a case distinction on $y \in \{z_1, \ldots, z_{m_\mathsf{p}}\}$. Assume firstly that $y \notin \{z_1, \ldots, z_{m_\mathsf{p}}\}$. The proof in this case is the same as in the case where $k = 0$ and c is a procedure call.
7. Now assume that $y = z_i$ for some i.
   a) Then, by (1) and (4) there is a path from *start* to *out* that contains a node $n \notin \{start, out\}$. Hence, we must show that $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$.
   b) By (2), (3), (4), and (5) paths from *in* to *out* in $PDG^k(CFG_c^{\{x\},\{y\}})$ are of the form $\langle in, a\text{-}in_{1,k_1,\mathsf{p}}, a\text{-}out_{1,k_2,\mathsf{p}}, out\rangle$, where
      i. $x \in vars(e_{k_1})$,
      ii. $k_2 = i$, and
      iii. there is a summary edge from $a\text{-}in_{1,k_1,\mathsf{p}}$ to $a\text{-}out_{1,k_2,\mathsf{p}}$.
   c) By (b iii), there is a $(k-1)$-realizable path from $f\text{-}in^\mathsf{p}_{k_1}$ to $f\text{-}out^\mathsf{p}_{k_2}$ in the procedure dependence graph of c_p. This is the case if and only if there is a path from *in* to *out* in $PDG^{k-1}(CFG_{c_\mathsf{p}}^{\{x^\mathsf{p}_{k_1}\},\{y^\mathsf{p}_{k_2}\}})$.
   d) We apply the induction hypothesis for $k-1$ and the command c_p. Let Γ'' be such that $\{\} \vdash_{\Psi_{k-1}} \Gamma_{id}\,\{c_\mathsf{p}\}\,\Gamma''$. Then $\Gamma''(y_{k_2}) = X'$, where X' is the set of all $x' \in Var$ such that there is a path from *in* to *out* in $PDG^{k-1}(CFG_{c_\mathsf{p}}^{\{x'\},\{y^\mathsf{p}_{k_2}\}})$.
   e) By (c) and (d), the statement (b iii) is equivalent to $x^\mathsf{p}_{k_1} \in \Gamma''(y^\mathsf{p}_{k_2})$ where Γ'' is such that $\{\} \vdash_{\Psi_{k-1}} \Gamma_{id}\,\{c_\mathsf{p}\}\,\Gamma''$. But this, by the definition of the sequence of the Ψ_k, is equivalent to $x^\mathsf{p}_{k_1} \in (\Psi_k)_\mathsf{p}(y^\mathsf{p}_{k_2})$.
   f) Hence, by (b) and (e), there is a path from *in* to *out* in $PDG^k(CFG_c^{\{x\},\{y\}})$ if and only if there exists j such that $x \in vars(e_j)$ and $x^\mathsf{p}_j \in (\Psi_k)_\mathsf{p}(y^\mathsf{p}_i)$.
   g) But then $\Gamma'(y) = pc \sqcup \big(\bigsqcup_{x \in X} \Gamma(x)\big)$ follows directly from (f), the typing rule [call], and the fact that $pc \vdash_{\Psi_k} \Gamma\,\{c\}\,\Gamma'$ is derivable in the type system.

Assume that k is arbitrary and $c = c_1; c_2$: The proof is the same as in the case for sequential composition in Lemma A.23 where the PDG is replaced by the PDG with summary edges.

Assume that k is arbitrary and $c =$ if e then c_1 else c_2 fi: The proof is the same as in the case for conditionals in Lemma A.23 where the PDG is replaced by the PDG with summary edges.

Assume that k is arbitrary and $c =$ while e do c_1 od: The proof is the same as in the case for while loops in Lemma A.23 where the PDG is replaced by the PDG with summary edges. □

Lemma A.25. Let $c \in Com$, $y \in Var$ and Γ be an environment. Let Γ' be such that $pc \vdash_{\Psi^*} \Gamma\,\{c\}\,\Gamma'$ is derivable in the security type system, and let X be the set of all $x \in Var$ such that there exists a path from *in* to *out* in $PDG^*(CFG_c^{\{x\},\{y\}})$. Then either

1. $\Gamma'(y) = pc \sqcup \left(\bigsqcup_{x \in X} \Gamma(x)\right)$ and there is a path from *start* to *out* in $PDG^k(CFG_c^{\{x\},\{y\}})$ that contains a node $n \notin \{start, out\}$ for some $x \in Var$, or
2. $\Gamma'(y) = \bigsqcup_{x \in X} \Gamma(x)$ and there is no such path in $PDG^k(CFG_c^{\{x\},\{y\}})$ for any $x \in Var$.

$\Diamond$

Proof. By the construction of the Ψ_k and k-realizable path, there exists $k \in \mathbb{N}$ such that $PDG^k(CFG_c^{\{x\},\{y\}}) = PDG^*(CFG_c^{\{x\},\{y\}})$ and $\Psi_k = \Psi^*$. As Lemma A.24 holds for arbitrary k, the Corollary follows immediately. $\square$

Proof of Theorem 6.5. Theorem 6.5 follows from Corollary A.25 for $pc = \{\}$. $\square$

To prove Theorem 6.7, we generalize the definition of $PDG^{\|}(CFG_c^{H,L}, mds)$ to the form $PDG^{\|}(CFG_c^{I,O}, mds)$ where I, O are arbitrary sets of variables.

Definition A.4. Let $c \in Com_{seq}$, $mds \in Mds$, $I, O \subseteq Var$, and $(N, E) = PDG(CFG_c^{I,O})$. We define the program dependence graph $PDG^{\|}(CFG_c^{I,O}, mds)$ by $(N, E \cup E')$ where $(n, n') \in E'$ if and only if one of the following conditions is satisfied:

1. $n = in$, there exists $x \in use_c(n')$ with $x \in I$, there exists $n'' \in N$ with $x \notin CFG\text{-}mds_{c,mds}(n'')(asm\text{-}no\text{-}w)$, and there is a path p from n'' to n' such that $x \notin def_c(n''')$ for every node n''' on p with $n''' \neq n''$ and $n''' \neq n'$,
2. $n' = out$, there exists $x \in def_c(n)$ with $x \in O$, there exists $n'' \in N$ with $x \notin CFG\text{-}mds_{c,mds}(n'')(asm\text{-}no\text{-}r)$, and there is a path p from n to n'' such that $x \notin def_c(n''')$ for every node n''' on p with $n''' \neq n$ and $n''' \neq n''$, or
3. $n \in \{1, \ldots, \sharp(c)\}$, $c[n] \in Exp$, and $n' = out$.

$\Diamond$

To prove Theorem 6.7, we establish the following more general lemma.

Lemma A.26. Let *mds* be a mode state, Λ be a partial environment that is consistent with *mds*, and $c \in Com_{seq}$. Assume that no partial environment Λ' exists such that $\vdash \Lambda\,\{c\}\,\Lambda'$ is derivable in the type system from Section 4.5. Then there exist $x, y \in Var$ with $\Lambda\langle x\rangle = high$ and $dma(y) = low$ and a path p of the form $\langle in, \ldots, n, out\rangle$ in $PDG^{\|}(CFG_c^{\{x\},\{y\}}, mds)$ where $n \in \{1, \ldots, \sharp(c)\}$ and one of the following conditions is satisfied:

1. $y \in def_c(n)$, there is a node n' with $y \notin CFG\text{-}mds_{c,mds}(n')(asm\text{-}no\text{-}r)$, and a definition of y at n reaches n' in CFG_c, or
2. $c[n] \in Exp$.

$\Diamond$

Proof. The proof is by induction on the structure of the command c.

Assume that $c = \mathsf{skip}$ **or** $c = stop$**.** Then $\vdash \Lambda\,\{c\}\,\Lambda$ is derivable, which contradicts the assumptions of the lemma.

Assume that $c = x{:=}e$**.** If $x \in dom(\Lambda)$ then $\vdash \Lambda\,\{c\}\,\Lambda'$ is derivable for some Λ' with rule $[\mathsf{ASS}_2]$. In consequence, $x \notin dom(\Lambda)$. Moreover, if $dma(x) = high$ then $\vdash \Lambda\,\{c\}\,\Lambda$ is derivable with rule $[\mathsf{ASS}_1]$. In consequence, $dma(x) = low$. Moreover, if $\Lambda\langle y\rangle = low$

for all $y \in \mathit{vars}(e)$ then $\vdash\ \Lambda\,\{c\}\,\Lambda$ would be derivable with rule [ASS_1]. In consequence, there exists $y \in \mathit{vars}(e)$ with $\Lambda\langle y\rangle = \mathit{high}$.

Since $y \in \mathit{vars}(e)$ it follows that $y \in \mathit{use}_c(1)$, and, hence, the pair $(\mathit{in}, 1)$ is an edge in $\mathit{PDG}(\mathit{CFG}_c^{\{y\},\{x\}})$. In consequence, the pair is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{y\},\{x\}}, \mathit{mds})$.

Since $x \notin \mathit{dom}(\Lambda)$, Λ is consistent with mds, and $\mathit{dma}(x) = \mathit{low}$, it follows that $x \notin \mathit{mds}(\mathit{asm\text{-}no\text{-}r})$. In consequence, $x \notin \mathit{CFG\text{-}mds}_{c,\mathit{mds}}(\mathit{stop})(\mathit{asm\text{-}no\text{-}r})$. Moreover, the definition of x at Node 1 reaches stop in CFG_c. Hence, $(1, \mathit{out})$ is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{y\},\{x\}}, \mathit{mds})$, and Condition (1) is satisfied for this edge.

Hence, $\langle \mathit{in}, 1, \mathit{out}\rangle$ is a path in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{y\},\{x\}}, \mathit{mds})$ that satisfies all required conditions.

Assume that $c = c_1; c_2$. If there exist Λ'' and Λ' such that $\vdash\ \Lambda\,\{c_1\}\,\Lambda''$ and $\vdash\ \Lambda''\,\{c_2\}\,\Lambda'$ are derivable then $\vdash\ \Lambda\,\{c\}\,\Lambda'$ is derivable with rule [SEQ]. In consequence, there does not exist Λ'' and Λ' such that $\vdash\ \Lambda\,\{c_1\}\,\Lambda''$ and $\vdash\ \Lambda''\,\{c_2\}\,\Lambda'$ are derivable. We distinguish the two cases (1) that there is no Λ'' such that $\vdash\ \Lambda\,\{c_1\}\,\Lambda''$ is derivable, and (2) that if $\vdash\ \Lambda\,\{c_1\}\,\Lambda''$ is derivable for some Λ'' then there is no Λ' such that $\vdash\ \Lambda''\,\{c_2\}\,\Lambda'$ is derivable.

1. Assume that there is no Λ'' such that $\vdash\ \Lambda\,\{c_1\}\,\Lambda''$ is derivable.
   a) By the induction hypothesis there are $x, y \in \mathit{Var}$ with $\Lambda\langle x\rangle = \mathit{high}$ and $\mathit{dma}(y) = \mathit{low}$ and a path p from in to out in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1}^{\{x\},\{y\}}, \mathit{mds})$ such that Condition (1) or Condition (2) is satisfied for the one-but-last node of p. To determine a path from in to out in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$ we distinguish between whether Condition (1) or Condition (2) is satisfied for that node.
      i. Assume that $y \in \mathit{def}_{c_1}(n)$, that there exists a node n' with $y \notin \mathit{CFG\text{-}mds}_{c_1,\mathit{mds}}(n')(\mathit{asm\text{-}no\text{-}r})$, and that a definition of y at n reaches n' in CFG_{c_1}.
      It follows from $y \in \mathit{def}_{c_1}(n)$ that $y \in \mathit{def}_{c_1;c_2}(n)$. Moreover, it follows from $y \notin \mathit{asm\text{-}no\text{-}r}_{c_1,\Lambda}(n)$ that $y \notin \mathit{asm\text{-}no\text{-}r}_{c_1;c_2,\Lambda}(n)$. Moreover, since a definition of y at n reaches n' in CFG_{c_1}, it also reaches n' in CFG_c.
      Hence, the last edge (n, out) of p is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$. Thus p is a path in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$.
      ii. Assume that $n \in \{1, \ldots, \sharp(c_1)\}$ and $c_1[n] \in \mathit{Exp}$. Then the last edge (n, out) of p is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$. Thus p is a path in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$.
2. Let Λ'' such that $\vdash\ \Lambda\,\{c_1\}\,\Lambda''$ is derivable, and such that there is no Λ' for which $\vdash\ \Lambda''\,\{c_2\}\,\Lambda'$ is derivable.
   a) Let $\mathit{mds}'' = \mathit{CFG\text{-}mds}_{c_1,\mathit{mds}}(\mathit{stop})$. It follows from the definition of the functions $\mathit{CFG\text{-}mds}$ and the typing rules that Λ'' is consistent with mds''.
   b) By the induction hypothesis there exist $z, y \in \mathit{Var}$ with $\Lambda''\langle z\rangle = \mathit{high}$ and $\mathit{dma}(y) = \mathit{low}$ such that there is a path p_2 in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_2}^{\{z\},\{y\}}, \mathit{mds}'')$ that is of the form $\langle \mathit{in}, \ldots, n, \mathit{out}\rangle$ with $n \in \{1, \ldots, \sharp(c_2)\}$ and where Condition (1) or Condition (2) is satisfied for the edge (n, out).
   c) Let Γ be the unique environment such that $\mathit{low} \vdash\ \Lambda\langle\cdot\rangle\,\{c_1\}\,\Gamma$ is derivable in the type system from Section 6.2. Then, by Theorem 6.3, $\Gamma(z) = \bigsqcup_{x\in X} \Lambda\langle x\rangle$ where X contains all $x \in \mathit{Var}$ such that there exists a path from in to out in $\mathit{PDG}(\mathit{CFG}_{c_1}^{\{x\},\{z\}})$. We denote this path with p_x for $x \in X$.
   Like in the proof of Theorem 6.3, it follows from the existence of the paths p_x and p_2 that there exists a path p from in to out in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1;c_2}^{\{x\},\{y\}}, \mathit{mds})$. That Condition (1) or Condition (2) is satisfied for the last edge in this path p follows

from the construction of p and from (a) and (b). Hence, we can conclude this case if there exists $x \in X$ with $\Lambda\langle x\rangle = \mathit{high}$.
Assume now that $\Lambda\langle x\rangle = \mathit{low}$ for all $x \in X$. Then $\Gamma(z) = \mathit{low}$. It follows from $\Gamma(z) = \mathit{low}$, $\Lambda''\langle z\rangle = \mathit{high}$, and the derivability of $\mathit{low} \vdash \Lambda\langle\cdot\rangle\,\{c_1\}\,\Gamma$ and $\vdash \Lambda\,\{c_1\}\,\Lambda''$ that $\mathit{dma}(z) = \mathit{high}$ and $z \notin \mathit{dom}(\Lambda'')$. Then, since Λ'' is consistent with mds'', $z \notin \mathit{mds}''(\mathit{asm\text{-}no\text{-}w})$. Let n' be a node such that $z \in \mathit{use}_{c_1}(n')$ and such that a definition of z at n' reaches *stop* in CFG_{c_1}, or let $n' = \mathit{out}$ if no such node exists. Then, by Definition A.4 the pair (in, n') is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1}^{\{x\},\{z\}}, \mathit{mds})$ for any x because $\mathit{mds}'' = \mathit{CFG\text{-}mds}_{c_1,\mathit{mds}}(\mathit{stop})$, and the pair (n', out) is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1}^{\{x\},\{z\}}, \mathit{mds})$ by construction of n'. Hence, $\langle \mathit{in}, n', \mathit{out}\rangle$ is a path in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1}^{\{x\},\{z\}}, \mathit{mds})$, and we can conclude as in the above case for $\Lambda\langle x\rangle = \mathit{high}$

Assume that $c = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$**.** Since there is no partial environment Λ' such that $\vdash \Lambda\,\{c\}\,\Lambda'$ is derivable in the type system, by the typing rule [IF] one of the following conditions is satisfied: $\Lambda\langle e\rangle = \mathit{high}$, or there is no Λ' such that $\vdash \Lambda\,\{c_1\}\,\Lambda'$ is derivable, or there is no Λ' such that $\vdash \Lambda\,\{c_2\}\,\Lambda'$ is derivable.

1. Assume that $\Lambda\langle e\rangle = \mathit{high}$. Then there exists $x \in \mathit{vars}(e)$ such that $\Lambda\langle x\rangle = \mathit{high}$. Hence, $(\mathit{in}, 1)$ is an edge in $\mathit{PDG}(\mathit{CFG}_c^{\{x\},\{y\}})$. In consequence, $(\mathit{in}, 1)$ is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$.
   Let y be a variable with $\mathit{dma}(y) = \mathit{low}$. Then the pair $(1, \mathit{out})$ is an edge in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$ because $c[1] \in \mathit{Exp}$.
   Hence, $\langle \mathit{in}, 1, \mathit{out}\rangle$ is a path from *in* to *out* in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$, and Condition (2) is satisfied for the last edge $(1, \mathit{out})$.
2. Assume that there is no Λ' such that $\vdash \Lambda\,\{c_1\}\,\Lambda'$ is derivable. Then, by the induction hypothesis for c_1, there are x, y, and a path p_1 in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1}^{\{x\},\{y\}}, \mathit{mds})$ with the properties stated by the lemma. But then one obtains a path p in the graph $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$ with the properties required by the lemma by increasing each node $n \in \{1, \ldots, \sharp(c_1)\}$ in p_1 by 1.
3. Assume that there is no Λ' such that $\vdash \Lambda\,\{c_2\}\,\Lambda'$ is derivable. The proof is as in the previous case, exploiting the induction hypothesis for c_2.

Assume that $c = \mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}$**:** Since there is no Λ' such that $\vdash \Lambda\{\mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}\}\Lambda'$ is derivable, by the typing rule [SUB] there is no Λ'' with $\Lambda \sqsubseteq \Lambda'' \sqsubseteq \Lambda'$ such that $\vdash \Lambda''\,\{\mathsf{while}\ e\ \mathsf{do}\ c_1\ \mathsf{od}\}\,\Lambda''$ is derivable. I.e., by the typing rule [WHILE], for all Λ'' with $\Lambda \sqsubseteq \Lambda'' \sqsubseteq \Lambda'$ one of the following conditions is satisfied: $\Lambda''\langle e\rangle = \mathit{high}$ or $\vdash \Lambda''\,\{c_1\}\,\Lambda''$ is not derivable.

1. Assume that $\Lambda''\langle e\rangle = \mathit{high}$. Then the proof is as in the case for conditionals, defining $p = \langle \mathit{in}, 1, \mathit{out}\rangle$.
2. Assume that $\vdash \Lambda''\,\{c_1\}\,\Lambda''$ is not derivable. Then, by the induction hypothesis for c_1, there are $x, y \in \mathit{Var}$ and a path p_1 in $\mathit{PDG}^{\parallel}(\mathit{CFG}_{c_1}^{\{x\},\{y\}}, \mathit{mds})$ with the properties stated by the lemma. But then one obtains a path p in $\mathit{PDG}^{\parallel}(\mathit{CFG}_c^{\{x\},\{y\}}, \mathit{mds})$ with the properties required by the lemma by increasing each node $n \in \{1, \ldots, \sharp(c_1)\}$ in p_1 by 1. □

Now we prove Theorem 6.7.

Proof. We show that it follows from the assumptions of the theorem that $\vdash \mathit{thr}$ is derivable in the type system from Section 4.5.

Let $i \in \{1, \ldots, \sharp(thr)\}$, and let Λ_i be a partial environment consistent with $mdss[i]$. We must show that there exists Λ_i' such that $\vdash \Lambda_i \ \{thr[i]\} \ \Lambda_i'$ is derivable. We prove this by contradiction. Assume that $\vdash \ \Lambda_i \ \{thr[i]\} \ \Lambda_i'$ is not derivable. Then, by Lemma A.26, there exist $x, y \in Var$ with $\Lambda_i\langle x\rangle = high$ and $dma(y) = low$, such that there is a path p from in to out in $PDG^{\parallel}(CFG^{\{x\},\{y\}}_{thr[i]}, mdss[i])$ that has at least length 3.

If $dma(x) = high$, then p is a path in $PDG^{\parallel}(CFG^{H,L}_{thr[i]}, mdss[i])$. This contradicts the assumptions of the theorem.

If $dma(x) = low$ then it follows from $\Lambda_i\langle x\rangle = high$ and the consistency of Λ_i with $mdss[i]$ that $x \in mdss[i](asm\text{-}no\text{-}r)$. Hence, if a definition of x at $start$ reaches a node n in $CFG_{thr[i]}$ then there is an edge from n to out in $PDG^{\parallel}(CFG^{H,L}_{thr[i]}, mdss[i])$. Since p has at least length 3, $p = \langle in, n, \ldots, out\rangle$ for some node n. Since (in, n) is an edge in $PDG^{\parallel}(CFG^{\{x\},\{y\}}_{thr[i]}, mdss[i])$ it follows that a definition of x at $start$ reaches n in $CFG_{thr[i]}$, and, hence (n, out) is an edge in $PDG^{\parallel}(CFG^{H,L}_{thr[i]}, mdss[i])$. Hence, $\langle in, 1, out\rangle$ is a path in $PDG^{\parallel}(CFG^{H,L}_{thr[i]}, mdss[i])$. This contradicts the assumptions of the theorem.

I.e., we have shown by contradiction that there exists Λ_i' such that $\vdash \Lambda_i \ \{thr[i]\} \ \Lambda_i'$ is derivable.

The remaining prerequisites of the derivation rule for the judgment $\vdash thr$ follow directly from the assumptions of the theorem. In consequence, $\vdash thr$ is derivable. Hence, by Theorem 4.8 (the soundness result for the type system), thr is $\mathcal{S}$-noninterferent for any scheduler.

APPENDIX B

Type System for Sound Modes

In this appendix, we present the simple type system sketched in Section 4.5.3 for checking that guarantees are respected, that no no-read assumptions occur at thread termination, and that modes are compatible with *obs*.

We use typing judgments of the form $\vdash mds\ \{c\}\ mds'$, where c is a command and mds, mds' are mode states. The interpretation of the judgment is that if mds is an upper bound on modes before c executes, then mds' is an upper bound on modes after the execution of c. The type system is displayed in Figure B.1.

We use auxiliary typing judgments $\vdash mds\ \{anns\}\ mds'$, where *anns* denotes a (possibly empty) sequence $/\!\!/ann_1/\!\!/ \ldots /\!\!/ann_n/\!\!/$ of annotations (see rule [ANNO]). We define $mds\text{-}update(mds, anns) = mds$ if *anns* is empty, and $mds\text{-}update(mds, anns) = mds\text{-}update(mds\text{-}update(mds, ann_1), /\!\!/ann_2/\!\!/, \ldots, /\!\!/ann_n/\!\!/)$ if $anns = /\!\!/ann_1/\!\!/ \ldots /\!\!/ann_n/\!\!/$. Moreover, the set X in rule [ANNO] is the largest set of variables to which *obs* is confined.

Subtyping (see rule [SUB]) makes use of the relation $\sqsubseteq$ on mode states that is defined by $mds \sqsubseteq mds'$ if and only if $mds(m) \subseteq mds'(m)$ for all modes $m \in Mod$.

Rules [ASS], [IF], and [WHILE] ensure that guarantees are respected. Moreover, rule [ANNO] ensures that modes are compatible with *obs*.

Theorem B.1. Let c be a command. Assume that the judgment $\vdash mds_1\ \{c\}\ mds_1'$ is derivable in the type system. Then for all mode states $mds_2 \sqsubseteq mds_1$ and memories $mem \in Mem$ the following conditions are satisfied:
(1) $\langle c, mds_2, mem\rangle$ respects guarantees,
(2) $\langle c, mds_2, mem\rangle$ has modes compatible with *obs*, and
(3) if $\langle stop, mds_2', mem'\rangle \in loc\text{-}reach(\langle c, mds_2, mem\rangle)$ then $mds_2' \sqsubseteq mds_1'$. ◊

It follows from Theorem B.1 that if $\vdash mds\ \{c\}\ mds_0$ is derivable then $\langle c, mds, mem\rangle$ respects guarantees, has modes compatible with *obs*, and no no-read assumptions occur at thread termination.

Proof of Theorem B.1. The proof is by induction on the derivation of the judgment $\vdash mds_1\ \{c\}\ mds_1'$.

$$[\mathsf{ANNO}]\ \frac{mds' = mds\text{-}update(mds, anns) \quad mds(asm\text{-}no\text{-}r) \cap X = mds'(asm\text{-}no\text{-}r) \cap X = \{\}}{\vdash mds\ \{anns\}\ mds'}$$

$$[\mathsf{SKIP}]\ \frac{\vdash mds\ \{anns\}\ mds'}{\vdash mds\ \{anns\ \mathsf{skip}\}\ mds'}$$

$$[\mathsf{ASS}]\ \frac{\vdash mds\ \{anns\}\ mds' \quad x \notin mds(guar\text{-}no\text{-}w) \quad vars(e) \cap mds(guar\text{-}no\text{-}r) = \{\}}{\vdash mds\ \{anns\ x{:=}e\}\ mds'}$$

$$[\mathsf{IF}]\ \frac{\vdash mds\ \{anns\}\ mds' \quad \vdash mds'\ \{c_1\}\ mds'' \quad \vdash mds'\ \{c_2\}\ mds'' \quad vars(e) \cap mds(guar\text{-}no\text{-}r) = \{\}}{\vdash mds\ \{anns\ \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\ mds''}$$

$$[\mathsf{WHILE}]\ \frac{\vdash mds\ \{anns\}\ mds' \quad \vdash mds'\ \{c\}\ mds' \quad vars(e) \cap mds(guar\text{-}no\text{-}r) = \{\}}{\vdash mds\ \{anns\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{od}\}\ mds'}$$

$$[\mathsf{SEQ}]\ \frac{\vdash mds\ \{c_1\}\ mds' \quad \vdash mds'\ \{c_2\}\ mds''}{\vdash mds\ \{c_1; c_2\}\ mds''}$$

$$[\mathsf{SPAWN}]\ \frac{\vdash mds\ \{anns\}\ mds' \quad \forall i \in \{1, \ldots, \sharp(thr)\} : \vdash spawn\text{-}mds(mds)\ \{thr[i]\}\ mds_0}{\vdash mds\ \{anns\ \mathsf{spawn}(thr)\}\ mds'}$$

$$[\mathsf{SUB}]\ \frac{\vdash mds_2\ \{c\}\ mds_2' \quad mds_1 \sqsubseteq mds_2 \quad mds_2' \sqsubseteq mds_1'}{\vdash mds_1\ \{c\}\ mds_1'}$$

Figure B.1: Type system for modes

– [SKIP]. If $\vdash mds_1\ \{c\}\ mds_1'$ is derived with rule [SKIP] then $c = anns\ \mathsf{skip}$ and $\vdash mds_1\ \{anns\}\ mds_1'$.

  Then, by rule [ANNO], $mds_1' = mds\text{-}update(mds_1, anns)$ and $mds_1(asm\text{-}no\text{-}r) \cap X = mds_1'(asm\text{-}no\text{-}r) \cap X = \{\}$. Let $mds_2' = mds\text{-}update(mds_2, anns)$. Then it follows from $mds_2 \sqsubseteq mds_1$ that $mds_2' \sqsubseteq mds_1'$, and, hence, $mds_2(asm\text{-}no\text{-}r) \cap X = mds_2'(asm\text{-}no\text{-}r) \cap X = \{\}$.

  If $\langle c', mds', mem' \rangle \in loc\text{-}reach(\langle c, mds_2, mem \rangle)$ then $c' = anns\ \mathsf{skip}$ and $mds' = mds_2$ or $c' = stop$ and $mds' = mds_2'$. Hence, $\langle c, mds_2, mem \rangle$ has modes compatible with obs. Moreover, since both $anns\ \mathsf{skip}$ and $stop$ do not read any variable and do not modify any variable it follows that $\langle c, mds_2, mem \rangle$ respects guarantees. Moreover, if $c' = stop$ then $mds' \sqsubseteq mds_1'$ because then $mds' = mds_2'$ and $mds_2' \sqsubseteq mds_1'$.

– [ASS]. If $\vdash mds_1\ \{c\}\ mds_1'$ is derived with rule [ASS] then $c = anns\ x{:=}e$, $\vdash mds_1\ \{anns\}\ mds_1'$, $x \notin mds_1(guar\text{-}no\text{-}w)$, and $vars(e) \cap mds_1(guar\text{-}no\text{-}r) = \{\}$.

  Then, by rule [ANNO], $mds_1' = mds\text{-}update(mds_1, anns)$ and $mds_1(asm\text{-}no\text{-}r) \cap X = mds_1'(asm\text{-}no\text{-}r) \cap X = \{\}$. Let $mds_2' = mds\text{-}update(mds_2, anns)$. Then it

follows from $mds_2 \sqsubseteq mds_1$ that $mds_2' \sqsubseteq mds_1'$, and, hence, $mds_2(\textit{asm-no-r}) \cap X = mds_2'(\textit{asm-no-r}) \cap X = \{\}$.

If $\langle c', mds', mem' \rangle \in \textit{loc-reach}(\langle c, mds_2, mem \rangle)$ then $c' = \textit{anns}\ x{:=}e$ and $mds' = mds_2$, or $c' = \textit{stop}\}$ and $mds' = mds_2'$. Hence, Items (1) and (2) of the theorem can be shown as in the case for rule [SKIP]. Moreover, the command *stop* does not read any variable and does not modify any variable, the only variable written by $x{:=}e$ is x, and the only variables read by $x{:=}e$ are the variables in the set $\textit{vars}(e)$ (by the definition of the operational semantics and our assumption on expression evaluation from Section 2.3.2). Hence, it follows from $x \notin mds_1(\textit{guar-no-w})$, $\textit{vars}(e) \cap mds_1(\textit{guar-no-r}) = \{\}$, and $mds_2 \sqsubseteq mds_1$ that $\langle c, mds_2, mem \rangle$ respects guarantees.

– [IF]. If $\vdash mds_1\ \{c\}\ mds_1'$ is derived with rule [IF], then $c = \textit{anns}$ if e then c_1 else c_2 fi, $\vdash mds_1\ \{\textit{anns}\}\ mds_1''$, $\vdash mds_1''\ \{c_1\}\ mds_1'$, and $\vdash mds_1''\ \{c_2\}\ mds_1'$ are derivable, and $\textit{vars}(e) \cap mds(\textit{guar-no-r}) = \{\}$.

Then, by rule [ANNO], $mds_1'' = \textit{mds-update}(mds_1, \textit{anns})$ and $mds_1(\textit{asm-no-r}) \cap X = mds_1''(\textit{asm-no-r}) \cap X = \{\}$. Let $mds_2'' = \textit{mds-update}(mds_2, \textit{anns})$. Then it follows from $mds_2 \sqsubseteq mds_1$ that $mds_2'' \sqsubseteq mds_1''$, and, hence, $mds_2(\textit{asm-no-r}) \cap X = mds_2''(\textit{asm-no-r}) \cap X = \{\}$.

If $\langle c', mds', mem' \rangle \in \textit{loc-reach}(\langle c, mds_2, mem \rangle)$ then $c' = c$ and $mds' = mds_2$, or $\langle c', mds', mem' \rangle \in \textit{loc-reach}(\langle c_1, mds_2'', mem'' \rangle)$ for some mem'', or $\langle c', mds', mem' \rangle \in \textit{loc-reach}(\langle c_2, mds_2'', mem'' \rangle)$ for some mem''. Hence, Items (2) and (3) of the theorem follow with the induction hypotheses for the derivations of $\vdash mds_1''\ \{c_1\}\ mds_1'$ and $\vdash mds_1''\ \{c_2\}\ mds_1'$. Moreover, the command c does not write any variables, and the only variables read by c are the variables in the set $\textit{vars}(e)$ (by the definition of the operational semantics and our assumption on expression evaluation from Section 2.3.2). Hence, it follows from $\textit{vars}(e) \cap mds_1(\textit{guar-no-r}) = \{\}$ and the induction hypotheses for the derivations of $\vdash mds_1''\ \{c_1\}\ mds_1'$ and $\vdash mds_1''\ \{c_2\}\ mds_1'$ that $\langle c, mds_2, mem \rangle$ respects guarantees.

– [SEQ]. In this case, $c = c_1; c_2$ and there is a mode state mds_1'' such that $\vdash mds_1\ \{c_1\}\ mds_1''$ and $\vdash mds_1''\ \{c_2\}\ mds_1'$ are derivable in the type system.

If $\langle c', mds', mem' \rangle \in \textit{loc-reach}(\langle c_1; c_2, mds_2, mem \rangle)$ then by the operational semantics for commands one of the following two conditions is satisfied:

1. $c' = c''; c_2$ and $\langle c'', mds', mem' \rangle \in \textit{loc-reach}(\langle c_1, mds_2, mem \rangle)$ or
2. $\langle \textit{stop}, mds'', mem'' \rangle \in \textit{loc-reach}(\langle c_1, mds_2, mem \rangle)$ and $\langle c', mds', mem' \rangle \in \textit{loc-reach}(\langle c_2, mds'', mem'' \rangle)$.

In the first case, the properties required for respecting guarantees and having compatible modes with *obs* follow from the induction hypothesis for the derivation of $\vdash mds_1\ \{c_1\}\ mds_1''$.

In the second case, we know from the induction hypothesis for the derivation of $\vdash mds_1\ \{c_1\}\ mds_1''$ that $mds'' \sqsubseteq mds_1''$. Hence, we obtain from the induction hypothesis for the derivation of $\vdash mds_1''\ \{c_1\}\ mds_1'$ that the properties required for respecting guarantees and having compatible modes with *obs* are satisfied, as well as that Item (3) of the theorem is satisfied.

The case for rule [WHILE] is proved using a combination of the proofs for rule [IF] and rule [SEQ].

– [SUB]. In this case, there exist mode states mds_3, mds_3' with $\vdash mds_3\ \{c\}\ mds_3'$, $mds_1 \sqsubseteq mds_3$, and $mds_3' \sqsubseteq mds_1'$.

From $mds_2 \sqsubseteq mds_1$ it follows that $mds_2 \sqsubseteq mds_3$. From the induction hypothesis for the derivation of $\vdash mds_3\ \{c\}\ mds'_3$ it follows that $\langle c, mds_2, mem\rangle$ respects guarantees and has modes compatible with obs. Moreover, it follows that if $\langle stop, mds'_2, mem'\rangle \in$ *loc-reach*$(\langle c, mds_2, mem\rangle)$ then $mds'_2 \sqsubseteq mds'_3$. Hence, $mds'_2 \sqsubseteq mds'_1$ because $mds'_3 \sqsubseteq mds'_1$. □

APPENDIX C

Semantics for Procedure Calls

A semantics for the language with procedures in Section 6.5 can be defined by extending the judgments for execution steps of commands from Section 2.2. To this end, we introduce judgments of the form $\langle coms, mems\rangle \rightarrowtriangle \langle coms', mems\rangle$, where $coms \in Com^*$ is a finite list of commands, and $mems \in Mem^*$ is a finite list of memories with $\sharp(coms) = \sharp(mems)$. The intuition is that $coms$ and $mems$ represent a call stack, where $coms[1]$ is the control state of the currently executing procedure, and $mems[1]$ is the memory of that procedure execution. The call stack grows by 1 when a procedure is called, and shrinks when a procedure terminates.

The derivation rules for the judgments of the form $\langle coms, mems\rangle \rightarrowtriangle \langle coms', mems\rangle$ are based on the judgments $\langle c, mem\rangle \rightarrowtriangle \langle c', mem'\rangle$ that model execution steps of programs without procedures. The derivation rules are displayed in Figure C.1, where mem_0 is a memory containing a default value for each variable.

$$\frac{\begin{array}{c} c = \mathcal{E}[\mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})] \\ \forall i \in \{1, \ldots, l_\mathsf{p}\} : eval(e_i, mem) = v_i \qquad mem' = mem_0[x_1^\mathsf{p} \mapsto v_1, x_{l_\mathsf{p}}^\mathsf{p} \mapsto v_{l_\mathsf{p}}] \end{array}}{\langle\langle c\rangle :: coms, \langle mem\rangle :: mems\rangle \rightarrowtriangle \langle\langle c_\mathsf{p}, c\rangle :: coms, \langle mem', mem\rangle :: mems\rangle}$$

$$\frac{\langle c, mem\rangle \rightarrowtriangle \langle c', mem'\rangle \qquad c' \neq stop}{\langle\langle c\rangle :: coms, \langle mem\rangle :: mems\rangle \rightarrowtriangle \langle\langle c'\rangle :: coms, \langle mem'\rangle :: mems\rangle}$$

$$\frac{\begin{array}{c} \langle c_1, mem_1\rangle \rightarrowtriangle \langle stop, mem_1'\rangle \qquad c_2 = \mathcal{E}[\mathsf{p}(e_1, \ldots, e_{l_\mathsf{p}}; z_1, \ldots, z_{m_\mathsf{p}})] \qquad c_2' = \mathcal{E}[stop] \\ mem_2' = mem_2[z_1 \mapsto mem_1'(y_1^\mathsf{p}), \ldots, z_{m_\mathsf{p}} \mapsto mem_1'(y_{m_\mathsf{p}}^\mathsf{p})] \end{array}}{\langle\langle c_1, c_2\rangle :: coms, \langle mem_1, mem_2\rangle :: mems\rangle \rightarrowtriangle \langle\langle c_2'\rangle :: coms, \langle mem_2'\rangle :: mems\rangle}$$

Figure C.1: Semantics for programs with procedures

Bibliography

[AB04] T. Amtoft and A. Banerjee. Information Flow Analysis in Logical Form. In *Proceedings of the 11th Symposium on Static Analysis (SAS)*, LNCS 3148, pages 100–115, Verona, Italy, 2004. Springer.

[ABC07] A. Almeida Matos, G. Boudol, and I. Castellani. Typing Noninterference for Reactive Programs. *Journal of Logic and Algebraic Programming*, 72(2):124–156, 2007.

[Aga00] J. Agat. Transforming out Timing Leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, USA, 2000. ACM.

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS)*, LNCS 5283, pages 333–348, Málaga, Spain, 2008. Springer.

[AP09] M. Abadi and G. D. Plotkin. A Model of Cooperative Threads. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 29–40, Savannah, GA, USA, 2009. ACM.

[AR80] G. R. Andrews and R. P. Reitman. An Axiomatic Approach to Information Flow in Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):56–76, 1980.

[AS07] A. Askarov and A. Sabelfeld. Localized Delimited Release: Combining the What and Where Dimensions of Information Release. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–60, San Diego, CA, USA, 2007. ACM.

[BC02] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science*, 281(1–2):109–130, 2002.

[BN02] A. Banerjee and D. A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270, Cape Breton, Nova Scotia, Canada, 2002. IEEE Computer Society.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In *Proceedings of the 10th Static Analysis Symposium (SAS)*, LNCS 2694, pages 55–72, San Diego, CA, USA, 2003. Springer.

[BPR07] A. Bossi, C. Piazza, and S. Rossi. Compositional Information Flow Security for Concurrent Programs. *Journal of Computer Security (JCS)*, 15(3):373–416, 2007.

[BRRS07] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of Multithreaded Programs by Compilation. In *Proceedings of the 12th European Symposium on Research in Computer Security (ESORICS)*, LNCS 4734, pages 2–18, Dresden, Germany, 2007. Springer.

[Bun11] Bundesamt für Sicherheit in der Informationstechnik. Die Lage der IT-Sicherheit in Deutschland 2011, 2011.

[CH08] D. Clark and S. Hunt. Non-Interference for Deterministic Interactive Programs. In *Proceedings of the 5th International Workshop on Formal Aspects in Security and Trust (FAST)*, LNCS 5491, pages 50–66, Málaga, Spain, 2008. Springer.

[Che93] J. Cheng. Slicing Concurrent Programs – A Graph-Theoretical Approach. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, LNCS 749, pages 223–240, Linköping, Sweden, 1993. Springer.

[Coh77] E. Cohen. Information Transmission in Computational Systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[CS02] M. Colón and H. Sipma. Practical Methods for Proving Program Termination. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV)*, LNCS 2404, pages 442–454, Copenhagen, Denmark, 2002. Springer.

[Den76] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DFPV09] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *Proceedings of the 18th European Symposium on Programming (ESOP)*, LNCS 5502, pages 363–377, York, UK, 2009. Springer.

[Din03] J. Dingel. Computer-Assisted Assume/Guarantee Reasoning with VeriSoft. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 138–148, Portland, OR, USA, 2003. IEEE Computer Society.

[Dur10] R. Durrett. *Probability: Theory and Examples*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University, 4th edition, 2010.

[FG95] R. Focardi and R. Gorrieri. A Taxonomy of Security Properties for Process Algebras. *Journal of Computer Security (JCS)*, 3(1):5–34, 1995.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[FRS05] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging Language-Based and Process Calculi Security. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures (FoSSaCS)*, LNCS 3441, pages 299–315, Edinburgh, UK, 2005. Springer.

[GFKD10] D. Garg, J. Franklin, D. Kaynar, and A. Datta. Compositional System Security with Interface-Confined Adversaries. In *Proceedings of the 26th Conference on Mathematical Foundations of Programming Semantics (MFPS)*, ENTCS 265, pages 49–71, Ottawa, ON, Canada, 2010. Elsevier.

[GH08] D. Giffhorn and C. Hammer. Precise Analysis of Java Programs Using JOANA. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 267–268, Beijing, China, 2008. IEEE Computer Society.

[GM82] 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, USA, 1982. IEEE Computer Society.

[GM05] R. Giacobazzi and I. Mastroeni. Timed Abstract Non-interference. In *Proceedings of the Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, LNCS 3829, pages 289–303, Uppsala, Sweden, 2005. Springer.

[GS12] D. Giffhorn and G. Snelting. Probabilistic Noninterference Based on Program Dependence Graphs. Technical Report 6, Karlsruhe Institute of Technology (KIT), 2012. Karlsruhe Reports in Informatics 2012.

[GTC+04] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust Management in Strand Spaces: A Rely-Guarantee Method. In *Proceedings of the 13th European Symposium on Programming (ESOP)*, LNCS 2986, pages 325–339, Barcelona, Spain, 2004. Springer.

[Ham09] C. Hammer. *Information Flow Control for Java*. PhD thesis, Universität Karlsruhe (TH), 2009.

[Ham10] C. Hammer. Experiences with PDG-Based IFC. In *Proceedings of the Symposium on Engineering Secure Software and Systems (ESSoS)*, LNCS 5965, pages 44–60, Pisa, Italy, 2010. Springer.

[HG11] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic. In *Proceedings of the 20th European Symposium on Programming (ESOP)*, LNCS 6602, pages 276–296, Saarbrücken, Germany, 2011. Springer.

[HomBC] Homer. Iliás, approx. 1000 BC.

[HQR98] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV)*, LNCS 1427, pages 440–451, Vancouver, BC, Canada, 1998. Springer.

[HRB90] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[HS06] S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 79–90, Charleston, SC, USA, 2006. ACM.

[HS09] C. Hammer and G. Snelting. Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control based on Program Dependence Graphs. *International Journal of Information Security (IJIS)*, 8(6):399–422, 2009.

[HS11] S. Hunt and D. Sands. From Exponential to Polynomial-Time Security Typing via Principal Types. In *Proceedings of the 20th European Symposium on Programming (ESOP)*, LNCS 6602, pages 297–316, Saarbrücken, Germany, 2011. Springer.

[HUM92] C. S. Hsieh, E. A. Unger, and R. A. Mata-Toledo. Using Program Dependence Graphs for Information Flow Control. *Journal of Systems and Software*, 17(3):227–232, 1992.

[HWS06] M. Huisman, P. Worah, and K. Sunesen. A Temporal Logic Characterisation of Observational Determinism. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–15, Venice, Italy, 2006. IEEE Computer Society.

[Jac89] J. Jacob. On the Derivation of Secure Components. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 242–247, Oakland, CA, USA, 1989. IEEE Computer Society.

[Jun11] Juniper Networks. 2011 Mobile Threats Report, February 2011.

[Jür00] J. Jürjens. Secure Information Flow for Concurrent Processes. In *Proceedings of the 11th International Conference on Concurrency Theory (Concur)*, LNCS 1877, pages 395–409, University Park, PA, USA, 2000. Springer.

[KM07] B. Köpf and H. Mantel. Transformational Typing and Unification for Automatically Correcting Insecure Programs. *International Journal of Information Security (IJIS)*, 6(2-3):107–131, 2007.

[Kri98] J. Krinke. Static Slicing of Threaded Programs. In *Proceedings of the SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 35–42, Montréal, QC, Canada, 1998. ACM.

[Kri03] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[Lam73] B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering (TSE)*, 3(2):125–143, 1977.

[LBMC94] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26(3):211–254, 1994.

[LM09] A. Lux and H. Mantel. Declassification with Explicit Reference Points. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, LNCS 5789, pages 69–85, Saint-Malo, France, 2009. Springer.

[LMP12] A. Lux, H. Mantel, and M. Perner. Scheduler-Independent Declassification. In *Proceedings of the 11th International Conference on Mathematics of Program Construction (MPC)*, LNCS 7342, pages 25–47, Madrid, Spain, 2012. Springer.

[Man00] H. Mantel. Possibilistic Definitions of Security – An Assembly Kit. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 185–199, Cambridge, UK, 2000. IEEE Computer Society.

[McL92] J. D. McLean. Proving Noninterference and Functional Correctness using Traces. *Journal of Computer Security (JCS)*, 1(1):37–57, 1992.

[MHC+10] J. P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro. Dynamically Checking Ownership Policies in Concurrent C/C++ Programs. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL)*, pages 457–470, Madrid, Spain, 2010. ACM.

[MMKM94] B. Mallo, J. D. McGregor, A. Krishnaswamy, and M. Medikonda. An Extensible Program Representation for Object-Oriented Software. *SIGPLAN Notices*, 29(12):38–47, 1994.

[MNZZ01] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow, Software release, 2001. http://www.cs.cornell.edu/jif.

[MR93] S. P. Masticola and B. G. Ryder. Non-concurrency Analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 129–138, San Diego, CA, USA, 1993. ACM.

[MR07] H. Mantel and A. Reinhard. Controlling the What and Where of Declassification in Language-Based Security. In *Proceedings of the 16th European Symposium on Programming (ESOP)*, LNCS 4421, pages 141–156, Braga, Portugal, 2007. Springer.

[MS03] H. Mantel and A. Sabelfeld. A Unifying Approach to the Security of Distributed and Multi-threaded Programs. *Journal of Computer Security (JCS)*, 11(4):615–676, 2003.

[MS04] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS)*, LNCS 3302, pages 129–145, Taipei, Taiwan, 2004. Springer.

[MS05] D. Meintrup and S. Schäffler. *Stochastik: Theorie und Anwendungen*. Statistik und ihre Anwendungen. Springer, 2005.

[MS07] H. Mantel and H. Sudbrock. Comparing Countermeasures against Interrupt-Related Covert Channels in an Information-Theoretic Framework. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, pages 326–340, Venice, Italy, 2007. IEEE Computer Society.

[MS09] H. Mantel and H. Sudbrock. Information-Theoretic Modeling and Analysis of Interrupt-Related Covert Channels. In *Proceedings of the 5th Workshop on Formal Aspects in Security and Trust (FAST)*, LNCS 5491, pages 67–81, Málaga, Spain, 2009. Springer.

[MS10] H. Mantel and H. Sudbrock. Flexible Scheduler-Independent Security. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS)*, LNCS 6345, pages 116–133, Athens, Greece, 2010. Springer.

[MS13] H. Mantel and H. Sudbrock. Types vs. PDGs in Information Flow Analysis. In *Proceedings of the 22nd International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2012)*, LNCS 7844, pages 106–121, Leuven, Belgium, 2013. Springer.

[MSK07] H. Mantel, H. Sudbrock, and T. Kraußer. Combining Different Proof Techniques for Verifying Information Flow Security. In *Proceedings the 16th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR)*, LNCS 4407, pages 94–110, Venice, Italy, 2007. Springer.

[MSS11] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232, Cernay-la-Ville, France, 2011. IEEE Computer Society.

[MSZ04] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 172–186, Pacific Grove, CA, USA, 2004. IEEE Computer Society.

[Mye99] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, USA, 1999. ACM.

[NA98] G. Naumovich and G. S. Avrunin. A Conservative Data Flow Algorithm for Detecting All Pairs of Statement That May Happen in Parallel. In *Proceedings of the the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 24–34, Lake Buena Vista, FL, USA, 1998. ACM.

[NAC99] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An Efficient Algorithm for Computing *MHP* Information for Concurrent Java Programs. In *Proceedings of the 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 338–354, Toulouse, France, 1999. ACM.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[Nul] NullLogic. Null httpd Web Server. http://nullhttpd.sourceforge.net/httpd/, accessed 2012 March 21.

[OBvEG10] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA)*, LIPIcs 6, pages 259–276. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

[OG76] S. S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6:319–340, 1976.

[O'H04] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *Proceedings of the 15th International Conference on Concurrency Theory (Concur)*, LNCS 3170, pages 49–67, London, UK, 2004. Springer.

[PBO07] M. J. Parkinson, R. Bornat, and P. W. O'Hearn. Modular Verification of a Non-blocking Stack. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL)*, pages 297–302, Nice, France, 2007. ACM.

[PDH99] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-Guarantee Model Checking of Software: A Comparative Case Study. In *Proceedings of the 5th and 6th Workshop on Theoretical and Practical Aspects of SPIN Model Checking*, LNCS 1680, pages 168–183, Toulouse, France, 1999. Springer.

[Pet81] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PS02] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 319–330, Portland, OR, USA, 2002. ACM.

[RA79] R. P. Reitman and G. R. Andrews. Certifying Information Flow Properties of Programs: An Axiomatic Approach. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages (POPL)*, pages 283–290, San Antonio, TX, USA, 1979. ACM.

[RHNS06] A. Russo, J. Hughes, D. A. Naumann, and A. Sabelfeld. Closing Internal Timing Channels by Transformation. In *Proceedings of the 11th Asian Computing Science Conference (ASIAN)*, LNCS 4435, pages 120–135, Tokyo, Japan, 2006. Springer.

[RHSR94] T. W. Reps, S. Horwitz, S. Sagiv, and G. Rosay. Speeding up Slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 11–20, New Orleans, LA, USA, 1994.

[Ros95] A. W. Roscoe. CSP and Determinism in Security Modelling. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–127, Oakland, CA, USA, 1995. IEEE Computer Society.

[RS06a] A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)*, pages 177–189, Venice, Italy, 2006. IEEE Computer Society.

[RS06b] A. Russo and A. Sabelfeld. Security for Multithreaded Programs Under Cooperative Scheduling. In *Proceedings of the Andrei Ershov 6th Conference on Perspectives of System Informatics (PSI)*, LNCS 4378, pages 474–480, Akademgorodok, Novosibirsk, Russia, 2006. Springer.

[RS09] A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler in the Presence of Synchronization. *Journal of Logic and Algebraic Programming*, 78(7):593–618, 2009.

[RWW94] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In *Proceedings of the 3rd European Symposium on Research in Computer Security (ESORICS)*, LNCS 875, pages 33–53, Brighton, UK, 1994. Springer.

[Sab01] A. Sabelfeld. The Impact of Synchronisation on Secure Information Flow in Concurrent Programs. In *Proceedings of the Andrei Ershov 4th Conference on Perspectives of System Informatics (PSI)*, LNCS 2244, pages 225–239, Akademgorodok, Novosibirsk, Russia, 2001. Springer.

[Sab03] A. Sabelfeld. Confidentiality for Multithreaded Programs via Bisimulation. In *Proceedings of the Andrei Ershov 5th Conference on Perspectives of System Informatics (PSI)*, LNCS 2890, pages 260–274, Akademgorodok, Novosibirsk, Russia, 2003. Springer.

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 317–331, Oakland, CA, USA, 2010. IEEE Computer Society.

[San09] D. Sangiorgi. On the Origins of Bisimulation and Coinduction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(4):111–151, 2009.

[Sim03] V. Simonet. The Flow Caml System, version 1.00, Documentation and User's Manual, 2003. http://www.normalesup.org/~simonet/soft/flowcaml/flowcaml-manual.pdf.

[SM03] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[SMH01] F. B. Schneider, J. G. Morrisett, and R. Harper. A Language-Based Approach to Security. In *Informatics: 10 Years Back, 10 Years Ahead*, LNCS 2000, pages 86–101, Dagstuhl, Germany, 2001. Springer.

[Smi01] G. Smith. A New Type System for Secure Information Flow. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 115–125, Cape Breton, Nova Scotia, Canada, 2001. IEEE Computer Society.

[Smi03] G. Smith. Probabilistic Noninterference through Weak Probabilistic Bisimulation. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 3–13, Pacific Grove, CA, USA, 2003. IEEE Computer Society.

[Sop11] Sophos. Security Threat Report 2011, 2011.

[SS99] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. In *Proceedings of the 8th European Symposium on Programming (ESOP)*, LNCS 1576, pages 50–59, Amsterdam, Netherlands, 1999. Springer.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 200–215, Cambridge, UK, 2000. IEEE Computer Society.

[Sta85] E. W. Stark. A Proof Technique for Rely/Guarantee Properties. In *Proceedings of the 5th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 206, pages 369–391, New Delhi, India, 1985. Springer.

[SV98] 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 (POPL)*, pages 355–364, San Diego, CA, USA, 1998. ACM.

[Sym11] Symantec. Symantec Internet Security Threat Report – Trends for 2010, April 2011.

[Var95] M. Y. Vardi. On the Complexity of Modular Model Checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS)*, pages 101–111, San Diego, CA, USA, 1995. IEEE Computer Society.

[VS97] D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *Proceedings of the 7th Conference on Theory and Practice of Software Development (TAPSOFT)*, LNCS 1214, pages 607–621, Lille, France, 1997. Springer.

[VS98] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop (CSFW)*, pages 34–43, Rockport, MA, USA, 1998. IEEE Computer Society.

[VS99] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security (JCS)*, 7(2,3):231–253, 1999.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security (JCS)*, 4(3):1–21, 1996.

[Was09] D. Wasserrab. Backing up Slicing: Verifying the Interprocedural Two-Phase Horwitz-Reps-Binkley Slicer. In *The Archive of Formal Proofs*. 2009.

[Was10] D. Wasserrab. Information Flow Noninterference via Slicing. In *The Archive of Formal Proofs*. 2010.

[Wei69] C. Weissman. Security Controls in the ADEPT-50 Time-sharing System. In *Proceedings of the Fall Joint Computer Conference*, AFIPS '69 (Fall), pages 119–133, Las Vegas, NV, USA, 1969. ACM.

[WL08] D. Wasserrab and A. Lochbihler. Formalizing a Framework for Dynamic Slicing of Program Dependence Graphs in Isabelle/HOL. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 5170, pages 294–309, Montréal, Québec, Canada, 2008. Springer.

[WLS09] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based Noninterference and its Modular Proof. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 31–44, Dublin, Ireland, 2009. ACM.

[WW94] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, Monterey, CA, USA, 1994. ACM.

[ZM03] S. Zdancewic and A. C. Myers. Observational Determinism for Concurrent Program Security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW)*, pages 29–43, Pacific Grove, CA, USA, 2003. IEEE Computer Society.

Symbols and Notation

Index