2-28-2011

Secure Information Flow via Stripping and Fast Simulation

Rafael H. Alpizar
Florida International University, rhAlpizar@ymail.com

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

SECURE INFORMATION FLOW VIA STRIPPING AND FAST SIMULATION

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Rafael Alpízar

2011

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Rafael Alpízar, and entitled Secure Information Flow via Stripping and Fast Simulation, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Dev Roy

Peter Clarke

Jinpeng Wei

Geoffrey Smith, Major Professor

Date of Defense: February 28, 2011

The dissertation of Rafael Alpízar is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Interim Dean Kevin O'Shea
University Graduate School

Florida International University, 2011

DEDICATION

To Marla, the love of my life, and to my children: Sofia, Leila, and Ariadna.

ACKNOWLEDGMENTS

I am grateful to my advisor and mentor, Geoffrey Smith, for guiding me though my most significant learning experience, showing me the wonders of theoretical computer science, and teaching me about life, integrity, and wisdom. I am also thankful to the members of my dissertation committee Peter Clarke, Dev Roy, and Jinpeng Wei for their many suggestions, advice, and diligence. This work was partially supported by the National Science Foundation under grants HRD-0317692 and CNS-0831114.

ABSTRACT OF THE DISSERTATION

SECURE INFORMATION FLOW VIA STRIPPING AND FAST SIMULATION

by

Rafael Alpízar

Florida International University, 2011

Miami, Florida

Professor Geoffrey Smith, Major Professor

Type systems for secure information flow aim to prevent a program from leaking information from H (high) to L (low) variables. Traditionally, bisimulation has been the prevalent technique for proving the soundness of such systems. This work introduces a new proof technique based on stripping and fast simulation, and shows that it can be applied in a number of cases where bisimulation fails. We present a progressive development of this technique over a representative sample of languages including a simple imperative language (core theory), a multiprocessing nondeterministic language, a probabilistic language, and a language with cryptographic primitives.

In the core theory we illustrate the key concepts of this technique in a basic setting. A fast low simulation in the context of transition systems is a binary relation where simulating states can match the moves of simulated states while maintaining the equivalence of low variables; stripping is a function that removes high commands from programs. We show that we can prove secure information flow by arguing that the stripping relation is a fast low simulation.

We then extend the core theory to an abstract distributed language under a nondeterministic scheduler. Next, we extend to a probabilistic language with a random assignment command; we generalize fast simulation to the setting of discrete time Markov Chains, and prove approximate probabilistic noninterference. Finally, we introduce cryptographic primitives into the probabilistic language and prove

computational noninterference, provided that the underling encryption scheme is secure.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION.

In the Summer of 2005 Geoffrey Smith, at the request of his doctoral students, taught a graduate seminar in cryptography. It was a wonderful seminar which immersed all of us in the area and out of which many ideas were generated; some of the ideas became the basis for [SA06]. In this paper we designed a language and type system for secure information flow which contained cryptographic primitives. In type systems for secure information flow, variables are assigned a security classification, e.g., high or low (H or L); then, the language semantics and typing rules ensure that the execution of well-typed programs is unable to leak information to variables with lower or incomparable security classification. However, the paper had a known weakness; it was missing the proof of a theorem that was used in the general soundness proof. We sketched the proof and assured our audience of its validity. It was fairly clear that the theorem was sound but the existing proof techniques were not useful. We had to look elsewhere and found the work of Baier, Katoen, Hermanns, and Wolf [BKHW05] on strong and weak simulation on Markov Chains which could be adapted to model the execution of probabilistic programs and might provide us with the needed proof technique. Strong simulation turned out to be too restrictive for us and while weak simulation could be used in the nondeterministic setting, it was not useful for the probabilistic setting. So we adapted this work to develop *fast simulation* which we now describe in a basic setting.

Consider a program which computes with H and L data. Suppose that there is a “stripping” function that can remove all the high commands from the program, so that the stripped program is limited to computing with low data. It turned out that *if* the stripped version of a program could *simulate* the execution of the original one, up to the final values of low variables, it meant that the original program could not leak high information.

Now, the precise formulation of simulation was critical; it could not be too restrictive (strong simulation) since that would require a lock-step execution. It could not be too loose; it would have to be faster (or at least as fast) as the original program for otherwise the simulation could loop forever and we did not want that.

So, we defined *fast simulation* and published [SA07]. A next obvious step was to expand our language into a multiprocessing system; there is significant work on multithreaded systems in the literature but most have severe typing restrictions on the language (typically, loop guards were restricted to be low) and we wanted a more expressive language. This in turn produced many problems that had to be resolved by typing and semantic rules; in the end we developed an abstract language for distributed program [AS09].

A contribution of [AS09] (over [SA07]) was a refined *stripping function* $\lfloor \cdot \rfloor$ that aggressively removed high computation without a trace; the previous version of the stripping function replaced secret computation with a no-op command **skip** (which took a step to execute). Also we introduced the concept of an empty command **done** to indicate a terminated execution. These changes greatly simplified the language analysis producing much simpler and more elegant theorems and proofs. For example, while before, the *simulation relation* R_L allowed a program to be simulated by many programs, including its stripped version, now the stripped version was the *only* simulation of the original one.

The concept of *fast simulation* is subtle and perhaps difficult to grasp, but it has been essential to prove the soundness of two of our languages (a probabilistic and a nondeterministic language). So in this work we present a comprehensive treatment of the theory that can be used as a good reference to researchers in this area in need of proof techniques and we apply the technique to a variety of languages.

1.1 Motivation

On confidentiality and trust. On any given day, information with varying levels of confidentiality flows back and forth throughout the world's computing systems. Whether you are doing a backup of medical records to an electronic vault in Colorado, using a credit card to purchase a book on Amazon, or reviewing the latest collider experiment at CERN from your office at FIU, the world computing systems exchange enormous amounts of confidential data. The assurance that confidential data remain so, is a daunting and perhaps unattainable goal.

At different levels of technical expertise, people perform different leaps of faith to reassure themselves about the confidentiality and integrity of data. The picture of a small lock on the lower right side of the screen is able to induce far more trust on the average user than volumes of security policies and communication protocols. Yet, while suggestive icons may carry more weight on certain audiences, there is never complete assurance of confidentiality or integrity. At the highest levels of formality, mathematicians and computer scientists face similar problems. While a proof of soundness is the definite statement about a model (if it has no mistakes), the model itself may not be adequate; as the abstraction approaches reality, the complexity of the model increases and its soundness proof becomes more intractable.

At the operational level technical managers agree on vast security policies and protocols so systems can share information. The correctness of such is highly questionable. A *best effort trust* is the best we currently have and such trust is based more on the perceived integrity and openness of the community of scientists and technologists that develop the techniques and work on the verification of the systems than in the actual policies and proofs that are beyond the understanding of most.

The kernel of trust. Nevertheless, while trust is what makes the world's systems viable, the technical work is the kernel without which there would be nothing to trust. An understanding of the vulnerability of systems is essential in designing new more secure systems; while a proof of soundness of, say, a new secure language design may still have problems at a different level of abstraction, it can still eliminate a vast class of security problems. Therefore, we recursively strive to refine systems to increase these security properties.

Secure information flow is motivated by the idea that information of diverse levels of confidentiality should flow through computing systems without leaking to lower or incomparable levels. As these are essentially sets of programs operating on memories, this field focuses on maintaining confidentiality at specific "observational" points during the execution of programs.

1.2 Problem Description and Contribution

In this work we wish to address the feasibility of a practical secure language. Ultimately, this would require an actual implementation which would be beyond the scope of a PhD dissertation. So instead, we propose the following criteria for success:

- the work should present successful implementations of languages with features that are representative of commercial languages; it should include at a minimum: probabilistic commands, concurrency, and cryptographic primitives.
- a single theory should handle the establishment of the security properties of all the languages.

- the languages should be simple and elegant, the programmer should not have to learn complex syntax or semantics and the type system should not restrict the programmer too unreasonably. At a minimum, the guards of **if** and **while** commands should not be restricted.

We try to meet the criteria above. We introduce a representative set of secure languages under one framework with features that are similar to commercial languages. In general, we see our contribution as follows:

- introduction of stripping, fast simulation, and fast low simulation as an analysis tool for soundness of languages with security policies.
- design and soundness analysis of four representative secure languages

  [a] core language,

  [b] abstract language for distributed systems,

  [c] probabilistic language, and

  [d] cryptographic language.

The stripping function has evolved over the years to what we now call aggressive stripping. Aggressive stripping actively removes high commands from sequential compositions. This is in contrast to the original function which replaced high commands with **skip**. A useful detail related to this is the introduction of the terminated command **done** into the standard language syntax, which yielded more elegant and intuitive soundness proofs.

More precisely we see our contribution as follows:

- Chapter 3 contains a detailed exposure of the core theory in a basic setting, provides easy understanding of the key concepts.

- Chapter 4 contains a treatment of a distributed language which is only limited by the Denning restrictions within individual processes.
- Chapter 5 has two main contributions. First, it has a quantitative account of information flows caused by nontermination in programs that satisfy just the Denning restrictions; this is important for understanding more precisely what is guaranteed in languages that allow termination channels. Second, it provides a technical contribution by introducing a new notion of simulation, *fast probabilistic simulation*, and applying it to the area of secure information flow; to our knowledge probabilistic simulation (unlike probabilistic *bisimulation*) has not previously been used in secure information flow. Also, this chapter provides a first treatment of a secure probabilistic language.
- Chapter 6: contains a first treatment of a cryptographic language within the computational cryptographic model of [BR05].

These contributions were originally published in four publications as described below.

Organization. In this work, our goal is to produce a model, a set of tools, and a set of applications that use these tools. These, we believe, can be useful to researchers and secure language designers to create more sophisticated and usable secure languages. It is organized as follows. In Chapter 2 we review the background areas upon which our theory rests and place our work within the research literature. Chapter 3 presents the core theory of stripping and fast simulation in the context of transition systems. The simple imperative language of [VSI96] is used to illustrate the core techniques. Chapter 4 applies the core theory to an abstract distributed system under a nondeterministic scheduler. This work is a revised version of our FAST[1]-09 paper [AS09]. Chapter 5 further extends the core theory to a probabilistic

[1] Formal Aspects of Security and Trust.

environment. The new context is now discrete time Markov Chains. An early version of this chapter was initially published at PLAS[2]-07 [SA07], but later it was expanded to a journal article in MSCS[3]-10 [SA11]. We consider this chapter the central contribution of this thesis. Chapter 6 is an application of Chapter 5 to a cryptographic language. Paradoxically, this was the first of our papers; it was first published in FMSE[4]-06. Finally, Chapter 7 presents future work and concludes.

[2] Programming Languages and Analysis for Security.

[3] Mathematical Structures in Computer Science.

[4] Formal Methods in Security Engineering

CHAPTER 2

BACKGROUND AND RELATED WORK.

In this chapter we provide a brief history of the research area with explanations of some of its key concepts and we relate the contributions of this work to the research literature.

2.1 Type Systems and Secure Information Flow

During the period between 1900-1902 mathematics suffered devastating challenges from the discovery of logical paradoxes; these threatened to invalidate all mathematical findings. Type systems were first devised then, to establish the soundness of logical systems. Somewhat independently, during the 1950s, type systems began emerging in programming languages, mostly to segregate floating point from integer operations in FORTRAN but, as the fields of computer science and logic merged, type system grew in sophistication and extent. Today, Benjamin Pierce [Pie02] defines type systems as tools for reasoning about programs, or more precisely, as tractable syntactic methods for proving the absence of certain program behaviors by classifying phrases according to the values they compute.

Secure information flow is concerned with a class of systems that compute on data with varying degrees of confidentiality; it intends to identify *illegal* flows of confidential data to variables with lower or incomparable classifications or to provide guarantees that such flows do not occur. The degrees of confidentiality form a security lattice that can range from simple $\{H$ (high), L(low)$\}$ to complex lattices. Hence, by classifying the data appropriately, the sources of programs need not be trusted as long as they meet a secure flow certification. Secure information flow was pioneered by Dorothy Denning in the 1970s. She introduced security classifications for variables in programs and studied ways to prevent information from leaking to variables of lower classification [Den75].

The Denning restrictions.. The seminal work in Secure Information Flow was the Dennings' 1977 paper [DD77], which proposed what we now call the *Denning restrictions*:

- An expression is classified as H if it contains any H variables; otherwise, it is classified as L.
- To prevent *explicit flows*, a H expression cannot be assigned to a L variable.
- To prevent *implicit flows*, an **if** or **while** command whose guard is H may not make *any* assignments to L variables.

Explicit flows are direct transfers of information to lower or incomparable variables; for example, let variable l be typed L ($l : L\ var$) and $h : H\ var$. The command $l := h$ does an explicit flow because it copies the value of h into a variable with a lower security type. Implicit flows are indirect transfers of information to lower or incomparable variables; e.g., if h can only be 0 or 1, the command **if** h **then** $l := 1$ **else** $l := 0$ copies the value of h to l without using a direct assignment; hence, the command does an implicit flow. The languages presented in this work are only limited by the *Denning restrictions*.

Noninterference. In the late 1970s Ellis Cohen defined the concept of *strong dependency* [Coh77] as a measure of whether any information is transmitted on a channel. Somewhat later, Goguen and Messeguer [GM82] defined the concept of *noninterference* as: given two sets of users, what one set does (the H commands for us) has no effect in what the other set sees as final outputs. Noninterference, as it is currently defined, is more related to Cohen's definition and has become the desired property of secure languages. Since its inception, variants of noninterference have emerged. The first variant was *probabilistic noninterference*, first proposed by McLean and Gray [McL90, Gra90] respectively, which extended the concept to probabilistic languages. To meet this new property, changes to the high inputs of programs could not change the probability distributions of the low outputs. The second variant was *computational noninterference*; first introduced by Backes and Pfitzmann in [BP02], it was a significant extension of the original definition. Under this property a program was allowed to fully leak values as long as recovering them would require exponential time on the size of a security parameter (usually the encryption key). The work by Sabelfeld and Myers [SM03] is a survey of secure information flow during this period (up to 2003).

The use of type systems for secure information flow was first introduced by Dennis Volpano, Geoffrey Smith, and Cynthia Irvine in [VSI96], to disallow leaking behavior in well-typed programs. The paper presented a simple imperative language with standard syntax and semantics; each variable was classified within a security lattice which captured the desired information flow policy. The paper integrated and clarified existing concepts of security (like confinement and subject reduction) and established noninterference as the key property for secure information flow. The type system now focused on the control of information flow within program executions rather than its traditional role as enforcer of data-type compatibility; as it was designed to enforce only the Denning restrictions, *the key contribution of the paper was a proof that implementing the Denning restrictions was sufficient for secure information flow on the simple imperative language.* Advantages of this approach are that the Security Policy (Type System) and Information Flow Policy (Security Lattice) can be treated separately from each other and from the language semantics, although soundness is established with respect to the complete system.

Cryptography. The complexity of language and type systems for secure information flow has increased over the years. The goal has been to provide practical applicability to the systems. To this end in early 2000, Peeter Laud pioneered the area of computationally-secure information-flow analysis in the presence of encryption. In his first works [Lau01], the analysis was not in the form of a type system but later, with Varmo Vene [LV05], they develop the first language and type system with encryption and a computational security property. Our work includes a cryptographic language in this vein, which was first published as [SA06]. More recently, Backes and Pfitzmann [BP05] establish secrecy properties for a rich Dolev-Yao-style cryptographic library.

Quantitative Information Flow. Rather than certifying a program secure, recently, there has been a great deal of work to quantify information leaks of programs that may not satisfy the Denning restrictions. Using Information Theory and other methods, the new approach has been to measure the leakage of information quantitatively; examples include Di Pierro, Hankin, and Wiklicky [DPHW02] and Clark, Hunt, and Malacaria [CHM02]. Later, in work by Malacaria [Mal07], he uses Shannon entropy to assign a quantitative measure to the amount of leakage caused by **while** loops that *violate* the Denning restrictions, for example by assigning to a L variable in the body of a **while** loop with a H guard. This work is in contrast with the traditional approach where the focus is on proving a program segment secure. For a general survey of recent work (up to 2005) on incorporating declassification into secure information-flow analyses see Sabelfeld and Sands [SS05].

The use of Shannon entropy provides an effective expectation of what it would take to guess a secret when the secrets is a uniformly distributed random variable. However, this expectation metric is not adequate as a general measure of the vulnerability of programs within secure information flow. The problem, as illustrated by Smith [Smi09], is that a secret may be highly guessable and yet have an arbitrarily high guessing entropy. To address this basic limitation, new metrics have been proposed in the areas of anonymity protocols [THV04, SW06] and more comprehensively within secure information flow [Smi08, Smi09]. The new metrics use *min-entropy* as a better measure of the vulnerability of a system.

Most recently, *Shannon entropy* and *min-entropy* have been applied to the problem of bounding the effectiveness of countermeasures against timing and side channel attacks; these attacks are methods of indirectly extracting information from a cryptosystem base on its physical performance and timing. For example, Kocher [Koc96] uses timing to extract bits of an RSA encryption key. Also, side channel attacks are very effective because they can be done remotely [BB05].

Input blinding aims to de-correlate the timing measure from the attacker's intended input by randomizing it prior to preforming decryptions. Bucketing established discrete time intervals for the computation of cryptographic operations. Kocher introduced *input blinding* [Koc96] and much later Köpf and Dürmuth introduced *bucketing* [KD09] and used Shannon entropy to bound the leakage under this countermeasure. Subsequently Köpf and Smith, using *min-entropy*, expanded this result to address when the attacker also possesses partial information of the cryptosystem.

Simulation and bisimulation. As detailed in [San09] bisimulation/coinduction was first introduced by Robin Milner in a series of papers whose goal was to establish when two programs are equivalent. In [Mil71], Milner first introduced simulation of total imperative programs and then introduced bisimulation as a symmetric simulation relation. Later [Mil82] refined the concept.

Ten years earlier Kemeny and Snell [KS60] had introduced the concept of *lumpability* in Markov Chains. Lumpability is a property of some Markov Chains where sets of states can be lumped together thereby reducing the state space without affecting its operations. States within the lumped sets were equivalent in some manner.

Following on Milner's work, Larsen and Skou [LS91] extended bisimulation to a probabilistic setting as an equivalence relation on probabilistic transition systems. Then, reintroducing lumpability on Markov Chains, probabilistic bisimulation was further expanded to allow for programs that don't quite "track" each other in *lock-step*. Baier, Katoen, Hermanns, and Wolf [BKHW05] expanded this area redefining the original concept of bisimulation to *strong bisimulation*, and introducing *weak bisimulation*. They also introduced the concepts of strong and weak *simulation*.[1]

From our perspective we would say that bisimulation is an equivalence relation where two transitioning systems behave indistinguishably with respect to a formal observer. The observer may be myopic in some cases and miss certain classes of transitions. For example in [Smi03] *weak probabilistic bisimulation* is established on a concurrent language and type system where the *observer* is not able to notice the time that the processes spends doing secret calculations. The effectiveness of *bisimulation* in establishing program equivalence, we believe, deemphasized *simulation* as a tool to prove noninterference on computing systems. But as we will see, there are many languages where executions of the same program/system multiple times are not bisimilar yet they posses a noninterference property due to an observational inability of the adversary. Hence, simulation, combined with observational determinism [ZM03], is a tool that can be used where bisimulation fails as well as instead of it.

[1] The setting for this work was discrete time and continuous time Markov Chains, but here we are only concerned with DTMC.

Practical secure languages. Recent work towards the creation of practical secure information-flow languages includes Jif [MCN$^+$06] and *Aura* [JVM$^+$08]. Jif is older, richer, fairly well introduced as a practical language but does not allow placement of secret data on public variables except by using explicit declassification. In contrast the extensive functional imperative language *Aura* maintains confidentiality and integrity properties of its constructs as specified by its label [JVM$^+$08] by "packing" them using asymmetric encryption before declassification. The cryptographic layer is hidden to the programmer making it easier to use. This system uses static and runtime checking to enforce security. Using a different approach, Zheng and Myers [ZM08] use a purely static type system to achieve confidentiality by splitting secrets under the assumption of non-collusion of repositories (e.g. key and data repositories). Under this model ciphertexts do not need to be public which allows relaxation of the type system while maintaining security. Further towards the practical end of the spectrum are efforts to provide assurance levels to software (as in EAL standard). In this line of work (Shaffer, Auguston, Irvine, Levin [SAIL08]) a security domain model is established and "real" programs are verified against it to detect flow violations. Also, Askarov, Hedin, and Sabelfeld [AHSS08] explores termination-insensitive noninterference (as guaranteed by the Denning restrictions) in the context of a deterministic programming language with an **output** command. They observe that such programs can leak an unbounded amount of information (albeit slowly) by going into an infinite loop at some point within a sequence of outputs.

2.2 Related Work

Our work encapsulates a number of contributions which we now relate to the literature. To the best of our knowledge this is the first cohesive treatment of a representative set of languages and type systems under one framework and one proof technique. The key concepts in this work are *stripping* and *fast low simulation.* Fast low simulation is based on *fast simulation on Markov Chains* from [SA07, SA11] which in turn was based on *strong* and *weak* simulation for discrete time Markov chains by Christel Baier, Joost-Pieter Katoen, Holger Hermanns, and Verena Wolf [BKHW05].

Stripping is somewhat reminiscent of the work of Agat [Aga00], which proposes to eliminate external timing leaks in programs through a transformation-based approach. But our stripping operation is not an implementation technique, but rather a *thought experiment* that we use to better understand the behavior of programs under the Denning restrictions.

The theorem that stripping is a fast low simulation shows that the theory of probabilistic simulation can be applied fruitfully to the secure information flow problem. This theorem adds a more powerful proof technique to the existing bisimulation-based approach of work like [LV05, SA06], and [FR08] on languages with cryptography, and [AFG98, SV98, Smi03, ACF06, FC08] on multi-threaded languages. In [AS09], the theorem was adapted from a probabilistic to a nondeterministic environment to prove secure information flow in a distributed language.

In this work we present a variety of language and type systems and prove different flavors of noninterference property on them using the techniques described above. Next we place each language and key concepts within the research literature.

The core language is intended to provide a clear exposition of the core techniques without having to deal with complicating issues like probabilistic or nondeterministic program executions. The core language is the same as [VSI96] except for the **done** command. However, the mechanisms to establish noninterference are now based on simulation rather than bisimulation.

The abstract distributed language. Previous work on multithreaded languages include [AFG98, SV98, Smi03, ACF06, FC08]. In order to attain soundness, these languages have restrictive type systems which strongly curtailed the language's expressive power. For example, [Smi01] achieves probabilistic noninterference on multithreaded programs by requiring that a command whose running time depends on H variables cannot be followed sequentially by an assignment to a L variable. An exception to these is the extensive work on observational determinism in concurrent languages by Zdancewic and Myers [ZM03]. In this paper the authors present a fairly rich concurrent language with message passing, memory sharing, and a relaxed language syntax, and prove a noninterference property with respect to a sound *race-condition* detection program that rejects any program that is found to have a race-condition. The paper separates the explicit/implicit flow leaks with the leaks due to thread-racing and mostly addresses the first. In contrast, our language is much simpler in its information sharing model, does not type the program counter, and its type system prevents race-conditions for well-typed programs. Hence, we gain intuition into a variety of information leaking mechanisms. Another language in a similar vein is the work of Fournet and Rezk [FR08]. The language has a rich set of cryptographic primitives and a treatment of integrity as well as confidentiality. The primary differences between these works are that their system does not handle concurrency and is subject to timing channels; on the other hand, their active adversary is more powerful, having the ability to modify public data.

Our random assignment language is an enhancement of Section 3 of [SA06]. As far as we know there is no other isolated treatment of a random assignment language in the literature. Laud and Vene's cryptographic language [LV05] has built in random assignment and their analysis could be done on programs that use random assignment but not encryption. However, this would establish that such programs, if well-typed, satisfy a *computational noninterference* property for polynomial-time programs; this is weaker than the *probabilistic noninterference* property that we show. Random assignment makes the language probabilistic, and hence similar to multi-threaded languages that have been previously investigated. But because the language is sequential, we do not have *races* between different threads, which greatly complicate secure information flow analysis and require more restrictive typing rules.

The cryptographic language. The cryptographic framework we used for this language is called the computational framework and is based on the work of Bellare and Rogaway [BR05], However, much of the work on cryptographic protocols is based on the Dolev-Yao model [DY83]. To relate these two, the pioneering work of Abadi and Rogaway [AR00] proves a computational justification for Dolev-Yao-style analysis of protocol messages under passive adversaries.

Based on [AR00], in [Lau03], Peeter Laud establishes a formal methods security property that is equivalent to IND-CCA. Then, building on these and [BP05], Laud [Lau05] presents a type system to ensure a computational secrecy property for communication protocols. The system is based on a process calculus (like *spi*) with a rich set of constructs for message passing, symmetric and asymmetric encryption and decryption, and key and nonce generation. The main difference with our cryptographic language is that the type system has a major limitation: its rules for typing conditionals, (IfH) and (IfR), do not allow branching on secret data; this may not be a problem for a protocol language but it certainly is a major restriction for a general-purpose programming language. Soon after, with Varmo Vene [LV05], they develop the first language and type system with cryptography and a computational security property. The main differences between [LV05] and our cryptographic language are that their type system does not address decryption, while ours does, and the security property of their model is equivalent to IND-CCA while we use both IND-CCA and IND-CPA, which is a weaker security and as such is easier to implement in a cryptographic scheme. On the other hand, their language is richer in that it supports key generation (although not inside loops) while our cryptographic language does not handle key manipulation and assumes a single, implicitly generated key for all the encryption and decryption operations; in compensation, we are able to use a much simpler type system and soundness proof. Much later, in [Lau08], Laud proves a computational noninterference property on a type system derived from the work of Askarov, Hedin, and Sabelfeld [AHS06]. Another language close to ours is by Focardi and Centenaro [FC08]. It treats a multiprogrammed language and type system over asymmetric encryption and proves a noninterference property on it. The main differences are that their type system is more restrictive, requiring low guards on loops, and they use a formal methods approach rather than computational complexity.

Recently there has been new work to relax noninterference by allowing *some* leaks of information while still preserving security (in some sense). For example, Volpano [Vol00] shows a computational justification for typing rules for one-way hash functions; because hashing is deterministic, however, his typing rules need to be more restrictive than the rules we use here. Another notable work is Li and Zdancewic [LZ05], which uses downgrading policies as security levels, so that the security level specifies what must be done to “sanitize” a piece of information.

More distantly related is the large body of recent work aimed at proving computational security properties of cryptographic protocols. An example is Warinschi's paper [War03] that proves the computational soundness of the Needham-Schroeder-Lowe public key protocol, provided that the encryption scheme is IND-CCA secure. However, it should be noted that the problem addressed by such works is very different from the secure information-flow problem: protocol work considers distributed systems in the presence of an *active adversary*, whose behavior is unconstrained but who does not have direct access to certain secrets; in contrast, secure information-flow analysis considers untrusted programs that *do* have direct access to secret information and that must be prevented (using typing rules, for example) from propagating it improperly.

CHAPTER 3
STRIPPING AND FAST SIMULATION ON THE CORE LANGUAGE.

In this chapter we define our core theory. Our goals are ease of understanding, simplicity of exposure, and completeness; we use the simple imperative language of [VSI96] to meet our goals. We first define the syntax, semantics, and type system of the core language and argue its basic properties, then we define *fast simulation* in the general context of transition systems; we define the *stripping* of "high computation" from program executions; and finally, we argue a noninterference property on well-typed programs.

Our language syntax is defined in Figure 3.1. In the syntax, metavariables x, y, z range over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true. A novelty of our language is that we have replaced the usual **skip** command with a **done** command instead; **done** can be used in much the same way as **skip** and (as will be seen later) it is also used to represent a terminated command in a configuration.

3.1 Imperative Language Semantics

In this section, we review the semantics for our simple imperative language. A program c is executed under a *memory* μ. Rather than modeling memory as an array

$$
\begin{array}{llll}
\textit{(phrases)} & p & ::= & e \mid c \\
\textit{(expressions)} & e & ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
\textit{(commands)} & c & ::= & \textbf{done} \mid x := e \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid c_1 ; c_2
\end{array}
$$

Figure 3.1: Core Language Syntax

of values which mutates upon updates, we use a purely functional abstraction where μ is a *partial function* that maps identifiers to integers, integers being the only values in our language. We assume that expressions are total and evaluated atomically, and we write $\mu(e)$ to denote the value of expression e in memory μ. In our semantics, a *configuration* is a pair (c, μ) where c is the command remaining to be executed and μ is a memory. We write $\mu[x := v]$ to denote a new memory function that is identical to μ, except that x is mapped to value v. Defining memory as a partial function can lead to some complications; for example, the absence of a variable in the domain of the memory-function could cause execution to get stuck. So we address these issues by assuming a "well-formedness" property on our configurations:

Assumption 3.1.1 (Well-Formed Configurations) *Given any configuration (c, μ) in our languages and type systems, the domain of the memory-function, dom(μ), contains all the identifiers in the command c.*

Note that the semantics do not allow the generation of new identifiers; the domain of the memory remains constant during program execution. *Terminal configurations* are written as $(\mathbf{done}, \mu)$ in our semantics. In previous work terminal configurations were written as μ, but this leads to a proliferation of cases in proofs; it is therefore more elegant to have only configurations of the form (c, μ). Also, we can now code "**if** e **then** c" using "**if** e **then** c **else done**".

The language uses a small-step semantics which is shown in Figure 3.2. We write $(c, \mu) \longrightarrow (c', \mu')$ to indicate the transition from configuration (c, μ) to configuration (c', μ'). The transition relation is governed by a set of semantic rules. *A semantic rule* is composed of a set of hypothesis, a separating line, and a conclusion. For

$$(\mathit{update}_s) \qquad \frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)])}$$

$$(\mathit{if}_s) \qquad \frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_2, \mu)}$$

$$(\mathit{while}_s) \qquad \frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow (\mathbf{done}, \mu)}$$

$$\frac{\mu(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu)}$$

$$(\mathit{compose}_s) \qquad \frac{(c_1, \mu) \longrightarrow (\mathbf{done}, \mu')}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$$

$$\frac{(c_1, \mu) \longrightarrow (c_1', \mu') \quad c_1' \neq \mathbf{done}}{(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')}$$

$$(\mathit{done}_s) \qquad (\mathbf{done}, \mu) \longrightarrow (\mathbf{done}, \mu)$$

Figure 3.2: Core Language Semantics

example, consider rule update_s:

$$(\mathit{update}_s)\ \frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)])}$$

that premises x is in the domain of μ. We can see that the well-formedness assumption prevents executions of this command from getting stuck. Based on the premise, executing command $x := e$ on memory μ, evaluates e and yields a new memory-function such x is now mapped to $\mu(e)$. Sometimes we need multiple semantic rules to handle a command; for example, consider the command $\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$; this command takes different actions depending on whether the guard is true or false. So we have two semantic rules, one for each premise. Here we see the first rule, when the guard is true:

$$(\mathit{if}_s)\ \frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu)}$$

then, the meaning of the command is that subcommand c_1 is selected for execution next; no changes to the memory occur. We now describe the remaining semantic rules in reference to Figure 3.2:

- *while$_s$*: when the guard is true, this rule prefixes the body of its loop with itself for future execution. When the guard is false it simply terminates.
- *compose$_s$*: when the first command of the composition finishes in one step, this rule selects the second command for "continuation". Otherwise it selects the resulting command of the first command together with the resulting memory.
- *done$_s$*: this rule does nothing in one step. It is a bit odd for **done** to make transitions, but necessary so that a command like "**done**; $x := 5$" does not get stuck.

Our semantics make the language deterministic; we now argue this point, starting with a proof that for any configuration there is one and only one "next step".

Lemma 3.1.1 (Unique Next Step) *For any configuration (c, μ) there is (c', μ') such that $(c, \mu) \longrightarrow (c', \mu')$ and for any (c'', μ'') such that $(c, \mu) \longrightarrow (c'', \mu'')$, it must be the case that $(c'', \mu'') = (c', \mu')$.*

Proof. By induction on the structure of c.

- If c is of the form **done**, then by rule *done$_s$*, $(c, \mu) \longrightarrow (\textbf{done}, \mu)$ which establishes that there is at least one (c', μ') such that $(c, \mu) \longrightarrow (c', \mu')$. But is there another (c'', μ'') such that $(c, \mu) \longrightarrow (c'', \mu'')$? No, because there is no other semantic rule by which $(\textbf{done}, \mu)$ may transition.

- If c is of the form $x := e$, then by rule *update*$_s$, and only by this rule, $(c, \mu) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)])$.

- If c is of the form **if** e **then** c_1 **else** c_2, first note that for fixed memory μ, $\mu(e)$ yields one unique value that can either be 0 or not. If it is not 0 then by first rule *if*$_s$, and only by this rule, $(c, \mu) \longrightarrow (c_1, \mu)$. Otherwise $\mu(e) = 0$, then, only by second rule *if*$_s$, $(c, \mu) \longrightarrow (c_2, \mu)$.

- If c is of the form **while** e **do** c_1, then the argument is similar as the case for the **if** command.

- If c is of the form $c_1; c_2$: then by induction there is unique (c_1', μ') such that $(c_1, \mu) \longrightarrow (c_1', \mu')$. c_1' can either be **done** or not. If it is, i.e., $(c_1, \mu) \longrightarrow (\mathbf{done}, \mu')$, then only by first rule *compose*$_s$ can $(c_1; c_2, \mu)$ transition, and it must be to (c_2, μ'). If c_1' is not **done**, then only by second rule *compose*$_s$ can $(c_1; c_2, \mu)$ transition, and it must be to $(c_1'; c_2, \mu')$.

□

Corollary 3.1.2 (Language Determinism) *For any configuration (c, μ), for all $k \geq 0$, there is (c', μ') such that $(c, \mu) \longrightarrow^k (c', \mu')$ and for any (c'', μ'') such that $(c, \mu) \longrightarrow^k (c'', \mu'')$, it must be the case that $(c'', \mu'') = (c', \mu')$.*

Proof. By induction on k.

- Base step: $k = 0$. Then $(c', \mu') = (c, \mu)$.
- Inductive step: $k = n + 1$. By induction, there exists a unique configuration (c'', μ'') such that $(c, \mu) \longrightarrow^n (c'', \mu'')$. By Lemma 3.1.1, there exists unique (c''', μ''') such that $(c'', \mu'') \longrightarrow (c''', \mu''')$. Finally let $(c', \mu') = (c''', \mu''')$.

□

We remark that every configuration (c, μ) has a unique infinite trace

$$(c, \mu) \longrightarrow (c_1, \mu_1) \longrightarrow (c_2, \mu_2) \longrightarrow \cdots$$

If a configuration (c_i, μ_i) where $c_i = \textbf{done}$ is reached then we have a terminating execution, otherwise we have a nonterminating execution. We are now ready to define a type system that will guarantee secure information flow.

3.2 Core Language Type System

Our typing rules simply enforce the Denning restrictions [DD77], which are described in Chapter 2. Here are the types we will use in this chapter:

$$\begin{array}{ll} (\textit{data types}) & \tau ::= L \mid H \\ (\textit{phrase types}) & \rho ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \end{array}$$

We use types H and L which represent the minimum set of types that is interesting for our purpose, but could straightforwardly generalize to an arbitrary lattice. Security types must at least form a partially ordered set (poset) but in our case a lattice is more useful. For example, while an expression like $e_1 + e_2$ might not have a unique least upper bound type in a poset, this would be guaranteed in a lattice. See [Smi07] for more details. Also, we only type the security aspect of data (since other than security, our data type is always integer). As usual, $\tau\ \textit{var}$ is the type of variables that store information of level τ, while $\tau\ \textit{cmd}$ is the type of command that assigns only to variables of level τ or higher; this implies that command types obey an *antimonotonic* or *contravariant* subtyping rule with respect to data type. Note that $\tau\ \textit{var}$ is *not* subsumable while τ and $\tau\ \textit{cmd}$ are.

Typing judgments have the form $\Gamma \vdash p : \rho$, where Γ is a partial function that maps identifiers to types of the form $\tau\ \textit{var}$. Generally, Γ is seldom mentioned.

Assumption 3.2.1 (Well-Formed Typing Judgments) *Γ is a partial function that maps identifiers to types. We assume that the domain of $\Gamma \vdash \rho$ includes all the identifiers that are needed in ρ.*

$$(\mathit{base}) \qquad L \subseteq H$$

$$(\mathit{cmd}) \qquad \frac{\tau \subseteq \tau'}{\tau' \ \mathit{cmd} \subseteq \tau \ \mathit{cmd}}$$

$$(\mathit{reflex}) \qquad \rho \subseteq \rho$$

$$(\mathit{trans}) \qquad \frac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$(\mathit{subsump}) \qquad \frac{\Gamma \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

$$(\mathit{done}_t) \qquad \Gamma \vdash \mathbf{done} : H \ \mathit{cmd}$$

$$(\mathit{int}_t) \qquad \Gamma \vdash n : L$$

$$(\mathit{rval}_t) \qquad \frac{\Gamma(x) = \tau \ \mathit{var}}{\Gamma \vdash x : \tau}$$

$$(\mathit{update}_t) \qquad \frac{\Gamma(x) = \tau \ \mathit{var} \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau \ \mathit{cmd}}$$

$$(\mathit{plus}_t) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$$

$$(\mathit{if}_t) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau \ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}}{\Gamma \vdash \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 : \tau \ \mathit{cmd}}$$

$$(\mathit{while}_t) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau \ \mathit{cmd}}{\Gamma \vdash \mathbf{while} \ e \ \mathbf{do} \ c : \tau \ \mathit{cmd}}$$

$$(\mathit{compose}_t) \qquad \frac{\Gamma \vdash c_1 : \tau \ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}}{\Gamma \vdash c_1 ; c_2 : \tau \ \mathit{cmd}}$$

Figure 3.3: Core Language Type System

Figure 3.3 shows the typing and subtyping rules of the core language. We say that a command c is *well typed* with respect to Γ if $\Gamma \vdash c : \tau \ \mathit{cmd}$, for some τ.

The typing rules are the same as those in [VSI96], except for the new rule $done_t$ which types **done** as a high command. We now briefly describe each typing rule in reference to Figure 3.3.

- *base*: Our security lattice for this work has only two points H (high) and L (low) with H higher than L.
- *cmd*: Commands types are contravariant with respect to the base types, e.g.: $H\ cmd \subseteq L\ cmd$.
- *reflex*, *trans*: The subtyping relation $\subseteq$ is reflexive and transitive.
- *subsump*: Any phrase can be retyped to a supertype. In our language, expressions and commands can be subsumed but there is no subsumption on variable types, which are fixed by Γ. This is critical for the soundness of the type system.
- $done_t$: The command **done** is a H command.
- int_t: Integer values are typed L.
- $rval_t$: The value of a variable acquires the security level of the variable.
- $update_t$: In a valid update command, the security type of the expression must be the same as (or be subsumable to) the type of the receiving variable. Then, the resulting command must have at least that type, e.g.: let $h : H\ var$ then $h := 3 : H\ cmd$. As a second example: let $l : L\ var$ then $l := h$ is not well typed because $L\ var$ cannot be subsumed to $H\ var$. Again, note that H commands are L commands because if the command only assigns to H variables or higher then it also assigns only to L variables or higher[1].
- $plus_t$: The type of a sum must equal the type of both operands.

[1] As another example, billionaires are millionaires.

- *if*$_t$: In a valid *if* command the type of the guard must be the same as the level of both subcommands.
- *while*$_t$: In a valid *while* command the type of the guard must be the same as the level of the body.
- *compose*$_t$: In a valid *sequential composition* the type of the command must be the same as the type of the subcommands.

Note that under our well-formedness assumptions some typing rules (like *plus*$_t$) do not reject phrases as one can always find a type for any sum, while others (like *if*$_t$) do. This would be the case in general provided that the security classes form a *lattice* and that Γ is "big enough". We now argue some standard soundness properties for our language and type system. The following example illustrates some of these issues:

Example 3.2.1 (Typing Derivations) *Consider the command*

$$\mathbf{c} = \ \textbf{if}\, h\, \textbf{then}\, h_1 := 1\, \textbf{else}\, h_1 := 0$$

where variables that start with h are typed H var, and those who start with l are typed L var. We should be able to type c, H cmd, or L cmd depending on our chosen typing derivation as follows:

$$\dfrac{\dfrac{\Gamma(h) = H\ var}{\Gamma \vdash h : H}\,(rval_t) \qquad \dfrac{\begin{array}{l}\dfrac{\begin{array}{l}\dfrac{}{\Gamma \vdash 1 : L}\,(int_t)\\ L \subseteq H\end{array}}{\Gamma \vdash 1 : H}\,(subsump)\\ \Gamma(h_1) = H\ var\end{array}}{\Gamma \vdash h_1 := 1 : H\ cmd}\,(update_t) \qquad \dfrac{\begin{array}{l}\dfrac{\begin{array}{l}\dfrac{}{\Gamma \vdash 0 : L}\,(int_t)\\ L \subseteq H\end{array}}{\Gamma \vdash 0 : H}\,(subsump)\\ \Gamma(h_1) = H\ var\end{array}}{\Gamma \vdash h_1 := 0 : H\ cmd}\,(update_t)}{\Gamma \vdash \textbf{if}\ h\ \textbf{then}\ h_1 := 1\ \textbf{else}\ h_1 := 0 : H\ cmd}\,(if_t)$$

Here, we type c : *H cmd. But we can further type c* : *L cmd, by subsump. On the other hand consider the command*

$$\mathbf{c} = \ \mathbf{if}\, l\, \mathbf{then}\, l_1 := 1\, \mathbf{else}\, h_1 := 0$$

Using rule if_t, we cannot *type c* : *H cmd because although we can always subsume $l : H$, the command $l_1 := 1$ cannot be typed H cmd. However going the other way, we can subsume $h_1 := 1 : H$ cmd to L cmd, and type the complete command L cmd. Here is a typing derivation:*

$$\dfrac{\dfrac{\Gamma(l) = L\ var}{\Gamma \vdash l : L}\,(rval_t) \qquad \dfrac{\begin{array}{l}\dfrac{}{\Gamma \vdash 1 : L}\,(int_t)\\ \Gamma(l_1) = L\ var\end{array}}{\Gamma \vdash l_1 := 1 : L\ cmd}\,(update_t) \qquad \dfrac{\begin{array}{l}\dfrac{\begin{array}{l}\dfrac{\begin{array}{l}\dfrac{}{\Gamma \vdash 0 : L}\,(int_t)\\ L \subseteq H\end{array}}{\Gamma \vdash 0 : H}\,(subsump)\\ \Gamma(h_1) = H\ var\end{array}}{\Gamma \vdash h_1 := 0 : H\ cmd}\,(update_t)\\ H\ cmd \subseteq L\ cmd\end{array}}{\Gamma \vdash h_1 := 0 : L\ cmd}\,(subsump)}{\Gamma \vdash \mathbf{if}\ l\ \mathbf{then}\ l_1 := 1\ \mathbf{else}\ h_1 := 0 : L\ cmd}\,(if_t)$$

Next, consider the command

$$\mathbf{c} = \ \mathbf{while}\, l\, \mathbf{do}\, h_1 := h_1 + 1$$

Again, this command can be typed H cmd or L cmd depending on the typing derivation. The following typing derivation types the while command H cmd:

$$\dfrac{\dfrac{\dfrac{\Gamma(l) = L\ \mathit{var}}{\Gamma \vdash l : L}\,(\mathit{rval}_t) \quad L \subseteq H}{\Gamma \vdash l : H}\,(\mathit{subsump}) \qquad \dfrac{\Gamma(h_1) = H\ \mathit{var} \quad \dfrac{\dfrac{\Gamma(h_1) = H\ \mathit{var}}{\Gamma \vdash h_1 : H}\,(\mathit{rval}_t) \quad \dfrac{\dfrac{}{\Gamma \vdash 1 : L}\,(\mathit{int}_t) \quad L \subseteq H}{\Gamma \vdash 1 : H}\,(\mathit{subsump})}{\Gamma \vdash h_1 + 1 : H}\,(\mathit{plus}_t)}{\Gamma \vdash h_1 := h_1 + 1 : H\ \mathit{cmd}}\,(\mathit{update}_t)}{\Gamma \vdash \textbf{while}\ l\ \textbf{do}\ h_1 := h_1 + 1 : H\ \mathit{cmd}}\,(\mathit{while}_t)$$

Then, we could, by adding an extra subsump step, type $c : L\ cmd$. Or alternatively we could reach $c : L\ cmd$ by leaving $l : L$ and subsuming $h_1 := h_1 + 1$ to $L\ cmd$.

Now, if we are given some command d with $\Gamma \vdash d : L\ cmd$, we only know that there is some typing derivation that types $d : L\ cmd$. There may be other derivations that reach other types, including any supertype. So given any typing judgement, we cannot tell what typing rule was used last so the analysis of the typing derivation is encumbered by a proliferation of cases that must be addressed. To address this, we shall follow [VS99] with the following key assumption about typing derivations:

Assumption 3.2.2 (Normal-Form Typing Derivation) *Given any typing derivation in our type system we assume, without loss of generality, that it ends with a single (perhaps trivial) application of rule subsump.*

There is no loss of generality because any sequence of subsump rules can be merged using *trans*, and in case where there is no *subsump* rule we can use the reflexive property of subsumption to insert a trivial application of rule *subsump*. We now proceed with basic properties of the type system.

Lemma 3.2.1 (Simple Security) *If* $\Gamma \vdash e : \tau$*, then* e *contains only variables of level* τ *or lower.*

Proof. By induction on the structure of e.

- If e is of the form x: by assumption 3.2.2, the last two steps of the typing derivation must look like the following:

  $$\begin{array}{ll} \Gamma(x) = \tau' \ var & \\ \cline{1-1} \Gamma \vdash x : \tau' \quad \tau' \subseteq \tau & (rval_t) \\ \cline{1-1} \Gamma \vdash x : \tau & (subsump) \end{array}$$

  Hence, $\Gamma(x) = \tau \ var$ or lower.

- If e is of the form n: e vacuously contains only variables of level τ or lower. (since e doesn't contain any variables).

- If e is of the form $e_1 + e_2$: by assumption 3.2.2, the last two steps of the typing derivation must look like:

  $$\begin{array}{ll} \Gamma \vdash e_1 : \tau' & \\ \Gamma \vdash e_2 : \tau' & \\ \cline{1-1} \Gamma \vdash e_1 + e_2 : \tau' \quad \tau' \subseteq \tau & (plus_t) \\ \cline{1-1} \Gamma \vdash e_1 + e_2 : \tau & (subsump) \end{array}$$

  Hence, $\Gamma \vdash e_1 : \tau'$ and $\Gamma \vdash e_2 : \tau'$. Then by induction, e_1 contains only variables of level τ' or lower and again by induction, e_2 contains only variables of level τ' or lower. But since $\tau' \subseteq \tau$, then e_1 and e_2 contain only variables of level τ or lower.

□

Lemma 3.2.2 (Confinement) *If $\Gamma \vdash c : \tau\ cmd$, then c assigns only to variables of level τ or higher.*

Proof. By induction on the structure of c.

- If c is of the form **done**: then it vacuously assigns to variables of level τ or higher (since c doesn't assign to any variables).

- If c is of the form $x := e$: by assumption 3.2.2, the last two steps of the typing derivation must look like:

$$\dfrac{\dfrac{\begin{array}{l}\Gamma(x) = \tau'\ var \\ \Gamma \vdash e : \tau'\end{array}}{\Gamma \vdash x := e : \tau'\ cmd}\ (update_t) \qquad \tau'\ cmd \subseteq \tau\ cmd}{\Gamma \vdash x := e : \tau\ cmd}\ (subsump)$$

Hence, $\Gamma(x) = \tau'\ var$ with $\tau \subseteq \tau'$, i.e., τ' is at least as high as τ. So in this case, c assigns only to variables of level τ or higher.

- If c is of the form **if** e **then** c_1 **else** c_2: by assumption 3.2.2, the last two steps of the typing derivation must look like:

$$\dfrac{\begin{array}{l}\Gamma \vdash e : \tau' \\ \Gamma \vdash c_1 : \tau'\ cmd \\ \Gamma \vdash c_2 : \tau'\ cmd\end{array}}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau'\ cmd}\ (if_t)$$

$$\dfrac{\tau'\ cmd \subseteq \tau\ cmd}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd}\ (subsump)$$

Hence, $\Gamma \vdash c_1 : \tau' \ cmd$ and $\Gamma \vdash c_2 : \tau' \ cmd$, where $\tau \subseteq \tau'$. Then by two applications of induction, c_1 and c_2 assigns only to variables of level τ' or higher. So c_1 and c_2 assign only to variables of level τ or higher.

- If c is of the form **while** e **do** c_1: the argument is similar to the **if** case.
- If c is of the form $c_1; c_2$: by assumption 3.2.2, the last two steps of the typing derivation must look like:

$$\cfrac{\cfrac{\Gamma \vdash c_1 : \tau' \ cmd \qquad \Gamma \vdash c_2 : \tau' \ cmd}{\Gamma \vdash c_1; c_2 : \tau' \ cmd}\ (compose_t) \qquad \tau' \ cmd \subseteq \tau \ cmd}{\Gamma \vdash c_1; c_2 : \tau \ cmd}\ (subsump)$$

Hence, $\Gamma \vdash c_1 : \tau' \ cmd$ and $\Gamma \vdash c_2 : \tau' \ cmd$. Then by two applications of induction, c_1 and c_2 assigns only to variables of level τ' or higher. So c_1 and c_2 assigns only to variables of level τ or higher.

□

Lemma 3.2.3 (Subject Reduction) *If* $\Gamma \vdash c : \tau \ cmd$ *and* $(c, \mu) \longrightarrow (c', \mu')$, *then* $\Gamma \vdash c' : \tau \ cmd$.

Proof. By induction on the structure of c.

- If c is of the form **done**: then by rule $done_s$, $(c, \mu) \longrightarrow (c, \mu)$, so $c' = c$, hence $c' : \tau \ cmd$.
- If c is of the form $x := e$: then by rule $update_s$, $(c, \mu) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)])$ with $\Gamma \vdash c' : H \ cmd$ (by rule $done_t$). Then by rule *subsump*, $\Gamma \vdash c' : \tau \ cmd$ for any $\tau \ cmd$.

- If c is of the form **if** e **then** c_1 **else** c_2: by assumption 3.2.2, the last two steps of the typing derivation must look like:

  $$\cfrac{\cfrac{\begin{array}{l}\Gamma \vdash e : \tau' \\ \Gamma \vdash c_1 : \tau' \ cmd \\ \Gamma \vdash c_2 : \tau' \ cmd\end{array}}{\Gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \tau' \ cmd}\ (if_t) \qquad \tau' \ cmd \subseteq \tau \ cmd}{\Gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 : \tau \ cmd}\ (subsump)$$

  Hence, $\Gamma \vdash c_1 : \tau' \ cmd$ and $\Gamma \vdash c_2 : \tau' \ cmd$ where $\tau' \ cmd \subseteq \tau \ cmd$. If transition is by first rule if_s then $(c, \mu) \longrightarrow (c_1, \mu)$, if transition is by second rule if_s then $(c, \mu) \longrightarrow (c_2, \mu)$. In either case by rule *subsump*, $\Gamma \vdash c' : \tau \ cmd$.

- If c is of the form **while** e **do** c_1: by assumption 3.2.2, the last two steps of the typing derivation must look like:

  $$\cfrac{\cfrac{\begin{array}{l}\Gamma \vdash e : \tau' \\ \Gamma \vdash c_1 : \tau' \ cmd\end{array}}{\Gamma \vdash \textbf{while } e \textbf{ do } c_1 : \tau' \ cmd}\ (while_t) \qquad \tau' \ cmd \subseteq \tau \ cmd}{\Gamma \vdash \textbf{while } e \textbf{ do } c_1 : \tau \ cmd}\ (subsump)$$

  Hence, $\Gamma \vdash c_1 : \tau' \ cmd$ where $\tau' \ cmd \subseteq \tau \ cmd$. If transition is by first rule $while_s$, then $(c, \mu) \longrightarrow (\textbf{done}, \mu)$. If transition is by second rule $while_s$, then $(c, \mu) \longrightarrow (c_1; c, \mu)$ and since both c_1 and c are typed $\tau' \ cmd$, then by rule $compose_t$, $\Gamma \vdash c_1; c : \tau' \ cmd$. In either case by rule *subsump*, $\Gamma \vdash c' : \tau \ cmd$.

- If c is of the form $c_1; c_2$: by assumption 3.2.2, the last two steps of the typing derivation must look like:

$$\frac{\begin{array}{l}\Gamma \vdash c_1 : \tau' \ cmd \\ \Gamma \vdash c_2 : \tau' \ cmd\end{array}}{\Gamma \vdash c_1; c_2 : \tau' \ cmd} \quad (compose_t)$$

$$\frac{\tau' \ cmd \subseteq \tau \ cmd}{\Gamma \vdash c_1; c_2 : \tau \ cmd} \quad (subsump)$$

Hence, $\Gamma \vdash c_1 : \tau' \ cmd$ and $\Gamma \vdash c_2 : \tau' \ cmd$ where $\tau' \ cmd \subseteq \tau \ cmd$. If transition is by first rule $compose_s$, then $(c, \mu) \longrightarrow (c_2, \mu')$ with $\Gamma \vdash c_2 : \tau' \ cmd$. If transition is by second rule $compose_s$, then $(c, \mu) \longrightarrow (c_1'; c_2, \mu')$. Then by induction $\Gamma \vdash c_1' : \tau' \ cmd$ and since we already established that $\Gamma \vdash c_2 : \tau' \ cmd$, then by rule $compose_t$, $\Gamma \vdash c_1'; c_2 : \tau' \ cmd$. In either case by rule *subsump*, $\Gamma \vdash c' : \tau \ cmd$.

□

In this section we have established the basic properties of our type system. We now pause for some background theory that will be needed before proceeding to more advanced properties of our type system.

3.3 Fast Simulation for Transition Systems

In this section we define *fast simulation* in the general setting of nondeterministic transition systems; later we will apply these concepts to specific languages and type systems but for now it is all in the abstract. This work is based on the *fast simulation* section from [SA07], which in turn was based on the concepts of strong and weak simulation for discrete time Markov chains by Christel Baier, Joost-Pieter Katoen, Holger Hermanns, and Verena Wolf [BKHW05]. This work is particularly suitable for us because it is based on local analysis (at the level of a transition

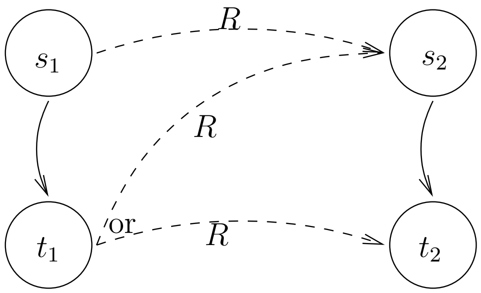


Figure 3.4: Graphical representation of fast simulation

step) yet it scales to the behavior of the entire system. We seek a definition for a nondeterministic setting, although later we will also extend it to a probabilistic setting, but first we define a transition system as follows:

Definition 3.3.1 (Transition System) *A transition system is a pair* $(S, \longrightarrow)$ *where* S *is a set of states and* $\longrightarrow$ *is a transition relation:* $(\longrightarrow\ \subseteq S \times S)$*.* S *may include terminal states. A* terminal state $t \in S$ *is defined as a state such that* $(\{t\} \times S) \cap \longrightarrow is \{(t,t)\}$ *or* $\emptyset$*, i.e.,* t *is a state without outgoing transitions or with a single transition to itself.*

In this setting we define *fast simulation* as a binary relation R such that if $s_1, s_2 \in S$ and $s_1 R\, s_2$, then any transition from s_1 can be "matched" by s_2 in zero or one steps. Formally,

Definition 3.3.2 (Fast Simulation) *Let* $(S, \longrightarrow)$ *be a transition system. A binary relation* R *on* S *is a* fast simulation *if whenever* $s_1 R s_2$*, for any state* t_1 *such that* $s_1 \longrightarrow t_1$ *either* $t_1 R s_2$ *or there is* t_2 *such that* $s_2 \longrightarrow t_2$ *and* $t_1 R t_2$*.*

Figure 3.4 illustrates the definition of *fast simulation*. In it, the dotted lines denote a fast simulation relation R, while the solid lines denote a transition relation $\longrightarrow$. A transition from state s_1 to t_1 is simulated either by s_2 in place, or else by t_2, where t_2 is a state reachable from s_2.

Example 3.3.1 (Simulation Relations) *To illustrate this concept, consider transition system $(S, \longrightarrow)$. Let us see if some of the common relations on S are fast simulations. Specifically we test the* identity, universal *and* empty *relations on S:*

- *Let R be the identity relation on S. Then, R is a* fast simulation *because if $s_1 R\, s_2$ then $s_1 = s_2$. Hence, for any state s_1 may transition to: $s_1 \longrightarrow t_1$; s_2 can do the same: $s_2 \longrightarrow t_1$ with $t_1 R\, t_1$.*
- *Let R be the universal relation on S. Then, R is a* fast simulation *because if $s_1 R s_2$ and $s_1 \longrightarrow t_1$, then $t_1 R s_2$.*
- *Let R be the empty relation on S. Then, R is a* fast simulation *vacuously.*

Next we argue the key property of fast simulation; essentially, that from the simulating state we can reach terminal states faster or as fast as from the simulated state, but first we have a simple lemma that extends *simulation* to multiple steps.

Lemma 3.3.3 (Simulation of Multiple Transitions) *Let R be a fast simulation and let $s_1 R\, s_2$. If s_1 can reach t_1 in k steps then there is a t_2 that s_2 can reach in k' steps, $k' \leq k$, such that $t_1 R\, t_2$.*

Proof. By induction on k.

1. Base step: $k = 0$, then $t_1 = s_1$ and $t_2 = s_2$ so $k' = 0 \leq k$.
2. $k = 1$, then by Definition 3.3.2, either $t_1 R\, s_2$ in which case $t_2 = s_2$ and $k' = 0$ or $s_2 \longrightarrow t_2$ for some t_2 with $t_1 R\, t_2$, in which case $k' = 1$. In either case $k' \leq k$.
3. Inductive step: $k = n + 1$ for some $n \geq 0$. We argue that if s_1 reaches t_1 in k steps then s_2 reaches t_2 in k' steps, with $t_1 R\, t_2$ and $k' \leq k$. Let t_1' be the state prior to reaching t_1, i.e., the n^{th} step from s_1. By induction, s_2 can reach some state t_2' in n' steps, with $t_1' R\, t_2'$ and $n' \leq n$. Then since we have that $t_1' \longrightarrow t_1$,

by Definition 3.3.2 either $t_1 R\, t_2'$ (in 0 transitions) or $t_2' \longrightarrow t_2$ (in 1 transition) for some t_2 with $t_1 R\, t_2$. We conclude that since $k = n + 1$, $n' \leq n$, and the last transition is simulated in 0 or 1 steps, it follows that $k' \leq k$.

$\square$

Definition 3.3.4 (Upwards Closed Set) *Let R be a binary relation on S. A set T of states is* upwards closed with respect to R *if, whenever $s \in T$ and $s R\, s'$, we also have $s' \in T$.*

Theorem 3.3.5 (Reachability) *Let R be a fast simulation, let T be an upwards closed set with respect to R, and let $s_1 R\, s_2$. If s_1 can reach T in k steps then s_2 can reach T in k' steps with $k' \leq k$.*

Proof. Let $t_1 \in T$ be the state that is reached in k steps. By Lemma 3.3.3, there is simulating state t_2 such that $s_2 \longrightarrow^{k'} t_2$ with $t_1 R\, t_2$ and $k' \leq k$, which implies by Definition 3.3.4, that $t_2 \in T$. $\square$

3.4 Stripping and Fast Low Simulation

In this section we explore *stripping* and *fast low simulation* within the context of the transition systems defined by our language semantics and type system. Our goal is to extend properties established at the level of individual steps to useful security properties of complete program executions.

Thus far, bisimulation has been the most common technique for proving noninterference and indeed it is possible to prove noninterference in this language by establishing some sort of *low bisimulation* between any two runs. Nevertheless, *bisimulation* works in a limited number of models where runs can be established to be step-wise *bisimilar* and some language models do not meet this requirement. For example, multiple executions of the same program with L-equivalent initial memories in the language of [AS09] are not bisimilar; however, if an execution reached a terminal state, this state would be the same for all executions. Section 2.1 has a brief history of bisimulation.

Leaking via nontermination. In our language the values of H variables can determine whether the execution will terminate or loop forever. For example, consider the program:

$$\begin{array}{l}\textbf{while}\, h = 1\, \textbf{do done};\\ l := 0\end{array}$$

This program terminates in two steps except when h equals 1, in which case it loops. Assuming that h is H and l is L, this program satisfies the Denning restrictions and it is well typed under our typing rules, however it does not satisfy termination sensitive noninterference. Intuitively, the program is "trying" to set l to 0, but the **while** loop can prevent assignments to l from being reached; i.e., the high computation within some executions is "messing things up" and so if we were able to remove it from the program then we could model the execution of the program as far as the low variables is concerned.

So we define a "stripped" version of c, which we denote by $\lfloor c \rfloor$ for any well-typed program c. In $\lfloor c \rfloor$, all high commands are removed; more precisely, we eliminate any subcommands that make no assignment to L variables; or more formally, we remove subcommands which have a typing derivation that ends in $H\ cmd$. Thus, the stripped version of the example above would look like:

$$l := 0$$

We emphasize that stripping is for us a *thought experiment*—it is not something that we would actually use in an implementation, but rather it is a means to *understand* a program's behavior.

Definition 3.4.1 (Stripping Function) *Let c be a well-typed command. We define $\lfloor c \rfloor = \textbf{done}$ if c has type H cmd; otherwise, define $\lfloor c \rfloor$ by*

- $\lfloor x := e \rfloor = x := e$
- $\lfloor \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rfloor = \textbf{if } e \textbf{ then } \lfloor c_1 \rfloor \textbf{ else } \lfloor c_2 \rfloor$
- $\lfloor \textbf{while } e \textbf{ do } c \rfloor = \textbf{while } e \textbf{ do } \lfloor c \rfloor$
- $\lfloor c_1 ; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \textit{if } c_1 : H \ cmd \\ \lfloor c_1 \rfloor & \textit{if } c_2 : H \ cmd \\ \lfloor c_1 \rfloor ; \lfloor c_2 \rfloor & \textit{otherwise} \end{cases}$

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ, define $\lfloor \cdot \rfloor$ as the relation that is derived from the stripping function such that $c \lfloor \cdot \rfloor d$ iff $d = \lfloor c \rfloor$, and we extend $\lfloor \cdot \rfloor$ to well-typed configurations by $\lfloor (c, \mu) \rfloor = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

Note that we use *stripping* as a function and as a relation[2]. Initially, stripping was introduced in [SA07], but in this paper stripping replaced H subcommands with **skip**; in contrast our new definition here aggressively *eliminates* such subcommands. Note also that $\lfloor \mu \rfloor$ and μ agree on L variables. Later, we shall formalize this notion as *low equivalence* and denote it with the binary relation: $\sim_L$. Now we argue a basic property of the *stripping function*:

Lemma 3.4.2 *For any command c, $\lfloor c \rfloor$ contains only L variables.*

Proof. By induction on the structure of c. If $c : H \ cmd$, then $\lfloor c \rfloor = \textbf{done}$, which vacuously contains only L variables. Otherwise there is no typing derivation that leads to $c : H \ cmd$. In this case, we consider the possible forms of c.

[2] In set theory, functions are defined as certain kinds of binary relations.

- If c is of the form $x := e$, then $\lfloor c \rfloor = x := e$. By rule *update*$_t$ and *subsump* (*subsump* is trivial throughout these cases), we must have that $x : L\ var$ and $e : L$, which implies by Simple Security that e contains only L variables.

- If c is of the form **if** e **then** c_1 **else** c_2, then $\lfloor c \rfloor =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$. By rule *if*$_t$ and *subsump*, $e : L$, by Simple Security, e contains only L variables. Then by induction, $\lfloor c_1 \rfloor$ contains only L variables and again by induction, $\lfloor c_2 \rfloor$ contains only L variables.

- If c is of the form **while** e **do** c_1, then $\lfloor c \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$. By rule *while*$_t$ and *subsump* $e : L$, which implies by Simple Security that e contains only L variables. And, by induction, $\lfloor c_1 \rfloor$ contains only L variables.

- If c is of the form $c_1; c_2$, $\lfloor c \rfloor$ can be of the form: $\lfloor c_1 \rfloor; \lfloor c_2 \rfloor$, $\lfloor c_1 \rfloor$, or $\lfloor c_2 \rfloor$. Then, by induction, $\lfloor c_1 \rfloor$ contains only L variables and by induction again, $\lfloor c_2 \rfloor$ contains only L variables.

□

Now, consider the execution of a program configuration (a program with its initial memory) and its stripped version. Using the results thus far, we can argue that the execution of the stripped configuration is absent of all high computation. Suppose that the stripped version simulates the original configuration up to the final values of the low variables. Then it does not matter whether the original execution contains high computation or not (unless the execution is delayed forever by the high computation). Ultimately, this is all that is needed to argue a noninterference property.

3.5 Noninterference of Well-Typed Programs

In this section we argue a termination-insensitive noninterference property on well-typed programs, i.e., we want to establish that two executions of a program with different H values produce identical L results (if they both terminate). We certainly need a termination-insensitive property since, under the Denning restrictions, H variables can affect termination.

Consider two executions of a program with different high memories. By definition, stripping the two initial configurations of their high commands must yield equal *low configurations*. Now, if these *low configurations* can *low simulate*[3] the original programs, then each pair of programs (a program and its stripped version) will execute together. At the first step, the executions of the original programs may diverge depending on the language (although not the case in this core language) and so will diverge the simulating executions. Nevertheless, if the simulating programs reach the same terminal state[4], then it must be the case that the original programs must reach terminal configuration with the same low results (if they are not delayed forever by the high computation). This implies that executions that reach terminal states are not affected by high computation, i.e., a *noninterference* property.

We now formalize these ideas, starting with the equivalence of memories with respect to their *low-variables*[5].

[3] Simulate up to the final values of the low variables.

[4] Or, in a probabilistic setting, reach terminal states with equal probability.

[5] Please note that there is no longer a requirement that the memories share the same domain. This is a departure from the historical definitions of low equivalence, but it is necessary because intuitively, if we were to strip a memory of its high variables, the result should be low equivalent to the original memory. Yet this would not be the case if we maintained a domain requirement.

$$\begin{tikzcd}
(c,\mu) \arrow[rr, "R"] \arrow[d] & & (d,\nu) \arrow[d] \\
(c',\mu') \arrow[urr, "R"] \arrow[rr, "R"'] \arrow[rr, phantom, "\text{or}", shift left=2ex] & & (d',\nu')
\end{tikzcd}$$

Figure 3.5: Graphical representation of fast low simulation

Definition 3.5.1 (Low-equivalent Memories) *Two memories μ and ν are L-*equivalent*, written $\mu \sim_L \nu$, if they agree on the values of all L variables.*

Next, we apply *fast simulation* (Section 3.3) to our language setting. The new definition is called *fast low simulation* and it aims to capture the idea of simulating up to the final values of the low variables.

Definition 3.5.2 (Fast Low Simulation) *A binary relation R on well-typed configurations is a* fast low simulation *if R is a fast simulation and whenever $(c_1, \mu_1)R(c_2, \mu_2)$, we have that $\mu_1 \sim_L \mu_2$.*

Figure 3.5 illustrates the definition of *fast low simulation.* Clearly, fast low simulations are fast simulations, so a fast low simulating state will *match* the simulated state in 0 or 1 steps. But now the *matching criteria* also requires that the memories be *low equivalent*.

As we can see, this definition meets our intuitive requirements that in stepwise execution within our core language, a simulating configuration is able to match the original program as far as the low variables and in at most the same number of steps (Theorem 3.3.5). We will use this concept extensively throughout this work and extend this definition to a probabilistic setting; we will also use variants of this definition to prove *noninterference* properties on three other languages.

Using our definition of *fast low simulation* within our language and type system we can now make up relations on the set of configurations and test to see if they are fast low simulations. In general, some *fast low simulations* will be more useful than others; the identity and empty relations of Example 3.3.1 are also *fast low simulations*[6] but they are of little use. However, we have the very relation we need since our *stripping* relation fulfills the intuition of removing all but the "low computation" from a program execution. So we continue with the key theorem of this chapter about the stripping relation $\lfloor\cdot\rfloor$:

Theorem 3.5.3 $\lfloor\cdot\rfloor$ *is a fast low simulation.*

Proof. Let $(c,\mu)\lfloor\cdot\rfloor(d,\nu)$. First, we quickly argue that whenever $(c,\mu)\lfloor\cdot\rfloor(d,\nu)$, we must have that $\mu \sim_L \nu$. By Definition 3.5.2, $\nu = \lfloor\mu\rfloor$ which implies that $\mu \sim_L \nu$. Next, we must show that the configuration reachable from (c,μ) in one step can be "matched" by (d,ν), in zero or one steps.

By Definition 3.4.1, $d = \lfloor c\rfloor$ and $\nu = \lfloor\mu\rfloor$. Suppose that $(c,\mu)\longrightarrow(c',\mu')$, it suffices to show that either:

Condition 1. $(d,\nu) = (\lfloor c'\rfloor, \lfloor\mu'\rfloor)$ (*match in place*)

or

Condition 2. $(d,\nu)\longrightarrow(d',\nu')$ *such that* $(d',\nu') = (\lfloor c'\rfloor, \lfloor\mu'\rfloor)$.

We make this argument by induction on the structure of c. If $c : H\ cmd$ then $d = \textbf{done}$. By Confinement we have $\mu \sim_L \mu'$, which implies that $\nu = \lfloor\mu'\rfloor$. By Subject Reduction we have $c' : H\ cmd$, which implies that $\lfloor c'\rfloor = \textbf{done}$. Hence, the move $(c,\mu)\longrightarrow(c',\mu')$ is matched in place by (d,ν) with $(d,\nu) = (\lfloor c'\rfloor, \lfloor\mu'\rfloor)$, which

[6] Note that the universal relation is not a *fast low simulation* because if $s_1 R s_2$ then they must have low equivalent memories.

meets Condition 1. Next, if c does not have type $H\ cmd$, then consider the possible forms of c:

1. c is of the form $x := e$. Here $d = x := e$. By rule *update*$_t$ and *subsump* (*subsump* is trivial throughout these cases), $x : L\ var$ and $e : L$. By *update*$_s$, $\mu' = \mu[x := \mu(e)]$ and $\nu' = \nu[x := \nu(e)]$, and by Simple Security $\nu(e) = \mu(e)$, which implies that $\nu' = \nu[x := \nu(e)] = \lfloor \mu[x := \mu(e)] \rfloor = \lfloor \mu' \rfloor$. Hence the move $(c, \mu) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)])$ is matched by the move $(d, \nu) \longrightarrow (\mathbf{done}, \nu[x := \nu(e)])$. This meets Condition 2.

2. c is of the form **if** e **then** c_1 **else** c_2. Here $d =$ **if** e **then** d_1 **else** d_2 where $d_1 = \lfloor c_1 \rfloor$ and $d_2 = \lfloor c_2 \rfloor$. By rule *if*$_t$ and *subsump*, $e : L$ and by Simple Security $\nu(e) = \mu(e)$. So if $\mu(e) \neq 0$, then by first rule *if*$_s$, $(c, \mu) \longrightarrow (c_1, \mu)$. By the same rule, $(d, \nu) \longrightarrow (d_1, \nu)$ so that the move from (c, μ) is matched by the move from (d, ν) in one step. This meets Condition 2. If $\mu(e) = 0$, then we have a similar result.

3. c is of the form **while** e **do** c_1. Here $d =$ **while** e **do** d_1 where $d_1 = \lfloor c_1 \rfloor$. By rule *while*$_t$ and *subsump*, $e : L$ and c_1 does not have type $H\ cmd$. By Simple Security, $\nu(e) = \mu(e)$. So in case $\mu(e) \neq 0$, then the move $(c, \mu) \longrightarrow (c_1; \mathbf{while}\ e\ \mathbf{do}\ c_1, \mu)$ (by second rule *while*$_s$) is matched by $(d, \nu) \longrightarrow (d_1; \mathbf{while}\ e\ \mathbf{do}\ d_1, \nu)$ (by the same rule). This meets Condition 2. When $\mu(e) = 0$, then by two applications of first rule *while*$_s$, the move $(c, \mu) \longrightarrow (\mathbf{done}, \mu)$ is matched by the move $(d, \nu) \longrightarrow (\mathbf{done}, \nu)$. This meets Condition 2.

Next we address the case when c is of the form $c_1; c_2$. First recall that under rule *compose*$_s$, any move from $(c_1; c_2, \mu)$ results from a move $(c_1, \mu) \longrightarrow (c_1', \mu')$ where c_1' may or may not be **done**. Depending on this, the transitions from c can be:

$$(c_1; c_2, \mu) \longrightarrow (c_2, \mu')$$

or

$$(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')$$

Simulating (c, μ) we have (d, ν) where $d = \lfloor c_1; c_2 \rfloor$ and $\nu = \lfloor \mu \rfloor$. Depending on the types of c_1 and c_2, d has three possible forms which we handle separately.

1. First, if neither c_1 nor c_2 has type H *cmd* then $d = \lfloor c_1 \rfloor; \lfloor c_2 \rfloor = d_1; d_2$ where $d_1 = \lfloor c_1 \rfloor$ and $d_2 = \lfloor c_2 \rfloor$. By induction, the move $(c_1, \mu) \longrightarrow (c_1', \mu')$ can be matched by (d_1, ν) using either Condition 1 or Condition 2 above. We now address each condition in turn.

   (a) If Condition 1 is met on $(c_1, \mu) \longrightarrow (c_1', \mu')$; i.e., (d_1, ν) can match the move of (c_1, μ) in place, then $d_1 = \lfloor c_1' \rfloor$ and $\nu = \lfloor \mu' \rfloor$, i.e., $(c_1', \mu') \lfloor \cdot \rfloor (d_1, \nu)$. Now observe that d_1 cannot be **done** because only H commands strip to **done** and c_1 cannot be typed H command. This implies that c_1' cannot have type H *cmd* and in particular c_1' cannot be **done**. Hence the move from c must be by the second rule *compose*$_s$, $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')$. Then, (d, ν) matches in place with $d = d_1; d_2 = \lfloor c_1' \rfloor; \lfloor c_2 \rfloor = \lfloor c' \rfloor$. This meets Condition 1.

   (b) If Condition 2 is met on $(c_1, \mu) \longrightarrow (c_1', \mu')$; then the transition must be matched by $(d_1, \nu) \longrightarrow (d_1', \nu')$ where $(c_1', \mu') \lfloor \cdot \rfloor (d_1', \nu')$. We now consider three distinct cases for c_1':

      i. If $c_1' =$ **done** then the move from c must be by the first rule *compose*$_s$: $(c_1; c_2, \mu) \longrightarrow (c_2, \mu')$. Then, d can make a matching move using first rule *compose*$_s$: $(d_1; d_2, \nu) \longrightarrow (d_2, \nu')$ with $(c', \mu') \lfloor \cdot \rfloor (d', \nu')$. Hence, condition 2 is met.

ii. If $c_1' : H\ cmd$ but $c_1' \neq \mathbf{done}$, then the move from c must be by the second rule $compose_s$: $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')$. But, $d_1' = \mathbf{done}$, hence, the move from d must be by the first rule $compose_s$: $(d_1; d_2, \nu) \longrightarrow (d_2, \nu')$ with $(c_1'; c_2, \mu') \lfloor \cdot \rfloor (d_2, \nu')$. Hence, condition 2 is met.[7]

iii. If c_1' does not have type $H\ cmd$, then the move from c is by the second rule $compose_s$: $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')$. Recalling that by induction $d_1' = \lfloor c_1' \rfloor$ we must have that $d_1' \neq \mathbf{done}$. And so the matching move from d must be by the second rule $compose_s$: $(d_1; d_2, \nu) \longrightarrow (d_1'; d_2, \nu')$, where $(c_1'; c_2, \mu') \lfloor \cdot \rfloor (d_1'; d_2, \nu')$. Hence, condition 2 is met.

2. Second, if $c_1 : H\ cmd$ then $d = \lfloor c_1; c_2 \rfloor = \lfloor c_2 \rfloor = d_2$. If the move from c is by the first rule $compose_s$, then we must have $(c_1, \mu) \longrightarrow (\mathbf{done}, \mu')$, where by Confinement $\mu \sim_L \mu'$. So the move $(c_1; c_2, \mu) \longrightarrow (c_2, \mu')$ is matched in zero steps by (d_2, ν) where $(c_1'; c_2, \mu') \lfloor \cdot \rfloor (d_2, \nu)$. This meets Condition 1.

   If instead the move from c is by the second rule $compose_s$, then we must have $(c_1, \mu) \longrightarrow (c_1', \mu')$, where by Confinement $\mu \sim_L \mu'$, and by Subject Reduction $c_1' : H\ cmd$. Hence the move $(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')$ is again matched in zero steps by (d_2, ν), where $(c_1'; c_2, \mu') \lfloor \cdot \rfloor (d_2, \nu)$. This meets Condition 1.

3. Third, if $c_2 : H\ cmd$ then $d = \lfloor c_1; c_2 \rfloor = \lfloor c_1 \rfloor = d_1$. In this case the argument is essentially the same as in the first case (a).

□

[7] An example illustrating this situation is when c is ($\mathbf{if}$ 0 $\mathbf{then}$ $l := 1$ $\mathbf{else}$ $h := 2$); $l := 3$. This goes in one step to $h := 2$; $l := 3$, which strips to $l := 3$. In this case, $\lfloor c \rfloor =$ ($\mathbf{if}$ 0 $\mathbf{then}$ $l := 1$ $\mathbf{else}$ $\mathbf{done}$); $l := 3$, which goes in one step to $l := 3$.

Corollary 3.5.4 (Terminal Reachability) *Let T be an* upwards closed set *of configurations with respect to $\lfloor\cdot\rfloor$. If (c,μ) can reach T in k steps, then $(\lfloor c\rfloor,\lfloor\mu\rfloor)$ can reach T in k' steps with $k'\le k$.*

Proof. By Theorem 3.5.3, $\lfloor\cdot\rfloor$ is a fast low simulation, by Definition 3.5.2, $\lfloor\cdot\rfloor$ is also a fast simulation. Then by Theorem 3.3.5, $k'\le k$. □

Corollary 3.5.4 establishes that a stripped configuration will terminate as fast or faster than the original configuration. We are now able to predict the behavior of complete executions of a well-typed program c from its stripped version $\lfloor c\rfloor$. Given two such executions of c, one under memories μ and ν, and assuming that μ and ν are low equivalent, we can now predict the behavior of two such complete executions as far as the low variables are concerned. The reason is that, by Lemma 3.4.2, $\lfloor c\rfloor$ contains only L variables, which means that its behavior under μ must be *identical* to its behavior under ν. Hence we can build a "bridge" between (c,μ) and (c,ν):

$$(c,\mu)\xleftrightarrow{\text{Cr 3.5.4}}(\lfloor c\rfloor,\lfloor\mu\rfloor)\equiv(\lfloor c\rfloor,\lfloor\nu\rfloor)\xleftrightarrow{\text{Cr 3.5.4}}(c,\nu)$$

because the stripped configurations are equal. Then, supposing that (c,μ) and (c,ν) both terminate, by our reachability result above, their stripped version would also terminate (at least as fast) and by the language determinism result and by Lemma 3.3.3, the stripped configuration would reach a unique final state. Hence, all final memories would be low equivalent. And so, we have informally argued the following corollary:

Corollary 3.5.5 *Let c be a well-typed command. Let μ and ν be memories such that $\mu\sim_L\nu$. Suppose that (c,μ) and (c,ν) can both execute successfully, reaching terminal configurations $(\mathbf{done},\mu')$ and $(\mathbf{done},\nu')$ respectively. Then $\mu'\sim_L\nu'$.*

$$\begin{tikzcd}
(c,\mu) \arrow[r, "\lfloor\cdot\rfloor"] \arrow[d, "*"'] & (\lfloor c\rfloor, \lfloor\mu\rfloor) \arrow[d, "*"'] & (c,\nu) \arrow[l, "\lfloor\cdot\rfloor"'] \arrow[d, "*"'] \\
(\mathbf{done}, \mu') \arrow[r, "\lfloor\cdot\rfloor"] & (\mathbf{done}, \lfloor\mu'\rfloor) & (\mathbf{done}, \nu') \arrow[l, "\lfloor\cdot\rfloor"']
\end{tikzcd}$$

Figure 3.6: Noninterference of terminating executions

Proof. By definition 3.4.1, $(c, \mu)\lfloor\cdot\rfloor\lfloor(c, \mu)\rfloor$ and $(c, \nu)\lfloor\cdot\rfloor\lfloor(c, \nu)\rfloor$ and since $\mu \sim_L \nu$ then $\lfloor\mu\rfloor = \lfloor\nu\rfloor$. Also, since $\lfloor\cdot\rfloor$ is a fast low simulation, and since the original configurations terminate, then by Corollary 3.5.4, $\lfloor(c, \mu)\rfloor$ and $\lfloor(c, \nu)\rfloor$ must also terminate successfully in at most the same number of steps, Now, by Lemma 3.3.3, $\lfloor(\mathbf{done}, \mu')\rfloor$ is reached from $\lfloor(c, \mu)\rfloor$ and $\lfloor(\mathbf{done}, \nu')\rfloor$ is reached from $\lfloor(c, \nu)\rfloor$. Then, by Corollary 3.1.2, $\lfloor(\mathbf{done}, \mu')\rfloor = \lfloor(\mathbf{done}, \nu')\rfloor$. It follows that $\mu' \sim_L \nu'$. □

Figure 3.6 illustrates the proof idea for the *noninterference* theorem. Assuming that $\mu \sim_L \nu$ (which implies that $\lfloor\mu\rfloor = \lfloor\nu\rfloor$) and that the two executions reach terminal configurations, we must have that the stripped version reaches a terminal configuration $(\mathbf{done}, \lfloor\mu'\rfloor)$ at least as fast as the original executions; since the language is deterministic, $(\mathbf{done}, \lfloor\mu'\rfloor)$ must be unique. So the bridge is reestablished and it follows that the two final memories must be low equivalent.

In this chapter we presented our core theory based on a simple imperative language and a fairly nonrestrictive type system. The language has been modified by the introduction of **done** and the removal of **skip** which simplify its formal treatment producing shorter and more elegant proofs. The core theory is based on a new technique to prove noninterference where other techniques are ineffective. The key concept of the new technique, *fast low simulation*, allows for executions to "diverge" as long as there are common convergent states. In this chapter, in our deterministic language we did not exercise this capability, instead we demonstrated the core technique of stripping and fast low simulation in lieu of bisimulation. However, in the following chapters, we use the core techniques within nondeterministic and probabilistic settings, as well as with a language with cryptographic primitives.

CHAPTER 4

SECURE INFORMATION FLOW FOR DISTRIBUTED SYSTEMS.

In this chapter, which is a revised version of [AS09], we extend our core theory to abstract distributed systems. The execution model of our language in this case is nondeterministic. Our goal is to provide the programmer with a simple and safe abstract language, with a built-in API to handle communication between processes. The abstract language should hide all communication protocols that control the data exchange between processes and all the cryptographic operations that ensure the confidentiality of the data transmitted. The language should have mechanisms to send and receive messages; and as in the previous chapter, the language should have the ability to classify data based on security levels; the language should also have a soundness property that says that distributed system cannot leak classified information to lower classifications. We would like our language to have the same clean familiar syntax and expressability of the previous language. Intuitively, each node should execute one or more imperative programs that can communicate with the other programs on the other nodes. The programmer should not have to worry about how the sending and receiving of messages is accomplished, except that messages are sent and received by processes safely.

A distributed system thus, is a group of programs executing in a group of nodes such that there is at least one program per node. An executing program with its local data is a process. As our processes may reside in separate nodes, they should have their own private memories and be able to send and receive messages from other processes.

```
-Δ1-
Process 1
  send(a, h1)
Process 2
  receive(a, l2)
```

```
-Δ2-
Process 1
  if (h1 is even) then
    run a long time;
  send(a, 1)
Process 2
  if (h2 is odd) then
    run a long time;
  send(a, 0)
Process 3
  run a short time;
  receive(a, l3)
```

```
-Δ3-
Process 1
  if (h1 is even) then
    run a long time;
  send(a, 1)
Process 2
  run a short time;
  if (channel a has data)
    then l2 := 0
    else l2 := 1
```

Figure 4.1: Distributed Attacks

4.1 An Abstract Language for Distributed Systems

Language requirements. In crafting our language we methodologically examine the obvious design pitfalls which would surely make it unsound and adjust the type system to eliminate the vulnerabilties. As in the previous chapter, we would like to classify data according to a security lattice, which in our case will be limited to H and L, and we would like to maintain the ability to transmit and receive H and L values. This implies that the language will need separate communication channels for each classification, otherwise we would only be able to receive messages using H variables. To demonstrate this, please note adversary Δ_1 of Figure 4.1. In this attack, Process 1 sends a H variable on channel a, but Process 2 receives it into a L variable. Because of subsumption, H channels can transmit H or L data but the receiving variable must be typed H, while L channels can only transmit L data. Therefore our communication channels must have a security classification. What else do they need? Unfortunately, communication channels will also need a specific source process and a specific destination process. The reason is exemplified by distributed system Δ_2 of Figure 4.1. In this attack, Process 1 and Process 2 both send on channel a. If we assume that h_1 and h_2 are initialized to the same secret value, then the last bit of this value is leaked into l_3 (assuming sufficiently "fair" scheduling). A solution to these two attacks (Δ_1 and Δ_2) is to restrict the type of communication channels. Hence our communication channels are typed $\tau\ ch_{i,j}$ which specifies the security level (τ) and the sending (i) and receiving (j) processes.

Another limitation of distributed systems is that processes cannot be allowed to test if a channel has data. This ability would render the language unsound by allowing internal timing channels[1]. This is illustrated by distributed system Δ_3 of Figure 4.1. This attack leaks the last bit of h_1 to l_2. When h_1 is even Process 1 takes a long time to send its message so when Process 2 checks, the channel will be empty. Therefore we do not allow processes to make such tests.

The astute reader may have noticed that in a nondeterministic world we would not really have to worry about some of these attacks, these being more related to a probabilistic environment. In general nondeterministic transitions are read as "could transition to". However we have presented the attacks this way for two reasons: first, our goal is a language that would also be sound under a probabilistic implementation; second, the attacks presented are intuitive and easy to understand. Hence, Figure 4.2 shows a nondeterministic attack which (although more complex) accomplish equivalent leaks as the probabilistic attacks of Figure 4.1. Δ_4, the nondeterministic version of Δ_1 is identical, but Δ_5 is very different from Δ_2; running it results in the last bit of h being leaked to l_1. Upon execution, Process 1 sends h to Process 2 and Process 3 on respective channels and waits for a response from either process on a common public-channel. If h is odd then Process 2 will respond immediately sending the public value 1 to Process 1 while Process 3 waits. If h is even

[1] Throughout this work we call internal timing channels, logical channels that do not require a time clock to leak information.

```
-$\Delta_4$-
  same as $\Delta_1$
Process 1
  $send(a, h_1)$
Process 2
  $receive(a, l_2)$
```

```
-$\Delta_5$-
Process 1
  $send(a, h); send(b, h)$
  $receive(c, l_1)$;
  $send(a, h); send(b, h)$
Process 2
  $receive(a, h_2)$;
  if ($h_2$ is even) then
    $receive(a, h_2)$;
  $send(c, 1)$
Process 3
  $receive(b, h_3)$;
  if ($h_3$ is odd) then
    $receive(b, h_3)$;
  $send(c, 0)$
```

```
-$\Delta_6$-
Process 1
  if ($h_1$ is even) then
    $receive(c, h_1)$;
  $send(a, 1)$
Process 2
  if (channel $a$ has data)
    then $l_2 := 0$
    else $l_2 := 1$;
  $send(c, 1)$
```

Figure 4.2: Distributed Attacks (nondeterministic)

then Process 1 receives a public value 0 from Process 3. Δ_6, the nondeterministic version of Δ_3, creates a dependence between the final value of the low variable l_2 and h_1. Whenever h_1 is even $l_2 = 0$ although otherwise l_2 could be 0 or 1, nonetheless, possibilistic noninterference is broken by this attack.

Because a process trying to receive from a channel must block until a message is available, the programmer has to be careful to ensure that for each *receive*, there is a corresponding *send*. The converse, however, is not required since a process may send a message that is never received. Indeed, processes should not be required to wait on *send* and should be able to send multiple times on the same channel. To handle this, we will need an unbounded buffer for each channel, where sent messages wait to be received. We would like to restrict processes as little as possible. To this end, we explore the possibility of typing processes using only the classic Denning restrictions [DD77], which disallows an assignment $l := e$ to a L variable if either e contains H variables or if the assignment is within an **if** or **while** command whose guard contains H variables. This would be in sharp contrast to prior works in secure information

$$
\begin{array}{lll}
\textit{(phrases)} & p & ::= \; e \mid c \\
\textit{(expressions)} & e & ::= \; x \mid n \mid e_1 + e_2 \mid \ldots \\
\textit{(commands)} & c & ::= \; \textbf{done} \mid x := e \mid \\
 & & \qquad \textit{send}(a, e) \mid \\
 & & \qquad \textit{receive}(a, x) \mid \\
 & & \qquad \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & \qquad \textbf{while } e \textbf{ do } c \mid c_1 ; c_2 \\
\textit{(variables)} & x, y, z, \ldots & \\
\textit{(channel ids)} & a, b, \ldots & \\
\textit{(process ids)} & i, j, \ldots &
\end{array}
$$

Figure 4.3: Abstract language syntax

flow for concurrent programs such as [SV98, SS00, Smi06, FC08] which have required severe restrictions to prevent H variables from affecting the *ordering* of assignments to shared memory. Our language, being based on message passing rather than shared memory, is much less dependent on timing and the behavior of the scheduler. Indeed, it turns out that our distributed systems are *observationally deterministic* which means that, despite our use of a purely nondeterministic process scheduler, the final result of a program is uniquely determined by the initial memories.

Language syntax. Formally, the language syntax (Figure 4.3) is that of the simple imperative language except that processes are added to the language and they may send or receive messages from other processes. This language was first published in [AS09]. In the syntax there are three types of identifiers: variables, for which we use metavariables x, y, and z; channel identifiers, for which we use metavariables a, and b; and process identifiers, for which we use metavariables i and j; metavariable n ranges over integer literals. As in the previous chapter command **done** is the terminated command also, integers are the only values; we use 0 for false and nonzero for true[2].

[2] Again, we can now code "**if** e **then** c" by "**if** e **then** c **else done**".

Semantics of processes ($\longrightarrow$). Each process can refer to its local memory μ, which maps variables to integers, and to the global *network memory* Φ, which maps channel identifiers to lists of messages currently waiting to be received; we start execution with an empty network memory Φ_0 such that $\Phi_0(a) = [\,]$, for all a. Thus we specify the semantics of a process via judgments of the form

$$(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi').$$

We use a standard small-step semantics with the addition of rules for the *send* and *receive* commands; the rules are shown in Figure 4.4. In the rules, we write $\mu(e)$ to denote the value of expression e in memory μ. The rule for $send(a, e)$ updates the network memory by adding the value of e to the end of the list of messages waiting on channel a. The rule for $receive(a, x)$ requires that there be at least one message waiting to be received on channel a; it removes the first such message and assigns it to x.

Semantics of distributed systems ($\Longrightarrow$). We model a distributed system as a function Δ that maps process identifiers to pairs (c, μ) consisting of a command and a local memory. A *global configuration* then has the form (Δ, Φ), and rule $global_s$ (Figure 4.4) defines the purely nondeterministic behavior of the process scheduler, which at each step can select any process that is able to make a transition; $global_s$ says that if a thread can make a transition to a new configuration then the complete *distributed system* transitions to a new global configuration containing the new state of the local thread. *The Type System.* Figure 4.5 shows the type system of the abstract language; its rules use an *identifier typing* Γ that maps identifiers to types. The typing rules enforce only the Denning restrictions [DD77]; in particular notice that we allow the guards of **while** loops to be H. Channels are restricted to carrying messages of one security classification from a specific process i to a specific process j

$$(\mathit{update}_s) \quad \frac{x \in \mathit{dom}(\mu)}{(x := e, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)], \Phi)}$$

$$(\mathit{if}_s) \quad \frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu, \Phi) \longrightarrow (c_1, \mu, \Phi)}$$

$$\frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu, \Phi) \longrightarrow (c_2, \mu, \Phi)}$$

$$(\mathit{while}_s) \quad \frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu, \Phi)}$$

$$\frac{\mu_(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu, \Phi) \longrightarrow (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu, \Phi)}$$

$$(\mathit{compose}_s) \quad \frac{(c_1, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu', \Phi')}{(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')}$$

$$\frac{(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi') \quad c_1' \neq \mathbf{done}}{(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')}$$

$$(\mathit{send}_s) \quad \frac{\Phi(a) = [m_1, \cdots, m_k] \quad k \geq 0}{(\mathit{send}(a, e), \mu, \Phi) \longrightarrow (\mathbf{done}, \mu, \Phi[a := [m_1, \ldots, m_k, \mu(e)]])}$$

$$(\mathit{receive}_s) \quad \frac{\Phi(a) = [m_1, \ldots, m_k] \quad k \geq 1}{(\mathit{receive}(a, x), \mu) \longrightarrow (\mathbf{done}, \mu[x := m_1], \Phi[a = [m_2, \ldots, m_k]])}$$

$$(\mathit{done}_s) \quad (\mathbf{done}, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu, \Phi)$$

$$(\mathit{global}_s) \quad \frac{\begin{array}{l}\Delta(i) = (c, \mu) \\ (c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')\end{array}}{(\Delta, \Phi) \Longrightarrow (\Delta[i := (c', \mu')], \Phi')}$$

Figure 4.4: Abstract language semantics

and accordingly are typed $\Gamma(a) = \tau\ ch_{i,j}$ where τ is the security classification of the data that can travel in the channel, i is the source process and j is the destination process. So to enable full communications between processes i and j we need four channels with types $H\ ch_{i,j}$, $H\ ch_{j,i}$, $L\ ch_{i,j}$, and $L\ ch_{j,i}$. In a typing judgment

$$\Gamma, i \vdash c : \tau\ cmd$$

$$(\mathit{sec})\qquad \tau\ ::= H \mid L$$

$$(\mathit{phrase})\qquad \rho\ ::= \tau \mid \tau\ \mathit{var} \mid \tau\ \mathit{cmd} \mid \tau\ \mathit{ch}_{i,j}$$

$$(\mathit{base})\qquad L \subseteq H$$

$$(\mathit{cmd})\qquad \frac{\tau \subseteq \tau'}{\tau'\ \mathit{cmd} \subseteq \tau\ \mathit{cmd}}$$

$$(\mathit{reflex})\qquad \rho \subseteq \rho$$

$$(\mathit{trans})\qquad \frac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$(\mathit{subsump})\qquad \frac{\Gamma, i \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma, i \vdash p : \rho_2}$$

$$(\mathit{done}_t)\qquad \Gamma, i \vdash \mathbf{done} : H\ \mathit{cmd}$$

$$(\mathit{int}_t)\qquad \Gamma, i \vdash n : L$$

$$(\mathit{rval}_t)\qquad \frac{\Gamma(x) = \tau\ \mathit{var}}{\Gamma, i \vdash x : \tau}$$

$$(\mathit{update}_t)\qquad \frac{\Gamma(x) = \tau\ \mathit{var} \quad \Gamma, i \vdash e : \tau}{\Gamma, i \vdash x := e : \tau\ \mathit{cmd}}$$

$$(\mathit{plus}_t)\qquad \frac{\Gamma, i \vdash e_1 : \tau \quad \Gamma, i \vdash e_2 : \tau}{\Gamma, i \vdash e_1 + e_2 : \tau}$$

$$(\mathit{if}_t)\qquad \frac{\Gamma, i \vdash e : \tau \quad \Gamma, i \vdash c_1 : \tau\ \mathit{cmd} \quad \Gamma, i \vdash c_2 : \tau\ \mathit{cmd}}{\Gamma, i \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ \mathit{cmd}}$$

$$(\mathit{while}_t)\qquad \frac{\Gamma, i \vdash e : \tau \quad \Gamma, i \vdash c_1 : \tau\ \mathit{cmd}}{\Gamma, i \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ \mathit{cmd}}$$

$$(\mathit{compose}_t)\qquad \frac{\Gamma, i \vdash c_1 : \tau\ \mathit{cmd} \quad \Gamma, i \vdash c_2 : \tau\ \mathit{cmd}}{\Gamma, i \vdash c_1; c_2 : \tau\ \mathit{cmd}}$$

$$(\mathit{send}_t)\qquad \frac{\Gamma(a) = \tau\ \mathit{ch}_{i,j} \quad \Gamma, i \vdash e : \tau}{\Gamma, i \vdash \mathit{send}(a, e) : \tau\ \mathit{cmd}}$$

$$(\mathit{receive}_t)\qquad \frac{\Gamma(a) = \tau\ \mathit{ch}_{j,i} \quad \Gamma(x) = \tau\ \mathit{var}}{\Gamma, i \vdash \mathit{receive}(a, x) : \tau\ \mathit{cmd}}$$

Figure 4.5: Abstract language type system

the process identifier i specifies to which process command c belongs; this is used to enforce the rule that only process i can send on a channel with type $\tau\ ch_{i,j}$ or receive on a channel with type $\tau\ ch_{j,i}$. We therefore say that a distributed system Δ is *well-typed* if $\Delta(i) = (c, \mu)$ implies that $\Gamma, i \vdash c : \tau\ cmd$ for some τ.

Language Soundness. We now argue soundness properties for our language and type system, starting with some standard properties, whose proofs are similar to those in Chapter 3.

Lemma 4.1.1 (Simple Security) *If $\Gamma, i \vdash e : \tau$, then e contains only variables of level τ or lower.*

Proof. By induction on the structure of e. □

Lemma 4.1.2 (Confinement) *If $\Gamma, i \vdash c : \tau\ cmd$, then c assigns only to variables of level τ or higher, and sends or receives only on channels of level τ or higher.*

Proof. By induction on the structure of c. □

Lemma 4.1.3 (Subject Reduction) *If $\Gamma, i \vdash c : \tau\ cmd$ and $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$, then $\Gamma, i \vdash c' : \tau\ cmd$.*

Proof. By induction on the structure of c. □

4.2 Observational Determinism for Nondeterministic Systems

In this section we turn to more interesting properties of the language. We begin by defining *terminal* global configurations; these are simply configurations in which all processes have terminated:

Definition 4.2.1 *A global configuration* (Δ, Φ) *is* terminal *if for all* i*,* $\Delta(i) = (\mathbf{done}, \mu_i)$ *for some* μ_i*.*

Notice that we do not require that Φ be an empty network memory—we allow it to contain unread messages.

Next we argue that, in spite of the nondeterminism of rule $global_s$, our distributed programs are *observationally deterministic* [ZM03], in the sense that executing any distributed system multiple times can reach at most one terminal configuration.

Theorem 4.2.2 (Observational Determinism) *Suppose that* Δ *is well-typed and that* $(\Delta, \Phi) \Longrightarrow^* (\Delta_1, \Phi_1)$ *and* $(\Delta, \Phi) \Longrightarrow^* (\Delta_2, \Phi_2)$*, where* (Δ_1, Φ_1) *and* (Δ_2, Φ_2) *are terminal configurations. Then* $(\Delta_1, \Phi_1) = (\Delta_2, \Phi_2)$*.*

Proof. We begin by observing that the behavior of each process i is completely independent of the rest of the distributed system, with the sole exception of its *receive* commands. Thus if we specify the sequence of messages $[m_1, m_2, \ldots, m_n]$ that process i receives during its execution, then process i's behavior is completely determined. (Notice that the sequence $[m_1, m_2, \ldots, m_n]$ merges together all of the messages that process i receives on any of its input channels.)

We now argue by contradiction. Suppose that we can run from (Δ, Φ) to two different terminal configurations, (Δ_1, Φ_1) and (Δ_2, Φ_2). By the discussion above, it must be that some process receives a different sequence of messages in the two runs. So consider the *first* place in the second run $(\Delta, \Phi) \Longrightarrow^* (\Delta_2, \Phi_2)$ where a process i receives a different message than it does in the first run $(\Delta, \Phi) \Longrightarrow^* (\Delta_1, \Phi_1)$. But for this to happen, there must be another process j that earlier *sent* a different message to i than it does in the first run. (Note that this depends on the fact that, in a well-typed distributed system, any channel can be sent to by just one process and received from by just one process). But for j to send a different message than in the first run, it must itself have received a different message earlier. This contradicts the fact that we chose the *first* place in the second run where a different message was received. □

Next we argue that well-typed distributed systems satisfy a termination-insensitive noninterference property. (We certainly need a termination-insensitive property since, under the Denning restrictions, H variables can affect termination.)

Definition 4.2.3 *Two memories μ and ν are* L-equivalent*, written $\mu \sim_L \nu$, if they agree on the values of all L variables. Similarly, two network memories Φ and Φ' are L-equivalent, also written $\Phi \sim_L \Phi'$, if they agree on the values of all L channels.*

4.3 Noninterference of Abstract Distributed Systems

Now we wish to argue that if we run a distributed system twice, using L-equivalent initial memories for each process, then, assuming that both runs reach terminal configurations, the final memories must be L-equivalent for each corresponding pair of processes. Historically, a standard way to prove such a result is by establishing some sort of *low bisimulation* between the two runs. However as we have seen, this does not seem to be possible for our abstract language, because changing the values of H variables can affect when *receive* commands are able to be executed. We now present an example that illustrates the difficulty:

Example 4.3.1 *Consider Figure 4.6 where h_i, l_i are local variables of $Process - i$. Suppose we run this program twice, using two L-equivalent memories for Process 1, namely $[h_1 = 1, l_1 = 0]$ and $[h_1 = 0, l_1 = 0]$, and the same memory for Process 2, $[h_2 = 0, l_2 = 0]$. Under the first memory, Process 1 may immediately send on channel $a_{H,1,2}$, which then unblocks Process 2 to do its receive and allows the assignment to*

```
Process 1
    if ($h_1$) then
        $send(a_{H,1,2}, 1)$
    else
        done
    $l_1 := 2$;
    $send(a_{H,1,2}, 2)$

Process 2
    $receive(a_{H,1,2}, h_2)$;
    $l_2 := 3$
```

Figure 4.6: A difficult example for low bisimulation

```
Process 1
    $l_1 := 2$;

Process 2
    $l_2 := 3$
```

Figure 4.7: Stripped version of Figure 4.6

l_2 to be done before l_1. But under the second memory, Process 1 does not send on channel $a_{H,1,2}$ until after assigning to l_1, which means that the assignment to l_2 must come after *the assignment to l_1. Thus the two runs are not low bisimilar.*

Note that the two runs are fine with respect to noninterference, however—in both cases we end up with $l_1 = 2$ and $l_2 = 3$. Because of this difficulty, we use a different approach to noninterference, via the concepts of *stripping* and *fast simulation*. These concepts were first developed in [SA07] and are used in Chapter 3 for the soundness analysis of our core language. Intuitively, the processes in Figure 4.6 contain H commands that are irrelevant to the L variables, except that they can cause *delays*. If we strip them out, we are left with Figure 4.7 which shows what will happen to the L variables if the system terminates. We therefore introduce a *stripping* operation that eliminates all subcommands of type H *cmd*, so that the delays that such subcommands might have caused are eliminated. More precisely, we have the following definition:

Definition 4.3.1 *Let c be a well-typed command. We define $\lfloor c \rfloor =$* **done** *if c has type H cmd; otherwise, define $\lfloor c \rfloor$ by*

- $\lfloor x := e \rfloor = x := e$
- $\lfloor \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rfloor = \textbf{if } e \textbf{ then } \lfloor c_1 \rfloor \textbf{ else } \lfloor c_2 \rfloor$
- $\lfloor \textbf{while } e \textbf{ do } c \rfloor = \textbf{while } e \textbf{ do } \lfloor c \rfloor$
- $\lfloor \mathit{send}(a, e) \rfloor = \mathit{send}(a, e)$
- $\lfloor \mathit{receive}(a, x) \rfloor = \mathit{receive}(a, x)$
- $\lfloor c_1; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \mathit{if}\ c_1 : H\ \mathit{cmd} \\ \lfloor c_1 \rfloor & \mathit{if}\ c_2 : H\ \mathit{cmd} \\ \lfloor c_1 \rfloor ; \lfloor c_2 \rfloor & \mathit{otherwise} \end{cases}$

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ, and $\lfloor \Phi \rfloor$ to be the result of deleting all H channels from Φ. We extend $\lfloor \cdot \rfloor$ to well-typed global configurations by $\lfloor (\Delta, \Phi) \rfloor = (\lfloor \Delta \rfloor, \lfloor \Phi \rfloor)$, where if $\Delta(i) = (c, \mu)$, then $\lfloor \Delta \rfloor (i) = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

We remark that stripping as defined in [SA07] replaces subcommands of type H *cmd* with **skip**; in contrast our new definition here aggressively *eliminates* such subcommands. Also, stripping in [AS09] does not extend to memories and channels, note that $\lfloor \mu \rfloor \sim_L \mu$ and $\lfloor \Phi \rfloor \sim_L \Phi$.

Next, we would like to know that the stripping function is effective in that when we strip a command all the high computation goes away and hence only L variables and channels remain in c to affect computation. So we have a simple lemma:

Lemma 4.3.2 *For any c, $\lfloor c \rfloor$ contains only L variables and channels.*

Proof. By induction on the structure of c. If c has type $H\ cmd$, then $\lfloor c \rfloor = \textbf{done}$, which (vacuously) contains only L variables and channels. If c does not have type $H\ cmd$, then consider the possible forms of c.

- If c is of the form $x := e$, then $\lfloor c \rfloor = x := e$. Since c does not have type $H\ cmd$[3], then by rule $update_t$ we must have that x is a L variable and $e : L$, which implies by Simple Security that e contains only L variables.
- If c is of the form $send(a, e)$, then $\lfloor c \rfloor = send(a, e)$. Since c does not have type $H\ cmd$, then by rule $send_t$, a is a L channel and $e : L$, which implies by Simple Security that e contains only L variables.
- If c is of the form $receive(a, x)$, then $\lfloor c \rfloor = receive(a, x)$. Since c does not have type $H\ cmd$, then by rule $receive_t$, a is a L channel and x is a L variable.
- If c is of the form $\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2$, then $\lfloor c \rfloor = \textbf{if}\ e\ \textbf{then}\ \lfloor c_1 \rfloor\ \textbf{else}\ \lfloor c_2 \rfloor$. By rule if_t, $e : L$, which implies by Simple Security that e contains only L variables. And, by two applications of induction, $\lfloor c_1 \rfloor$ and $\lfloor c_2 \rfloor$ contain only L variables and channels.
- If c is of the form $\textbf{while}\ e\ \textbf{do}\ c_1$, then $\lfloor c \rfloor = \textbf{while}\ e\ \textbf{do}\ \lfloor c_1 \rfloor$. By rule $while_t$, $e : L$, which implies by Simple Security that e contains only L variables. And, by induction, $\lfloor c_1 \rfloor$ contains only L variables and channels.

□

Key result. The key result that we wish to establish in this Chapter is that $\lfloor(\Delta, \Phi)\rfloor$ can *simulate* (Δ, Φ), up to the final values of L variables. To this end

[3] Meaning that no typing derivation in c ends with $c : H\ cmd$.

$$\begin{tikzcd}
(\Delta, \Phi) \arrow[r, "R"] \arrow[d, Rightarrow] & (\Upsilon, \Psi) \arrow[d, Rightarrow] \\
(\Delta', \Phi') \arrow[ur, "R"] \arrow[r, "R"'] \arrow[r, phantom, "\text{or}", shift left=2] & (\Upsilon', \Psi')
\end{tikzcd}$$

Figure 4.8: Graphical representation of fast low Simulation

it suffices to extend the core theory of Chapter 3 to our language. Previously we accomplished this result in [AS09] but the language and the theory were different, making for a more complex theory and proofs. Originally, *fast simulation* was introduced in [SA07] for a probabilistic setting and derived from the work of Baier, Katoen, Hermanns, and Wolf on strong and weak simulation [BKHW05]. Now, recalling the definition of fast simulation: Definition 3.3.2 from Chapter 3, we define *fast low simulation* under our nondeterministic environment.

Definition 4.3.3 (Fast Simulation) *A binary relation R on global configurations is a* fast low simulation *if whenever $(\Delta, \Phi) R \, (\Upsilon, \Psi)$ then R is a fast simulation and (Δ, Φ) and (Υ, Ψ) agree on the values of the L variables and channels.*

Figure 4.8 illustrates the definition of *fast low simulation*. Viewing our stripping function $\lfloor \cdot \rfloor$ as a *relation*, we write $(\Delta, \Phi) \ \lfloor \cdot \rfloor \ (\Upsilon, \Psi)$ if $\lfloor (\Delta, \Phi) \rfloor = (\Upsilon, \Psi)$. We are now ready for the key theorem of this chapter which states that a stripped global configuration is able to simulate the original one as far as the "low computation".

Theorem 4.3.4 $\lfloor \cdot \rfloor$ *is a fast low simulation.*

Proof. Let $(\Delta, \Phi) \lfloor \cdot \rfloor (\Upsilon, \Psi)$ First, it is immediate from the definition of $\lfloor \cdot \rfloor$ that (Δ, Φ) and (Υ, Ψ) agree on the values of L variables and channels.

Next we must show that any move from (Δ, Φ) can be matched by (Υ, Ψ) in zero or one steps. Suppose that the move from (Δ, Φ) involves a step on process i.

Then we must have $\Delta(i) = (c, \mu)$, $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$, and $\Delta' = \Delta[i := (c', \mu')]$. To show that (Υ, Ψ) can match this move in zero or one steps (note that $(\Upsilon, \Psi) = (\lfloor \Delta \rfloor, \lfloor \Phi \rfloor)$ and that $\Upsilon(i) = (d, \nu) = (\lfloor c \rfloor, \lfloor \mu \rfloor))$ then, it suffices to show that either

Condition 1. $(d, \nu, \Psi) = (\lfloor c' \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$ *(match in place)*

or

Condition 2. $(d, \nu, \Psi) \longrightarrow (d', \nu', \Psi')$ *such that* $(d', \nu', \Psi') = (\lfloor c' \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$.

We argue this by induction on the structure of c. If c has type H cmd, then $d = \mathbf{done}$. Also, by Confinement we have $\mu \sim_L \mu'$ and $\Phi \sim_L \Phi'$, which implies that $\nu = \lfloor \mu' \rfloor$, and $\Psi = \lfloor \Phi' \rfloor$. And by Subject Reduction we have $c' : H\ cmd$, which implies that $\lfloor c' \rfloor = \mathbf{done}$. So the move $(c, \mu, \Phi) \longrightarrow (c', \mu', \Phi')$ is matched in zero steps by $(\mathbf{done}, \nu, \Psi)$ (note that $d = \mathbf{done}$). This meets Condition 1. If c does not have type H cmd, then consider the possible forms of c:

1. c is of the form $x := e$. Here $d = x := e$. By $update_t$, $x : L\ var$ and $e : L$. By $update_s$, $\mu' = \mu[x := \mu(e)]$ and $\nu' = \nu[x := \nu(e)]$, and by Simple Security $\nu(e) = \mu(e)$, which implies that $\nu' = \nu[x := \nu(e)] = \lfloor \mu[x := \mu(e)] \rfloor = \lfloor \mu' \rfloor$. Hence the move $(c, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu[x := \mu(e)], \Phi)$ is matched by the move $(d, \nu, \Psi) \longrightarrow (\mathbf{done}, \nu[x := \nu(e)], \Psi)$. This meets Condition 2.

2. c is of the form $send(a, e)$. Here $d = send(a, e)$. By $send_t$, a is a low channel and $e : L$. By $send_s$, $\Phi'(a) = [m_1, \ldots, m_k, \mu(e)]$ for some k and $\Psi'(a) = [m_1, \ldots, m_k, \nu(e)]$. Also, by Simple Security, $\nu(e) = \mu(e)$. Hence the move $(c, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu, \Phi[a := [m_1, \ldots, m_k, \mu(e)])$ is matched by the move $(d, \nu, \Psi) \longrightarrow (\mathbf{done}, \nu, \Psi[a := [m_1, \ldots, m_k, \nu(e)])$. This meets Condition 2.

3. c is of the form $receive(a, x)$. Here $d = receive(a, x)$ By $receive_t$, a is a low channel and $x : L\ var$. By $receive_s$, $(c, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu[x := m_1], \Phi[a :=$

$[m_2, \ldots, m_k]])$ (this is because the receive command is only executed if there is a message waiting on the queue). Now by simple security, $\Psi(a) = \lfloor\Phi\rfloor(a) = \Phi(a) = [m_1, \ldots, m_k]$, where $k \geq 1$. And by *receive*$_s$ again, $(d, \nu, \Psi) \longrightarrow (\mathbf{done}, \nu[x := m_1], \Psi[a := [m_2, \ldots, m_k]])$. Therefore, the move from (c, μ, Φ) is matched by the move from (d, ν, Ψ) in one step. This meets Condition 2.

4. c is of the form **if** e **then** c_1 **else** c_2. Here $d =$ **if** e **then** d_1 **else** d_2 where $d_1 = \lfloor c_1 \rfloor$ and $d_2 = \lfloor c_2 \rfloor$. By rule *if*$_t$, $e : L$ and by Simple Security $\nu(e) = \mu(e)$. So if $\mu(e) \neq 0$, then by first rule *if*$_s$, $(c, \mu, \Phi) \longrightarrow (c_1, \mu, \Phi)$. Then by the same rule, (d, ν, Ψ) can match this move in one step: $(d, \nu, \Psi) \longrightarrow (d_1, \nu, \Psi)$ This meets Condition 2. If $\mu(e) = 0$, then we have a similar result.

5. c is of the form **while** e **do** c_1. Here $d =$ **while** e **do** d_1 where $d_1 = \lfloor c_1 \rfloor$. By rule *while*$_t$, $e : L$ and c_1 cannot reach type $H\ cmd$. By Simple Security, $\nu(e) = \mu(e)$. So in case $\mu(e) \neq 0$, then the move $(c, \mu, \Phi) \longrightarrow (c_1; \mathbf{while}\ e\ \mathbf{do}\ c_1, \mu, \Phi)$ (by second rule *while*$_s$) is matched in one step by $(d, \nu, \Psi) \longrightarrow (d_1; \mathbf{while}\ e\ \mathbf{do}\ d_1, \nu, \Psi)$ (by the same rule). This meets Condition 2. When $\mu(e) = 0$, then by two applications of first rule *while*$_s$, the move $(c, \mu, \Phi) \longrightarrow (\mathbf{done}, \mu, \Phi)$ is matched in one step by the move $(d, \nu, \Psi) \longrightarrow (\mathbf{done}, \nu, \Psi)$. This meets Condition 2.

Next we address the case when c is of the form $c_1; c_2$. First recall that under rule *compose*$_s$, any move from $(c_1; c_2, \mu, \Phi)$ results from a move $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$ where c_1' may or may not be **done**. Depending on this, the transitions from c can be:

$$(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')$$

or

$$(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$$

Simulating (c, μ, Φ) we have (d, ν, Ψ) where $d = \lfloor c_1; c_2 \rfloor$, $\nu = \lfloor \mu \rfloor$, and $\Psi = \lfloor \Phi \rfloor$.

Depending on the types of c_1 and c_2, d has three possible forms which we handle separately.

1. First, if neither c_1 nor c_2 has type $H\ cmd$ then $d = \lfloor c_1 \rfloor; \lfloor c_2 \rfloor = d_1; d_2$ where $d_1 = \lfloor c_1 \rfloor$ and $d_2 = \lfloor c_2 \rfloor$. By induction, the move $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$ can be matched by (d_1, ν, Φ) using either Condition 1 or Condition 2 above. We now address each condition in turn.

   (a) If Condition 1 is met on $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$; i.e., (d_1, ν, Ψ) can match the move of (c_1, μ, Φ) in place, then $(d_1, \nu, \Psi) = (\lfloor c_1' \rfloor, \lfloor \mu' \rfloor, \lfloor \Phi' \rfloor)$. Now observe that d_1 cannot be **done** because only H commands strip to **done** and c_1 cannot be typed H command. This implies that c_1' cannot have type $H\ cmd$ and in particular c_1' cannot be **done**. Hence the move from c must be by the second rule $compose_s$, $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi)$. Then, (d, ν, Ψ) matches in place. This meets Condition 1.

   (b) If Condition 2 is met on $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$; then the transition must be matched by $(d_1, \nu, \Psi) \longrightarrow (d_1', \nu', \Psi')$ where $(c_1', \mu', \Phi') \lfloor \cdot \rfloor (d_1', \nu', \Phi')$. We now consider three distinct cases for c_1':

      i. If $c_1' =$ **done** then the move from c must be by the first rule $compose_s$: $(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')$. Then, d can make a matching move using first rule $compose_s$: $(d_1; d_2, \nu, \Psi) \longrightarrow (d_2, \nu', \Psi')$ with $(c', \mu', \Phi')\ \lfloor \cdot \rfloor\ (d', \nu', \Phi')$. Hence, condition 2 is met.

      ii. If $c_1' : H\ cmd$ but $c_1' \neq$ **done**, then the move from c must be by the second rule $compose_s$: $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$. But, $d_1' =$ **done**, hence, the move from d must be by the first rule $compose_s$: $(d_1; d_2, \nu, \Psi) \longrightarrow (d_2, \nu', \Psi')$ with $(c_1'; c_2, \mu', \Phi') \lfloor \cdot \rfloor (d_2, \nu', \Psi')$. Hence, condition 2 is met.

iii. If c_1' does not have type $H\ cmd$, then the move from c must be by the second rule $compose_s$: $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$. Recalling that by induction $d_1' = \lfloor c_1' \rfloor$ we must have that $d_1' \neq \textbf{done}$. And so the matching move from d must be by the second rule $compose_s$: $(d_1; d_2, \nu, \Psi) \longrightarrow (d_1'; d_2, \nu', \Psi')$, where $(c_1'; c_2, \mu', \Phi')\ \lfloor\cdot\rfloor\ (d_1'; d_2, \nu', \Psi')$. Hence, condition 2 is met.

2. Second, if $c_1 : H\ cmd$ then $d = \lfloor c_1; c_2 \rfloor = \lfloor c_2 \rfloor = d_2$. If the move from c is by the first rule $compose_s$, then we must have $(c_1, \mu, \Phi) \longrightarrow (\textbf{done}, \mu', \Phi')$, where by Confinement $\mu \sim_L \mu'$ and $\Phi \sim_L \Phi'$. So the move $(c_1; c_2, \mu, \Phi) \longrightarrow (c_2, \mu', \Phi')$ is matched in zero steps by (d_2, ν, Ψ), where $(c_1'; c_2, \mu', \Phi') \lfloor\cdot\rfloor (d_2, \nu, \Psi)$. This meets Condition 1.

   If instead the move from c is by the second rule $compose_s$, then we must have $(c_1, \mu, \Phi) \longrightarrow (c_1', \mu', \Phi')$, where by Confinement $\mu \sim_L \mu'$ and $\Phi \sim_L \Phi'$, and by Subject Reduction $c_1' : H\ cmd$. Hence the move $(c_1; c_2, \mu, \Phi) \longrightarrow (c_1'; c_2, \mu', \Phi')$ is again matched in zero steps by (d_2, ν, Ψ), where $(c_1'; c_2, \mu', \Phi') \lfloor\cdot\rfloor (d_2, \nu, \Psi)$. This meets Condition 1.

3. Third, if $c_2 : H\ cmd$ then $d = \lfloor c_1; c_2 \rfloor = \lfloor c_1 \rfloor = d_1$. In this case the argument is essentially the same as in the first case (a).

□

We are now ready to use these results in establishing our termination-insensitive noninterference result:

Theorem 4.3.5 *Let Δ_1 be a well-typed distributed program and let Δ_2 be formed by replacing each of the initial memories in Δ_1 with a L-equivalent memory. Let Φ_1 and Φ_2 be L-equivalent channel memories. Suppose that (Δ_1, Φ_1) and (Δ_2, Φ_2)*

$$\begin{tikzcd}
(\Delta_1, \Phi_1) \arrow[r, "\lfloor \cdot \rfloor"] \arrow[d, Rightarrow, "*"] & (\lfloor \Delta_1 \rfloor, \lfloor \Phi_1 \rfloor) \arrow[d, Rightarrow, "*"] & (\Delta_2, \Phi_2) \arrow[l, "\lfloor \cdot \rfloor"'] \arrow[d, Rightarrow, "*"] \\
(\Delta_1', \Phi_1') \arrow[r, "\lfloor \cdot \rfloor"] & (\lfloor \Delta_1' \rfloor, \lfloor \Phi_1' \rfloor) & (\Delta_2', \Phi_2') \arrow[l, "\lfloor \cdot \rfloor"']
\end{tikzcd}$$

Figure 4.9: Noninterference of distributed systems

can both execute successfully, reaching terminal configurations (Δ_1', Φ_1') *and* (Δ_2', Φ_2') *respectively. Then the corresponding local memories of* Δ_1' *and* Δ_2' *are* L*-equivalent, and* $\Phi_1' \sim_L \Phi_2'$.

Proof. By definition, $(\Delta_1, \Phi_1)\lfloor\cdot\rfloor\lfloor(\Delta_1, \Phi_1)\rfloor$ and $(\Delta_2, \Phi_2)\lfloor\cdot\rfloor\lfloor(\Delta_2, \Phi_2)\rfloor$. Hence, since $\lfloor\cdot\rfloor$is a fast low simulation, we know that $\lfloor(\Delta_1, \Phi_1)\rfloor$ and $\lfloor(\Delta_2, \Phi_2)\rfloor$ can also execute successfully, and can reach terminal configurations whose local memories are L-equivalent to the corresponding memories of Δ_1' and Δ_2'. Moreover, by Theorem 4.2.2 we know that those terminal configurations are unique.

But $\lfloor(\Delta_1, \Phi_1)\rfloor$ is *identical* to $\lfloor(\Delta_2, \Phi_2)\rfloor$, since neither contains H variables or channels. Hence they must reach the *same* terminal configuration. It follows that the corresponding local memories of Δ_1' and Δ_2' are L-equivalent and that $\Phi_1' \sim_L \Phi_2'$. □

Figure 4.9 illustrates the proof idea for this theorem. Running a distributed system twice and assuming that for each process the local memories are low equivalent[4] and that the two executions reach terminal configurations, we must have that the stripped version reaches a terminal configuration $(\lfloor\Delta_1'\rfloor, \lfloor\Phi_1'\rfloor)$ at least as fast as the original executions; also, since the language is observationally deterministic $(\lfloor\Delta_1'\rfloor, \lfloor\Phi_1'\rfloor)$ must be unique. So we are able to reestablish the bridge and it follows that corresponding final memories and channels must be low equivalent.

[4] Note that initially all channels are empty.

4.4 Towards a Concrete Implementation

In this section we explore the implementation of our distributed language in a concrete setting over a public network, where external adversary (Eve) is able to see but not modify the traffic and where she is not able to measure time intervals (although she can observe messages in the order that they are sent). We present a series of attacks on the distributed systems that help us define minimal confidentiality requirements on the network traffic, propose a middleware (similar to the CSMA protocol) that handles channels and message transmission, and sketch a proof of soundness for such a system. This exploration was first published in our FAST-09 paper [AS09].

The abstract language of Chapter 4 requires private channels for all communications which limits its applicability to secure settings, but we would like to implement our language in a more practical setting where communications between programs happen via a public network. We would like a setting like a wireless LAN but then we are faced with significant challenges to ensure confidentiality. Eavesdroppers can easily see all messages between processes; what would it take to implement our language for distributed systems in a wireless environment? Clearly, secret data cannot be transmitted in a wireless LAN with any expectation of confidentiality as any computer with a receiver can get all the data that has been transmitted. Hence our *first requirement*, we need cryptography to hide the information being transmitted. Asymmetric cryptography seems to be the appropriate style for our setting. We should keep in mind that the introduction of cryptography means that from an information theoretic perspective the language is completely insecure, as encrypted data completely leaks its value. Although we expect that recovering the

```
-$\Delta_4$-
Process 1
    if ($h_1$ is even) then run a long time
    $send(a_{H,1,7}, 1)$
Process 2
    if ($h_2$ is odd) then run a long time
    $send(a_{H,2,8}, 0)$
```

```
-$\Delta_5$-
Process 1
    if ($h_1$ is even) then
        $C \overset{?}{\leftarrow} \mathcal{E}_{pk}(someMsg);$
        $send(a_{L,1,3}, C)$
        $send(a_{L,1,3}, C)$
```

Figure 4.10: Attacks (second wave)

secrets from the ciphertext will take too long and thus becomes unfeasible for the attacker.

Having decided on cryptography to hide the information that is being transmitted, we really want to encrypt only what is necessary to maintain the soundness of the distributed system, since encryption and decryption are expensive operations. In a wireless LAN, communications happen via electromagnetic signals which contain not only the message (payload) but also other information like source, destination, and data classification (header). Clearly we have to encrypt the payload but do we have to encrypt the header? In fact we do, for otherwise the language confidentiality would be lost as exemplified in Δ_4 of Figure 4.10. The channel used in both processes is a high channel (encrypted payload) yet an eavesdropper can still discern the value of the least bit of the secret h_1 by looking in the header of each packet for which process sends first; if Process 1 sends first then h_1 is odd and the least bit is 1. Therefore our *second requirement:* we have to encrypt the header and the payload of packets on high channels to prevent the leakage of secret information. Yet we are not done because surprisingly, this attack works even if the message sent is public and it is being sent on a public channel. Consider Δ_4 again and let's assume that we are encrypting all headers (secret and public) and the secret payloads, but we are allowing the public data to be transmitted in the clear. This seems reasonable enough since the adversary will get all public data at the end of the execution. Nevertheless, if we observe a public value of 1 being transmitted first we will know with high probability that the least bit of h_1 is 1. Hence our *third requirement:* we have to encrypt all transmitted data.

We remark that if we had an active adversary which was able to drop packets, modify them and resend them, in addition to the attacks that we have seen she could modify packets to leak information and to affect the integrity of the distributed system. For example if the packet header was not encrypted she could change the packet classification from H to L thereby declassifying the payload which could cause the packet to be received by a low channel buffer in the receiving process. Then she could wait until the end of the execution to pick up the leaked secret from the process' public memory.

This distinction may play a role in deciding what kind of security property will be necessary in our encryption scheme. Specifically a passive adversary might only require IND-CPA security while the active adversary will definitely require IND-CCA security. As an illustration of this distinction consider the following variation of the Warinschi attack on the Needham-Schroeder-(Lowe) protocol [War03] where an IND-CPA scheme has the flaw where there is a function $C' := f(C)$ that takes an encrypted plaintext (like a packet header) and returns the ciphertext of an identical plaintext but with a certain location within it changed to L. This would not affect the security of the encryption scheme in any other way i.e., an adversary would not be able to know anything about the content of the ciphertext but by simply substituting the header of any packet with C' and re-sending it, the adversary would be able to declassify the payload of the message.

But continuing with our analysis, are we done? We have decided to encrypt all data that is transmitted in the network, yet it is not enough to ensure confidentiality.

```
-$\Delta_6$-
Process 1
    $send(a_{L,1,2}, 0)$
    if ($h_1$ is even) then delay 1 sec;
    $send(a_{L,1,2}, 0)$
```

```
-$\Delta_7$-
Process 1
    if ($h_1$ is even) then
        $send(a_{H,1,2}, 0)$
```

Figure 4.11: Attacks (third wave)

Consider adversary Δ_5 of Figure 4.10, it encrypts a message and sends it two times if a secret is even. Meanwhile Eve scans every transmission waiting for two identical ciphertexts; if they are found she knows with high probability that the least bit of h_1 is 1, if all ciphers are distinct, it is 0. Therefore our *fourth requirement:* all transmitted data must be composed of freshly generated ciphertexts to ensure the confidentiality of our distributed systems.

Finally, there are two more attacks that we need to consider. The first one is the classical timing attack. If we know how long a typical execution step takes and can measure time intervals then we can leak information. Δ_6 of Figure 4.11 exemplifies this attack. If Eve is able to measure the time interval between the two transmissions she will have the value of the least bit of h_1. A related timing attack is to count the number of message transmitted as in Δ_7 of Figure 4.11. In this attack, the last bit of h_1 is leaked by the transmission of one or zero packets. These attacks can be generalized to changing the statistics of packet transmission; for example, one can conceive an attack where the time distribution for packet transmission is uniform to leak a 0 and χ-square or normal for a 1.

However, this area of study seems out of scope for us as it is addressed in other areas of research. For example recently, Köpf and Dürmuth [KD09] use Shannon entropy to bound the leakage of timing and side channel attacks under the *input blinding* and *bucketing* countermeasures. The advantage of this work over more traditional approaches is that it carries provable security guarantees both using

Shannon-entropy and min-entropy [KS10] to bound the leakage under this countermeasure. Other more traditional solutions include imposing a super-density of packet transmission at regular intervals, where only some of the packets are real messages. This might eliminate the problem but significantly increase the network bandwidth utilization. A solution on the same vein could be the NRL-Pump [KM93a, KMC05] which is currently available at the local (government-owned) electronics shop. The pump obfuscates Eve's ability to measure the time between messages. It does this by inserting random delays based on an adaptive mechanism that adjusts the delays based on network traffic statistics. However, the pump cannot prevent timing channels based on the order of distinguishable messages and does not come with security guarantees.

To summarize, to ensure confidentiality of our distributed system, we not only have to somehow hide the meaning of messages, but also hide anything in a message that makes it distinguishable to Eve. Obviously, this includes payload, but also the message header, since the source or destination of a message can be used to distinguish it from another. Although L values are public, they cannot be seen within the network traffic. This is not a problem when computation happens within a processor (like in some multithreaded environments) because it is not reasonable that Eve would have access to the public memory in real time.

Soundness: Next we sketch a possible way to argue a computational noninterference property for our concrete language:

Random Transfer Language Definition and Soundness: First, we should be able to move our language, at the most basic level, from a nondeterministic to a probabilistic setting by constructing a subset language which we will call *Random Transfer Language*, and prove PNI on it. The PNI property on this language shall establish that if we allow only fresh-random traffic, the language is safe and sound.

Message Transfer Language Definition and Soundness: Then, we should be able to construct a *Message Transfer Language* and prove CNI on it. This language simply has the regular send command encrypt its payload before transmission on a public channel but keeps private channels for transmission of header information.

Header Transfer Language Definition and Soundness: Next, we should be able to construct *Header Transfer Language* and prove CNI on it. In this language the send command encrypts its header before transmission, but keeps private channels for transmission of the payload.

Hybrid Cryptographic Argument: Finally, we should be able to argue that since the Message and Header Transfer languages both satisfy CNI, the combination also should via a hybrid cryptographic argument.

In this chapter we extended the core language of Chapter 3 to an abstract language for distributed systems under a nondeterministic scheduler. We have maintained the fairly nonrestrictive type system for individual processes while keeping the noninterference property on it. As far as we know, this is a new result in the area of Secure Information Flow. While in the past, noninterference under a nondeterministic scheduler has not meant that the concrete implementation of the language would be secure, we are fairly certain that the implementation of this language as detailed in Section 4.4 would be (although this is not shown).

CHAPTER 5

PROBABILISTIC SIMULATION AND NONTERMINATION.

So far we have not restricted our languages beyond the Denning restrictions and our results have been valid as long as the executions reach terminal states, although later on (when we deal with cryptographic languages) our results will only apply to polynomial-time programs. We now focus on the behavior of potentially nonterminating programs looking to *quantify* the effects of nontermination. In this work, which is a revised version of [SA11], we extend the core language of Chapter 3 to use a probabilistic semantics by the introduction of a random assignment operation. We find that the nontermination of program executions can be used to leak information even when the program meets the Denning Restrictions. Then, we reason about the behavior of nonterminating programs and develop a property that says that the probability of a nonterminating execution must be conserved in that it must come directly from the probabilities of the execution reaching other terminal configurations. We call this the *bucket property* . We then extend the core theory of Chapter 3 to a probabilistic setting by introducing *fast probabilistic simulation* on Markov chains in general and further, *fast low probabilistic simulation* on the Markov chains resulting from the execution of our extended language under probabilistic semantics. Finally, we provide some applications of the theory including a proof of probabilistic noninterference on our random assignment language.

In this setting, *fast simulation* and *fast low simulation* are now probabilistic concepts. Also, *noninterference* becomes *probabilistic noninterference* which says roughly, that in the execution of a program the final values of L variables are independent of the initial values of H variables [VS99]. This is formalized using the concept of *low equivalence* of memories which we recall from Chapter 3:

Definition 5.0.1 *Two memories μ and ν are L-*equivalent*, written $\mu \sim_L \nu$, if they agree on the values of all L variables.*

The idea is that if initial memories μ and ν are low equivalent, then running c under μ should be (in some sense) equivalent to running c under ν, as far as L variables are concerned. But the possibility of nontermination causes complications, because the Denning restrictions allow programs whose termination behavior depends on the values of H variables. For example, the program:

while $h = 1$ **do done**

where h is a H variable, might terminate under μ and loop under ν. One reaction to the termination issue is to say that further restrictions are needed. A number of studies have proposed forbidding H variables in the guards of **while** loops (e.g. [VS97]) or forbidding assignments to L variables that sequentially follow commands whose termination depends on H variables (e.g. [Smi06]). But such additional restrictions may in practice be overly stringent, making it difficult to write useful programs. For this reason, practical secure information-flow languages like Jif [MCN$^+$06] have chosen not to impose extra restrictions to control termination leaks.

In this chapter, therefore, we study the behavior of potentially nonterminating programs typed just under the Denning restrictions. To be able to make *quantitative* statements about the effects of nontermination, we consider a *probabilistic* language. In such a language, we would like to achieve *probabilistic noninterference* [VS99], which asserts that the probability distribution on the final values of L variables is independent of the initial values of H variables.

Example 5.0.1 *Consider the program in Figure 5.1. Note that $t \overset{?}{\leftarrow} \{0, 1\}$ is a* random assignment *that assigns either 0 or 1 to t, each with probability* $1/2$. *Assuming that h is H and t and l are L, this program satisfies the Denning restrictions and*

```
$t \overset{?}{\leftarrow} \{0,1\}$;
if $t = 0$ then (
    while $h = 1$ do skip;
    $l := 0$
)
else (
    while $h = 0$ do skip;
    $l := 1$
)
```

Figure 5.1: A random assignment program

it is well-typed under the typing rules that we will present in Section 5.1. But, if $h = 0$*, then this program terminates with* $l = 0$ *with probability* $1/2$ *and fails to terminate with probability* $1/2$*. And if* $h = 1$*, then it terminates with* $l = 1$ *with probability* $1/2$ *and fails to terminate with probability* $1/2$*. Thus this program does not satisfy probabilistic noninterference.*

What goes wrong? Intuitively, the program is "trying" to set l to either 0 or 1, each with probability $1/2$*, regardless of the value of h. But the* **while** *loops, whose termination depends on the value of h, sometimes prevent assignments to l from being reached. As a result, the probabilities of some of the terminal states of the execution (with specific values of l) are lowered, because the paths that would have led to them become infinite loops. This suggests, that if the probability of nontermination of a well-typed program is small, then it will "almost" satisfy probabilistic noninterference.*

To make these intuitions precise, we consider a thought experiment. For any well-typed program c, we define a "stripped" version of c, denoted by $\lfloor c \rfloor$. In $\lfloor c \rfloor$, certain subcommands of c (namely, those that make no assignments to L variables) are removed. For example, the stripped version of the program in Figure 5.1 is shown in Figure 5.2. We emphasize that stripping is for us a *thought experiment*—it

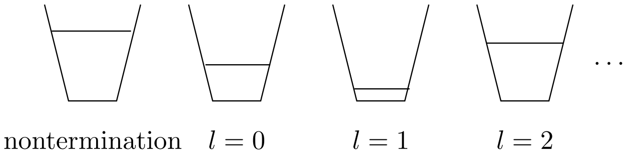
```
$t \overset{?}{\leftarrow} \{0,1\}$;
**if** $t = 0$ **then**
    $l := 0$
**else**
    $l := 1$
```

Figure 5.2: Stripped version of the program

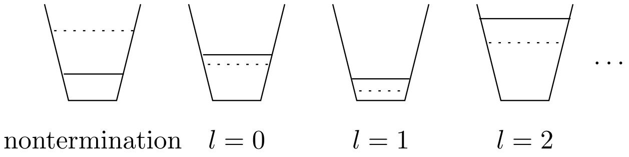
is not something that we would actually use in an implementation, but rather it is a means to *understand* a program's behavior.

The major technical effort of this chapter is to prove a precise relationship between the behavior of a well-typed program c and of its stripped version $\lfloor c \rfloor$. We will show that the only effect of the stripping operation is to boost the probabilities of certain L outcomes by lowering the probability of nontermination. For example, consider the program in Figure 5.1 when $h = 0$. Stripping boosts the probability of terminating with $l = 1$ from 0 up to 1/2 by lowering the probability of nontermination from 1/2 down to 0; it leaves the probability of terminating with $l = 0$ unchanged at 1/2.

More precisely, we will prove in Theorem 5.4.3 that the probability that c terminates with certain values for its L variables is always less than or equal to the corresponding probability for $\lfloor c \rfloor$. To visualize this theorem, imagine that the results of running c and $\lfloor c \rfloor$ are shown as two sequences of buckets, one for each possible final value of their L variables, and one bucket each to represent nontermination. The probability of each outcome is indicated by the amount of water in each bucket; there is a total of one gallon of water among all the buckets in each set of buckets. Suppose that c's buckets look like this:



Then Theorem 5.4.3 tells us that $\lfloor c \rfloor$'s buckets are gotten simply by pouring some of the water from c's nontermination bucket into some of the other buckets:



Note that the amount of water moved is defined by c and that no water moves from any terminal bucket.

The need for Theorem 5.4.3 first arose in [SA06] where it was claimed without proof. Furthermore, it was claimed there that the proof could be done using a *strong probabilistic simulation* as defined by Jonsson and Larsen [JL91]. But that claim was incorrect; strong simulation turns out to be too restrictive for this purpose and while weak simulation [BKHW05] could be used for our result here, it is needlessly complex for our setting. For this reason, we introduced a new simulation in [SA07], which we call *fast simulation*, refine it here as part of our extended theory, and show that it can be used in proving Theorem 5.4.3.

Theorem 5.4.3 gives us a way of bounding the effect of nontermination. For example, if c's nontermination bucket is empty, then $\lfloor c \rfloor$'s buckets are identical to c's, because there is no water to pour! More generally, we prove in Theorem 5.5.2 that if a well-typed program c fails to terminate with probability at most p, then c's deviation from probabilistic noninterference is at most $2p$.

$$\begin{array}{llll}
\textit{(phrases)} & p & ::= & e \mid c \\
\textit{(expressions)} & e & ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
\textit{(commands)} & c & ::= & \mathbf{done} \mid x := e \mid \\
 & & & x \overset{?}{\leftarrow} \mathcal{D} \mid \\
 & & & \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \\
 & & & \mathbf{while}\ e\ \mathbf{do}\ c \mid c_1; c_2 \\
\textit{(variables)} & x, y, z, \ldots & &
\end{array}$$

Figure 5.3: Probabilistic language syntax

5.1 A Probabilistic Language

In this section, we extend the core language of Chapter 3 to a probabilistic setting. Previously, we designed a similar language in [SA06]. Formally, the language syntax (Figure 5.3) is that of the simple imperative language except that we introduce a random assignment command (we will see that the addition of this simple command greatly changes the language semantics and significantly increases the complexity of the theory). In the syntax there is only one type of identifier variable, for which we use metavariables x, y, and z; metavariable n ranges over integer literals. Integers are the only values; we use 0 for false and nonzero for true. As in previous chapters we have replaced the traditional **skip** command with a **done** command instead; **done** can be used in much the same way as **skip** and (as will be seen below) it is also used to represent a terminated command in a configuration.

The command $x \overset{?}{\leftarrow} \mathcal{D}$ is a random assignment; here $\mathcal{D}$ ranges over some set of probability distributions on the integers. In examples, we use notation like $x \overset{?}{\leftarrow} \{0, 1, 2\}$ to denote a random assignment command that assigns either 0, 1, or 2 to x, each with equal probability. Please note that the language allows random *assignments* but not random *expressions*; this lets us use simpler semantics and does not limit the language's computational power.

A program c is executed under a *memory* μ, which maps identifiers to integers. We assume that expressions are total and evaluated atomically, and we write $\mu(e)$ to denote the value of expression e in memory μ. In our semantics, a *configuration* is a pair (c, μ) where c is a command and μ is a memory. Note that *terminal* configurations are written as $(\mathbf{done}, \mu)$ in our semantics. We remark that the more common semantic approach is to have both non-terminal configurations (c, μ) and also terminal configurations μ, but this tends to lead to a proliferation of cases in proofs; it is therefore simpler to have only configurations of the form (c, μ).

Probabilistic semantics: to model the language with the new random assignment command, the standard transition relation on configurations, which we use on the core theory, is not adequate; the semantics needs to be extended with probabilities—we write $(c, \mu) \xrightarrow{p} (c', \mu')$ to indicate that the probability of going from configuration (c, μ) to configuration (c', μ') is p. The semantic rules are given in Figure 5.4. They define a *Markov chain* [Fel68] on the set of configurations. Notice that terminal configurations $(\mathbf{done}, \mu)$ are *absorbing states* in the Markov chain.

The type system for this language simply enforces the Denning restrictions. Here are the types we will use:

$$\begin{array}{lrcl} (\mathit{data\ types}) & \tau & ::= & L \mid H \\ (\mathit{phrase\ types}) & \rho & ::= & \tau \mid \tau\ \mathit{var} \mid \tau\ \mathit{cmd} \end{array}$$

Typing judgments have the form $\Gamma \vdash p : \rho$, where Γ is an *identifier typing* that maps each variable to a type of the form $\tau\ \mathit{var}$. In this chapter Γ is fixed throughout. Intuitively, $\tau\ \mathit{var}$ is the type of variables that store information of level τ, while $\tau\ \mathit{cmd}$ is the type of commands that assign only to variables of level τ or higher.[1] The typing and subtyping rules are given in Figures 5.5. The rules are the same as

[1] Again as in Chapter 3, this implies that command types obey a *contravariant* subtyping rule.

$$(\mathit{update}_s) \quad \frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \xrightarrow{1} (\mathbf{done}, \mu[x := \mu(e)])}$$

$$(\mathit{random}_s) \quad \frac{x \in \mathit{dom}(\mu) \quad \mathcal{D}(v) > 0}{(x \overset{?}{\leftarrow} \mathcal{D}, \mu) \xrightarrow{\mathcal{D}(v)} (\mathbf{done}, \mu[x := v])}$$

$$(\mathit{if}_s) \quad \frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \xrightarrow{1} (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \xrightarrow{1} (c_2, \mu)}$$

$$(\mathit{while}_s) \quad \frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \xrightarrow{1} (\mathbf{done}, \mu)}$$

$$\frac{\mu_(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \xrightarrow{1} (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu)}$$

$$(\mathit{compose}_s) \quad \frac{(c_1, \mu) \xrightarrow{p} (\mathbf{done}, \mu')}{(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')}$$

$$\frac{(c_1, \mu) \xrightarrow{p} (c_1', \mu') \quad c_1' \neq \mathbf{done}}{(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')}$$

$$(\mathit{done}_s) \quad (\mathbf{done}, \mu) \xrightarrow{1} (\mathbf{done}, \mu)$$

Figure 5.4: Probabilistic semantics

those in Chapter 3, except for the new rule random_t for random assignment. Rule random_t says that a random assignment can be done to any variable x, but if x has type $\tau\ \mathit{var}$, then the assignment gets type $\tau\ \mathit{cmd}$; this type is used to prevent improper implicit flows.

Now we have the usual Simple Security, Confinement, and Subject Reduction properties for this language:

Lemma 5.1.1 (Simple Security) *If $\Gamma \vdash e : \tau$, then e contains only variables of level τ or lower.*

Proof. By induction on the structure of e. □

$$(\mathit{base}) \qquad L \subseteq H$$

$$(\mathit{cmd}) \qquad \frac{\tau \subseteq \tau'}{\tau' \ \mathit{cmd} \subseteq \tau \ \mathit{cmd}}$$

$$(\mathit{reflex}) \qquad \rho \subseteq \rho$$

$$(\mathit{trans}) \qquad \frac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$(\mathit{subsump}) \qquad \frac{\Gamma \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

$$(\mathit{done}_t) \qquad \Gamma \vdash \mathbf{done} : H \ \mathit{cmd}$$

$$(\mathit{int}_t) \qquad \Gamma \vdash n : L$$

$$(\mathit{rval}_t) \qquad \frac{\Gamma(x) = \tau \ \mathit{var}}{\Gamma \vdash x : \tau}$$

$$(\mathit{update}_t) \qquad \frac{\Gamma(x) = \tau \ \mathit{var} \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau \ \mathit{cmd}}$$

$$(\mathit{random}_t) \qquad \frac{\Gamma(x) = \tau \ \mathit{var}}{\Gamma \vdash x \overset{?}{\leftarrow} \mathcal{D} : \tau \ \mathit{cmd}}$$

$$(\mathit{plus}_t) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$$

$$(\mathit{if}_t) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau \ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}}{\Gamma \vdash \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 : \tau \ \mathit{cmd}}$$

$$(\mathit{while}_t) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau \ \mathit{cmd}}{\Gamma \vdash \mathbf{while} \ e \ \mathbf{do} \ c : \tau \ \mathit{cmd}}$$

$$(\mathit{compose}_t) \qquad \frac{\Gamma \vdash c_1 : \tau \ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau \ \mathit{cmd}}{\Gamma \vdash c_1; c_2 : \tau \ \mathit{cmd}}$$

Figure 5.5: Probabilistic language type system

Lemma 5.1.2 (Confinement) *If $\Gamma \vdash c : \tau \ cmd$, then c assigns only to variables of level τ or higher.*

Proof. By induction on the structure of c. □

Lemma 5.1.3 (Subject Reduction) *If $\Gamma \vdash c : \tau\ cmd$ and $(c, \mu) \xrightarrow{p} (c', \mu')$ for some $p > 0$, then $\Gamma \vdash c' : \tau\ cmd$.*

Proof. By induction on the structure of c. □

But, as discussed at the beginning of this chapter, well-typed programs need not satisfy probabilistic noninterference. This is because changes to H variables can result in infinite loops that block subsequent assignments to L variables, affecting the probabilities of different L outcomes. Therefore as we seek a relationship between the program's full execution and its "low execution", i.e., its execution as far as the low variables is concerned, we now develop key results in the theory of probabilistic simulation that will be needed later.

5.2 Probabilistic Simulation for Transition Systems

In this chapter, we discuss the theory of probabilistic simulation and bisimulation in the abstract setting of *discrete time Markov Chains*.

Definition 5.2.1 (DTMC) *A (discrete-time)* Markov chain *[Fel68] is a pair $(S, \mathbf{P})$ where*

- *S is a countable set of states, and*
- *$\mathbf{P} : S \times S \to [0, 1]$ is a probability matrix satisfying $\sum_{t \in S} \mathbf{P}(s, t) = 1$ for all $s \in S$.*

If $\mathbf{P}(s, t) > 0$, then we say that t is a successor *of s. Also, for $T \subseteq S$, we write $\mathbf{P}(s, T)$ to denote $\sum_{t \in T} \mathbf{P}(s, t)$, the probability of going in one step from s to a state in T.*

A classic equivalence relation on Markov chains is *probabilistic bisimulation* , due to Kemeny and Snell [KS60] and Larsen and Skou [LS91].

Definition 5.2.2 (Strong Bisimulation) *Let R be an equivalence relation on S. R is a* strong bisimulation *if whenever sRt we have $\mathbf{P}(s, T) = \mathbf{P}(t, T)$ for every equivalence class T of R.*

Both strong and weak versions of bisimulation have been applied fruitfully in secure information flow analysis, for example by Gray [Gra90], Sabelfeld and Sands [SS00], and Smith [Smi06]. The basic idea is that a secure program should behave (in some sense) "indistinguishably" when run under two low-equivalent initial memories; this indistinguishability can be formalized as a bisimulation.

However, in this chapter, we apply instead non-symmetric *probabilistic simulation* relations, explored earlier by Jonsson and Larsen [JL91] and Baier, Katoen, Hermanns, and Wolf [BKHW05]. Roughly speaking, a binary relation R is a strong simulation if whenever sRt and s has a successor s', t has a "matching" successor t' that simulates s' (i.e. $s'Rt'$). But in the probabilistic setting, we must also make sure that the probabilities are preserved. Suppose for example that $\mathbf{P}(s, s') = p$. Then t must be able to match that much probability. But t need not have a single successor t' such that $\mathbf{P}(t, t') = p$. Instead, it is enough for t to have several successors, each simulating s', such that the total probability is p. However, in doing this simulation we must not "double count" t's probabilities—for example, if s goes to s' with probability $1/3$ and t goes to t' with probability $1/2$, then if we use t' to match the move to s' we must remember that $1/3$ of t''s probability is "used up", leaving just $1/6$ to be used in matching other moves of s. These considerations lead to what is called a *weight function* Δ to specify how the probabilities are matched up, giving the following definition (adapted from Definition 16 of [BKHW05]):

Definition 5.2.3 (Strong Simulation) *Let R be a binary relation on S. R is a* strong simulation *if, whenever sRt, there exists a function $\Delta : S \times S \to [0, 1]$ such that*

1. $\Delta(s', t') > 0$ *implies that* $s'Rt'$,
2. $\mathbf{P}(s, s') = \sum_{u \in S} \Delta(s', u)$ *for all* $s' \in S$,
3. $\mathbf{P}(t, t') = \sum_{u \in S} \Delta(u, t')$ *for all* $t' \in S$.

For our exploration of the stripping operation $\lfloor \cdot \rfloor$ in Section 5.3, however, it turns out that strong simulation isn't quite what we want, because it does not allow the simulating state t to run "faster" than the simulated state s. The issue is that s could make "insignificant" moves to states that are already simulated by t; in this case t shouldn't need to make a matching move. Such "insignificant" moves are allowed by the more flexible notion of *weak simulation* in Definition 34 of [BKHW05]. But their definition also allows t to make "insignificant" moves, which are not appropriate for us, since we want t to run at least as fast as s. So here we develop a restricted kind of weak simulation, which we call a *fast simulation*.

Figure 5.6 illustrates the concept of a state s simulated by a state t. In the figure, sRt is represented by a dashed arrow from state s to state t, while a transition relation from say s to u_1 is represented by a solid arrow. We partition successors of s into two sets, $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_m\}$. The states in V represent "insignificant" moves, and we require that t itself simulates each of them (note the dashed arrows from each element of V to t. The states in U represent "significant" moves, and we require that t be able to match such moves. The state t then, must transition to a set of states $W = \{w_1, w_2, \ldots, w_r\}$ such that there exist some weight function $\Delta(i, j)$ of size $m \times r$ where the sum of its rows projects to U and the sum of its columns projects to W. Intuitively then, t matches s's behavior either in place

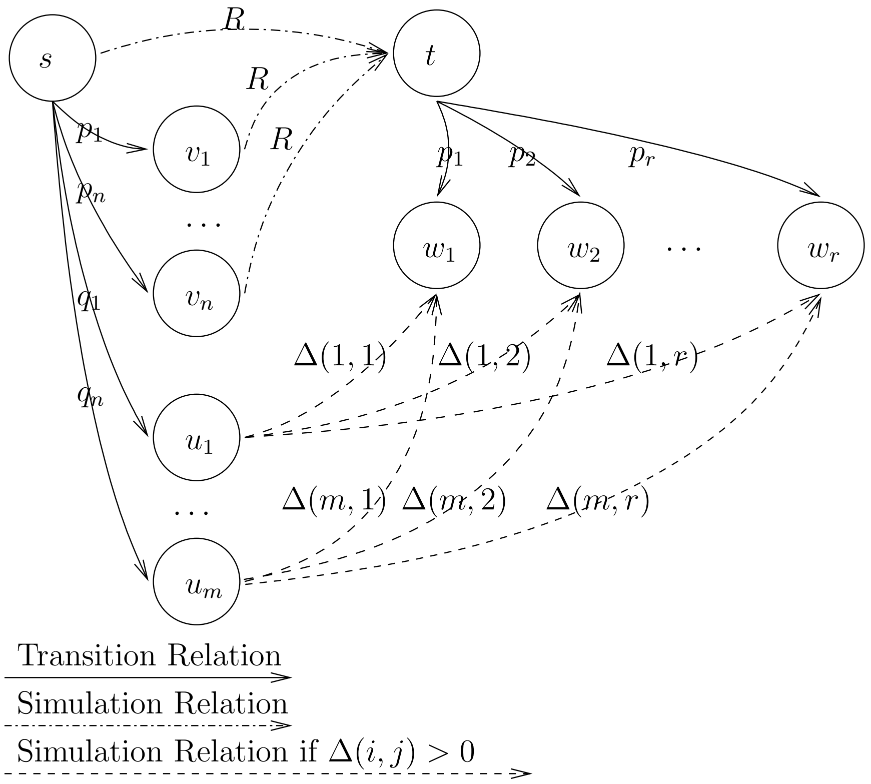


Figure 5.6: Fast Probabilistic Simulation

or by moving to a simulating state in one step. Formally, we have the following definition:

Definition 5.2.4 (Fast Simulation) *Let R be a binary relation on S. R is a* fast simulation *if, whenever sRt, the successors of s can be partitioned into two sets U and V such that*

1. *vRt for every $v \in V$, and*
2. *letting $K = \sum_{u \in U} \mathbf{P}(s,u)$, if $K > 0$ then there exists a function $\Delta : S \times S \to [0,1]$ such that*

   (a) $\Delta(u,w) > 0$ implies that uRw,

   (b) $\mathbf{P}(s,u)/K = \sum_{w \in S} \Delta(u,w)$ for all $u \in U$, and

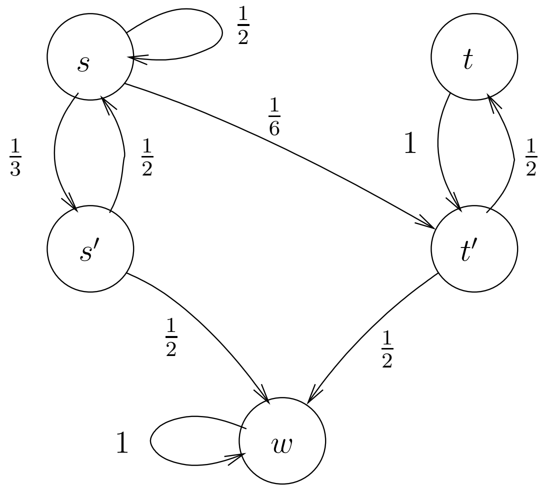


Figure 5.7: An example Markov chain

(c) $\mathbf{P}(t, w) = \sum_{u \in U} \Delta(u, w)$ *for all* $w \in S$.

Notice that in condition 2(b), $\mathbf{P}(s, u)/K$ is the *conditional probability* of going from s to u, given that s goes to U. The reason for using a conditional probability here may be intuitively unclear; a justification for this is to normalize the probability of significant transitions; in fact the best justification for this definition is its utility in the proof of the key Theorem 5.2.7 below. We now illustrate this concept with an example:

Example 5.2.1 *Consider the Markov chain in Figure 5.7, where* $S = \{s, t, s', t', w\}$. *Define* R *by* sRt, $s'Rt'$, *together with* uRu *for every* $u \in S$. *(In other words,* R *is the* reflexive closure *of* $\{(s, t), (s', t')\}$.*) Then we can show that* R *is a fast simulation:*

- *For pairs of the form* xRx, *we can always satisfy the requirements of Definition 5.2.4 by choosing* U *to be the set of successors of* x *and* V *to be* $\emptyset$. *Then* $K = 1$, *and for each* $u \in U$ *we can choose* $\Delta(u, u) = \mathbf{P}(x, u)$.

  Here is the Delta function for sRs. *The top row has the possible destinations of the* simulating *state with their probabilities while the left most column has* *the possible destinations of the* simulated *state also with their probabilities, although in this case they are the same. We have omitted rows and columns that do not contribute to the function.*

$\Delta_{s,s}$	s	s'	t'
s	$\frac{1}{2}$	0	0
s'	0	$\frac{1}{3}$	0
t'	0	0	$\frac{1}{6}$

It is straightforward to verify that these choices satisfy conditions 1, 2(a), 2(b), and 2(c).

- *For sRt we can choose $U = \{s', t'\}$ and $V = \{s\}$, which makes $K = \frac{1}{2}$, and we can choose $\Delta(s', t') = \frac{2}{3}$ and $\Delta(t', t') = \frac{1}{3}$.*

$\Delta_{s,t}$	t'
s'	$\frac{2}{3}$
t'	$\frac{1}{3}$

- *Finally, for $s'Rt'$ we can choose $U = \{s, w\}$ and $V = \emptyset$, which makes $K = 1$, and we can choose $\Delta(s, t) = \frac{1}{2}$ and $\Delta(w, w) = \frac{1}{2}$.*

$\Delta_{s',t'}$	t	w
s	$\frac{1}{2}$	0
w	0	$\frac{1}{2}$

We remark that every strong simulation is also a fast simulation, since a strong simulation is just a fast simulation in which all the V sets are empty. Furthermore, every fast simulation is also a weak simulation as defined in Definition 34 of [BKHW05].

Properties of fast simulation. We now develop the key properties of fast simulation starting with terminal states.

Definition 5.2.5 (Upwards Closed Set) *Let R be a binary relation on S. A set T of states is* upwards closed with respect to R *if, whenever $s \in T$ and sRs', we also have $s' \in T$.*

If $s \in S$, n is a natural number, and $T \subseteq S$, then let us write $\Pr(s, n, T)$ to denote the probability of reaching a state in T from s in at most n steps. Following [BKHW05], we can calculate $\Pr(s, n, T)$ with a recurrence:

$$\Pr(s, n, T) \quad = \quad \begin{cases} 1, & \text{if } s \in T \\ \sum_{s' \in S} \mathbf{P}(s, s')\Pr(s', n-1, T), & \text{if } n > 0 \text{ and } s \notin T \\ 0, & \text{if } n = 0 \text{ and } s \notin T \end{cases}$$

Note that $\Pr(s, n, T)$ increases monotonically with n:

Lemma 5.2.6 $\Pr(s, n, T) \leq \Pr(s, n+1, T)$*, for all n, s, and T.*

Proof. By induction on n. For the basis, if $s \notin T$, then $\Pr(s, 0, T) = 0 \leq \Pr(s, 1, T)$. And if $s \in T$, then $\Pr(s, 0, T) = 1 \leq \Pr(s, 1, T)$.

For the induction, assume that for some $k \geq 0$ we have $\Pr(s, k, T) \leq \Pr(s, k+1, T)$ for all s and T. We must show that $\Pr(s, k+1, T) \leq \Pr(s, k+2, T)$. First note that if $s \in T$ then we have $\Pr(s, k+1, T) = 1 = \Pr(s, k+2, T)$. It remains to consider the case when $s \notin T$. In this case we have

$$\begin{aligned} \Pr(s, k+2, T) &= \sum_{s' \in S} \mathbf{P}(s, s')\Pr(s', k+1, T) \\ &\geq \sum_{s' \in S} \mathbf{P}(s, s')\Pr(s', k, T) \\ &\quad \text{(by induction)} \\ &= \Pr(s, k+1, T) \end{aligned}$$

□

We now proceed to the key theorem about fast simulation; its proof is similar to the proof of Theorem 54 of [BKHW05], though that theorem refers to *strong simulation.*

Theorem 5.2.7 (Reachability) *If R is a fast simulation, T is upwards closed with respect to R, and $s_1 R s_2$, then* $\Pr(s_1, n, T) \leq \Pr(s_2, n, T)$ *for every n.*

Proof. By induction on n. For the basis, note that if $s_1 \notin T$, then $\Pr(s_1, 0, T) = 0 \leq \Pr(s_2, 0, T)$. And if $s_1 \in T$, then $s_2 \in T$, since $s_1 R s_2$ and T is upwards closed. So we have $\Pr(s_1, 0, T) = 1 = \Pr(s_2, 0, T)$.

For the induction, first note that if $s_1 \in T$ then as above we have $\Pr(s_1, k{+}1, T) = 1 = \Pr(s_2, k{+}1, T)$, and if $s_2 \in T$ then we have $\Pr(s_1, k{+}1, T) \leq 1 = \Pr(s_2, k{+}1, T)$. It remains to consider the case when $s_1 \notin T$ and $s_2 \notin T$. In this case we have

$$\begin{aligned}\Pr(s_1, k+1, T) &= \sum_{s \in S} \mathbf{P}(s_1, s)\Pr(s, k, T) \\ &= \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T)\end{aligned}$$

where U and V are as specified in Definition 5.2.4. (Note that the rearrangement is valid because the series are absolutely convergent.)

Now, for every $v \in V$, since vRs_2 we get by induction that $\Pr(v, k, T) \leq \Pr(s_2, k, T)$. Also, letting $K = \sum_{u \in U} \mathbf{P}(s_1, u)$, we note that $\sum_{v \in V} \mathbf{P}(s_1, v) = (1 - K)$. Hence we have

$$\begin{aligned}\sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) &\leq \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(s_2, k, T) \\ &= (\sum_{v \in V} \mathbf{P}(s_1, v))\Pr(s_2, k, T) \\ &= (1 - K)\Pr(s_2, k, T) \\ &\leq (1 - K)\Pr(s_2, k+1, T)\end{aligned}$$

Note that if $K = 0$, then $U = \emptyset$, which implies that

$$\begin{aligned}
\Pr(s_1, k+1, T) &= \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T) \\
&\leq (1-0)\Pr(s_2, k+1, T) + 0 \\
&= \Pr(s_2, k+1, T)
\end{aligned}$$

We are left with the case when $K > 0$. In that case, by Definition 5.2.4 there exists a function Δ satisfying conditions 2(a), 2(b), and 2(c). Hence we have

$$\begin{aligned}
\sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T) &= \sum_{u \in U}(K \sum_{w \in S} \Delta(u, w))\Pr(u, k, T) \\
&= K \sum_{u \in U}(\sum_{w \in S} \Delta(u, w)\Pr(u, k, T)) \\
&\leq K \sum_{u \in U}(\sum_{w \in S} \Delta(u, w)\Pr(w, k, T)) \\
&\quad \text{(by induction, as } \Delta(u, w) > 0 \text{ implies } uRw) \\
&= K \sum_{w \in S}(\sum_{u \in U} \Delta(u, w)\Pr(w, k, T)) \\
&= K \sum_{w \in S}(\sum_{u \in U} \Delta(u, w))\Pr(w, k, T) \\
&= K \sum_{w \in S} \mathbf{P}(s_2, w)\Pr(w, k, T) \\
&= K\Pr(s_2, k+1, T)
\end{aligned}$$

Finally, we have

$$\begin{aligned}
\Pr(s_1, k+1, T) &= \sum_{v \in V} \mathbf{P}(s_1, v)\Pr(v, k, T) + \sum_{u \in U} \mathbf{P}(s_1, u)\Pr(u, k, T) \\
&\leq (1-K)\Pr(s_2, k+1, T) + K\Pr(s_2, k+1, T) \\
&= \Pr(s_2, k+1, T)
\end{aligned}$$

□

We can illustrate Theorem 5.2.7 by considering the Markov chain in Example 5.2.1 above and its fast simulation R. Because $s'Rt'$ and $\{w\}$ is upwards closed with respect to R, Theorem 5.2.7 tells us that $\Pr(s', n, \{w\}) \leq \Pr(t', n, \{w\})$, for every n. Direct calculation confirms these inequalities for $n \leq 5$:

n	$\Pr(s', n, \{w\})$	$\Pr(t', n, \{w\})$
0	0	0
1	$\frac{1}{2}$	$\frac{1}{2}$
2	$\frac{1}{2}$	$\frac{1}{2}$
3	$\frac{5}{8}$	$\frac{3}{4}$
4	$\frac{11}{16}$	$\frac{3}{4}$
5	$\frac{73}{96}$	$\frac{7}{8}$

We remark that the *universal relation* $R_U = S \times S$ is trivially a fast simulation. But under R_U the only upwards closed sets are $\emptyset$ and S itself, which means that Theorem 5.2.7 is uninteresting in that case.

Note that Theorem 5.2.7 also holds if R is a strong simulation, since every strong simulation is also a fast simulation. Interestingly, Theorem 5.2.7 *fails* if R is a weak simulation as defined in Definition 34 of [BKHW05]. Here is a counterexample:

Example 5.2.2 *Consider the Markov chain in Figure 5.8, from page 197 of [BKHW05]. If R is the reflexive closure of $\{(s, s'), (u_1, u_2)\}$, then R is a weak simulation and $\{w\}$ is upwards closed with respect to R, yet* $\Pr(s, 3, \{w\}) = 7/16$ *and* $\Pr(s', 3, \{w\}) = 6/16$*. Notice that in this case R is not a fast simulation.*

Now that we have developed fast simulation on abstract Markov chains, we are almost ready to apply it to our study of the stripping operation $\lfloor\cdot\rfloor$. It turns out, however, that when we study the behavior of $\lfloor\cdot\rfloor$ we will see that we do not need the great flexibility allowed by fast simulation as defined in Definition 5.2.4. We therefore introduce a restricted kind of fast simulation as follows:

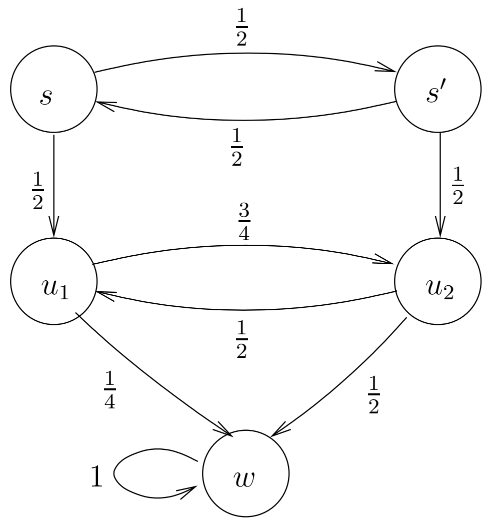


Figure 5.8: Another example Markov chain

Definition 5.2.8 *A binary relation R on S is a* simple fast simulation *if, whenever sRt, either*

1. *for every successor s' of s, we have $s'Rt$; or else*
2. *there is a bijection δ from the successors of s to the successors of t such that for every successor s' of s, we have $\mathbf{P}(s,s') = \mathbf{P}(t,\delta(s'))$ and $s'R\delta(s')$.*

Compared with fast simulation in Definition 5.2.4, we can see that here case 1 corresponds to the situation where all of the moves from s are "inessential," allowing us to take $U = \emptyset$. And case 2 corresponds to the situation where all of the moves from s are "essential," allowing us to take $V = \emptyset$; moreover we can use a very simple weight function that pairs up the successors of s and the successors of t in a one-to-one manner. Figure 5.9 illustrates case 2 for some states s and t where sRt. A distinct simulating state t_i must exist for each successor s_i of s and the probability p_i of reaching s_i must be matched by the probability of reaching t_i from t. Next we show that every simple fast simulation is indeed a fast simulation.

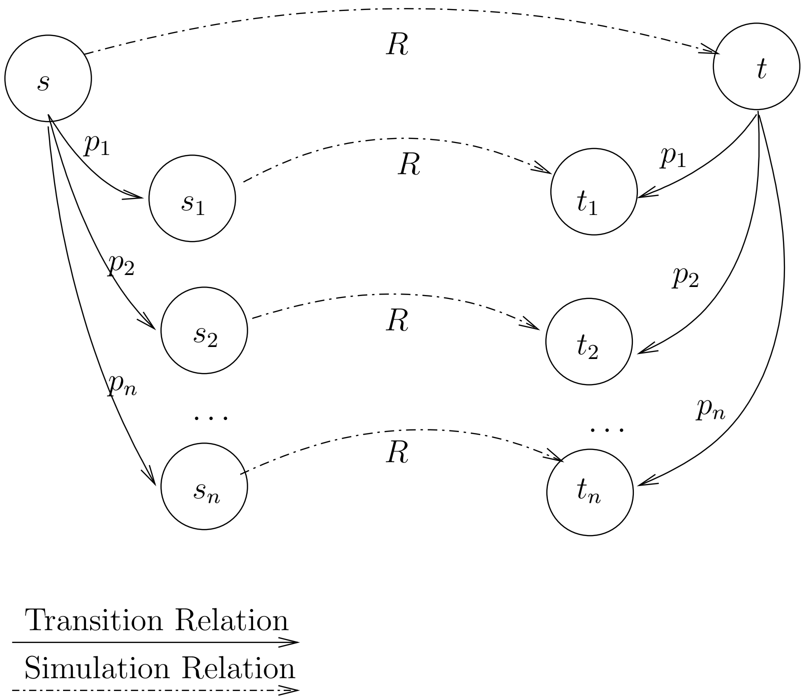


Figure 5.9: Simple Fast Probabilistic Simulation

Theorem 5.2.9 *Every simple fast simulation R is a fast simulation.*

Proof. Suppose that R is a simple fast simulation and that sRt.

If case 1 holds, then we satisfy Definition 5.2.4 by letting V be the set of successors of s and letting U be $\emptyset$. Note in this case that $K = 0$, so condition 2 of Definition 5.2.4 is satisfied vacuously.

If case 2 holds, then we let U be the set of successors of s and let V be $\emptyset$. Now condition 1 of Definition 5.2.4 is satisfied vacuously. To satisfy condition 2, note that $K = 1$ and define $\Delta(s', \delta(s')) = \mathbf{P}(s, s') = \mathbf{P}(t, \delta(s'))$. (All other values of Δ are 0.) It is straightforward to verify that Δ satisfies conditions 2(a), 2(b), and 2(c). $\square$

This concludes our treatment of *fast simulation* on abstract Markov chains. We are now ready to apply these results to our study of secure information flow. We do this next.

5.3 The Stripping Relation in a Probabilistic Setting

In this section, we formally define the stripping operation on well-typed commands in our random assignment language and use our results about fast simulation to prove a fundamental result about the relationship between the behavior of c and of $\lfloor c \rfloor$. Intuitively, $\lfloor c \rfloor$ eliminates all subcommands of c that contain no assignments to L variables; it is easy to see that this is the same as eliminating subcommands of type $H\ cmd$. More precisely, we have the following definition, an early version of which appeared in [SA06]:

Definition 5.3.1 (Stripping Function) *Let c be a well-typed command. We define $\lfloor c \rfloor = \textbf{done}$ if c has type $H\ cmd$; otherwise, define $\lfloor c \rfloor$ by*

- $\lfloor x := e \rfloor = x := e$
- $\lfloor x \overset{?}{\leftarrow} \mathcal{D} \rfloor = x \overset{?}{\leftarrow} \mathcal{D}$
- $\lfloor \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \rfloor = \textbf{if}\ e\ \textbf{then}\ \lfloor c_1 \rfloor\ \textbf{else}\ \lfloor c_2 \rfloor$
- $\lfloor \textbf{while}\ e\ \textbf{do}\ c \rfloor = \textbf{while}\ e\ \textbf{do}\ \lfloor c \rfloor$
- $\lfloor c_1; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \textit{if } c_1 : H\ cmd \\ \lfloor c_1 \rfloor & \textit{if } c_2 : H\ cmd \\ \lfloor c_1 \rfloor; \lfloor c_2 \rfloor & \textit{otherwise} \end{cases}$

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ and we extend $\lfloor \cdot \rfloor$ to well-typed configurations by $\lfloor (c, \mu) \rfloor = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

This definition is very close to that of Chapter 3, expect for the random command; it is derived from [SA07] but in that paper the stripping relation replaced subcommands of type H *cmd* with **skip**; in contrast our new definition here aggressively *eliminates* such subcommands in sequential compositions and replaces them with **done** in the other cases. Note that $\lfloor \mu \rfloor \sim_L \mu$. As expected, stripping removes all variables that are not typed L; here is a simple lemma for this:

Lemma 5.3.2 *For any command c, $\lfloor c \rfloor$ contains only L variables.*

Proof. By induction on the structure of c. If c has type H *cmd*, then $\lfloor c \rfloor =$ **done**, which (vacuously) contains only L variables. If c does not have type H *cmd*, then consider the form of c.

- If c is of the form $x := e$, then $\lfloor c \rfloor = x := e$. Since c does not have type H *cmd*, then by rule $update_t$ we must have that x is a L variable and $e : L$, which implies by Simple Security that e contains only L variables.

- The case of $x \overset{?}{\leftarrow} \mathcal{D}$ is similar.

- If c is of the form **if** e **then** c_1 **else** c_2, then $\lfloor c \rfloor =$ **if** e **then** $\lfloor c_1 \rfloor$ **else** $\lfloor c_2 \rfloor$. By rule if_t we have $e : L$, which implies by Simple Security that e contains only L variables. And, by induction, $\lfloor c_1 \rfloor$ and $\lfloor c_2 \rfloor$ contain only L variables.

- If c is of the form **while** e **do** c_1, then $\lfloor c \rfloor =$ **while** e **do** $\lfloor c_1 \rfloor$. By rule $while_t$, $e : L$, which implies by Simple Security that e contains only L variables. And, by induction, $\lfloor c_1 \rfloor$ contains only L variables.

- If c is of the form $c_1; c_2$ then if $c_1 : H$ *cmd* then $\lfloor c \rfloor = \lfloor c_2 \rfloor$ which by induction, contains only L variables. Similarly if $c_2 : H$ *cmd* then $\lfloor c \rfloor = \lfloor c_1 \rfloor$ which by induction, contains only L variables. Otherwise, $\lfloor c \rfloor = \lfloor c_1 \rfloor; \lfloor c_2 \rfloor$. In this case by two applications of induction, $\lfloor c_1 \rfloor$ and $\lfloor c_2 \rfloor$ contain only L variables.

□

5.4 Fast Low Probabilistic Simulation

We now specialize fast simulation from arbitrary Markov chains to the particular Markov chain of well-typed configurations (c, μ) of our random assignment language. In addition, we impose the requirement that the simulating configuration's memory must be low equivalent to the simulated configuration's memory:

Definition 5.4.1 *A binary relation R on well-typed configurations is a* fast low simulation *if R is a fast simulation such that whenever $(c_1, \mu_1)R(c_2, \mu_2)$, we have $\mu_1 \sim_L \mu_2$.*

Recalling that any function is also a relation, our stripping function can also be viewed as one. When we view it that way, we use the notation $\lfloor\cdot\rfloor$ to denote the stripping relation, and we write $(c_1, \mu_1)\lfloor\cdot\rfloor(c_2, \mu_2)$ to denote $\lfloor(c_1, \mu_1)\rfloor = (c_2, \mu_2)$.

We now are ready for the key result of this chapter which says that the relation $\lfloor\cdot\rfloor$ is a *fast low simulation* with respect to our language semantics. But first we give an example that illustrates how the behaviors of (c, μ) and $(\lfloor c\rfloor, \mu)$ can differ. Consider the program

$$\begin{array}{l}\textbf{while } h \textbf{ do } h \overset{?}{\leftarrow} \{0, 1\};\\ l := 1\end{array}$$

and its stripped version

$$l := 1$$

Notice that under memory $\{h = 1, l = 0\}$ the original program can run for an arbitrary number of steps; it has infinitely many terminating traces, whose probabilities sum to 1. In contrast, the stripped program always terminates in exactly one step. We now proceed to our key result which we recall from [SA11].

Theorem 5.4.2 $\lfloor\cdot\rfloor$ *is a fast low simulation.*

Proof. First note that if $(c, \mu)\lfloor\cdot\rfloor(d, \nu)$, then we have $\nu = \lfloor\mu\rfloor$, which implies $\mu \sim_L \nu$. More substantially, we must show that $\lfloor\cdot\rfloor$ is a fast simulation. By Theorem 5.2.9, it suffices to show the stronger result that $\lfloor\cdot\rfloor$ is a simple fast simulation. Note that the stronger result is *easier* to prove, because it gives us a stronger induction hypothesis to use in the case of $c_1; c_2$.[2]

Now suppose that $(c, \mu)\lfloor\cdot\rfloor(d, \nu)$, which means that that $d = \lfloor c\rfloor$ and $\nu = \lfloor\mu\rfloor$. We must show that either condition 1 or condition 2 of Definition 5.2.8 holds. We make this argument by induction on the structure of c.

First, if c has type $H\ cmd$, then $d = \mathbf{done}$. Now consider the (possibly infinite) set of successors of (c, μ):

$$\begin{array}{lcl} (c, \mu) & \xrightarrow{p_1} & (c_1, \mu_1) \\ (c, \mu) & \xrightarrow{p_2} & (c_2, \mu_2) \\ (c, \mu) & \xrightarrow{p_3} & (c_3, \mu_3) \\ \ldots & & \end{array}$$

For every i, by Subject Reduction we have $c_i : H\ cmd$, and by Confinement we have $\mu_i \sim_L \mu$. Hence we have $(c_i, \mu_i)\lfloor\cdot\rfloor(d, \nu)$. Thus we can see that condition 1 of Definition 5.2.8 is satisfied.

Next, if c does not have type $H\ cmd$, then consider the possible forms of c:

1. $c = x := e$.

   Here $d = c$. By $update_t$, $x : L\ var$ and $e : L$. So by Simple Security and the fact that $\nu = \lfloor\mu\rfloor$, we have $\mu(e) = \nu(e)$, which implies that $\nu[x := \nu(e)] = \lfloor\mu[x := \mu(e)]\rfloor$. Hence the move $(c, \mu)\xrightarrow{1}(\mathbf{done}, \mu[x := \mu(e)])$ is matched by the move $(d, \nu)\xrightarrow{1}(\mathbf{done}, \nu[x := \nu(e)])$. Formally, condition 2 of Definition 5.2.8 is satisfied by choosing $\delta((\mathbf{done}, \mu[x := \mu(e)])) = (\mathbf{done}, \nu[x := \nu(e)])$.

[2] This is an example of what George Pólya called the "Inventor's Paradox".

2. $c \;=\; x \overset{?}{\leftarrow} \mathcal{D}$.

   This case is similar to case 1; again $d = c$ and each move $(c, \mu)\xrightarrow{\mathcal{D}(v)}(\textbf{done}, \mu[x := v])$ is matched by $(d, \nu)\xrightarrow{\mathcal{D}(v)}(\textbf{done}, \nu[x := v])$. Formally, for each v such that $\mathcal{D}(v) > 0$, we choose $\delta((\textbf{done}, \mu[x := v])) = (\textbf{done}, \nu[x := v])$.

3. $c \;=\; \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2$.

   Here $d = \textbf{if}\ e\ \textbf{then}\ \lfloor c_1 \rfloor\ \textbf{else}\ \lfloor c_2 \rfloor$. By if_t, $e : L$ and by Simple Security $\mu(e) = \nu(e)$. So if $\mu(e) \neq 0$, then $(c, \mu)\xrightarrow{1}(c_1, \mu)$ is matched by $(d, \nu)\xrightarrow{1}(\lfloor c_1 \rfloor, \nu)$. Formally, we choose $\delta((c_1, \mu)) = (\lfloor c_1 \rfloor, \nu)$. The case when $\mu(e) = 0$ is similar.

4. $c \;=\; \textbf{while}\ e\ \textbf{do}\ c_1$.

   Here $d = \textbf{while}\ e\ \textbf{do}\ \lfloor c_1 \rfloor$. By while_t, $e : L$ and c_1 does not have type $H\ \mathit{cmd}$. By Simple Security, we have $\mu(e) = \nu(e)$. So if $\mu(e) \neq 0$, then the move $(c, \mu)\xrightarrow{1}(c_1; \textbf{while}\ e\ \textbf{do}\ c_1, \mu)$ is matched by $(d, \nu)\xrightarrow{1}(\lfloor c_1 \rfloor; \textbf{while}\ e\ \textbf{do}\ \lfloor c_1 \rfloor, \nu)$. (Notice that because c_1 does not have type $H\ \mathit{cmd}$, we have $\lfloor c_1; \textbf{while}\ e\ \textbf{do}\ c_1 \rfloor = \lfloor c_1 \rfloor; \textbf{while}\ e\ \textbf{do}\ \lfloor c_1 \rfloor$.)

   Formally, we choose $\delta((c_1; \textbf{while}\ e\ \textbf{do}\ c_1, \mu)) = (\lfloor c_1 \rfloor; \textbf{while}\ e\ \textbf{do}\ \lfloor c_1 \rfloor, \nu)$. The case when $\mu(e) = 0$ is similar.

5. $c \;=\; c_1; c_2$.

   First note that under rule $\mathit{compose}_s$, any move from $(c_1; c_2, \mu)$ results from a move

   $$(c_1, \mu)\xrightarrow{p}(c_1', \mu')$$

   where c_1' may or may not be **done**. If so, then the move is

   $$(c_1; c_2, \mu)\xrightarrow{p}(c_2, \mu')$$

   and if not, then the move is

   $$(c_1; c_2, \mu)\xrightarrow{p}(c_1'; c_2, \mu').$$

Now we split into subcases, depending on the types of c_1 and c_2.

(a) If c_1 has type $H\ cmd$, then $d = \lfloor c_2 \rfloor$. By Subject Reduction we have $c_1' : H\ cmd$, and by Confinement we have $\mu' \sim_L \mu$. Hence (d, ν) can match either of the possible moves from $(c_1; c_2, \mu)$ by doing nothing, since we have both $(c_2, \mu')\lfloor\cdot\rfloor(d, \nu)$ and $(c_1'; c_2, \mu')\lfloor\cdot\rfloor(d, \nu)$. So condition 1 of Definition 5.2.8 is satisfied.[3]

(b) If neither c_1 nor c_2 has type $H\ cmd$, then $d = \lfloor c_1 \rfloor; \lfloor c_2 \rfloor$. Define $d_1 = \lfloor c_1 \rfloor$ and $d_2 = \lfloor c_2 \rfloor$. Now, by induction (d_1, ν) can match the moves from (c_1, μ) so that either condition 1 or 2 of Definition 5.2.8 is satisfied. We consider these two possibilities in turn:

- If condition 1 is satisfied, then we always have $(c_1', \mu')\lfloor\cdot\rfloor(d_1, \nu)$. Now observe that d_1 cannot be **done**, since $d_1 = \lfloor c_1 \rfloor$ and c_1 does not have type $H\ cmd$. This implies that c_1' cannot have type $H\ cmd$ and in particular that c_1' is not **done**. Hence each move from c must be $(c_1; c_2, \mu)\xrightarrow{p}(c_1'; c_2, \mu')$. Moreover we have $(c_1'; c_2, \mu')\lfloor\cdot\rfloor(d_1; d_2, \nu)$. Hence condition 1 is satisfied for $(c_1; c_2, \mu)$ and $(d_1; d_2, \nu)$.

- If condition 2 is satisfied, then there is a bijection δ from the successors of (c_1, μ) to the successors of (d_1, ν). If we let (d_1', ν') denote $\delta((c_1', \mu'))$, then under condition 2 we have $(d_1, \nu)\xrightarrow{p}(d_1', \nu')$ and $(c_1', \mu')\lfloor\cdot\rfloor(d_1', \nu')$. Now consider c_1'.

[3] We remark that this is the critical case that prevents us from showing that $\lfloor\cdot\rfloor$ is a strong simulation. For example, if c is $h := 2; l := 3$ and μ is $\{h = 0, l = 1\}$, then d is $l := 3$ and ν is $\{l = 1\}$. In this case the move $(c, \mu)\xrightarrow{1}(l := 3, \{h = 2, l = 1\})$ cannot be matched by the move $(d, \nu)\xrightarrow{1}(\mathbf{done}, \{l = 3\})$, since the resulting memories are not low equivalent. Instead (d, ν) must match the move by doing nothing.

If $c_1' = \textbf{done}$, then the move from $(c_1; c_2, \mu)$ is $(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')$. In this case we have $d_1' = \textbf{done}$ as well, so we have the matching move $(d_1; d_2, \nu) \xrightarrow{p} (d_2, \nu')$.

And if $c_1' \neq \textbf{done}$, then the move from $(c_1; c_2, \mu)$ is $(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')$. This move is matched by $(d_1; d_2, \nu) \xrightarrow{p} (d_1'; d_2, \nu')$, but only if c_1' does not have type $H\ cmd$. For if $c_1' : H\ cmd$, then $\lfloor c_1'; c_2 \rfloor = \lfloor c_2 \rfloor = d_2 \neq d_1; d_2$. But in this case we have $d_1' = \textbf{done}$, which means that we actually have the matching move $(d_1; d_2, \nu) \xrightarrow{p} (d_2, \nu')$.[4]

(c) Finally, if c_1 does not have type $H\ cmd$ and $c_2 : H\ cmd$, then the argument is essentially the same as in case (b).

□

Now we are able to prove that if (c, μ) is well-typed and can terminate within at most n steps with its L variables having certain values, then $\lfloor (c, \mu) \rfloor$ can do the same, with probability at least as great. (This property was claimed, without proof, as Theorem 3.6 of [SA06].)

Let us say that a *low memory property* Φ is any property that depends only on the values of L variables. For example, if x and y are L variables, then "$x = 5$ and y is even" is a low memory property.

Theorem 5.4.3 (The Bucket Property) *Let c be well-typed and let Φ be a low memory property. For any n, the probability that (c, μ) terminates within n steps in a final memory satisfying Φ is less than or equal to the corresponding probability for $\lfloor (c, \mu) \rfloor$.*

[4] An example illustrating this scenario is when c is $(\textbf{if}\ 0\ \textbf{then}\ l := 1\ \textbf{else}\ h := 2); l := 3$. This goes in one step to $h := 2; l := 3$, which strips to $l := 3$. In this case, $\lfloor c \rfloor = (\textbf{if}\ 0\ \textbf{then}\ l := 1\ \textbf{else done}); l := 3$, which goes *in one step* to $l := 3$.

Proof. By Theorem 5.4.2 $\lfloor\cdot\rfloor$ is a *fast simulation*. Then by Definition 5.4.1, $\lfloor\cdot\rfloor$is a *fast simulation*. Now, let $T = \{(\mathbf{done}, \nu) \mid \nu \text{ satisfies } \Phi\}$. It is easy to see that T is upwards closed with respect to $\lfloor\cdot\rfloor$. For if $(\mathbf{done}, \nu_1) \in T$ and $(\mathbf{done}, \nu_1)\lfloor\cdot\rfloor(\mathbf{done}, \nu_2)$, then ν_1 satisfies Φ and $\nu_1 \sim_L \nu_2$, which implies that ν_2 also satisfies Φ. Therefore we can apply Theorem 5.2.7 to deduce that

$$\Pr((c, \mu), n, T) \leq \Pr((\lfloor c\rfloor, \lfloor\mu\rfloor), n, T)$$

for every n. □

We can extend this result to the case of *eventually* terminating in T, since the probability of eventually terminating in T is just $\lim_{n\to\infty} \Pr((c, \mu), n, T)$. We will use these results about $\lfloor c\rfloor$ in the next section.

5.5 Applications

Theorem 5.4.3 gives us the ability to quantify how the behavior of a well-typed program c can deviate from its stripped version $\lfloor c\rfloor$. But it also gives us a way to quantify how the behavior of c under memory μ can deviate from its behavior under ν, assuming that μ and ν are low equivalent. The reason is that, by Lemma 5.3.2, $\lfloor c\rfloor$ contains only L variables, which means that its behavior under μ must be *identical* to its behavior under ν. Hence we can build a "bridge" between (c, μ) and (c, ν):

$$(c, \mu) \xleftrightarrow{\text{Thm 5.4.3}} (\lfloor c\rfloor, \lfloor\mu\rfloor) \equiv (\lfloor c\rfloor, \lfloor\nu\rfloor) \xleftrightarrow{\text{Thm 5.4.3}} (c, \nu)$$

In this section, we develop several applications of these ideas.

First, suppose that c is well-typed and *probabilistically total*, which means that it halts with probability 1 from all initial memories. Then (c, μ)'s nontermination bucket is empty, which implies by Theorem 5.4.3 that (c, μ)'s buckets are identical to $(\lfloor c \rfloor, \lfloor \mu \rfloor)$'s buckets. Similarly, (c, ν)'s buckets are identical to $(\lfloor c \rfloor, \lfloor \nu \rfloor)$'s buckets. Hence (c, μ)'s buckets are identical to (c, ν)'s buckets. So we have proved the following corollary:

Corollary 5.5.1 *If c is well-typed and probabilistically total, then c satisfies probabilistic noninterference.*

This same result was proved in a different way as Corollary 3.5 of [SA06]; the proof there used a weak probabilistic bisimulation.

More interestingly, we can now prove an *approximate probabilistic noninterference* result for well-typed programs whose probability of nontermination is bounded.

Corollary 5.5.2 *Suppose that c is well-typed and fails to terminate from any initial memory with probability at most p. If μ and ν are low equivalent, then the deviation between the distributions of L outcomes under μ and under ν is at most $2p$.*

Proof. Since (c, μ)'s nontermination bucket contains at most p units of water, the sum of the absolute value of the differences between the L outcome buckets of (c, μ) and of $(\lfloor c \rfloor, \lfloor \mu \rfloor)$ is at most p. Similarly for (c, ν). Hence the sum of the absolute value of the the differences between the L outcome buckets of (c, μ) and of (c, ν) is at most $2p$. □

Notice that the program in Figure 5.1 achieves the upper bound of this corollary. From any initial memory, this program fails to terminate with probability at most $1/2$, so here $p = 1/2$. When $h = 0$, it terminates with $l = 0$ with probability $1/2$ and terminates with $l = 1$ with probability 0. When $h = 1$, it terminates with $l = 0$ with probability 0 and terminates with $l = 1$ with probability $1/2$. Hence the deviation between the two distributions of L outcomes is $|1/2 - 0| + |0 - 1/2| = 1$, which is $2p$. In general, applying Corollary 5.5.2 usefully requires a good bound p on the probability of nontermination; of course such bounds may be hard to obtain.

As an application of the approximate noninterference result, notice that an adversary $\mathcal{A}$, given the final values of c's L variables, might try to distinguish between initial memories μ and ν through statistical hypothesis testing. Assuming that the probability p of nontermination is small, then the approximate noninterference property gives us a way to bound $\mathcal{A}$'s ability to do this.[5]

In this chapter we extended the concept of *fast simulation* to a probabilistic setting, applied *stripping* to operate on a language with a random assignment command, and extended fast low simulation to a probabilistic semantics. Then, we used the theory to prove the language's termination insensitive probabilistic noninterference property. We have also shown that, under the Denning restrictions, well-typed probabilistic programs are guaranteed to satisfy an *approximate* probabilistic noninterference property, provided that their probability of nontermination is small. This property will be critical in Chapter 6, to prove computational noninterference on a language with cryptographic primitives.

Our proofs are based on a new notion of *fast simulation*, which builds on the work of Baier, Katoen, Hermanns, and Wolf [BKHW05] on strong and weak simulation on discrete and continuous Markov chains. The theorem that stripping is a fast simulation shows that the theory of probabilistic simulation can be applied fruitfully to the secure information flow problem, giving another proof technique in addition to the more common bisimulation-based approach.

[5] Similar ideas are considered (in the context of a process algebra) in the work of Di Pierro, Hankin, and Wiklicky [DPHW02].

CHAPTER 6

SECURE INFORMATION FLOW WITH ENCRYPTION.

In this Chapter, which is a revised version of [SA06], we extend the probabilistic language of Chapter 5 with encryption and decryption primitives which transform messages to ciphertexts and back. We model these primitives as algorithms and so they are only as effective as the algorithm's ability to hide the information in the message; this ability cannot be perfect. Nonetheless, our intuition is that *encrypting* a H plaintext yields a L ciphertext and decrypting a L (or H) ciphertext yields a H plaintext.

We craft a new type system for *secure information flow*. But while the language of Chapter 5 has an *approximate noninterference* property, this is not possible for a cryptographic language since the encryption and decryption algorithms cannot be perfectly effective. Instead we argue:

- that well-typed, polynomial-time programs in our language and type system extended with the *encryption* primitive satisfy *computational probabilistic noninterference* [BP02], provided that the encryption scheme is IND-CPA secure, and

- that well-typed, polynomial-time programs in our language and type system extended with the *encryption* and *decryption* primitives satisfy a *computational probabilistic noninterference* provided that the encryption scheme is IND-CCA secure.

It may seem that the first item above is subsumed by the second but the security property of the encryption scheme for the language with only encryption primitives (IND-CPA), is considerably weaker than that for the second language (IND-CCA). As we shall see there remains a question as to whether IND-CPA security would be sufficient for the soundness of the language with decryption primitives; we think that it would, but do not have a proof for this. In work by Peeter Laud [Lau02] relating formal methods to computational cryptography, IND-CCA security is equated to Dolev Yao with encryption cycles disallowed; as far as we know, IND-CPA security has no relative in the formal methods world.

Collecting requirements: throughout all of this work, we have classified variables as H (high) or L (low); an expression is classified as H if it contains any H variables; otherwise, it is classified as L. So far the Denning restrictions alone have been sufficient to achieve the desired security property; recalling them: first, to prevent *explicit flows*, a H expression may not be assigned to a L variable; second, to prevent *implicit flows*, an **if** or **while** command whose guard is H may not make *any* assignments to L variables. The crucial question, then, is whether a type system under this setting and which includes the intuition that the encryption of a secret is public, can be justified.

More precisely, let $\mathcal{E}$ and $\mathcal{D}$ denote encryption and decryption under some (properly generated and properly protected) shared key. We would like typing rules like:

- if e is H, then $\mathcal{E}(e)$ is L, and
- if e is either L or H, then $\mathcal{D}(e)$ is H.

But do these rules make sense? Clearly, they don't make sense if the encryption algorithm does not do a good job of hiding the information, but even if it did, it is not clear that the new type system would be sound as illustrated in the following example.

Example 6.0.1 *Consider a deterministic encryption scheme where the encryption algorithm is "perfect", i.e., a perfect* pseudo random permutation *which for any message-key pair always generates the same ciphertext (an example of this is a block* *cipher like AES in Electronic Code Book mode), then the following well-typed program can efficiently leak a secret.*

Let secret be a H n-bit variable, leak and mask be L variables, "|" denote bitwise-or, and "$\gg 1$" denote right shift by one bit. Then, the program:

```
$leak := 0$;
$mask := 2^{n-1}$;
while $mask \neq 0$ do (
    if $\mathcal{E}(secret \mid mask) = \mathcal{E}(secret)$ then
        $leak := leak \mid mask$;
    $mask := mask \gg 1$
)
```

copies secret to leak in time linear in n. This is because since $\mathcal{E}$ is a deterministic encryption function, then the test in the **if** *command is true iff $secret \mid mask = secret$. And this is true iff the bit of secret specified by mask is 1.*

Furthermore, under the Denning restrictions together with the above rule for typing $\mathcal{E}$, this program is well-typed—the guards of the **while** and **if** are both L, which means that the nested assignments to the L variables *leak* and *mask* are allowed.

Probabilistic encryption. It is well known that deterministic encryption is not sufficient for confidentiality in an stateless encryption scheme [BR05], and that one needs a probabilistic encryption algorithm, so that encrypting the same plaintext repeatedly yields different ciphertexts with high probability. Note that if $\mathcal{E}$ were probabilistic, the leaking program above would not work—then likely all the tests in the **if** command would be false, which means that *leak* would just end up with value 0.

We will use two strong properties that are used in the cryptographic community to define the security of symmetric encryption schemes, namely *IND-CPA* and *IND-CCA* security [BR05]. These abbreviations stand for "indistinguishability under chosen-plaintext attack" and "indistinguishability under chosen-ciphertext attack".

Security property of language and type system: from an information theoretic perspective, even the strongest security property on an encryption scheme is completely insecure in the sense that encrypting a message leaks the entire content into the ciphertext.

Example 6.0.2 *Let variable h be typed H, variable l be L, and consider the following well-typed program*

$$l \overset{?}{\leftarrow} \mathcal{E}(h)$$

which encrypts h and puts the result into l. We use $\overset{?}{\leftarrow}$ here to reflect the fact that the encryption algorithm ($\mathcal{E}$) is probabilistic. Probabilistic noninterference *requires that the probability distribution on the final values of L variables be* independent *of the initial values of H variables. That plainly cannot hold here, because any two distinct plaintexts must give rise to* disjoint *sets of possible ciphertexts—otherwise decryption would not be possible.*

Hence, as the classical *probabilistic noninterference* property is inappropriate for this setting, we shall seek a *computational probabilistic noninterference* result that says that, if program c is well-typed, then changes to the initial values of H variables lead to probability distributions on the final values of L variables that are *indistinguishable* to observers with limited computational resources. By its very definition, this property cannot hold for *all* well-typed programs. The following program exhaustively searches and discovers the implicit key K, then, uses it to leak the secret h:

$$\begin{array}{lll}
\textit{(phrases)} & p & ::= \; e \mid c \\
\textit{(expressions)} & e & ::= \; x \mid n \mid e_1 + e_2 \mid \ldots \\
\textit{(commands)} & c & ::= \; \textbf{done} \mid x := e \mid \\
 & & \; x \overset{?}{\leftarrow} \mathcal{D} \mid \\
 & & \; \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & \; \textbf{while } e \textbf{ do } c \mid c_1 ; c_2 \\
\textit{(variables)} & x, y, z, \ldots &
\end{array}$$

Figure 6.1: Probabilistic language syntax

1. Pick a few L plaintexts and encrypt each with $\mathcal{E}$, producing a few L ciphertexts.
2. Go through all possible keys, searching for one that successfully decrypts each of the ciphertexts. Decryption is done not by calling the decryption primitive $\mathcal{D}$, but by directly implementing the underlying decryption algorithm. Eventually the key will be found, and it can be stored in a L variable.
3. Encrypt the H variable h using $\mathcal{E}$, producing a L ciphertext.
4. Decrypt the ciphertext using the key found in Step 2, and write the result into the L variable l.

Of course, the running time of this program is exponential in the size of K. Therefore, our security property will apply only to programs that run in polynomial time in the size of K.

Recalling probabilistic language results: we now recall several results from Chapter 5 that will be needed as foundations for the cryptographic languages. First, are the syntax and semantic and typing rules; they are presented here in Figure 6.1, 6.2, and 6.3. Next are some properties of the type system:

Lemma 6.0.3 (Simple Security) *If $\Gamma \vdash e : \tau$, then e contains only variables of level τ or lower.*

Proof. By induction on the structure of e. □

$$(\mathit{update}_s) \qquad \frac{x \in \mathit{dom}(\mu)}{(x := e, \mu) \xrightarrow{1} (\mathbf{done}, \mu[x := \mu(e)])}$$

$$(\mathit{random}_s) \qquad \frac{x \in \mathit{dom}(\mu) \quad \mathcal{D}(v) > 0}{(x \overset{?}{\leftarrow} \mathcal{D}, \mu) \xrightarrow{\mathcal{D}(v)} (\mathbf{done}, \mu[x := v])}$$

$$(\mathit{if}_s) \qquad \frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \xrightarrow{1} (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \xrightarrow{1} (c_2, \mu)}$$

$$(\mathit{while}_s) \qquad \frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \xrightarrow{1} (\mathbf{done}, \mu)}$$

$$\frac{\mu_(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \xrightarrow{1} (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu)}$$

$$(\mathit{compose}_s) \qquad \frac{(c_1, \mu) \xrightarrow{p} (\mathbf{done}, \mu')}{(c_1; c_2, \mu) \xrightarrow{p} (c_2, \mu')}$$

$$\frac{(c_1, \mu) \xrightarrow{p} (c_1', \mu') \quad c_1' \neq \mathbf{done}}{(c_1; c_2, \mu) \xrightarrow{p} (c_1'; c_2, \mu')}$$

$$(\mathit{done}_s) \qquad (\mathbf{done}, \mu) \xrightarrow{1} (\mathbf{done}, \mu)$$

Figure 6.2: Structural Operational Semantics

Lemma 6.0.4 (Confinement) *If $\Gamma \vdash c : \tau\ cmd$, then c assigns only to variables of level τ or higher.*

Proof. By induction on the structure of c. □

Lemma 6.0.5 (Subject Reduction) *If $\Gamma \vdash c : \tau\ cmd$ and $(c, \mu) \xrightarrow{p} (c', \mu')$ for some $p > 0$, then $\Gamma \vdash c' : \tau\ cmd$.*

Proof. By induction on the structure of c. □

Next we recall a series of results from Chapter 5 that well-typed probabilistically total programs satisfy *probabilistic noninterference*. We shall need these formaliza-

(base) $\quad L \subseteq H$

$$(\mathit{cmd}) \qquad \frac{\tau \subseteq \tau'}{\tau'\ \mathit{cmd} \subseteq \tau\ \mathit{cmd}}$$

(reflex) $\quad \rho \subseteq \rho$

$$(\mathit{trans}) \qquad \frac{\rho_1 \subseteq \rho_2 \quad \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$(\mathit{subsump}) \qquad \frac{\Gamma \vdash p : \rho_1 \quad \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

(done_t) $\quad \Gamma \vdash \mathbf{done} : H\ \mathit{cmd}$

(int_t) $\quad \Gamma \vdash n : L$

$$(\mathit{rval}_t) \qquad \frac{\Gamma(x) = \tau\ \mathit{var}}{\Gamma \vdash x : \tau}$$

$$(\mathit{update}_t) \qquad \frac{\Gamma(x) = \tau\ \mathit{var} \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\ \mathit{cmd}}$$

$$(\mathit{random}_t) \qquad \frac{\Gamma(x) = \tau\ \mathit{var}}{\Gamma \vdash x \overset{?}{\leftarrow} \mathcal{D} : \tau\ \mathit{cmd}}$$

$$(\mathit{plus}_t) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$$

$$(\mathit{if}_t) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau\ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau\ \mathit{cmd}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ \mathit{cmd}}$$

$$(\mathit{while}_t) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau\ \mathit{cmd}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ \mathit{cmd}}$$

$$(\mathit{compose}_t) \qquad \frac{\Gamma \vdash c_1 : \tau\ \mathit{cmd} \quad \Gamma \vdash c_2 : \tau\ \mathit{cmd}}{\Gamma \vdash c_1; c_2 : \tau\ \mathit{cmd}}$$

Figure 6.3: Probabilistic Typing Rules

tions and theorems for proving the soundness of the cryptographic languages. We start with L-equivalent memories:

Definition 6.0.6 *Two memories μ and ν are L-*equivalent*, written $\mu \sim_L \nu$, if they agree on the values of all L variables.*

Definition 6.0.7 (Upwards Closed Set) *Let R be a binary relation on S. A set T of states is* upwards closed with respect to R *if, whenever $s \in T$ and sRs', we also have $s' \in T$.*

Next, we need the main result from Section 5.2 which says that in a Discrete Time Markov Chain, if a relation R is a *fast simulation* and there are two states s_1 and s_2 such that $s_1 R s_2$, then the probability of reaching an *upwards closed set T* from s_1 is at most that of s_2, i.e., s_2 gets to its destination faster or as fast as s_1.

Theorem 6.0.8 (Reachability) *If R is a fast simulation, T is upwards closed with respect to R, and $s_1 R s_2$, then $\Pr(s_1, n, T) \leq \Pr(s_2, n, T)$ for every n.*

We also need the concept of a stripping relation in a probabilistic setting from Section 5.3 to help us "remove" high from low computation:

Definition 6.0.9 *Let c be a well-typed command. We define $\lfloor c \rfloor =$ **done** if c has type H cmd; otherwise, define $\lfloor c \rfloor$ by*

- $\lfloor x := e \rfloor = x := e$
- $\lfloor x \xleftarrow{?} \mathcal{D} \rfloor = x \xleftarrow{?} \mathcal{D}$
- $\lfloor \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rfloor = \textbf{if } e \textbf{ then } \lfloor c_1 \rfloor \textbf{ else } \lfloor c_2 \rfloor$
- $\lfloor \textbf{while } e \textbf{ do } c \rfloor = \textbf{while } e \textbf{ do } \lfloor c \rfloor$
- $\lfloor c_1; c_2 \rfloor = \begin{cases} \lfloor c_2 \rfloor & \textit{if } c_1 : H \ cmd \\ \lfloor c_1 \rfloor & \textit{if } c_2 : H \ cmd \\ \lfloor c_1 \rfloor ; \lfloor c_2 \rfloor & \textit{otherwise} \end{cases}$

Also, we define $\lfloor \mu \rfloor$ to be the result of deleting all H variables from μ and we extend $\lfloor \cdot \rfloor$ to well-typed configurations by $\lfloor (c, \mu) \rfloor = (\lfloor c \rfloor, \lfloor \mu \rfloor)$.

Next, we recall, *fast low simulation* to capture the idea that a state can simulate another as far as the low variables are concerned.

Definition 6.0.10 *A binary relation R on configurations is a fast low simulation if R is a* fast simulation *and whenever $(c_1, \mu_1)R(c_2, \mu_2)$, $\mu_1 \sim_L \mu_2$.*

Next we recall, the key result from Chapter 5: to capture the idea that the *stripped* version of a configuration is able to simulate the original configuration.

Theorem 6.0.11 $\lfloor \cdot \rfloor$*is a fast low simulation.*

Next we recall the result that says that the stripped version of a configuration can reach a terminal state faster (or as fast) as the original configuration. Let a *low memory property* Φ be any property that depends only on the values of L variables. For example, if x and y are L variables, then "$x = 5$ and y is even" is a low memory property.

Theorem 6.0.12 (The Bucket Property) *Let c be well-typed and let Φ be a low memory property. For any n, the probability that (c, μ) terminates within n steps in a final memory satisfying Φ is less than or equal to the corresponding probability for $\lfloor (c, \mu) \rfloor$.*

This result is valid in the case of *eventually* terminating in T, since the probability of eventually terminating in T is just $\lim_{n \to \infty} \Pr((c, \mu), n, T)$. We have recalled a number of results on the probabilistic language of Chapter 5 as a preamble to applying them to our cryptographic languages. Finally, we have the probabilistic noninterference result on our language:

6.1 Elements of Cryptographic Security

In this section we recall some elements of cryptographic theory mainly from the book Introduction to modern cryptography by Bellare and Rogaway [BR05]. We start with the definition of *symmetric encryption schemes* and *IND-CPA security* [BDJR97, BR05]:

Definition 6.1.1 *A* symmetric encryption scheme $\mathcal{SE}$ *with security parameter* k *is a triple of algorithms* $(\mathcal{K}, \mathcal{E}, \mathcal{D})$*, where*

- $\mathcal{K}$ *is a randomized key-generation algorithm that generates a* k*-bit key; we write* $K \overset{?}{\leftarrow} \mathcal{K}$*.*
- $\mathcal{E}$ *is a randomized encryption algorithm that takes a key and a plaintext and returns a ciphertext; we write* $C \overset{?}{\leftarrow} \mathcal{E}_K(M)$*.*
- $\mathcal{D}$ *is a deterministic decryption algorithm that takes a key and a ciphertext and returns a plaintext; we write* $M := \mathcal{D}_K(C)$*.*

Next, we recall the concept of *IND-CPA security* (*indistinguishability under chosen-plaintext attack* [BR05] pg. 102). Intuitively, a symmetric encryption scheme $\mathcal{SE}$ is *IND-CPA secure* if, given limited computational resources, there is no way to distinguish the ciphertexts of two different messages unless the encryption key is known.

Formally an adversary $\mathcal{A}$ exists within a setting or a *world* which is either 0 or 1 (there are only two possible worlds) and is denoted by b. In each world there is an *LR oracle* which is of the form

$$\mathcal{E}_K(\mathrm{LR}(\cdot, \cdot, b)),$$

where K is a randomly generated key. The *LR oracle* in *World 0* receives a pair of equal-length messages (M_0, M_1), encrypts M_0 using $\mathcal{E}_K$, and returns the resulting ciphertext. The *LR oracle* in *World 1* does the same thing using M_1. $\mathcal{A}$'s job is to produce a guess of which world it is in; it is able to use the oracle as much as it wants, so since $\mathcal{A}$ queries the oracle with pairs of messages, it gets back encryptions of the *left* messages if $\mathcal{A}$ is in World 0 ($b = 0$) or else the *right* messages if $\mathcal{A}$ is in World 1 ($b = 1$). As $\mathcal{A}$ is an algorithm, when it is finished its execution, it returns a bit with its guess of its world. Intuitively, if the encryption scheme is effective, $\mathcal{A}$ will not be able to guess which world it is in with a significant probability, but we have to be careful how we measure $\mathcal{A}$'s ability to guess its world. For example: $\mathcal{A}$ could simply return 1 on any input; then, it will always guess right if it is in World 1 regardless of how strong $\mathcal{SE}$ is. Formally, $\mathcal{A}$ is executed in two *experiments*:

```
Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-1}}(\mathcal{A})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $d \overset{?}{\leftarrow} \mathcal{A}^{\mathcal{E}_K(\text{LR}(\cdot,\cdot,1))}$;
    return $d$

Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-0}}(\mathcal{A})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $d \overset{?}{\leftarrow} \mathcal{A}^{\mathcal{E}_K(\text{LR}(\cdot,\cdot,0))}$;
    return $d$
```

Definition 6.1.2 *The* IND-CPA advantage *of $\mathcal{A}$ is defined as*

$$\begin{aligned}\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cpa}}(\mathcal{A}) = \ & \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-1}}(\mathcal{A}) = 1] - \\ & \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-0}}(\mathcal{A}) = 1].\end{aligned}$$

We say that $\mathcal{SE}$ is *IND-CPA secure* if there is no adversary $\mathcal{A}$ running in polynomial time in the security parameter k that can achieve a non-negligible advantage.

Next, we recall the concept of *IND-CCA security* (*indistinguishability under chosen-ciphertext attack* [BR05] pg. 127). Like the IND-CPA adversary, IND-CCA adversary $\mathcal{A}$ is uncertain about which of the two worlds (World 0 or World 1) it is in. $\mathcal{A}$ retains access to the LR-encryption-oracle $\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,b))$ of its counterpart; but now $\mathcal{A}$ has access to a new oracle: a decryption oracle ($\mathcal{D}_K(\cdot)$) which is unavailable to the IND-CPA adversary and which can be used to decrypt any ciphertext *except* one that has been obtained using the LR-encryption oracle; otherwise, $\mathcal{A}$ would trivially determine its world by:

```
$\boxed{\text{-}\Delta_7\text{-}}$
    $c \overset{?}{\leftarrow} \mathcal{E}_K(\mathrm{LR}(0,1,b))$;
    $world := \mathcal{D}_K(c)$
```

So, to prevent $\mathcal{A}$ from winning trivially, it is forbidden from querying the decryption oracle on any ciphertext that it previously received from the LR-oracle. Aside from these differences, the definitions of $\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cca-1}}(\mathcal{A})$ and $\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cca-0}}(\mathcal{A})$, of the *IND-CCA advantage* $\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cca}}(\mathcal{A})$, and of *IND-CCA security* are just like the definitions in the IND-CPA case: $\mathcal{A}$ is executed in two *experiments*:

```
Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cca-1}}(\mathcal{A})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $d \overset{?}{\leftarrow} \mathcal{A}^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1)),\mathcal{D}_K(\cdot)}$;
    If $\mathcal{A}$ queried $\mathcal{D}_K(\cdot)$ on a ciphertext previously returned by $\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,1))$
        then return 0
        else return $d$
```

```
Experiment $\mathbf{Exp}^{\text{ind-cca-0}}_{\mathcal{SE}}(\mathcal{A})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $d \overset{?}{\leftarrow} \mathcal{A}^{\mathcal{E}_K(\text{LR}(\cdot,\cdot,0))}$;
    If $\mathcal{A}$ queried $\mathcal{D}_K(\cdot)$ on a ciphertext previously returned by $\mathcal{E}_K(\text{LR}(\cdot,\cdot,0))$
        then return 0
        else return $d$
```

Definition 6.1.3 *The* IND-CCA advantage *of $\mathcal{A}$ is defined as*

$$\begin{aligned}\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(\mathcal{A}) = \;& \Pr[\mathbf{Exp}^{\text{ind-cca-1}}_{\mathcal{SE}}(\mathcal{A}) = 1] - \\ & \Pr[\mathbf{Exp}^{\text{ind-cca-0}}_{\mathcal{SE}}(\mathcal{A}) = 1].\end{aligned}$$

We say that $\mathcal{SE}$ is *IND-CPA secure* if there is no adversary $\mathcal{A}$ running in polynomial time in the security parameter k that can achieve a non-negligible advantage. (As usual, $s(k)$ is *negligible* if for any positive polynomial $p(k)$, there exists k_0 such that $s(k) \leq \frac{1}{p(k)}$, for all $k \geq k_0$.).

Finally we need a concept of *computational probabilistic noninterference* property. Work on this area was first introduced by Backes and Pfitzmann [BP02]; we use formulations similar to Laud's [Lau03, LV05]. Suppose that a well-typed, polynomial-time program c is run twice, under two L-equivalent memories μ and ν. Upon termination, the pair of resulting L memories is fed into an adversary program which tries to discern which memory was used. If there is no adversary capable of guessing the correct memory with probability non-negligibly greater than 1/2, then the system has *computational probabilistic noninterference* property[1].

[1] Note that "the system" here means the language, type system and the encryption scheme.

6.2 A Language With Encryption

In this section, we first extend our probabilistic language with a symmetric encryption primitive $\mathcal{E}$ and extend the type system with a rule that says that the encryption of a H expression is L. We argue that our new language has *computational probabilistic noninterference* property if the encryption scheme is *IND-CPA secure*. Then we further extend the language with a decryption primitive and argue *computational probabilistic noninterference* if the encryption scheme is *IND-CCA secure*. Note that we do not include key generation and manipulation in our language. Instead, we assume that a single key K is generated before executing the program and is used implicitly in all encryption and decryption operations. Before proceeding, we need to address issues of message length which can make our language unsound. Our IND-CPA and IND-CCA secure symmetric encryption are *message-length revealing*, because the size of the ciphertext depends on the size of the plaintext.

Example 6.2.1 *Consider an encryption scheme that uses* cipher-block chaining *with a random initial vector. Under this scheme, an m-block plaintext would encrypt to an $(m+1)$-block ciphertext. Then, the following program could be used to leak a secret efficiently:*

```
if $secret \% 2 = 0$ then
    $h :=<$ some 1-block plaintext $>$
else
    $h :=<$ some 2-block plaintext $>$;
$l \overset{?}{\leftarrow} \mathcal{E}(h)$;
if $l$ is 2 blocks long then $leak := 0$
else $leak := 1$
```

This code leaks the last bit of secret, yet it is well-typed (using our intended typing rule for encryption) if secret and h are H and l and leak are L.

Laud and Vene [LV05] address this difficulty by assuming that encryption is *length concealing*, as defined by Abadi and Rogaway [AR00]. Here we adopt an operationally equivalent restriction: we assume that all integer values in our language are n bits long, for some n (perhaps 128). This way, we never encrypt plaintexts of different sizes.

Intuitively, the probabilistic encryption operation is composed of two parts: a pseudo random permutation of the message and the injection of random information into the result. A message of n bits permutes to a ciphertext of the same length; it seems only natural to inject an amount of random information equal to n bits, making the result of an encryption $2n$-bits long. Also in practice, if n is the block size of a block cipher, and we use CBC\$ mode (cipher block chaining with random initial vector) or CTR\$ mode (counter mode with random initial vector) [BR05], then our ciphertexts will be two n-bit blocks long. Hence, here is the syntax we choose for the encryption operation in our language:

$$(x, y) \overset{?}{\leftarrow} \mathcal{E}(e)$$

Next we extend the language syntax of Figure 6.1 with our new encryption command which encrypts the n-bit value of e with the implicit key K and produces $2n$-bit ciphertext; it then puts the first n bits into x and the second n bits into y. Here is the typing and semantic rules for encryption, which we add to the rules in Figure 6.3 and Figure 6.2 respectively:

$$(\mathit{encrypt}_s)\quad \frac{x, y \in \mathit{dom}(\mu) \quad \mathcal{D}_{\mathcal{E}(e)}(v, u) > 0}{((x, y) \overset{?}{\leftarrow} \mathcal{E}(e), \mu) \overset{\mathcal{D}_{\mathcal{E}(e)}(v,u)}{\longrightarrow} (\mathbf{done}, \mu[(x, y) := (v, u)])}$$

$$(\mathit{encrypt}_t) \quad \frac{\begin{array}{l}\Gamma(x) = \tau_1\ \mathit{var} \quad \Gamma(y) = \tau_2\ \mathit{var} \\ \Gamma \vdash e : H \\ \tau \subseteq \tau_1 \quad \tau \subseteq \tau_2\end{array}}{\Gamma \vdash (x, y) \overset{?}{\leftarrow} \mathcal{E}(e) : \tau\ \mathit{cmd}}$$

The semantic rule for encryption requires that under any given key, the encryption of any particular value must have a fixed distribution, $\mathcal{D}_{\mathcal{E}(e)}$; this is a natural enough requirement and is the way encryption works in practice. Then given that a pair of values have a corresponding probability p in the distribution and that there is a pair of variables in memory, the encryption command simply assigns the values to the memories with probability p. These rules formalize our intuition that the encryption of a H expression e is to be assigned to a pair of variables x and y, regardless of whether they are H or L[2].

We now argue that extending the language and type system with the encryption command and with the rules $\mathit{encrypt}_t$ and $\mathit{encrypt}_s$ is *sound*. We begin by showing that no well-typed program can (with probability significantly greater than $\frac{1}{2}$) efficiently leak a randomly-chosen, 1-bit H variable h into a L variable l, provided that encryption is IND-CPA secure. Formally, we define a *leaking adversary* $\mathcal{B}$ as a program that contains a H variable h, a L variable l, and other variables that can be typed arbitrarily. $\mathcal{B}$ is executed in the following experiment:

[2] To control implicit flows, however, the type of the command must record the minimum level of variable that is assigned to; this is the purpose of the last two hypotheses in the rule.

```
Experiment $\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $h_0 \overset{?}{\leftarrow} \{0,1\}$;
    $h := h_0$;
    initialize all other variables of $\mathcal{B}$ to 0;
    run $\mathcal{B}^{\mathcal{E}_K(\cdot)}$;
    if $l = h_0$ then return 1 else return 0
```

$\mathcal{B}$'s purpose is to leak h to l; the experiment measures $\mathcal{B}$'s effectiveness in doing this. In the experiment a proper key and a random 1-bit secret are generated, the secret is transferred to the program variable h (while h_0 does not occur in $\mathcal{B}$); then, the *leaking adversary* $\mathcal{B}$ is executed; when $\mathcal{B}$ stops, its guess is tested against the original secret. We define the leaking advantage of $\mathcal{B}$ as follows:

Definition 6.2.1 *Given leaking adversary $\mathcal{B}$ in the random assignment language with encryption, the* leaking advantage *of $\mathcal{B}$ is*

$$\mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 2 \cdot \Pr[\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 1] - 1.$$

Note that by a random guess, $\mathcal{B}$ can guess h with probability $\frac{1}{2}$, so we adjust our metric to measure advantage only in the range from $\frac{1}{2}$ to 1. We now proceed with the key theorem of this chapter which says that if $\mathcal{B}$ is effective in leaking its secret then we we can use it to break the underlying encryption scheme.

Theorem 6.2.2 (IND-CPA Reduction) *Given a well-typed leaking adversary $\mathcal{B}$ (in the random assignment language with encryption) that runs in polynomial time $p(k)$, there exists an IND-CPA adversary $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\text{ind-cpa}}_{\mathcal{SE}}(\mathcal{A}) \geq \frac{1}{2} \cdot \mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}).$$

Moreover, $\mathcal{A}$ runs in $O(p(k))$ time.

Proof. From $\mathcal{B}$, we construct an IND-CPA adversary $\mathcal{A}$ that runs $\mathcal{B}$ with a randomly-chosen 1-bit value of h. Whenever $\mathcal{B}$ makes a call $\mathcal{E}(e)$ to its encryption primitive, $\mathcal{A}$ passes $(0^n, e)$ to its LR oracle and returns the result to $\mathcal{B}$. When $\mathcal{B}$ terminates, $\mathcal{A}$ checks whether $\mathcal{B}$ has succeeded in leaking h to l; if so, $\mathcal{A}$ guesses that it is in World 1; if not, $\mathcal{A}$ guesses that it is in World 0. Also, for reasons that will be explained shortly, $\mathcal{A}$ limits $\mathcal{B}$'s execution to $p(k)$ steps, returning 0 if $\mathcal{B}$ does not terminate within that time. Formally, $\mathcal{A}$ is defined as:

```
Adversary $\mathcal{A}^{\mathcal{E}_K(\mathrm{LR}(\cdot,\cdot,b))}$
  $h_0 \overset{?}{\leftarrow} \{0,1\}$;
  $h := h_0$;
  initialize all other variables of $\mathcal{B}$ to 0;
  run $\mathcal{B}^{\mathcal{E}_K(\mathrm{LR}(0^n,\cdot,b))}$ for at most $p(k)$ steps;
  **if** $\mathcal{B}$ did not terminate **then return** 0;
  **if** $l = h_0$ **then return** 1 **else return** 0
```

Cryptanalysis, World 1: first note that $\mathcal{A}$ runs in $O(p(k))$ time; if $\mathcal{A}$ is in World 1, then $\mathcal{B}$ is run faithfully—each encryption $\mathcal{E}(e)$ is converted to $\mathcal{E}_K(\mathrm{LR}(0^n, e, 1))$, which returns $\mathcal{E}_K(e)$. Also $p(k)$ steps are enough for $\mathcal{B}$ to terminate, by assumption. Hence $\mathcal{A}$ returns 1 precisely if $\mathcal{B}$ succeeds in its leaking experiment. Hence

$$\begin{aligned}\Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-1}}(\mathcal{A}) = 1] &= \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{leak}}(\mathcal{B}) = 1]\\ &= \tfrac{1}{2} \cdot \mathbf{Adv}_{\mathcal{SE}}^{\text{leak}}(\mathcal{B}) + \tfrac{1}{2}\end{aligned}$$

World 0: if $\mathcal{A}$ is in World 0, $\mathcal{B}$ is *not* run faithfully—each encryption $\mathcal{E}(e)$ is converted to $\mathcal{E}_K(\mathrm{LR}(0^n, e, 0))$, which returns $\mathcal{E}_K(0^n)$. This has nothing to do with e—it just returns a $2n$-bit value from some fixed probability distribution (depending only on K). So in this case, $\mathcal{B}$ is actually run as $\mathcal{B}^{\mathcal{E}_K(0^n)}$, a well-typed program in the random assignment language of Chapter 5; but now we have to deal with information leaks due to nontermination, i.e., $\mathcal{B}^{\mathcal{E}_K(0^n)}$ might now run forever or at least *longer* than polynomial time. For example: $\mathcal{B}$ might call $\mathcal{E}(0^n)$ and $\mathcal{E}(1^n)$ and test whether the two ciphertexts are equal. In $\mathcal{B}^{\mathcal{E}_K(\cdot)}$, the test would certainly be false, but in $\mathcal{B}^{\mathcal{E}_K(0^n)}$, it would be true with a small nonzero probability[3]; a new execution path is created by the change to World 0 which can allow nontermination. So as much as we would like to, it is not possible to claim that the language in World 0 has *probabilistic noninterference*; if it was so, then

$$\Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-0}}(\mathcal{A}) = 1] = \frac{1}{2}$$

However, recalling Theorem 6.0.12, we argue that the probability above is upper bounded by $\frac{1}{2}$; the intuition here is that whenever $\mathcal{B}$ loops, $\mathcal{A}$ knows that it is in World 0 and using this information, $\mathcal{A}$ may be less likely to guess World 1.

Let μ_0 and μ_1 be identical memories except that $\mu_0[h] = 0$ and $\mu_1[h] = 1$. Let upwards closed sets $T_0 = \{(\mathbf{done}, \nu) | \nu[l] = 0\}$ and $T_1 = \{(\mathbf{done}, \nu) | \nu[l] = 1\}$; these sets contain the terminal configurations for when $\mathcal{B}^{\mathcal{E}_K(0^n)}$ guesses 0 or 1 respectively. Note that $\mathcal{A}$ guesses World 1 precisely when running $\mathcal{B}^{\mathcal{E}_K(0^n)}$ on memory μ_0 reaches T_0 within $p(k)$ steps or when running $\mathcal{B}^{\mathcal{E}_K(0^n)}$ on memory μ_1 reaches T_1 within $p(k)$ steps. Let p_0 be the probability of $(\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_0)$ reaching T_0 within $p(k)$ steps and let p_1 be the probability of $(\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_1)$ reaching T_1 within $p(k)$ steps.

Now, since $\lfloor(\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_0)\rfloor = \lfloor(\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_1)\rfloor$, let p'_0 and p'_1 be the probabilities of reaching T_0 and T_1 (within $p(k)$ steps) respectively from the stripped initial configuration. Note that $p'_0 + p'_1 \leq 1$. Then, by Theorem 6.0.8 (Reachability)

$$\Pr((\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_0), p(k), T_0) \leq \Pr((\lfloor \mathcal{B}^{\mathcal{E}_K(0^n)} \rfloor, \lfloor \mu_0 \rfloor), p(k), T_0)$$

which implies that $p_0 \leq p'_0$ and

$$\Pr((\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_1), p(k), T_1) \leq \Pr((\lfloor \mathcal{B}^{\mathcal{E}_K(0^n)} \rfloor, \lfloor \mu_1 \rfloor), p(k), T_1)$$

[3] Note that this test does not help $\mathcal{B}$ leak h.

which implies that $p_1 \leq p'_1$. Therefore,

$$\begin{aligned}
\Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cpa-0}}(\mathcal{A}) = 1] &= \tfrac{1}{2}\Pr[(\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_0)] + \tfrac{1}{2}\Pr[(\mathcal{B}^{\mathcal{E}_K(0^n)}, \mu_1)] \\
&= \frac{1}{2}p_0 + \frac{1}{2}p_1 \\
&\leq \frac{1}{2}p'_0 + \frac{1}{2}p'_1 \\
&\leq \frac{1}{2}
\end{aligned}$$

then, by Definition 6.1.2,

$$\begin{aligned}
\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cca}}(\mathcal{A}) = \quad & \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cca-1}}(\mathcal{A}) = 1] - \\
& \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cca-0}}(\mathcal{A}) = 1].
\end{aligned}$$

and substituting above results

$$\begin{aligned}
\mathbf{Adv}_{\mathcal{SE}}^{\text{ind-cca}}(\mathcal{A}) &\geq \Pr[\mathbf{Exp}_{\mathcal{SE}}^{\text{ind-cca-1}}(\mathcal{A}) = 1] - \frac{1}{2} \\
&\geq \frac{1}{2} \cdot \mathbf{Adv}_{\mathcal{SE}}^{\text{leak}}(\mathcal{B}) + \frac{1}{2} - \frac{1}{2} \\
&\geq \frac{1}{2} \cdot \mathbf{Adv}_{\mathcal{SE}}^{\text{leak}}(\mathcal{B}).
\end{aligned}$$

□

Corollary 6.2.3 *If $\mathcal{SE}$ is IND-CPA secure, then there is no polynomial-time, well-typed leaking adversary $\mathcal{B}$ that achieves a non-negligible advantage.*

Proof. Given such a $\mathcal{B}$ with non-negligible leaking advantage $s(k)$, then by Theorem 6.2.2 there exists polynomial-time $\mathcal{A}$ with IND-CPA advantage $s(k)/2$, contradicting the IND-CPA security of $\mathcal{SE}$. □

Computational probabilistic noninterference: Our final result for this language and type system is to establish the standard *computational probabilistic noninterference* property [BP02]. We argue that after running a well-typed program c under two memories (μ and ν) with different H values, there is no polynomial time program $\mathcal{N}$ that inputs the final L values of c and can distinguish with a significant probability the execution using the first memory from the execution using the second memory; we argue this by reduction.

Let μ and ν be L-equivalent memories and let c be a well-typed command in our language and type system with encryption. Let $\mathcal{N}$ be a *noninterference adversary* which does not refer to H variables and generates a guess (0 or 1) into the new variable g; $\mathcal{N}$'s purpose is to examine the final low memory after the execution of well-typed program c and to generate a guess as to which initial memory (μ or ν) was used; $\mathcal{N}$ is executed in the following experiment, where h_0 is a new variable:

```
Experiment $\mathbf{Exp}^{\mathrm{NI}}_{\mathcal{SE},c,\mu,\nu}(\mathcal{N})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $h_0 \overset{?}{\leftarrow} \{0,1\}$;
    if $h_0 = 0$ then initialize the variables of $c$ to $\mu$
                  else initialize the variables of $c$ to $\nu$;
    $c$; $\mathcal{N}$;
    if $g = h_0$ then return 1 else return 0
```

Next formalizing $\mathcal{N}$'s effectiveness: the *noninterference advantage* of $\mathcal{N}$ is defined as

$$\mathbf{Adv}^{\mathrm{NI}}_{\mathcal{SE},c,\mu,\nu}(\mathcal{N}) = 2 \cdot \Pr[\mathbf{Exp}^{\mathrm{NI}}_{\mathcal{SE},c,\mu,\nu}(\mathcal{N}) = 1] - 1.$$

again, this is because $\mathcal{N}$ can guess correctly which memory was used with probability $\frac{1}{2}$ and so $\mathcal{N}$'s effectiveness corresponds to guessing correctly with probability ranging from $\frac{1}{2}$ to 1. We are ready to argue that if $\mathcal{N}$ is able to effectively guess which initial memory was used then we can construct a well-typed leaking adversary $\mathcal{B}$ that is similarly effective.

Theorem 6.2.4 *If c is a well-typed, polynomial-time program and μ and ν are two L-equivalent memories, then no polynomial-time noninterference adversary $\mathcal{N}$ for c, μ, and ν can achieve a significant noninterference advantage.*

Proof. Assuming that such a noninterference adversary $\mathcal{N}$ exists, we can build a leaking adversary $\mathcal{B}$ which uses two new variables h and l, of type H and L, respectively. $\mathcal{B}$ is defined as follows:

```
Adversary $\mathcal{B}$
    initialize the $L$ variables of $c$ according to $\mu$ and $\nu$;
    if $h = 0$ then initialize the $H$ variables of $c$ according to $\mu$
        bidelse initialize the $H$ variables of $c$ according to $\nu$;
    $c$; $\mathcal{N}$;
    $l := g$
```

Note that $\mathcal{B}$ runs in polynomial time. We argue that $\mathcal{B}$ is well-typed and that $\mathcal{B}$'s *leaking advantage* is the same as $\mathcal{N}$'s. The initialization code is well-typed under rules update_t and if_t. (this would not be true if μ and ν were not L-equivalent, because then the initialization of the L variables of c would depend on h.) Next, c is well-typed by hypothesis but $\mathcal{N}$ looks like a regular program. We can argue that $\mathcal{N}$ is well-typed. This does *not* follow by hypothesis—$\mathcal{N}$ should be thought of as a *passive adversary*, which cannot be assumed to be well-typed. But because $\mathcal{N}$ cannot refer to the H variables of c, we can give all of its variables type L, which makes it *automatically* well-typed under our rules. Finally, the leaking advantage of $\mathcal{B}$ is non-negligible, because it is the same as the noninterference advantage of $\mathcal{N}$. This contradicts Corollary 6.2.3. □

6.3 A Language With Encryption and Decryption

We now argue that further extending our language and type system with a decryption primitive and with the rule *decrypt*$_t$ is *sound*. Our decryption primitive $\mathcal{D}(e_1, e_2)$ decrypts under the implicit key K just like the language's encryption command and since $\mathcal{E}$ takes an n-bit plaintext to a $2n$-bit ciphertext, then our decryption primitive $\mathcal{D}$ shall take two n-bit expressions as input (one expression for each of the n bits blocks of a ciphertext) and return an n-bit plaintext. Since decryption is deterministic, we can model it simply as a new expression and add it to our language syntax in Figure 6.1. Here is the typing rule for decryption, which we add to the rules in Figure 6.3:

$$(\mathit{decrypt}_t)\quad \frac{\begin{array}{l}\Gamma \vdash e_1 : H \\ \Gamma \vdash e_2 : H\end{array}}{\Gamma \vdash \mathcal{D}(e_1, e_2) : H}$$

Notice that, because of subtyping, e_1 and e_2 can be either L or H. But, whatever they are, $\mathcal{D}(e_1, e_2)$ is always H. With our extended language, we use exactly the same definition of leaking adversary $\mathcal{B}$ as before, except that the adversary can now perform decryption, $\mathcal{B}$ is executed in the following experiment:

```
Experiment $\mathbf{Exp}_{\mathcal{SE}}^{\text{leak}}(\mathcal{B})$
    $K \overset{?}{\leftarrow} \mathcal{K}$;
    $h_0 \overset{?}{\leftarrow} \{0,1\}$;
    $h := h_0$;
    initialize all other variables of $\mathcal{B}$ to 0;
    run $\mathcal{B}^{\mathcal{E}_K(\cdot),\mathcal{D}_K(\cdot,\cdot)}$;
    if $l = h_0$ then return 1 else return 0
```

and has the same advantage as the previous version:

$$\mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 2 \cdot \Pr[\mathbf{Exp}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}) = 1] - 1.$$

Intuitively, it does not seem that decryption should help a leaking adversary, since a decryption always has type H. However, we have so far been unable to adapt the proof of Theorem 6.2.2 to deal with decryption. But we are able to prove an analogous theorem in the case where the symmetric encryption scheme $\mathcal{SE}$ satisfies a stronger security property, namely *IND-CCA security* (*indistinguishability under chosen-ciphertext attack* [BR05]). We now show that our type system with both encryption and decryption is sound, assuming that $\mathcal{SE}$ is IND-CCA secure.

Theorem 6.3.1 (IND-CCA Reduction) *Given adversary $\mathcal{B}$ as above, there exists IND-CCA adversary $\mathcal{A}$ such that*

$$\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(\mathcal{A}) \geq \frac{1}{2} \cdot \mathbf{Adv}^{\text{leak}}_{\mathcal{SE}}(\mathcal{B}).$$

Moreover, $\mathcal{A}$ runs about as quickly as $\mathcal{B}$.

Proof. The proof is quite similar to that of Theorem 6.2.2. Given leaking adversary $\mathcal{B}$, we construct an IND-CCA adversary $\mathcal{A}$ that runs $\mathcal{B}$ with a randomly-chosen 1-bit value of h, for at most $p(k)$ steps, As before, whenever $\mathcal{B}$ makes a call $\mathcal{E}(m)$ to its encryption primitive, $\mathcal{A}$ passes $(0^n, m)$ to its LR oracle and returns the result, which is a two-block ciphertext (c_1, c_2), to $\mathcal{B}$. But $\mathcal{A}$ also remembers $((c_1, c_2), m)$ in a hash table. Whenever $\mathcal{B}$ makes a call $\mathcal{D}(c_1, c_2)$ to its decryption primitive, $\mathcal{A}$ first checks whether (c_1, c_2) is in the hash table. If so, it returns the corresponding plaintext m to $\mathcal{B}$. If not, it answers the query using its decryption oracle $\mathcal{D}_K(\cdot)$. As before, $\mathcal{A}$ guesses that it is in World 1 if $\mathcal{B}$ terminates within $p(k)$ steps and successfully leaks h; if not, then $\mathcal{A}$ guesses that it is in World 0.

Note that although there will be some cost associated with maintaining the hash table, $\mathcal{A}$ still runs about as quickly as $\mathcal{B}$. Also note that $\mathcal{A}$ never "cheats" by calling its decryption oracle on a ciphertext previously returned by its LR-oracle.

As before, when $\mathcal{A}$ is in World 1, it runs $\mathcal{B}$ faithfully. And when $\mathcal{A}$ is in World 0, it runs $\mathcal{B}$ as a program in the random assignment language of Chapter 5. (Note that the decryption primitive $\mathcal{D}$ can be viewed as an ordinary operation like $+$, since its typing rule conforms to the Simple Security lemma.) As before, we cannot claim probabilistic noninterference on the random assignment language because of nontermination; instead we use the Bucket property to bound $\mathcal{B}$'s probability of terminating and leaking h to l. The calculation of $\mathbf{Adv}^{\text{ind-cca}}_{\mathcal{SE}}(\mathcal{A})$ follows as before. $\square$

Finally, Using Theorem 6.3.1, we can obtain results analogous to Corollary 6.2.3 and Theorem 6.2.4, using exactly the same arguments; hence, we establish computational probabilistic noninterference on the language with encryption and decryption primitives.

In this chapter we have extended the probabilistic language of Chapter 5 with encryption and decryption primitives. We have kept the same non-restrictive type system as in all the other work and have shown that the intuition that the result of encryption can be public, is sound. Finally we have shown a computational noninterference property on our language and type systems for IND-CPA and IND-CCA secure schemes.

CHAPTER 7

CONCLUSION AND FUTURE WORK.

In this thesis we set out to address the feasibility of a practical secure language. We proposed the following success criteria:

- the work should present successful implementations of languages with features that are representative of commercial languages; it should include at a minimum: probabilistic commands, concurrency, and cryptographic primitives.
- a single theory should handle the establishment of the security properties of all the languages.
- the languages should be simple and elegant, the programmer should not have to learn complex syntax or semantics and the type system should not restrict the programmer too unreasonably. At a minimum, the guards of **if** and **while** commands should not be restricted to public.

We have met this criteria by the introduction of a new powerful technique for proving noninterference on languages that were not previously provable under bisimulation, the presentation of four elegant languages that are only limited by the Denning Restrictions, and the use of a single theory for their proof of soundness.

Research in the area of secure information flow is motivated by the need for programs that can handle sensitive information with the assurance that it will not be leaked to unauthorized parties. Yet, it is not yet clear how feasible it would be to develop a practical language with a strong security property like noninterference, or whether the language could be used to develop "real" applications. There is also no guarantee of complete security through applying secure information flow techniques, just the elimination of a class of information leaks.

For example, in a distributed language, an eavesdropper could presumably sense the *time* when information is transmitted, introducing the possibility of *external timing channels*. Such channels are notoriously difficult to prevent. However, there are effective countermeasures such as the techniques *input blinding* [Koc96] and *bucketing* [KD09], and more traditionally, the NRL Pump [KM93b] which can limit the effectiveness of timing attacks.

Therefore, we also conclude that currently there is no major impediment in the creation of such languages. To support this, in this work we have shown several languages which exemplify aspects of some of the key features of a typical general-purpose language; this adds to existing work in secure implementation of other aspects of such a language; for example, [DS04] explores a secure implementation of arrays.

Correctness. One issue that should be addressed is the correctness of theorems in this work and in future work involving complex languages. The theory in this thesis was published in four peer-reviewed publications and the proofs are elegant and can be followed carefully. Also, the findings are not incongruent with related work.

However, as we move to the implementation of practical languages the proofs are likely to become extremely tedious and long, making them impregnable. The principal problem with this is the introduction of errors which would not be found easily. A solution to this problem is the introduction of automatic proving tools to verify the theory. A variety of such tools are now available for use and have become part of university curricula. For example, Benjamin Pierce [Pie08] has incorporated *Coq* at U-Penn for the Foundations of Programming Languages course.

Coq, was developed at INRIA et al [The04] and creates an environment where theorems can be specified and their proofs can be verified. To do this, one first fully describes the theory using the tool's formal language; this language is similar to that used in functional languages like Haskell and F#. Next, a theorem can be specified and, finally, the proof of the theorem is done; usually, this is an easier part of the process. Proofs are simply done via a succession of claimed properties which are verified by the tool.

It is likely that practical applications commissioned by non-academic organizations would require some sort of automatic verification of the properties of systems being developed. Also, proving tools can provide an insight into the theory being developed since this is done within the formal environment provided by the tool; hence, design errors can be found and corrected immediately.

Future work. One situation that seems promising is an implementation of Chapter 4 in a concrete setting over a public network. Section 4.4 sketches this work but without a proof of soundness.

Another interesting result would be to extend the cryptographic language of Chapter 6 to a richer language, for example with public-key encryption, and to carry out case studies to explore the usefulness of the language and type system in developing realistic applications with provable security properties. It would be interesting to determine whether IND-CCA secure encryption is really necessary for the soundness of the cryptographic language. It appears possible to construct pathological IND-CPA secure encryption schemes that allow well-typed programs that use both encryption and decryption to leak H variables, but it is unclear whether "real" IND-CPA secure schemes would allow similar attacks.

Finally, a main goal of secure information flow is to enable practical secure applications. A practical language guaranteeing a noninterference property would not be for everybody, nor for regular applications. The complexity and subtlety of modern systems makes the detection of "leaking code" hard (in the rare case where the code is available to inspect). In secure information flow analysis, code inspection (at least by a human) would not be necessary to certify the security property of a program. This is because the certifying authority would only need to compile the program under the secure language exposing the security lattice with the certificate. This, of course, would require a certificate authority as well as tools for verification of programs. But to have a "real" secure information flow system we would need to address many other practical problems. For example, it seems unlikely that one security lattice would be able to be fixed for any real project. This could be addressed by allowing multiple security lattices so that each module could have its own. Then, if the language semantics and type system are well set, we could separate the lattice from the language, allowing for a certification of programs with respect to a security lattice. Under this model we would still have to work out the relations between elements of the lattices and the interfaces between programs with different lattices.

Even after the language is specified and an operational scope is set to put boundaries on the type of programs that would be created, a development environment would be needed and programmers would need to learn the new environment; the security lattices would have to be designed and related and users would need to know the proper classification of data. This is a daunting and expensive proposition, yet, there are environments where this kind of security is needed. National security, banking and medicine seem to be the prime candidates as the confidentiality of the information in these industries is valued. Selecting a workable scope for implementing secure information flow within our national security would seem more feasible given that there has been extensive security classification work in this area and that the security latices seems more tractable. This is in contrast with a lattice for a medical system where who is able to inspect a patient's medical data depends on the patient and on the institution. A banking or medical model would probably be composed of a federated set of entities with a central certifying authority which would assert the absence of leaks with respect to a security lattice.

The current approach of fixing vulnerabilities as they are found seems inadequate for some applications. When a system cannot afford careless or devious programming, secure information flow may provide a solution that maintains the confidentiality of data from inception.

NOMENCLATURE

Γ. In type systems for secure information flow, Γ is a partial function that maps variables to their security classification. Section 3.2, pg. 25.

Aura. Aura is a language for secure information flow that maintains confidentiality and integrity properties of its constructs as specified by their labels. During execution data is “packed” according to its label using asymmetric encryption. Section 2.1, pg. 14.

bisimulation. Bisimulation is an equivalence relation where two transitioning systems behave indistinguishably with respect to a formal observer. Section 5.2, pg. 87.

bucket property. Given a program for secure information flow and its stripped version, if running the program can reach a number of final configurations, each with some probability greater than zero or can result in nontermination with some other probability, the bucket property says that the final probabilities of the stripped program can only grow and this growth can only come from the looping probability of the original program. Section 5, pg. 77.

ciphertexts. In cryptography, a ciphertext is a bit-string that is the result of an encryption operation. Section 2.1, pg. 14.

cryptography. Cryptography is the study of algorithms to embed information in bit-strings so that it is hard to discover and recover the information for unauthorized parties. Section 2.1, pg. 11.

Denning restrictions. The Denning restrictions are a set of restrictions to prevent the leaking of information during the executions of programs. Section 2.1, pg. 9.

Dolev-Yao. Dolev-Yao is a formal model to reason about the behavior of cryptographic protocols. The essence of Dolev-Yao is that encrypting an object with a key produces another object that is not inspectable and can only be decrypted if one has the encryption key. Also, the network in the Dolev-Yao model is hostile in that attackers control message delivery being able to drop, substitute, or initiate messages at will. Section 2.2, pg. 17.

Encryption Oracle. In the context of cryptography, an LR-encryption oracle is a pair of probabilistic functions [MRST06] of which only one is used, but it is not known which. Also, the probability distribution of the functions is not known. Given such an oracle, our ability to determine which function it uses is related to the effectiveness of the encryption scheme. Section 6.1, pg. 117.

fast low simulation. A fast low simulation is a fast simulation that maintains low equivalence of memories. Section 3.5.2, pg. 43.

fast simulation. Intuitively, fast simulation is a binary relation on a transition system such that if R is a fast simulation and sRt (where s and t are states of the transition system), t can match, in zero or one steps, any transition from s. However there are varying definitions of fast simulation in Chapters 3, 4, and 5 (Section 3.3, pg. 35; Section 4.3.3, pg. 65; Section 5.2.4, pg. 89).

IND-CCA security. IND-CCA stands for Indistinguishability Under Chosen Ciphertext Attack. Similarly to IND-CPA security, an adversary is provided with an encryption oracle, but now the adversary also has a decryption oracle. The encryption oracle works just like in IND-CPA security. The decryption oracle decrypts ciphertexts but the adversary is not allowed to decrypt a ciphertext that has been generated with the encryption oracle. The adversary then tries to guess which message the oracle is encrypting. The encryption scheme is secure if there is no adversary that can guess the oracle with a significant probability and in polynomial time. Section 6.1, pg. 118.

IND-CPA security. IND-CPA stands for Indistinguishability Under Chosen Plaintext Attack. IND-CPA security is a property of encryption schemes where no adversary is capable of distinguishing any two ciphertexts that are generated by the encryption algorithm unless the adversary has the encryption key. More precisely, an adversary is provided with an encryption oracle that is given two messages and always encrypts one or the other, but we don't know which. The adversary then, tries to guess which message the oracle is encrypting. The encryption scheme is secure if there is no adversary that can guess the oracle with a significant probability and in polynomial time. Section 6.1, pg. 117.

Jif. Jif is a practical language for secure information flow that can include information flow and access control policies. For example, the language supports labels such as

int $\{Alice \rightarrow Bob\}$ x;

which means that x is owned by Alice and she allows Bob to read it. Section 2.1, pg. 14.

lumpability. In Markov chains, lumpability is a property where sets of states can be grouped together, thereby reducing the state space without affecting its operations. States within the lumped sets are equivalent in some manner. Section 2.1, pg. 12.

Markov chain. A Discrete Time Markov Chain (DTMC) is a pair composed of a countable set of states and a probability matrix (with the transition probabilities from any state to any other state). The probability matrix must have the property that for any state the sum of all the transition probabilities (the sum of any row of the matrix) must be 1. Section 2.1, pg. 12.

Needham-Schroeder-Lowe protocol. The Needham-Schroeder-Lowe protocol is a cryptographic protocol for secure communications; it is defined as follows:
$A \to B : \{N_A, A\}_{pkB}$
$B \to A : \{N_A, , N_B, B\}_{pkA}$
$A \to B : \{N_B\}_{pkB}$
where A and B are parties, N is a fresh nonce, and pkA is the public key of A. In the protocol A sends B a fresh nonce and a self-identification encrypted using B's public key. Then, B replies with A's nonce, its own fresh nonce, and its own self-identification encrypted using A's public key. Finally, A replies by returning B's nonce. Section 4.4, pg. 73.

noninterference. Noninterference is a formal property of a system that guarantees the absence of information leaks within an execution model. Section 2.1, pg. 9.

observational determinism. Observational determinism is a property of type systems such that multiple execution yield the same result for a formal observer. Section 2.2, pg. 16.

plaintext. In cryptography, a plaintext is a message, a bit-string. Section 6, pg. 108.

reachability property. The essence of the reachability property is that if a state simulates another the simulating state must reach its destination in less steps then the simulated state. For this work we require that the simulating state reaches at least as fast as the simulated state; although this is not strictly necessary for all our results, it is required for some. Section 3.3.5, pg. 38.

RSA. RSA is an encryption scheme invented by Ron Rivest, Adi Shamir, and Leonard Adleman and is the basis for public key cryptography. Section 2.1, pg. 12.

Secure Information Flow. Secure Information Flow is an area of research in computer security that emphasizes the maintaining of confidentiality and integrity of information as it is processed by a computing machine. Section 2.1, pg. 8.

security lattice. A security lattice is a lattice that models the relation between the security types of the objects in a security policy. A lattice is a partially ordered set with least upper bound and greatest lower bound for all pair of elements. Section 2.1, pg. 8.

security policy. A security policy is a set of objects, definitions, and rules that restrict the access of information for a system. Section 2.1, pg. 10.

Shannon entropy. Suppose a random variable has a certain number of outcomes each with some probability. You are tasked with encoding these outcomes using bit strings so that the generation of values from the random variable produces on average the least number of bits. The Shannon entropy is the lower bound for this. It is the expected value of the log_2 of the probability of the outcomes in non-increasing order. Entropy in general has wide application and intuitions. Section 2.1, pg. 11.

simulation. A simulation relation is a preorder (reflexive and transitive) on a set of states such that if state s is simulates by t (sRt) then t can match all transitions of s. Section 5.2, pg. 87.

stripping. The stripping function is a function that when applied to a program removes all subcommands that do not assign to low variables; when applied to a memory it removes all the high variables. The stripping function is denoted as $\lfloor \cdot \rfloor$ when used as a relation. Section 2.2, pg. 15.

symmetric encryption scheme. A symmetric encryption scheme is a triple of algorithms: one for the generation of keys, one for the randomized encryption of messages using a key, and one for the deterministic decryption of ciphertexts using a key. Section 6.1.1, pg. 117.

termination-insensitive noninterference. Termination-insensitive noninterference is a variant of the noninterference property that only applies to executions that terminate. Section 3.5, pg. 42.

transition system. A transition system is a pair composed of a set of states and a transition relation. Section 3.3.1, pg. 36.

type system. A type system is an aspect of a system composed of objects, types, and inference rules that can be used to reason about the behavior of the complete system. Section 2.1, pg. 8.

upwards closed set. Within this work, an upwards closed set is a set that includes all the simulating states of all the states in itself. Section 3.3.4, pg. 38.

BIBLIOGRAPHY

[ACF06] M. Abadi, R.J. Corin, and C. Fournet. Computational secrecy by typing for the pi calculus. In N. Kobayashi, editor, *Fourth Asian Symposium on Programming Languages and Systems (APLAS 2006)*, volume 4279 of *Lecture Notes in Computer Science*, pages 253–269, London, November 2006. Springer Verlag.

[AFG98] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *LICS '98: Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science*, page 105, Washington, DC, USA, 1998. IEEE Computer Society.

[Aga00] Johan Agat. Transforming out timing leaks. In *POPL*, pages 40–53, Boston, MA, January 2000.

[AHS06] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. Cryptographically-masked flows. In *Proceedings of the 13th International Static Analysis Symposium*, pages 353–369, Seoul, Korea, 2006.

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, 2008.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). In *TCS '00: Proceedings of the IFIP International Conference on Theoretical Computer Science*, pages 3–22, August 2000.

[AS09] Rafael Alpízar and Geoffrey Smith. Secure information flow for distributed systems. In *Proc. Formal Aspects of Security and Trust (FAST 2009)*, Eindhoven, Netherlands, November 2009.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Comput. Netw.*, 48:701–716, August 2005.

[BDJR97] M. Bellare, E. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of DES modes of operation. In *Proceedings of the 38th Symposium on Foundations of Computer Science*, 1997.

[BKHW05] Christel Baier, Joost-Pieter Katoen, Holger Hermanns, and Verena Wolf. Comparative branching-time semantics for Markov chains. *Information and Computation*, 200(2):149–214, 2005.

[BP02] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. In *Proceeding 7th ESORICS*, pages 1–23, 2002.

[BP05] Michael Backes and Birgit Pfitzmann. Relating symbolic and cryptographic secrecy. In *Proceeding 26th IEEE Symposium on Security and Privacy*, Oakland, California, 2005.

[BR05] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, page 207, 2005.

[CHM02] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative analysis of the leakage of confidential data. *Electronic Notes in Theoretical Computer Science*, 59(3), 2002.

[Coh77] Ellis S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139, 1977.

[DD77] Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den75] Dorothy Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, West Lafayette, IN, May 1975.

[DPHW02] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In *Proceedings15thIEEE Computer Security Foundations Workshop*, pages 1–17, Cape Breton, Nova Scotia, Canada, June 2002.

[DS04] Zhenyue Deng and Geoffrey Smith. Lenient array operations for practical secure information flow. In *Proceedings17thIEEE Computer Security Foundations Workshop*, pages 115–124, Pacific Grove, California, June 2004.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[FC08] Riccardo Focardi and Matteo Centenaro. Information flow security of multi-threaded distributed programs. In *PLAS '08: Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2008. ACM.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume I. John Wiley & Sons, Inc., Third edition, 1968.

[FR08] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings35thSymposium on Principles of Programming Languages*, San Francisco, California, January 2008.

[GM82] Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings1982IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[Gra90] James W. Gray, III. Probabilistic interference. In *Proceedings1990IEEE Symposium on Security and Privacy*, pages 170–179, Oakland, CA, May 1990.

[JL91] Bengt Jonsson and Kim Larsen. Specification and refinement of probabilistic processes. In *Proc. 6th IEEE Symposium on Logic in Computer Science*, pages 266–277, 1991.

[JVM$^+$08] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: a programming language for authorization and audit. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 27–38, New York, NY, USA, 2008. ACM.

[KD09] Boris Köpf and Markus Dürmuth. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *Proc. 22nd IEEE Computer Security Foundations Symposium (CSF '09)*, pages 324–335. IEEE, 2009.

[KM93a] Myong H. Kang and Ira S. Moskowitz. A pump for rapid, reliable, secure communication. In *CCS '93: Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 119–129, New York, NY, USA, 1993. ACM.

[KM93b] Myong H. Kang and Ira S. Moskowitz. A pump for rapid, reliable secure communication. In *Proceedings of the 1st ACM Conference on Computer & Communications Security*, pages 119–129, November 1993.

[KMC05] Myong H. Kang, Ira S. Moskowitz, and Stanley Chincheck. The pump: A decade of covert fun. In *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*, pages 352–360. IEEE Computer Society, 2005.

[Koc96] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems. In *Proceedings 16th Annual Crypto Conference*, August 1996.

[KS60] John Kemeny and J. Laurie Snell. *Finite Markov Chains.* D. Van Nostrand, 1960.

[KS10] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *CSF*, pages 44–56, 2010.

[Lau01] Peeter Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings 10th ESOP (European Symposium on Programming*, pages 77–91, 2001.

[Lau02] Peeter Laud. Encryption cycles and two views of cryptography. In *In NORDSEC 2002 - Proceedings of the 7th Nordic Workshop on Secure IT Systems (Karlstad University Studies 2002:31*, pages 85–100, 2002.

[Lau03] Peeter Laud. Handling encryption in an analysis for secure information flow. In *Proceedings 12th ESOP (European Symposium on Programming*, pages 159–173, 2003.

[Lau05] Peeter Laud. Secrecy types for a simulatable cryptographic library. In *Proceedings 12th CCS (ACM Conference on Computer and Communications Security)*, pages 26–35, 2005.

[Lau08] Peeter Laud. On the computational soundness of cryptographically masked flows. In *Proceedings35thSymposium on Principles of Programming Languages*, San Francisco, California, January 2008.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[LV05] Peeter Laud and Varmo Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Symposium on Fundamentals of Computational Theory*, volume 3623 of *Lecture Notes in Computer Science*, pages 365–377, Lübeck, Germany, 2005.

[LZ05] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings32ndSymposium on Principles of Programming Languages*, pages 158–170, January 2005.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *Proceedings34thSymposium on Principles of Programming Languages*, pages 225–235, Nice, France, January 2007.

[McL90] John McLean. Security models and information flow. In *Proceedings1990IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, 1990.

[MCN$^+$06] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. *Jif: Java + information flow*. Cornell University, 2006. Available at `http://www.cs.cornell.edu/jif/`.

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *IJCAI'71: Proceedings of the 2nd international joint conference on Artificial intelligence*, pages 481–489, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[MRST06] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. *Theor. Comput. Sci.*, 353:118–164, March 2006.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pie08] Benjamin C. Pierce. Using a proof assistant to teach programming language foundations, or, Lambda, the ultimate TA, April 2008. White paper.

[SA06] Geoffrey Smith and Rafael Alpízar. Secure information flow with random assignment and encryption. In *Proc. 4th ACM Workshop on Formal Methods in Security Engineering*, pages 33–43, Fairfax, Virginia, November 2006.

[SA07] Geoffrey Smith and Rafael Alpízar. Fast probabilistic simulation, nontermination, and secure information flow. In *Proc. 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 67–71, San Diego, California, June 2007.

[SA11] Geoffrey Smith and Rafael Alpízar. Nontermination and secure information flow. *Mathematical Structures in Computer Science*, 2011.

[SAIL08] Alan B. Shaffer, Mikhail Auguston, Cynthia E. Irvine, and Timothy E. Levin. A security domain model to assess software for exploitable covert channels. In Úlfar Erlingsson and Marco Pistoia, editors, *PLAS*, pages 45–56. ACM, 2008.

[San09] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):1–41, 2009.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[Smi01] Geoffrey Smith. A new type system for secure information flow. In *Proceedings14thIEEE Computer Security Foundations Workshop*, pages 115–125, Cape Breton, Nova Scotia, Canada, June 2001.

[Smi03] Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings16thIEEE Computer Security Foundations Workshop*, pages 3–13, Pacific Grove, California, June 2003.

[Smi06] Geoffrey Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[Smi07] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 297–307. Springer-Verlag, 2007.

[Smi08] Geoffrey Smith. Adversaries and information leaks. In Gilles Barthe and Cédric Fournet, editors, *TGC 2007 (Trustworthy Global Comput-*

ing), volume 4912 of *Lecture Notes in Computer Science*, pages 383–400. Springer-Verlag, 2008.

[Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings FoSSaCS 2009: Twelfth International Conference on Foundations of Software Science and Computation Structures*, volume 5504 of *Lecture Notes in Computer Science*, pages 288–302. Springer-Verlag, 2009.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings13thIEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, UK, July 2000.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings18thIEEE Computer Security Foundations Workshop*, June 2005.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multithreaded imperative language. In *Proceedings25thSymposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, January 1998.

[SW06] Vitaly Shmatikov and Ming-Hsiu Wang. Measuring relationship anonymity in mix networks. In *WPES '06: Proceedings of the 5th ACM workshop on Privacy in Electronic Society*, pages 59–62, 2006.

[The04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004.

[THV04] Gergely Tóth, Zoltán Hornák, and Ferenc Vajda. Measuring anonymity revisited. In S. Liimatainen and T. Virtanen, editors, *Proceedings of the Ninth Nordic Workshop on Secure IT Systems*, pages 85–90, Espoo, Finland, 2004.

[Vol00] Dennis Volpano. Secure introduction of one-way functions. In *Proceedings13thIEEE Computer Security Foundations Workshop*, pages 246–254, Cambridge, UK, June 2000.

[VS97] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings10thIEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

[VS99] Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.

[War03] Bogdan Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. In *Proceedings16thIEEE Computer Security Foundations Workshop*, pages 248–262, Pacific Grove, California, June 2003.

[ZM03] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proceedings16thIEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003.

[ZM08] Lantian Zheng and Andrew C. Myers. Securing nonintrusive web encryption through information flow. In *PLAS '08: Proceedings of the third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 125–134, New York, NY, USA, 2008. ACM.

VITA

RAFAEL ALPIZAR

August 4, 1961	Born, Santiago, Cuba
1984	B.S., Electrical Engineering University of Miami Coral Gables, Florida
1986	M.S., Computer Science George Washington University Washington D.C.
1992	15 Cr. towards MSEE Catholic University Washington D.C.
Press:	Ph.D. Candidate Computer Science Florida International University Miami FL.
1984–1990	Electronic Engineer David Taylor Research Center Bethesda, Maryland
1990–1995	Programmer & Systems Analyst Dade County DERM Miami, Florida
1995–2000	Chief Information Officer City of Hialeah Hialeah, Florida
2000–2003	Academic Chair Design Technology, MDCC Miami, Florida
2003–2006	Graduate Assistant Florida International University Miami, Florida
2006–Press	IT Director City of Doral Doral, Florida

PUBLICATIONS AND PRESENTATIONS

Smith, G., Alpízar, R., (2006). *Secure Information Flow with Random Assignment and Encryption.* Proc. 4th ACM Workshop on Formal Methods in Security Engineering (FMSE 2006).

Smith, G., Alpízar, R., (2007). *Fast Probabilistic Simulation, Nontermination, and Secure Information Flow.* Proc. 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2007).

Presentation, Nov 2009. *Secure Information Flow for Distributed Systems.* Formal Aspects of Security and Trust (FAST 2009 Affiliate of FM Week). Technische Universiteit Eindhoven NL.

Alpízar, R., Smith, G.,(2009). *Secure Information Flow for Distributed Systems.* Formal Aspects of Security and Trust (FAST 2009).

Smith, G., Alpízar, R., (2011). *Nontermination and secure information flow.* Mathematical Structures in Computer Science (MSCS 2011).