PROGRAMMING LANGUAGES FOR INFORMATION SECURITY

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Stephan Arthur Zdancewic

August 2002

PROGRAMMING LANGUAGES FOR INFORMATION SECURITY

Stephan Arthur Zdancewic, Ph.D.
Cornell University 2002

Our society's widespread dependence on networked information systems for everything from personal finance to military communications makes it essential to improve the security of software. Standard security mechanisms such as access control and encryption are essential components for protecting information, but they do not provide end-to-end guarantees. Programming-languages research has demonstrated that security concerns can be addressed by using both program analysis and program rewriting as powerful and flexible enforcement mechanisms.

This thesis investigates *security-typed programming languages*, which use static typing to enforce information-flow security policies. These languages allow the programmer to specify confidentiality and integrity constraints on the data used in a program; the compiler verifies that the program satisfies the constraints.

Previous theoretical security-typed languages research has focused on simple models of computation and unrealistically idealized security policies. The existing practical security-typed languages have not been proved to guarantee security. This thesis addresses these limitations in several ways.

First, it establishes *noninterference*, a basic information-flow policy, for languages richer than those previously considered. The languages studied here include recursive, higher-order functions, structured state, and concurrency. These results narrow the gap between the theory and the practice of security-typed languages.

Next, this thesis considers more practical security policies. Noninterference is often too restrictive for real-world programming. To compensate, a restricted form of declassification is introduced, allowing programmers to specify a richer set of information-flow policies. Previous work on information-flow security also assumed that all computation occurs on equally trusted machines. To overcome this unrealistic premise, additional security constraints for systems distributed among heterogeneously trusted hosts are considered.

Finally, this thesis describes Jif/split, a prototype implementation of *secure program partitioning*, in which a program can automatically be partitioned to run securely on

heterogeneously trusted hosts. The resulting communicating subprograms collectively implement the original program, yet the system as a whole satisfies the security requirements without needing a universally trusted machine. The theoretical results developed earlier in the thesis justify Jif/split's run-time enforcement mechanisms.

BIOGRAPHICAL SKETCH

Steve was born on June 26, 1974 in Allentown, Pennsylvania to Arthur and Deborah Zdancewic. After living briefly in Eastern Pennsylvania and California, his family, which includes his brother, David, and sister, Megan, settled in Western Pennsylvania in the rural town of Friedens. His family remained there until the autumn of 1997, when his parents moved back to Eastern PA.

Steve attended Friedens Elementary School and Somerset Area Junior and Senior High Schools. His first computer, a Commodore 64, was a family Christmas gift in 1982. Although he learned a smattering of Commodore BASIC[1], he mainly used the computer to play games, the best of which were Jumpman, Archon, and the classic Bard's Tale. Steve pursued his interest in computers through senior high school, although he never took the programming courses offered there. His most influential high school teacher was Mr. Bruno, who taught him Precalculus, Calculus I & II, and Statistics.

After graduating with Honors from Somerset Area Senior High in 1992, Steve enrolled in Carnegie Mellon University's Department of Electrical and Computer Engineering. Shortly into his second semester there, he decided that the computer science courses were more fun than the engineering ones and transferred into the School of Computer Science.

Steve graduated from Carnegie Mellon University with a B.S. in Computer Science and Mathematics. He decided to continue his education by obtaining a Ph.D. and entered Cornell's CS department in the fall of 1996. There, he met Stephanie Weirich, also a computer scientist, when they volunteered to organize the department's Fall picnic. Both Steve and Stephanie were recipients of National Science Foundation Fellowships and Intel Fellowships; they also both spent the Summer of 1999 doing internships at Lucent Technologies in Murray Hill, New Jersey. On August 14, 1999 Steve and Stephanie were married in Dallas, Texas.

Steve received a M.S. in Computer Science from Cornell University in 2000, and a Ph.D. in Computer Science in 2002.

[1] Anyone familiar with the Commodore machines will recall with fondness the arcane command `poke 53281, 0` and the often used `load *,8,1`.

ACKNOWLEDGEMENTS

First, I thank my wife, Stephanie Weirich, without whom graduate school would have been nearly impossible to survive. She has been my best friend, my unfaltering companion through broken bones and job interviews, my source of sanity, my reviewer and editor, my dinner partner, my bridge partner, my theater date, my hockey teammate, my most supportive audience, my picnic planner, and my love. I cannot thank her enough.

Next, I thank my parents, Arthur and Deborah Zdancewic, my brother Dave and my sister Megan for their encouragement, love, and support. Thanks also to Wayne and Charlotte Weirich, for welcoming me into their family and supporting me as they do Stephanie.

I also thank my thesis committee. Andrew Myers, my advisor and friend, made it fun to do research; his ideas, suggestions, questions, and feedback shaped this dissertation more than anyone else's. Greg Morrisett advised me for my first three years at Cornell and started me on the right path. Fred Schneider, with his sharp insights and unfailingly accurate advice, improved not only this thesis, but also my writing and speaking skills. Karen Vogtmann challenged my mathematical abilities in her algebraic topology course.

I also thank Jon Riecke, whom I worked with one fun summer at Lucent Technologies; our discussions that summer formed the starting point for the ideas in this dissertation.

I am especially indebted to Nate Nystrom and Lantian Zheng, who not only did the bulk of the programming for the Jif and Jif/split projects, but also contributed immensely to the results that make up Chapter 8.

Many, many thanks to my first set of officemates, Tuğkan Batu, Tobias Mayr, and Patrick White, who shared numerous adventures with me during our first years as graduate students. Thanks also to my second set of officemates: Dan Grossman and Yanling Wang, from whom I've learned much. I also thank Dan for coffee filters, for grammatical and editorial acumen, and for always being prepared to talk shop.

Lastly, I would like to add to all of the above, a big thanks to many others who made Ithaca such a fun place to be for the last six years:

Bert Adams, Gary Adams, Kavita Bala, Matthew Baram, Jennifer Bishop, James Cheney, Bob Constable, Karl Crary, Jim Ezick, Adam Florence, Annette Florence, Neal

Glew, Mark Hayden, Jason Hickey, Takako Hickey, Kim Hicks, Mike Hicks, Timmy Hicks, Amanda Holland-Minkley, Nick Howe, Susannah Howe, David Kempe, Dan Kifer, Jon Kleinberg, Dexter Kozen, Lillian Lee, Lyn Millet, Tonya Morrisett, Riccardo Pucella, Andrei Sabelfeld, Dave Walker, Vicky Weisman, and Allyson White.

This research was supported in part by a National Science Foundation Fellowship (1996 through 1999) and an Intel Fellowship (2001 through 2002).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The widespread use of computers to archive, process, and exchange information via the Internet has led to explosive growth in e-commerce and on-line services. This increasing connectivity of the web means that more and more businesses, individual users, and organizations have come to depend critically on computers for day-to-day operation. In a world where companies exist whose sole purpose is to buy and sell electronic data and everyone's personal computer is connected to everyone else's, it is information itself that is valuable.

Protecting valuable information has long been a concern for security—cryptography, for example, has been in use for centuries [Sch96] Ironically, the features that make computers so useful—the ease and speed with which they can duplicate, process, and transmit data—are the same features that threaten information security.

This thesis focuses on two fundamental types of policies that relate to information security. *Confidentiality policies* deal with disseminating data [BL75, Den75, GM82, GM84]. They restrict who is able to learn information about a piece data and are intended to prevent secret information from becoming available to an untrusted party. *Integrity policies* deal with generating data [Bib77]. They restrict what sources of information are used to create or modify a piece of data and are intended to prevent an untrusted party from corrupting or destroying it.

The approach is based on *security-typed languages*, in which extended type systems express security policies on programs and the data they manipulate. The compiler checks the policy before the program is run, detecting potentially insecure programs before they can possibly leak confidential data, tamper with trusted data, or perform unsafe actions. Security-typed languages have been used to enforce *information-flow* policies that protect the confidentiality and integrity of data [ABHR99, HR98, Mye99, PC00, SV98, VSI96, ZM01b].

This thesis addresses the problem of how to provably enforce confidentiality and integrity policies in computer systems using security-typed languages. [1]

For example, the following program declares `h` to be a secret integer and `l` to be a public integer:

```
int{Secret} h;
int{Public} l;
... code using h and l...
```

Conceptually, the computer's memory is divided into a *low-security* portion visible to all parts of the system (the `Public` part) and a *high-security* portion visible only to highly trusted components (the `Secret` part). Intuitively, the declaration that `h` is `Secret` means that it is stored in the `Secret` portion of the memory and hence should not be visible to any part of the system that does not have clearance to access secret data.

Of course, simply dividing memory into regions does not prevent learning about high-security data indirectly, for instance by observing the behavior of a program that alters the `Public` portion of the memory. For example, a program that copies `Secret` data to a `Public` variable is insecure. When the observable behavior of the program is affected by the `Secret` data, the low-clearance program might be able to deduce confidential information, which constitutes a security violation.

This model assumes that the low-security observer knows which program is being run and hence can correlate the observed behaviors of the program with its set of possible behaviors to make deductions about confidential data. If the `Public` observer is able to infer some information about the contents of the `Secret` portion of data, there is said to be an *information flow* from `Secret` to `Public`. Information flows from `Public` to `Secret` are possible too, but they are permitted.

These information flows arise for many reasons:

1. **Explicit flows** are information channels that arise from the ways in which the language allows data to be assigned to memory locations or variables. Here is an example that shows an explicit flow from the high-security variable `h` to a low-security variable `l`:

   ```
   l := h;
   ```

   Explicit flows are easy to detect because they are readily apparent from the text of the program.

[1] Confidentiality and integrity of data are of course not the only cause for concern in networked information systems, but they are essential components of information security. See *Trust in Cyberspace* [Sch99] for a comprehensive review of security challenges. Security-typed languages can enforce security policies other than information flow, for example arbitrary safety policies [Wal00].

2. **Implicit flows** arise from the control-flow structure of the program. For example, whenever a conditional branch instruction is performed, information about the condition variable is propagated into each branch. The program below shows an implicit flow from the high-security variable `h` to a low-security variable `l`; it copies one bit of the integer `h` into the variable `l`:

   ```
   if (h > 0) then l := 1 else l := 0
   ```

   Similar information flows arise from other control mechanisms such as function calls, `goto`'s, or exceptions.

3. **Alias channels** arise from sharing of a mutable resource that can be affected by both high- and low-security data. For example, if memory locations are first-class constructs in the programming language, aliases between references can leak information. In the following example, the expression `ref 0` creates a reference to the integer `0`, the expression `!y` reads the value stored in the reference `y`, and the statement `x := 1` updates the location pointed to by reference `x` to hold the value `1`:

   ```
   x = ref 0;        Create a reference x to value 0
   y = x;            Create an alias y of x
   x := h;           Assignment through x affects contents of y
   l := !y;          Contents of h are stored in l
   ```

   Because the problem of determining when two program variables alias is, in general undecidable, the techniques for dealing with alias channels make use of conservative approximations to ensure that potential aliases (such as `x` and `y`) are never treated as though their contents have different security levels.

4. **Timing channels** are introduced when high-security data influences the amount of time it takes for part of a program to run. The code below illustrates a timing channel that transmits information via the shared system clock.

   ```
   l := time();              Get the current time
   if h then delay(10);      Delay based on h
   if (time() < l + 10)      See whether there was delay
     then l := 0             h is false
     else l := 1;            h is true
   ```

   The kind of timing channel shown above is *internal* to the program; the program itself is able to determine that time has passed by invoking the `time()` routine.

This particular flow can be avoided by making the clock high-security, but concurrent threads may time each other without using the system clock.

A second kind of timing channel is *external* to the program, in the sense that a user observing the time it takes for a program to complete is able to determine extra information about secret data, even if the program itself does not have access to the system clock. One approach to dealing with external timing channels is to force timing behavior to be independent of the high-security data by adding extra delays [Aga00] (at a potentially severe performance penalty).

5. **Abstraction-violation channels** arise from under-specification of the context in which a program will be run. The level of abstraction presented to the programmer by a language may hide implementation details that allow someone with knowledge of run-time environment to deduce high-security information.

   For example, the memory allocator and garbage collector might provide an information channel to an observer who can watch memory consumption behavior, even though the language semantics do not rely on a particular implementation of these features. Similarly, caching behavior might cause an external timing leak by affecting the program's running time. External timing channels are a form of abstraction-violation—they are apparent only to an observer with access to the "wall clock" running time of the program.

   These are the hardest sources of information flows to prevent as they are not covered by the language semantics and are not apparent from the text or structure of the program. While it is nearly impossible to protect against all abstraction-violation channels, it is possible to rule out more of them by making the language semantics more specific and detailed. For instance, if one were to model the memory manager formally, then that class of covert channels might be eliminated. Of course making such refined assumptions about the run-time environment means that the assumptions are harder to validate—any implementation must meet the specific details of the model.

Noninterference is the basic information-flow policy enforced by the security-typed languages considered in this thesis. It prohibits all explicit, implicit, and internal timing information flows from `Secret` to `Public`.

Although the above discussion has focused on confidentiality, similar observations hold for integrity: A low-integrity (`Tainted`) variable should not be able to influence the contents of a high-integrity (`Untainted`) variable. Thus, a security analysis should also rule out explicit and implicit flows from `Tainted` to `Untainted`.

The security-typed languages in this thesis are designed to ensure noninterference, but noninterference is often not the desired policy in practice. Many useful security

policies include intentional release of confidential information. For example, although passwords are `Secret`, the operating system authentication mechanism reveals information about the passwords—namely whether a user has entered a correct password.

Noninterference should be thought of as a baseline security policy from which others are constructed. Practical security-typed languages include *declassification* mechanisms that allow controlled release of confidential data, relaxing the strict requirements of noninterference. Although noninterference results are the focus, this thesis also discusses declassification and controlling its use.

1.1 Security-typed languages

Language-based security is a useful complement to traditional security mechanisms like access control and cryptography because it can enforce different security policies.

Access-control mechanisms grant or deny access to a piece of data at particular points during the system's execution. For example, the read–write permissions provided by a file system prevent unauthorized processes from accessing the data at the point when they try to open the file. Such discretionary access controls are well-studied [Lam71, GD72, HRU76] and widely used in practice.

Unlike traditional discretionary access-control mechanisms, a security-typed language provides *end-to-end* protection—the data is protected not just at certain points, but throughout the duration of the computation. To the extent that a system can be described as a program or a collection of communicating programs written in a security-typed language, the compositional nature of the type-system extends this protection system-wide.

As an example of the difference between information flow and access control, consider this policy: "the information contained in this e-mail may be obtained only by me and the recipient." Because it controls information rather than access, this policy is considerably stronger than the similar access-control policy: "only processes authorized by me or the recipient may open the file containing the e-mail." The latter policy does not prohibit the recipient process from forwarding the contents of the e-mail (perhaps cleverly encoded) to some third party.

Program analysis is a useful addition to run-time enforcement mechanisms such as reference monitors because such purely run-time mechanisms can enforce only safety properties, which excludes many useful information-flow policies [Sch01][2]. Run-time mechanisms can monitor sequences of actions and allow or deny them; thus, they can enforce access control and capability-based policies. However, dynamic enforcement of

[2]This analysis assumes that the run-time enforcement mechanism does not have access to the program text; otherwise the run-time mechanism could itself perform program analysis. Run-time program analysis is potentially quite costly.

information-flow policies is usually expensive and too conservative because information flow is a property of all possible executions of a program, not just the single execution available during the course of one run [Den82].

Encryption is another valuable tool for protecting information security, and it is crucial in settings where data must be transmitted via an untrusted medium—for example sending a secret over the Internet. However, encryption works by making it infeasible to extract information from the ciphertext without possessing a secret key. This property is exactly what is needed for transmitting the data, but it also makes it (nearly) impossible to compute usefully over the data; for instance it is difficult to create an algorithm that sorts an encrypted array of data.[3] For such non-trivial computation to take place over encrypted data, the data must be decrypted, at which point the problem again becomes regulating information flow through a computation.

The following examples illustrate scenarios in which access control and cryptography alone are insufficient to protect confidential data, but where security-typed languages can be used:

1. A home user wants a guarantee that accounting software, which needs access to both personal financial data and a database of information from the software company, doesn't send her credit records or other private data into the Internet whenever it accesses the web to query the database. The software company does not want the user to download the database because then proprietary information might fall into the hands of a competitor. The accounting software, however, is available for download from the company's web site.

   Security-typed languages offer the possibility that the user's home computer could verify the information flows in the tax program after downloading it. That verification gives assurance that the program will not leak her confidential data, even though it communicates with the database.

   With the rise of the Internet, such examples of mobile code are becoming a widespread phenomenon: Computers routinely download Java applets, web-scripts and Visual Basic macros. Software is distributed via the web, and dynamic software updates are increasingly common. In many cases, the downloaded software comes from untrusted or partially untrustworthy parties.

2. The ability for the sender of an e-mail to regulate how the recipient uses it is an information-flow policy and would be difficult to enforce via access control.

[3] There are certain encryption schemes that support arithmetic operations over ciphertext so that $encrypt(x) \oplus encrypt(y) = encrypt(x + y)$, for example. They are too impractical to be used for large amounts of computation [CCD88].

While cryptography would almost certainly be used to protect confidential e-mail and for authenticating users, the e-mail software itself could be written in a security-typed language.

3. Many programs written in C are vulnerable to buffer overrun and format string errors. The problem is that the C standard libraries do not check the length of the strings they manipulate. Consequently, if a string obtained from an untrusted source (such as the Internet) is passed to one of these library routines, parts of memory may be unintentionally overwritten with untrustworthy data—this vulnerability can potentially be used to execute an arbitrary program such as a virus.

   This situation is an example of an integrity violation: low-integrity data from the Internet should not be used as though it is trustworthy. Security-typed languages can prevent these vulnerabilities by specifying that library routines require high-integrity arguments [STFW01, Wag00].

4. A web-based auction service allows customers to bid on merchandise. Multiple parties may bid on a number of items, but the parties are not allowed to see which items others have bid on nor how much was bid. Because the customers do not necessarily trust the auction service, the customer's machines share information sufficient to determine whether the auction service has been honest. After the bidding period is over, the auction service reveals the winning bids to all participants.

   Security policies that govern how data is handled in this auction scenario can potentially be quite complex. Encryption and access control are certainly useful mechanisms for enforcing these policies, but the client software and auction server can be written in a security-typed language to obtain some assurance that the bids are not leaked.

Despite the historical emphasis on policies that can be enforced by access control and cryptographic mechanisms, computer security concerns have advanced to the point where richer policies are needed.

Bill Gates, founder of Microsoft, called for a new emphasis on what he calls "Trustworthy Computing" in an e-mail memorandum to Microsoft employees distributed on January 15, 2002. Trustworthy Computing incorporates not only the reliability and availability of software, but also security in the form of access control and, of particular relevance to this thesis, privacy [Gat02]:

> Users should be in control of how their data is used. Policies for information use should be clear to the user. Users should be in control of when and if they receive information to make best use of their time. It should be easy for

> users to specify appropriate use of their information including controlling the use of email they send.[4]
>
> –Bill Gates, January 15, 2002

Trustworthy Computing requires the ability for users and software developers to express complex security policies. Commercial operating systems offer traditional access control mechanisms at the file-system and process level of granularity and web browsers permit limited control over how information flows to and from the Internet. But, as indicated in Gates' memo, more sophisticated, end-to-end policies are desired.

Security-typed languages provide a formal and explicit way of describing complex policies, making them auditable and enforceable via program analysis. Such automation is necessitated both by the complexity of security policies and by the sheer size of today's programs. The security analysis can potentially reveal subtle design flaws that make security violations possible.

Besides complementing traditional enforcement mechanisms, security-typed languages can help software developers detect security flaws in their programs. Just as type-safe languages provide memory safety guarantees that rule out a class of program errors, security-typed languages can rule out programs that contain potential information leaks or integrity violations. Security-typed languages provide more confidence that programs written in them are secure.

Consider a developer who wants to create digital-signature software that is supposed to run on a smart card. The card provides the capability to digitally sign electronic data based on a password provided by the user. Because the digital signatures authorize further computations (such as transfers between bank accounts), the password must be protected—if it were leaked, anyone could forge the digital signatures and initiate bogus transactions. Consequently, the developer would like some assurance that the digital-signature software does not contain any bugs that unintentionally reveal the password. Writing the digital-signature software in a security-typed language would help improve confidence in its correctness.

There is no magic bullet for security. Security-typed languages still rely in part on the programmer to implement the correct policy, just as programmers are still trusted to implement the correct algorithms. Nevertheless, security-typed languages provide a way to ensure that the policy implemented by the programmer is self-consistent and that it agrees with the policy provided at the program's interface to the external environment. For example, the operating system vendor can specify a security policy on the data passed between the file system and applications written to use the file system. The compiler of a security-typed language can verify that the application obeys the policy

[4] Is it ironic that the text of this e-mail was available on a number of web sites shortly after it was sent?

specified in the OS interface; therefore the OS vendor need not trust the applications programmer. Symmetrically, the application writer need not trust the OS vendor.

Absolute security is not a realistic goal. Improved confidence in the security of software systems *is* a realistic goal, and security-typed programming languages offer a promising way to achieve it.

1.2 Contributions and Outline

This thesis develops the theory underlying a variety of security-typed languages, starting with a simple toy language sufficient for sequential computation on a trusted computer and building up to a language for describing multithreaded programs. It also address the problem of secure computation in a concurrent, distributed setting in which not all the computers are equally trusted.

Chapter 2 introduces the lattice-model of information-flow policies and the notation used for it in this thesis. This chapter defines noninterference—making precise what it means for a security-typed language to protect information security. This chapter is largely based on the existing work on using programming language technology to enforce information-flow policies.

Chapter 3 gives an elementary proof of noninterference for a security-typed, pure lambda calculus. This is not a new result, but the proof and the language's type system serve as the basis for the more complex ones presented later. Chapter 3 explains the proof and discusses the difficulties of extending it to more realistic programming languages.

The subsequent chapters describe the main contributions of this thesis. The contributions are:

1. The first proof of noninterference for a security-typed language that includes higher-order functions and state. This result is described in Chapter 4. The material there is drawn from a conference paper [ZM01b] and its extended version, which appears in the Journal of Higher Order and Symbolic Computation special issue on continuations [ZM01a]. The proofs of Soundness and Noninterference for the language that appear in Sections 4.3 and 4.4 are adapted from a technical report [ZM00]. Since the original publication of this result, other researchers have proposed alternatives to this approach [PS02, HY02, BN02].

2. An extension of the above noninterference proof to the case of multithreaded programs. The main difficulty in a concurrent setting is preventing information leaks due to timing and synchronization behavior. The main contribution of Chapter 5 is a proposal that, contrary to what is done in existing security-typed languages

for concurrent programs, *internal timing channels* should be controlled by eliminating race conditions entirely. This chapter gives a type system for concurrent programs that eliminates information leaks while still allowing threads to communicate in a structured way.

3. The observation that declassification, or intentional release of confidential data, ties together confidentiality and integrity constraints. Because declassification is a necessary part in any realistic secure system, providing a well-understood mechanism for its use is essential. Chapter 6 explains the problem and a proposed solution that is both simple and easy to put into practice. Intuitively, the *decision* to declassify a piece of confidential information must be protected from being tampered with by an untrusted source.

4. A consideration of the additional security requirements imposed when the system consists of a collection of distributed processes running on heterogeneously trusted hosts. Previous security-typed languages research has assumed that the underlying execution platform (computers, operating systems, and run-time support) is trusted equally by all of the principals whose security policies are expressed in a program. This assumption violates the principle of least privilege. Furthermore, it is unrealistic for scenarios involving multiple parties with mutual distrust (or partial distrust)—the very scenarios for which multilevel security is most desirable

   This approach, described in Chapter 7, is intended to serve as a model for understanding confidentiality and integrity in distributed settings in which the hosts carrying out the computation are trusted to varying degrees.

5. An account of a prototype implementation for obtaining end-to-end information-flow security by automatically partitioning a given source program to run in a network environment with heterogeneously trusted hosts. This prototype, called Jif/split, extends Jif [MNZZ01], a security-typed variant of Java, to include the heterogeneous trust model. Jif/split serves both as a test-bed and motivating application for the theoretical results described above.

   The Jif/split prototype described in Chapter 8, which is adapted from a paper that appeared in the Symposium on Operating Systems Principles in 2001 [ZZNM01] and a subsequent journal version that will appear in Transactions on Computer Systems [ZZNM02]. The proof from 8.4 is taken in its entirety from the latter.

Finally, Chapter 9 concludes with a summary of the contributions and some future directions.

Chapter 2

Defining Information-Flow Security

This chapter introduces the lattice model for specifying confidentiality and integrity levels of data manipulated by a program. It then shows how to use those security-level specifications to define the noninterference security policy enforced by the type systems in this thesis.

2.1 Security lattices and labels

Security-typed languages provide a way for programmers to specify confidentiality and integrity requirements in the program. They do so by adding explicit annotations at appropriate points in the code. For example, the declaration `int{Secret} h` indicates that `h` has confidentiality label `Secret`.

Following the work on multilevel security [BP76, FLR77, Fei80, McC87, MR92b] and Denning's original work on program analysis [Den75, Den76, DD77], the security levels that can be ascribed to the data should form a lattice.

Definition 2.1.1 (Lattice) *A* **lattice** $\mathcal{L}$ *is a pair* $\langle L, \sqsubseteq \rangle$*. Where* L *is a set of* **elements** *and* $\sqsubseteq$ *is a reflexive, transitive, and anti-symmetric binary relation (a partial order) on* L*. In addition, for any subset* X *of* L*, there must exist both least upper and greatest lower bounds with respect to the* $\sqsubseteq$ *ordering.*

An **upper bound** *for a subset* X *of* L *is an element* $\ell \in L$ *such that* $x \in X \Rightarrow x \sqsubseteq \ell$*. The* **least upper bound** *or* **join** *of* X *is an upper bound* ℓ *such that for any other upper bound* z *of* X*, it is the case that* $\ell \sqsubseteq z$*. It is easy to show that the least upper bound of a set* X*, denoted by* $\bigsqcup X$*, is uniquely defined. In the special case where* X *consists of two elements* x_1 *and* x_2*, the notation* $x_1 \sqcup x_2$ *is used to denote their join.*

A **lower bound** *for a subset* X *of* L *is an element* $\ell \in L$ *such that* $x \in X \Rightarrow \ell \sqsubseteq x$*. The* **greatest lower bound** *or* **meet** *of* X *is a lower bound* ℓ *such that for any other lower bound* z *of* X*, it is the case that* $z \sqsubseteq \ell$*. It is easy to show that the greatest lower*

bound of a set X, denoted by $\bigsqcap X$, is uniquely defined. In the special case where X consists of two elements x_1 and x_2, the notation $x_1 \sqcap x_2$ is used to denote their meet.

Note that because a lattice is required to have a join for all subsets of L there must be a join for L itself, denoted by $\top \stackrel{\text{def}}{=} \bigsqcup L$. By definition, it must be the case that $\ell \sqsubseteq \top$ for any element $\ell \in L$, that is, $\top$ is the **greatest** *or* **top** *element of the lattice. Similar reasoning establishes that there is a* **least** *or* **bottom** *element of the lattice, denoted by $\bot \stackrel{\text{def}}{=} \bigsqcap L$.*

One example of a confidentiality lattice is the classification used by the Department of Defense in their "Orange Book" [DOD85]:

$$\texttt{Unclassified} \sqsubseteq \texttt{Confidential} \sqsubseteq \texttt{Secret} \sqsubseteq \texttt{Top Secret}$$

An even simpler lattice that will be useful for examples in what follows is the two point lattice:

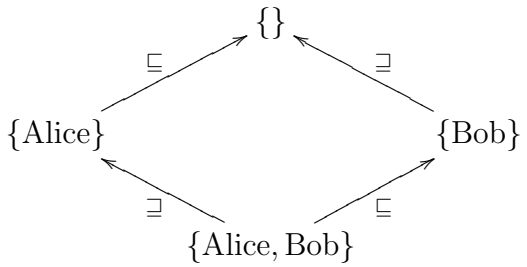
$$\bot \sqsubseteq \top$$

This lattice is just a renaming of the lattice already used in the examples at the beginning of this chapter:

$$\texttt{Public} \sqsubseteq \texttt{Secret}$$

Another example is a *readers* lattice that is generated from a set of principal identifiers, P. The elements of the lattice are given by $\mathcal{P}(P)$, the powerset of P. The order $\sqsubseteq$ is the reverse of the usual set inclusion. Intuitively, information about a piece of data labeled with the set of principals $\{p_1, \ldots, p_n\} \subseteq P$ should only be observable by members p_1 through p_n. Thus the set P itself is the most public element, and the empty set (indicating that the information should be invisible to all principals) is the most confidential.

As an example of a readers lattice, consider the case where there are two principals, Alice and Bob. The resulting label lattice is:



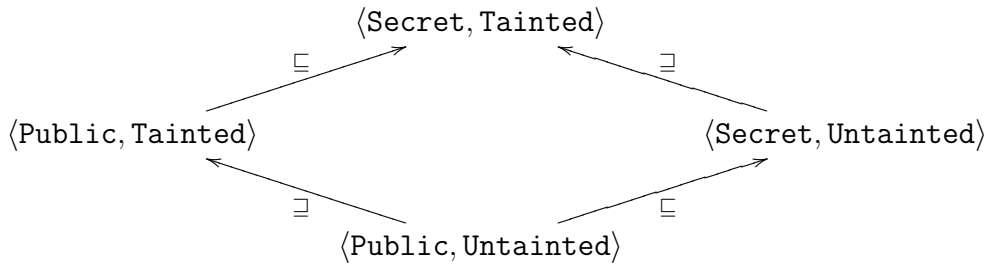
All of the lattices shown above are intended to describe confidentiality policies; lattices can also describe integrity policies. The simplest such lattice is:

$$\texttt{Untainted} \sqsubseteq \texttt{Tainted}$$

Note that this lattice is isomorphic to the `Public` $\sqsubseteq$ `Secret` lattice. Why is that? Intuitively, `Secret` information has more restrictions on where it can flow than `Public` information—`Secret` data should not flow to a `Public` variable, for instance. Similarly, `Tainted` information has more restrictions on its use than `Untainted` information. Both `Secret` and `Tainted` data should be prevented from flowing to points lower in the lattice. Formally, confidentiality and integrity are duals [Bib77].

In view of this duality, in this thesis, *high security* means "high confidentiality" or "low integrity" and *low security* means "low confidentiality" or "high integrity." *High* and *low* are informal ways of referring to relative heights in a lattice where $\ell_1 \sqsubseteq \ell_2$ means that ℓ_1 is bounded above by ℓ_2 and $\ell_1 \not\sqsubseteq \ell_2$ means that ℓ_1 is not bounded above by ℓ_2. The terminology "ℓ_1 is protected by ℓ_2" will also be used to indicate that $\ell_1 \sqsubseteq \ell_2$—intuitively it is secure to treat data with label ℓ_1 as though it has label ℓ_2 because the latter label imposes *more* restrictions on how the data is used.

As a final example of a security lattice, both integrity and confidentiality can be combined by forming the appropriate product lattice, as shown below:



The lattice elements are also used to describe the privileges of users of the program, hence determining what data should be visible to them. For instance, in the DoD lattice, a user with clearance `Secret` is able to learn information about `Unclassified`, `Classified`, and `Secret` data, but should not learn anything about `Top Secret` data.

The choice of which lattice to use is dependent on the desired security policies and level of granularity at which data is to be tracked. For simple security, the DoD style lattice may suffice; for finer control over the security levels of data more complex lattices, such as those found in Myers' and Liskov's decentralized label model [ML98, ML00] should be used.

Despite the importance of the security lattice with regard to the security policies that can be expressed, it is useful to abstract from the particular lattice in question. Consequently, all of the results in this thesis are derived for an arbitrary choice of security lattice.

2.1.1 Lattice constraints

The type systems in this thesis can be thought of as generating a system of lattice inequalities based on security annotations of a program in question. For example, consider the program that assigns the contents of the variable x to the variable y:

```
y := x
```

Suppose that the labels assigned to the variables x and y are $\mathsf{label}(\mathtt{x})$ and $\mathsf{label}(\mathtt{y})$ respectively. The assignment is permissible if $\mathsf{label}(\mathtt{x}) \sqsubseteq \mathsf{label}(\mathtt{y})$, because this constraint says that x contains more public (or less tainted) data than y is allowed to hold. Concretely, suppose that $\mathsf{label}(\mathtt{x}) = \mathtt{Secret}$ and $\mathsf{label}(\mathtt{y}) = \mathtt{Public}$. The program above would generate the constraint $\mathtt{Secret} \sqsubseteq \mathtt{Public}$, which is not satisfiable in the simple security lattice. On the other hand, if $\mathsf{label}(\mathtt{y}) = \mathtt{Secret}$, then the constraint $\mathsf{label}(\mathtt{x}) \sqsubseteq \mathtt{Secret}$ is satisfiable no matter what $\mathsf{label}(\mathtt{x})$ is.

The lattice structure is used to approximate the information contained in a piece of data that results from combining two pieces of data. For example, the expression `x + y` is given the security label $\mathsf{label}(\mathtt{x}) \sqcup \mathsf{label}(\mathtt{y})$, which intuitively says that the expression `x + y` may reveal information about either x or y. If either x or y is Secret, then the result of `x + y` is Secret.

To determine whether the assignment `z := x + y` is legal, we determine whether the constraint $\mathsf{label}(\mathtt{z}) \sqsubseteq \mathsf{label}(\mathtt{x}) \sqcup \mathsf{label}(\mathtt{y})$ is satisfiable.

The type system generates similar lattice inequations for all of the program statements, reducing the problem of determining whether a program is secure to a lattice-inequality constraint satisfaction problem. The correctness theorem for a security-type system says that if the constraints are satisfiable then the program does not leak information. The meaning of "does not leak information" is made precise in the next section.

The complexity of determining whether a program obeys the noninterference policy rests on the ability to solve systems of lattice inequalities. In general, this problem is NP-complete for finite lattices: it is simple to reduce 3SAT to the lattice constraint satisfaction problem because Boolean algebras constitute lattices and implication can be encoded via $\sqsubseteq$.

There are properties of the security lattice and the system of inequalities that can make it easier to determine whether a solution exists [RM96]. One possibility is that the system has only inequalities that can be written in the form $a \sqsubseteq b \sqcup c$, for example, and does not contain more complex constraints like $a \sqcap b \sqsubseteq c \sqcup d$. Disallowing meets on the left of inequalities reduces the search space of candidate solutions.

Another useful lattice property is distributivity, which means that:

$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

Distributivity alone is not enough to admit a polynomial time constraint satisfaction algorithm (Boolean algebras are also distributive). However, distributitivy allows inequalities to be put into normal forms that, with additional restrictions like the one above, make efficient constraint satisfaction algorithms possible.

Despite the importance of obtaining tractable constraint sets from the type system, this thesis is not concerned with the complexity of solving the lattice constraints. Happily, however, practical applications often make use of distributive lattices (see Myers' and Liskov's Decentralized Label Model [ML98, ML00] for example). The constraints generated by the type systems in this thesis also do not contain meets on the left of inequalities.

2.2 Noninterference

This section describes a general methodology for defining information-flow security in programming languages. The goal is a formal definition of *noninterference*, a basic security policy that intuitively says that high-security information cannot affect the results of low-security computation.

This thesis is concerned with regulating information flows that are *internal* to a program. In particular, the type systems presented attempt to address only information flows that can be detected because they alter the behavior of a program as it runs. This means that programs deemed to be secure might still contain external timing leaks or abstraction-violation channels.

For instance, the following Java-like[1] insecure, under the assumption that the method `System.print` prints to a public location:

```
class C {
  public static void main(string[] args) {
    String{Secret} combination = System.input();
    System.print("The secret combination is : " + combination);
  }
}
```

Here, the value of the string stored in the variable `combination` (which has been explicitly declared to be secret) affects the behavior of the program. The purpose of the security-typed languages is to rule out these kind of information flows.

The basic approach to defining noninterference is the following. Each step is described in more detail below.

[1] Java's keyword `public`, in contrast to the label `Public`, describes the scope of fields or methods, not their confidentiality level. Such scoping mechanisms are considerably weaker than the information-flow policies described in this thesis.

1. Choose an appropriate formal model of computation equipped with a meaningful (implementable) semantics. The language should have *values*—data objects—and programs should describe computations over those values.

$$3, \texttt{true}, \ldots \in \mathsf{Values} \qquad \texttt{z := x + 3}, \ldots \in \mathsf{Programs}$$

2. Derive from the semantics a definition of program equivalence, starting from an apparent equivalence on the values of the language. This equivalence should be sound with respect to the language semantics in the sense that equivalent programs should produce equivalent observable results.

$$\forall P_1, P_2 \in \mathsf{Programs}.\ P_1 \approx P_2 \Leftrightarrow \ldots$$

3. Enrich the program model using a security lattice as described in the previous section. This yields a way of specifying the high- and low-security interfaces (written with a Γ) to a program P.

   An interface Γ to a program describes a set of contexts in which it makes sense to run the program. In this thesis, the interfaces will be type environments that describe what variables or memory locations are available for use within the program P. Assertions like the following say that program P has high- and low-security interfaces Γ_{High} and Γ_{Low}:

$$\Gamma_{\mathsf{High}}, \Gamma_{\mathsf{Low}} \vdash P$$

4. Define the powers of the low-security observers of the system. This is done by coarsening the standard notion of process equivalence $\approx$ to ignore the high-security parts of the program. This new equivalence, $\approx_{\mathsf{Low}}$ represents the low-security view of the computation; it depends on the low-security interface to the program (Γ_{Low}). Treating the equivalence relations as sets, coarsening $\approx$ is the requirement that $\approx \subseteq \approx_{\mathsf{Low}}$.

5. Define a set of high-security inputs for the program, these values should match the interface Γ_{High}, so that $v \in \mathsf{Values}(\Gamma_{\mathsf{High}})$.

6. Define noninterference from the above components: There is no illegal information flow through a program P iff the low-security behavior of the program is independent of what high-security inputs are given to the program. Formally, $P \in \mathsf{Programs}$ is information-flow secure (satisfies noninterference) iff

$$\Gamma_{\mathsf{High}}, \Gamma_{\mathsf{Low}} \vdash P \ \Rightarrow\ \forall v_1, v_2 \in \mathsf{Values}(\Gamma_{\mathsf{High}}).\ P(v_1) \approx_{\mathsf{Low}} P(v_2)$$

This basic recipe for defining information-flow security will be used in this thesis for a variety of different programming languages. For each language, a type system that establishes noninterference for programs written in the language are given. However, there are many details left unspecified in the high-level overview given above, so it is worth going into each of the steps in more depth.

Step 1: Language definition

The first step is to choose a notion of program (or process, or system) that includes a computationally meaningful semantics. For example, one could pick the untyped lambda calculus and give its semantics via β-reduction. Another choice could be the Java programming language with semantics given by translation to the bytecode interpreter (which in turn has its own semantics).

The language semantics should include an appropriate notion of the observable behavior of programs written in the language. The observable behavior is usually formalized as an evaluation relation between program terms and values computed (large-step operational semantics), or perhaps a finer-grained model of the computation via a suitable abstract machine (small-step operational semantics).

Step 2: Program equivalence

The next step is to choose a basic definition of program equivalence; typically this equivalence is derived from and must respect the behavioral semantics of the language. For example, one might choose β-η equivalence for the untyped lambda calculus. Giving an appropriate definition of equivalence for Java programs is considerably harder; nevertheless, some well-defined notion of equivalence is necessary. (Programmers and compiler writers make use of program equivalences all the time to reason about changes they make to a program, so this is not an unreasonable requirement.)

The choice of language behavioral semantics, together with the accompanying equivalence, determines the level of detail in the model. For example, the lambda calculus provides a very abstract model of computation that is quite far from the behavior of actual computers, whereas, in principle, one could characterize the precise operational specification of a particular microprocessor.

There is a trade off between the accuracy of the information-flow analysis and the generality of the results. This thesis concentrates on a relatively abstract level of detail in an idealized computer.

Step 3: Security types

It is impossible to define security without specifying a security policy to be enforced. Consequently, the next step in defining information-flow security is to enrich the programming language so it can describe the confidentiality or integrity of the data it manipulates. This is done by associating a label, drawn from a security lattice, with the types of the data manipulated by the program.

Consider the example Java-like program from the introduction:

```
class C {
  public static void main(string[] args) {
    String{Secret} combination = System.get();
    System.print("The secret combination is : " + combination);
  }
}
```

The declaration `String{Secret}` indicates that the variable contains secret data. A similar annotation on the `print` method can indicate that its `String` argument is printed to a console visible to the public—`print` constitutes a channel through which the program's behavior can be observed. In Java-like notation,[2] `print`'s type can be written as: `void System.print(String{Public} x)`

Except where Java or Jif programs are considered (see Chapters 6 and 8), this thesis adopts a more abstract syntax for security types. If t is a type in the language and ℓ is a security label, then t_ℓ is a security type. This notation is more compact than the $t\{\ell\}$ used in the example above.

Step 4: Low-security behavioral equivalence

The next step is to define an appropriate notion of low-level or low-security equivalence. Intuitively, this equivalence relation hides the parts of the program that should not be visible to a low-level observer.

For example, consider the set of states consisting of pairs $\langle h, l\rangle$, where h ranges over some high-security data and l ranges over low-security data. An observer with low-security access (only permitted to see the l component) can see that the states $\langle \texttt{attack at dawn}, 3\rangle$ and $\langle \texttt{do not attack}, 4\rangle$ are different (because $3 \neq 4$), but will be unable to distinguish the states $\langle \texttt{attack at dawn}, 3\rangle$ and $\langle \texttt{do not attack}, 3\rangle$. Thus, with respect to this view ($\approx_{\mathsf{Low}}$):

$$\langle \texttt{attack at dawn}, 3\rangle \approx_{\mathsf{Low}} \langle \texttt{do not attack}, 3\rangle$$
$$\langle \texttt{attack at dawn}, 3\rangle \not\approx_{\mathsf{Low}} \langle \texttt{do not attack}, 4\rangle$$

[2] In more traditional type-theoretic notation, this type might be written as: `System.print : String{Public}` $\rightarrow$ `unit`

It is necessary to generalize this idea to include other parts of the program besides its state—the computations must also have a suitable notion of low equivalence. The choice of observable behavior impacts the strength of the noninterference result. For example, if the equivalence on computations takes into account running time, then noninterference will require that high-security information not affect the timing behavior of the program. This thesis, as elsewhere in the security literature, generalizes low-equivalence to computations via appropriate bisimulation relations [LV95, Mil89].

Also, because the security lattice contains many points, and the program should be secure only if *all* illegal information flows are ruled out, we must also generalize to equivalence relations indexed by an arbitrary lattice element ℓ. The relation $\approx_\ell$ represents the portion of the computation visible to an observer at security level ℓ.

Step 5: High-security inputs

Because we are interested in preventing information flows from high-security sources to lower-security computation, we must specify how the high-security information is generated. The next step of defining information flows is to pick an appropriate notion of high-security inputs.

For simple datatypes such as Booleans and integers, any value of the appropriate type is suitable as a high-security input. However, if the high-security input is a function or some other higher-order datatype (like an object), then this input itself can lead to insecure behavior—when the insecure function is invoked, for instance.

Any security analysis that establishes noninterference must guarantee that insecure inputs are not used by the program. In practice, this can be accomplished by analyzing the inputs, i.e. requiring them to type check.

Step 6: Noninterference

Combining the steps above, we obtain a suitable definition of noninterference:

$$\Gamma_{\mathsf{High}}, \Gamma_{\mathsf{Low}} \vdash P \;\Rightarrow\; \forall v_1, v_2 \in \mathsf{Values}(\Gamma_{\mathsf{High}}).\; P(v_1) \approx_{\mathsf{Low}} P(v_2)$$

This definition says that a program P is secure if changing the high-security values of the initial state does not affect the low-security observable behavior of the program.

2.3 Establishing noninterference

The security-typed languages studied in this thesis rule out insecure information flows by augmenting the type system to constrain how high-security data is handled by the program. To connect these nonstandard type systems to information security, we must

prove that well-typed programs satisfy an appropriate definition of noninterference. As we have seen, noninterference is a statement about how the program *behaves*. Therefore one must connect the static analysis of a program to the program's operational behavior. As with ordinary type systems, the main connection is a *soundness theorem* that implies that well-typed programs do not exhibit undesired behavior (such as accessing initialized memory locations).

In the case of information-flow properties, we take this proof technique one step further: we instrument the operational semantics of the programming language to include labels. This nonstandard operational semantics is constructed so that it tracks information flows during program execution. For example, suppose that the standard semantics for the language specifies integer addition using rules like $3+4 \Downarrow 7$, where the $\Downarrow$ symbol can be read as "evaluates to". The labeled operational semantics requires that the values 3 and 4 to be tagged with security labels. Supposing that the labels are drawn from the two point lattice, we might have $3_\top$ and $4_\bot$. The nonstandard rule for arithmetic addition would show that $3_\top + 4_\bot \Downarrow 7_{(\top \sqcup \bot)}$, where we use the lattice join operation ($\sqcup$) to capture that the resulting value reveals information about both of the operands.

Importantly, the instrumented operational semantics agrees with the original semantics: erasing all of the additional label information from an execution trace of the nonstandard programs yields a valid execution trace of the standard program. This implies that any results about the nonstandard operational semantics apply to the standard program as well. This *erasure* property is also important, because it means that, even though the instrumented operational semantics makes use of labels at run time, a real implementation of the security-typed language does not need to manipulate labels at run time.

The strategy adopted in this thesis for establishing noninterference thus consists of four steps.

1. Construct a labeled operational semantics safely approximates the information flows in a program.

2. Show that the security type system is sound with respect to the nonstandard semantics.

3. Use the additional structure provided by the labeled semantics to show that noninterference conditions hold for instrumented programs.

4. Use the erasure property to conclude that the standard behavior of a program agrees with the nonstandard behavior, which implies that the standard program satisfies noninterference.

The next three chapters illustrate this process for three different languages that incorporate increasingly rich programming features.

2.4 Related work

There is a considerable amount of work related to specifying noninterference-style information-policies and generalizing those definitions to various models of computation.

The enforcement of information-flow policies in computer systems has its inception in Bell and La Padula's work on a multi-level security policy for the MULTICS operating system [BL75]. At roughly the same time, Denning proposed the lattice-model of secure information flow [Den76] followed by a means of certifying that programs satisfy a strong information-flow policy [DD77]. However, no correctness result was given for this approach, partly due to a lack of a formal characterization of what it means for a program to be insecure.

Goguen and Meseguer addressed this problem of formalizing information-security by proposing the first definition of noninterference in 1982 [GM82]. The intuitions underlying their definition of noninterference are the same as those used to motivate the definition of noninterference in this thesis. Their original definition was suitable for deterministic state machines and used traces of states to represent systems, rather than the language and context formulation used here.

Many definitions of information security similar to noninterference have been proposed, and there is no general agreement about which definition is appropriate for what scenarios. Two major classifications of security properties have emerged.

In the *possibilistic* setting, the set of possible outcomes that might result from a computation are considered the important factor [Sut86, McC88, McL88b, McL88a, McL90, WJ90, McL94, ZL97, Zha97]. A system is considered possibilistically secure if the actions of a high-security observer do not affect the set of possible outcomes. *Probabilistic* security, in contrast, requires that high-security events are not able to affect the probability distribution on the possible outcomes of a system [Gra90, Gra91, GS92]. For sequential programs, possibilistic and probabilistic security coincide—there is only one possible outcome of running the system and it occurs with probability 1.

The results in the work discussed above are presented at the level of state machines that represent an entire system, typically a whole computer or complete program. Security properties are expressed as predicates over sets of traces which correspond to runs of the state machine on various inputs. This level of detail abstracts away from the implementation details of the system, which is good from the point of view of specification, but does not contain enough detail to give rise to any principle for building secure system. Sabelfeld and Mantel bridge the gap between the labeled transition models and programming-languages approaches to information security [MS01] by showing how to encode the actions of a simple programming language in the labeled transition model.

The definition of noninterference used here is closer to those used in the programming languages community [VSI96, HR98, PC00] and is equivalent to them for se-

quential programs. The presentation of nointerference in this thesis draws on the idea contextual equivalence [Mor68].

Language-based security extends beyond information-flow control [SMH00]. Work on Typed Assembly Language [MWCG99, MCG+99, CWM99] and proof-carrying code [Nec97] emphasizes static checking of program properties. In-lined reference monitors [ES99, ET99] use code rewriting techniques to enforce security policies on existing software. Buffer overflow detection, a common source of security holes, has also been treated via static program analysis [LE01] and dynamic techniques [CPM+98].

Chapter 3

Secure Sequential Programs

This chapter introduces two secure source calculi. They serve as examples of the basic methodology introduced in Chapter 2, and the remainder of this thesis builds on them.

The first language, λ_{SEC}, is a case study for introducing the basic definitions and notation. It is a purely functional, simply-typed lambda calculus that includes the minimal extensions for expressing confidentiality policies. Section 3.1 describes λ_{SEC} in detail, explains its type system, and proves that well-typed programs enjoy the noninterference security property.

The second language, $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$, serves as a vehicle for discussing the problems of information flows that can occur through *side effects* in a program. It extends λ_{SEC} with mutable state and recursion, to obtain a Turing-complete language. The type system for $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ must be more complicated to account for information flows that can arise from aliasing and mutations to the store. Section 3.2 describes the language, its operational semantics and the type system for ensuring security. Noninterference is not proved for $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ directly; instead, that result is obtained in Chapter 4 using a semantics-preserving translation into a CPS-style language.

3.1 λ_{SEC}: a secure, simply-typed language

Figure 3.1 describes the grammar for λ_{SEC}, a purely functional variant of the simply-typed lambda calculus that includes security annotations. This language is a simplified variant of the SLam calculus, developed by Heintze and Riecke [HR98].

In the grammar, the metavariables ℓ and `pc` range over elements of the security lattice. The possible types include the type `bool` of Boolean values and the types of functions $(s \rightarrow s)$ that expect a security-annotated value as an argument and produce a security-annotated type as a result. Security types, ranged over by the metavariable s, are just ordinary types labeled with an element from the security lattice.

ℓ, pc	$\in$	$\mathcal{L}$	Security labels
t	$::=$	`bool`	Boolean type
	$\mid$	$s \to s$	Function type
s	$::=$	t_ℓ	Security types
bv	$::=$	`t` $\mid$ `f`	Boolean base values
	$\mid$	$\lambda x\!:\!s.\ e$	Functions
v	$::=$	x	Variables
	$\mid$	bv_ℓ	Secure Values
e	$::=$	v	Values
	$\mid$	$e\ e$	Function application
	$\mid$	$e \oplus e$	Primitive operations
	$\mid$	`if` e `then` e `else` e	Conditional
$\oplus$	$::=$	$\wedge \mid \vee \mid \ldots$	Boolean operations

Figure 3.1: λ_{SEC} grammar

Base values, in the syntactic class bv, include the Boolean constants for true and false as well as function values. All computation in a security-typed language operates over secure-values, which are simply base values annotated with a security label. Variables, ranged over by the metavariable x, denote secure values.

Expressions include values, primitive Boolean operations such as the logical "and" operation $\wedge$, function application, and a conditional expression.

To obtain the underlying unlabeled lambda-calculus term from a λ_{SEC} term, we simply erase the label annotations on security types and secure values. For any λ_{SEC} term e, let $\mathsf{erase}(e)$ be its label erasure. The resulting language is identical to standard definitions of the simply typed lambda-calculus [Mit96].

Definition 3.1.1 (Free and Bound Variables) *Let* $\mathrm{vars}(e)$ *be the set of all variables occurring in an expression* e*. The* **free** *and* **bound** *variables of an expression* e *are defined as usual for the lambda calculus. They are denoted by the functions* $\mathrm{fv}(-)$ *and* $\mathrm{bv}(-)$ *respectively.*

$$
\begin{array}{rcl}
\mathrm{fv}(\mathtt{t}_\ell) & = & \emptyset \\
\mathrm{fv}(\mathtt{f}_\ell) & = & \emptyset \\
\mathrm{fv}((\lambda x\!:\!s.\ e)_\ell) & = & \mathrm{fv}(e) \setminus \{x\} \\
\mathrm{fv}(e_1\ e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(e_1 \oplus e_2) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) & = & \mathrm{fv}(e) \cup \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
 & & \\
\mathrm{bv}(e) & = & \mathrm{vars}(e) \setminus \mathrm{fv}(e)
\end{array}
$$

Following Barendregt[Bar84], this thesis uses the **bound variable convention**: the terms are identified up to consistent renaming of their bound variables. Two such terms are said to be α**-equivalent**, and this is indicated by the notation $e_1 =_\alpha e_2$. Usually, however, terms will be considered to implicitly stand for their $=_\alpha$-equivalence classes; consequently, bound variables may be renamed so as not to conflict.

Definition 3.1.2 (Program) *A* **program** *is an expression e such that $\mathrm{fv}(e) = \emptyset$. Such an expression is said to be* **closed***. Expressions that contain free variables are* **open***.*

3.1.1 Operational semantics

For simplicity, we give λ_{SEC} a large-step operational semantics. The standard evaluation relation is of the form $e \Downarrow_S v$, which means that the (closed) program e evaluates to the value v. The definition of the $\Downarrow_S$ relation is given in Figure 3.2.[1] Figure 3.3 shows the instrumented operational semantics, which is derived from the standard operational semantics by adding labels.

Values evaluate to themselves; they require no further computation, as indicated by the rule λ_{SEC}-EVAL-VAL.

Binary Boolean operators are evaluated using the rule λ_{SEC}-EVAL-BINOP. Here, the notation $[\![\oplus]\!]$ is the standard semantic function on primitive values corresponding to the syntactic operation $\oplus$. For example:

$$
\begin{array}{rcl}
\mathtt{t}\ [\![\wedge]\!]\ \mathtt{t} & = & \mathtt{t} \\
\mathtt{t}\ [\![\wedge]\!]\ \mathtt{f} & = & \mathtt{f} \\
\mathtt{f}\ [\![\wedge]\!]\ \mathtt{t} & = & \mathtt{f} \\
\mathtt{f}\ [\![\wedge]\!]\ \mathtt{f} & = & \mathtt{f}
\end{array}
$$

[1] The box at the top of the figure (and subsequent figures in this thesis) illustrates the form of the relation defined by the figure. Rules are named by appending a short description in SMALL CAPS to the name of the language to which the rule pertains.

$\boxed{e \Downarrow_S v}$

λ_{SEC}-SEVAL-VAL
$$v \Downarrow_S v$$

λ_{SEC}-SEVAL-BINOP
$$\frac{e_1 \Downarrow_S bv_1 \quad e_2 \Downarrow_S bv_2}{e_1 \ \oplus \ e_2 \Downarrow_S bv_1 [\![\oplus]\!] bv_2}$$

λ_{SEC}-SEVAL-COND1
$$\frac{e \Downarrow_S \mathtt{t} \quad e_1 \Downarrow_S v}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow_S v}$$

λ_{SEC}-SEVAL-COND2
$$\frac{e \Downarrow_S \mathtt{f} \quad e_2 \Downarrow_S v}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow_S v}$$

λ_{SEC}-SEVAL-APP
$$\frac{e_1 \Downarrow_S \lambda x{:}s.\, e \quad e_2 \Downarrow_S v \quad e\{v/x\} \Downarrow_S v'}{e_1\ e_2 \Downarrow_S v'}$$

Figure 3.2: Standard large-step operational semantics for λ_{SEC}

$\boxed{e \Downarrow v}$

λ_{SEC}-EVAL-VAL
$$v \Downarrow v$$

λ_{SEC}-EVAL-BINOP
$$\frac{e_1 \Downarrow (bv_1)_{\ell_1} \quad e_2 \Downarrow (bv_2)_{\ell_2}}{e_1 \ \oplus \ e_2 \Downarrow (bv_1 [\![\oplus]\!] bv_2)_{(\ell_1 \sqcup \ell_2)}}$$

λ_{SEC}-EVAL-COND1
$$\frac{e \Downarrow \mathtt{t}_\ell \quad e_1 \Downarrow v}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v \sqcup \ell}$$

λ_{SEC}-EVAL-COND2
$$\frac{e \Downarrow \mathtt{f}_\ell \quad e_2 \Downarrow v}{\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Downarrow v \sqcup \ell}$$

λ_{SEC}-EVAL-APP
$$\frac{e_1 \Downarrow (\lambda x{:}s.\, e)_\ell \quad e_2 \Downarrow v \quad e\{v/x\} \Downarrow v'}{e_1\ e_2 \Downarrow v' \sqcup \ell}$$

Figure 3.3: Labeled large-step operational semantics for λ_{SEC}

As shown in the evaluation rule, the labels on the arguments to the binary operation are joined to produce the label on the result. This is necessary because it is possible to learn information about the arguments based on the outcome of the operation. As a simple example, we have:

$$\mathtt{t}_\top \wedge \mathtt{f}_\bot \Downarrow \mathtt{f}_\top$$

The pair of rules λ_{SEC}-EVAL-COND1 and λ_{SEC}-EVAL-COND2 describe the behavior of conditional expressions. First, the conditional expression is evaluated to a Boolean value. If the result is $\mathtt{t}_\ell$ the first branch is evaluated, otherwise the second branch is. The confidentiality label ℓ of the `if` guard propagates to the result of the condition expression because the result depends on information contained in the guard.

The notation $v \sqcup \ell$ in these rules is a convenient abbreviation used throughout this thesis. This operation simply joins the label ℓ into the label tagging a secure value:

Definition 3.1.3 (Label Stamping) *Let bv_ℓ be any secure value and ℓ' be any label in the security lattice.*

$$bv_\ell \sqcup \ell' \stackrel{\mathrm{def}}{=} bv_{(\ell \sqcup \ell')}$$

As an example of how this operational semantics propagates the security labels to account for information flows, we have the following derivation tree, which says that the results of branching on high-security data are high-security:

$$\frac{\mathtt{t}_\top \Downarrow \mathtt{t}_\top \quad \mathtt{t}_\bot \Downarrow \mathtt{t}_\bot}{\mathtt{if}\ \mathtt{t}_\top\ \mathtt{then}\ \mathtt{t}_\bot\ \mathtt{else}\ \mathtt{f}_\bot \Downarrow \mathtt{t}_\top}$$

Finally, rule λ_{SEC}-EVAL-APP shows the operational behavior of function application. The left expression must evaluate to a function value. The right expression must evaluate to a value. Finally the actual parameter to the function call is substituted for the bound variable in the body of the function to obtain the result.

The application rules make use of *capture-avoiding substitution*, a concept used throughout this thesis:

Definition 3.1.4 (Capture-Avoiding Substitution) *Let e_1 and e_2 be expressions and let x be a variable. The* **capture-avoiding substitution** *of e_1 for x within e_2 is written $e_2\{e_1/x\}$. Such a substitution is well defined when $\mathrm{fv}(e_1) \cap \mathrm{bv}(e_2) = \emptyset$, that is, whenever none of the binding occurrences of variables in e_2 can capture the free variables of e_1. Note that it is always possible to choose a term α-equivalent to e_2 so that substitution may occur.*

A substitution $e_2\{e_1/x\}$ results in a new term in which the free occurrences of the variable x in e_1 have been replaced by the expression e_1. It is defined inductively on the structure of e_2.

$$
\begin{array}{rcll}
x\{e_1/x\} & \stackrel{\text{def}}{=} & e_1 & \\
y\{e_1/x\} & \stackrel{\text{def}}{=} & y & \textit{(when } x \neq y\textit{)} \\
\mathtt{t}_\ell\{e_1/x\} & \stackrel{\text{def}}{=} & \mathtt{t}_\ell & \\
\mathtt{f}_\ell\{e_1/x\} & \stackrel{\text{def}}{=} & \mathtt{f}_\ell & \\
(\lambda y{:}t.\, e)_\ell\{e_1/x\} & \stackrel{\text{def}}{=} & (\lambda y{:}t.\, e\{e_1/x\})_\ell & \textit{(}x \neq y \textit{ by assumption)} \\
(e\ e')\{e_1/x\} & \stackrel{\text{def}}{=} & (e\{e_1/x\}\ e'\{e_1/x\}) & \\
(\mathtt{if}\ e\ \mathtt{then}\ e'\ \mathtt{else}\ e'')\{e_1/x\} & \stackrel{\text{def}}{=} & \mathtt{if}\ e\{e_1/x\}\ \mathtt{then}\ e'\{e_1/x\}\ \mathtt{else}\ e''\{e_1/x\} &
\end{array}
$$

In λ_{SEC}-EVAL-APP the security label on the function being applied is stamped into the results of calling the function. Such a restriction, in combination with the rule for conditionals, prevents information flows that arise when high-security data influences the choice of which function gets applied to a piece of data.

As an example, consider the following program that applies either the Boolean identity or Boolean negation, based on high-security information. It also results in a high-security value:

$$(\mathtt{if}\ \mathtt{f}_\top\ \mathtt{then}\ (\lambda x{:}\mathtt{bool}_\bot.\, x)_\bot\ \mathtt{else}\ (\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\bot)\ \mathtt{t}_\bot\ \Downarrow\ \mathtt{f}_\top$$

This program shows the propagation of a high-security label through the conditional expression and then through the resulting function application, as seen in these subderivations that are part of its evaluation:

$$\frac{\mathtt{f}_\top \Downarrow \mathtt{f}_\top \quad (\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\bot \Downarrow (\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\bot}{(\mathtt{if}\ \mathtt{f}_\top\ \mathtt{then}\ (\lambda x{:}\mathtt{bool}_\bot.\, x)_\bot\ \mathtt{else}\ (\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\bot)\ \Downarrow\ (\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\top}$$

$$\frac{(\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\top \Downarrow (\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\top \quad \mathtt{t}_\bot \Downarrow \mathtt{t}_\bot \quad \dfrac{\mathtt{t}_\bot \Downarrow \mathtt{t}_\bot \quad \mathtt{f}_\bot \Downarrow \mathtt{f}_\bot}{(x \Rightarrow \mathtt{f}_\bot)\{\mathtt{t}_\bot/x\} \Downarrow \mathtt{f}_\bot}}{(\lambda x{:}\mathtt{bool}_\bot.\, x \Rightarrow \mathtt{f}_\bot)_\top\ \mathtt{t}_\bot\ \Downarrow\ \mathtt{t}_\top}$$

Finally, we note that the instrumented operational semantics of λ_{SEC} terms corresponds to the standard operational semantics.

Lemma 3.1.1 (Erasure) *If* $e \Downarrow v$ *then* $\mathsf{erase}(e) \Downarrow \mathsf{erase}(v)$.

Proof (sketch): By induction on the derivation of $e \Downarrow v$. For λ_{SEC}-EVAL-APP one must show that erasure commutes with substitution:

$$\mathsf{erase}(e)\{\mathsf{erase}(v)/x\} = \mathsf{erase}(e\{v/x\})$$

□

3.1.2 An aside on completeness

It is worth noting that the labels used in the operational semantics of this simple language are an approximation to the true information flows. For example, the following program could be deemed to produce a low-security result because its result does not depend on the high-security value used in the conditional. Nevertheless, it is considered to return a high-security value

$$\texttt{if } \texttt{t}_\top \texttt{ then } \texttt{t}_\bot \texttt{ else } \texttt{t}_\bot$$

λ_{SEC}, a toy language, is not Turing complete: all λ_{SEC} programs eventually halt because their label erasures are simply-typed lambda calculus programs, which are known to terminate [Mit96]. In principle, it would be possible to fully characterize the information flows that arise in a λ_{SEC} program, but such an analysis would amount to running the program on all possible high-security inputs and determining whether it produced the same low-security output in each run. In the worst case, such an analysis could take time more than exponential in the program size, so approximating the information flows with the security lattice elements is justified.

In Turing-complete languages, like those presented later in this thesis, the problem of determining security is even harder. Because it is possible to reduce the halting problem to the problem of determining whether a program is secure, the question is undecidable. Thus, some form of approximation, like the lattice elements used here, is necessary.

3.1.3 λ_{SEC} type system

The type system for λ_{SEC} is designed to prevent unwanted information flows. The basic idea is to associate security-labels to the type information of the program and then take the confidentiality lattice into account when type checking so as to rule out illegal (downward) information flows.

This section examines the type system for λ_{SEC} in detail, and proves some basic properties that establish its soundness. The next section proves the desired noninterference result: well-typed programs are secure.

Because upward information flows are allowed (e.g. low-confidentiality data may flow to a high-confidentiality variable), the lattice ordering is incorporated as a subtyping relationship [VSI96]. This subtyping eliminates the need for the programmer to make explicit when information flows are permissible.

The subtype relationship is shown in Figure 3.4, which contains mutually-recursive rules of the form $\vdash t_1 \leq t_2$ and $\vdash s_1 \leq s_2$. The rules establish that $\leq$ is a reflexive, transitive relation that obeys the expected contravariance for function types.

The interesting rule is λ_{SEC}-SLAB, which allows a low-security type to be treated as a high-security type. For example, $\vdash \texttt{bool}_\bot \leq \texttt{bool}_\top$ because anywhere a high-security

$\boxed{\vdash t_1 \leq t_2}$ $\boxed{\vdash s_1 \leq s_2}$

$$\lambda_{\mathrm{SEC}}\text{-TREFL} \qquad \vdash t \leq t$$

$$\lambda_{\mathrm{SEC}}\text{-TTRANS} \qquad \frac{\vdash t \leq t' \quad \vdash t' \leq t''}{\vdash t \leq t''}$$

$$\lambda_{\mathrm{SEC}}\text{-TFUNSUB} \qquad \frac{\vdash s_1' \leq s_1 \quad \vdash s_2 \leq s_2'}{\vdash s_1 \to s_2 \quad \leq \quad s_1' \to s_2'}$$

$$\lambda_{\mathrm{SEC}}\text{-SLAB} \qquad \frac{\vdash t \leq t' \quad \ell \sqsubseteq \ell'}{\vdash t_\ell \leq t'_{\ell'}}$$

Figure 3.4: Subtyping for pure λ_{SEC}

Boolean can be safely used, a low-security Boolean can also be used. Intuitively, if the program is sufficiently secure to protect high-security data, it also provides sufficient security to "protect" low-security data.

The rules for type checking terms of λ_{SEC} are given in Figure 3.5. They are judgments of the form $\Gamma \vdash e : s$, which says "under the assumptions provided by Γ, the term e is a secure program that evaluates to a value of type s." Here, Γ is a type context that maps the free variables of the term e to their types:

Definition 3.1.5 (Type Environment) *A* **type environment** *is a finite map from variables to security types. Syntactically, type environments are written as terms in the following grammar:*

$$\Gamma ::= \cdot \mid \Gamma, x : s$$

Here, $\cdot$ stands for the empty type environment, and if Γ is any environment, then $\Gamma, x : s$ stands for a new environment in which the variable x is mapped to the type s.

The **domain** *of a type environment Γ, written $\mathrm{dom}(\Gamma)$, is simply the set of variables on which the finite map is defined. The notation $\Gamma(x)$ is used to indicate the type s to which x is mapped by Γ, and is undefined if Γ does not contain a mapping for x.*

To avoid unnecessary clutter, whenever the type environment is empty, the symbol $\cdot$ will be elided from the judgment. For example $\cdot \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell$ will be written as $\vdash \mathtt{t}_\ell : \mathtt{bool}_\ell$.

The rules of the type system make use of the intuitions formed from the operational semantics of the language—appropriate security information is propagated in such a

$\boxed{\Gamma \vdash e : s}$

λ_{SEC}-TRUE $\quad \Gamma \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell$

λ_{SEC}-FALSE $\quad \Gamma \vdash \mathtt{f}_\ell : \mathtt{bool}_\ell$

λ_{SEC}-VAR $\quad \dfrac{\Gamma(x) = s}{\Gamma \vdash x : s}$

λ_{SEC}-FUN $\quad \dfrac{\Gamma, x{:}s_1 \vdash e : s_2 \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda x{:}s_1.\, e)_\ell : (s_1 \to s_2)_\ell}$

λ_{SEC}-BINOP $\quad \dfrac{\Gamma \vdash e_1 : \mathtt{bool}_{\ell_1} \quad \Gamma \vdash e_2 : \mathtt{bool}_{\ell_2}}{\Gamma \vdash e_1 \ \oplus \ e_2 : \mathtt{bool}_{(\ell_1 \sqcup \ell_2)}}$

λ_{SEC}-APP $\quad \dfrac{\Gamma \vdash e_1 : (s_2 \to s)_\ell \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash e_1\ e_2 : s \sqcup \ell}$

λ_{SEC}-COND $\quad \dfrac{\Gamma \vdash e : \mathtt{bool}_\ell \quad \Gamma \vdash e_i : s \sqcup \ell \quad i \in \{1, 2\}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : s \sqcup \ell}$

λ_{SEC}-SUB $\quad \dfrac{\Gamma \vdash e : s \quad \vdash s \leq s'}{\Gamma \vdash e : s'}$

Figure 3.5: Typing λ_{SEC}

way that the potential dependencies of computation on high-security information are tracked. Because information is propagated only when data is examined in some way—that is, its value is used to alter the behavior of the computation—the interesting rules are the so-called *elimination* forms, which deconstruct values.

Rule λ_{SEC}-BINOP is a typical elimination rule: it ensures that the binary Boolean operations are applied to Boolean values, but it additionally carries along the appropriate security information. If the two operands are of security label ℓ_1 and ℓ_2 respectively, then the results of the binary operation should be labeled with their join, $\ell_1 \sqcup \ell_2$ to approximate the information that the resulting Boolean reveals about the operands.

Similarly, the rule for function application λ_{SEC}-APP ensures not only that the function is applied to an argument of the correct type, but also that the results of the function call will be no less secure than the function itself, namely $s \sqcup \ell$.

Lastly, the `if` expression must propagate the security label of the condition to the results of the computation, thus avoiding implicit information flows. Rule λ_{SEC}-COND incorporates this constraint.

The *introduction* rules show how to create data values; consequently it is in these rules that security annotations supplied by the programmer affect the labels that appear in the types of the expressions.

Rules λ_{SEC}-TRUE and λ_{SEC}-FALSE say that Boolean constants have type `bool`, and that they inherit whatever security annotation was declared in the program. Variables are simply looked up in the appropriate type environment, as indicated in rule λ_{SEC}-VAR. Similarly, the rule for function values, λ_{SEC}-FUN is standard: it says that the body of the function is well-typed when the formal parameter is assumed to have the type declared by the programmer.

The remaining rule, λ_{SEC}-SUB, plays an important role in the type system: it allows an expression that results in a low-security value to be used in a position where high-security data is expected.

Definition 3.1.6 (Substitution) *A* **substitution** γ *is a finite map from variables to values. If* Γ *is a typing environment and* γ *is a substitution, we write* $\gamma \models \Gamma$ *to mean that* γ *assigns each variable a value of the type required by* Γ*. It should be read "substitution* γ *satisfies environment* Γ*." Formally, we have:*

$$\mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma) \ \wedge \ \forall x \in \mathrm{dom}(\Gamma).\ \vdash \gamma(x) : \Gamma(x)$$

The notation $\gamma(e)$ *is short-hand for the simultaneous capture-avoiding substitutions:*[2]

$$\gamma(e) \stackrel{\mathrm{def}}{=} e\{\gamma(x_1)/x_1\}\{\gamma(x_2)/x_2\}\ldots\{\gamma(x_n)/x_n\} \ \textit{where} \ \{x_1, \ldots, x_n\} = \mathrm{dom}(\gamma)$$

In order to show that the λ_{SEC}-EVAL-APP rule preserves typing, we must show that substitution of well-typed values does not yield an ill-typed term. This requirement is captured in the following lemma.

Lemma 3.1.2 (Value Substitutions) *If* $\Gamma \vdash e : s$ *and* $\gamma \models \Gamma$ *then* $\vdash \gamma(e) : s$*.*

Proof: Standard proof by induction on the derivation of $\Gamma \vdash e : s$. □

In order to assess the outcome of evaluating a program, it is helpful to associate the types of the result with their possible values. The connection is quite strong: The types of values *determine* their syntactic structure.

[2] Note that because the values in the range of γ are closed, the substitutions may be done in any order.

Lemma 3.1.3 (Canonical Forms)

- *If* $\vdash v : \texttt{bool}_\ell$ *then* $v = \texttt{t}_{\ell'}$ *or* $v = \texttt{f}_{\ell'}$ *and* $\ell' \sqsubseteq \ell$.
- *If* $\vdash v : (s_1 \to s_2)_\ell$ *then* $v = (\lambda x{:}s_1'.\ e)_{\ell'}$ *and* $\vdash s_1 \leq s_1'$ *and* $\ell' \sqsubseteq \ell$.

Proof: By inspection of the forms for values and the typing rules. □

Lemma 3.1.4 (λ_{SEC} Preservation) *If* $\vdash e : s$ *and there exists a value* v *such that* $e \Downarrow v$ *then* $\vdash v : s$.

Proof: Standard proof by induction on the derivation that $\vdash e : s$, appealing to Lemma 3.1.2 in the case of λ_{SEC}-EVAL-APP. □

Preservation is weaker than full type soundness because it doesn't say that a well-typed program does not go "wrong" (makes progress). The standard way to prove such a result for a language with large-step operational semantics is to provide evaluation rules that result in errors and then show that well-typed programs never produce an error. Although such a soundness result could easily be formulated for this language, there is no reason to include it here.

3.1.4 Noninterference for λ_{SEC}

This section establishes a noninterference result for λ_{SEC}. In this simple setting, the only possible observation of a program is the value to which it evaluates, consequently, noninterference simply says that confidential information should not alter the public results of any expression. Formally, we want to establish:

Theorem 3.1.1 (Noninterference) *If* $x{:}t_H \vdash e : \texttt{bool}_L$ *and* $\vdash v_1, v_2 : t_H$ *then*

$$e\{v_1/x\} \Downarrow v \ \Leftrightarrow\ e\{v_2/x\} \Downarrow v$$

Proof: This theorem follows by using the method of logical relations as a special case of Lemma 3.1.6 below. □

The intuition behind the proof comes from thinking of the behavior of a secure program from the perspective of a low-security observer. For the program to be secure, you (the low-security observer) should not be able to see any of the actions performed on high-security data. On the other hand, you can see the low-security data and computations. The program is secure if you can't see any of the high-security data.

To formalize this intuition, we need to mathematically model what it means for an observer to be able to "see" (or not "see") a piece of data. If you can "see" a value,

you can distinguish it from other values of the same type: for instance if you can see the Boolean value `t`, you should be able to tell that it is not the value `f`. On the other hand, if you cannot see some Boolean value x, you should not be able to distinguish it from either `t` or `f`. Thus, to model the fact that two values are indistinguishable, we simply use an appropriate equivalence relation—two values are related if they can't be distinguished.

Whether or not a piece of data is visible to the low-security observer depends on its security annotation. Consequently, which equivalence relation to use to capture the observer's view of the data depends on the relationship between the observer's security clearance and the label on the value. Thus, we parameterize the equivalence relations with ζ, the security level of the observer.

Using the standard technique of logical relations [Mit96], we can extend these equivalence relations to higher-order pieces of data and computations over the data as follows:

Definition 3.1.7 (Security Logical Relations) *For an arbitrary element ζ of the security lattice, the ζ-**level security logical relations** are type-indexed binary relations on closed terms defined inductively as follows. The notation $v_1 \approx_\zeta v_2 : s$ indicates that v_1 is related to v_2 at type s. Similarly, the notation $e_1 \approx_\zeta e_2 : \mathcal{C}(s)$ indicates that e_1 and e_2 are related computations that produce values of type s.*

$$\begin{array}{rcl} v_1 \approx_\zeta v_2 : \mathtt{bool}_\ell & \Leftrightarrow & \vdash v_i : \mathtt{bool}_\ell \ \wedge \ \ell \sqsubseteq \zeta \Rightarrow v_1 = v_2 \\ v_1 \approx_\zeta v_2 : (s_1 \to s_2)_\ell & \Leftrightarrow & \vdash v_i : (s_1 \to s_2)_\ell \ \wedge \\ & & \ell \sqsubseteq \zeta \Rightarrow \forall\, v_1' \approx_\zeta v_2' : s_1.\ (v_1\ v_1') \approx_\zeta (v_2\ v_2') : \mathcal{C}(s_2 \sqcup \ell) \end{array}$$

$$e_1 \approx_\zeta e_2 : \mathcal{C}(s) \ \Leftrightarrow \ \vdash e_i : s \ \wedge \ e_1 \Downarrow v_1 \ \wedge \ e_2 \Downarrow v_2 \ \wedge \ v_1 \approx_\zeta v_2 : s$$

To show that a well-typed program e that produces a ζ-observable output of type s (i.e. $\mathsf{label}(s) \sqsubseteq \zeta$) is secure, we simply show that $e \approx_\zeta e : \mathcal{C}(s)$.

To do so, we must first show that the logical relations are well-behaved with respect to the subtyping relation. Intuitively, the following lemma shows that if two values with some security label are indistinguishable to an observer, they remain indistinguishable if given a higher-security label.

Lemma 3.1.5 (Subtyping Relations) *If $v_1 \approx_\zeta v_2 : s_1$ and $\vdash s_1 \leq s_2$ then $v_1 \approx_\zeta v_2 : s_2$. If $e_1 \approx_\zeta e_2 : \mathcal{C}(s_1)$ and $\vdash s_1 \leq s_2$ then $e_1 \approx_\zeta e_2 : \mathcal{C}(s_2)$.*

Proof: We strengthen the hypothesis with the auxiliary claims (and similarly for relations on computations):

$$v_1 \approx_\zeta v_2 : t_\ell \quad \wedge \quad \vdash t \leq t' \ \Rightarrow \ v_1 \approx_\zeta v_2 : t'_\ell$$

$$v_1 \approx_\zeta v_2 : t_\ell \quad \wedge \quad \ell \sqsubseteq \ell' \ \Rightarrow \ v_1 \approx_\zeta v_2 : t_{\ell'}$$

We proceed by induction on this strengthened hypothesis. For λ_{SEC}-TREFL, the result follows immediately. The case for λ_{SEC}-TTRANS follows by straightforward induction.

The interesting case is when rule λ_{SEC}-TFUNSUB is the next-to-last rule used in the derivation that $\vdash s_1 \leq s_2$. Assume $s_1 = t_{1\ell_1}$ and $s_2 = t_{2\ell_2}$. By λ_{SEC}-SLAB it must be the case that $\ell_1 \sqsubseteq \ell_2$. We want to show that $v_1 \approx_\zeta v_2 : t_{2\ell_2}$.

If $\ell_2 \not\sqsubseteq \zeta$, then any two values of the correct type are related, and the typing rule λ_{SEC}-SUB allows us to show that $\vdash v_i : s_2$, so we are done. Otherwise, we have $\ell_2 \sqsubseteq \zeta$, from which we conclude that $\ell_1 \sqsubseteq \zeta$. It must be that $t_1 = s_a \to s_b$ and $t_2 = s'_a \to s'_b$ such that $\vdash s'_a \leq s_a$ and $\vdash s_b \leq s'_b$. It remains to show that

$$\forall\ v'_1 \approx_\zeta v'_2 : s'_a.\ (v_1\ v'_1) \approx_\zeta (v_2\ v'_2) : \mathcal{C}(s'_b \sqcup \ell_2)$$

But, by the induction hypothesis on $\mathcal{S}$ relations it follows that $v'_1 \approx_\zeta v'_2 : s_a$ and by the assumption that $v_1 \approx_\zeta v_2 : s_1$ it follows that $(v_1\ v'_1) \approx_\zeta (v_2\ v'_2) : \mathcal{C}(s_b \sqcup \ell_1)$. Using the induction hypothesis on the $\mathcal{C}$ relations and an easy induction that shows $\vdash s_b \sqcup \ell_1 \leq s'_b \sqcup \ell_2$ we obtain the desired result that $(v_1\ v'_1) \approx_\zeta (v_2\ v'_2) : \mathcal{C}(s'_b \sqcup \ell_2)$.

The inductive step on computation relations $\mathcal{C}$s relations follows directly from the mutual induction with the value relations. □

We need to prove the noninterference result for open programs, but to do so, we must establish that the substitution operation preserves the ζ-equivalence relations. First, we define a notion of related substitutions; two substitutions are related if they are component-wise related.

Definition 3.1.8 (Related Substitutions) *Two substitutions γ_1 and γ_2 are related, indicated by writing $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, if $\gamma_i \models \Gamma$ and*

$$\forall x \in \mathrm{dom}(\Gamma).\ \gamma_1(x) \approx_\zeta \gamma_2(x) : \Gamma(x)$$

We next must show that substitution preserves the logical relations:

Lemma 3.1.6 (Substitution) *If $\Gamma \vdash e : s$ and $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ then $\gamma_1(e) \approx_\zeta \gamma_2(e) : \mathcal{C}(s)$.*

Proof: By induction on the derivation that e has type s. Consider the last step used in the derivation:

λ_{SEC}-**TRUE** Then $\Gamma \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell$ and $s = \mathtt{bool}_\ell$. By the definition of substitutions, $\gamma_1(e) = \gamma_2(e) = \mathtt{t}_\ell$. By two applications of the rule λ_{SEC}-EVAL-VAL, it follows that $\gamma_1(e) \Downarrow \mathtt{t}_\ell$ and $\gamma_2(e) \Downarrow \mathtt{t}_\ell$. By definition, $\mathtt{t}_\ell \approx_\zeta \mathtt{t}_\ell : s$ as required.

λ_{SEC}-**FALSE** Analogous to the previous case.

λ_{SEC}-**VAR** Follows immediately from the facts that substitutions map variables to values and that $\gamma_1(x) \approx_\zeta \gamma_2(x) : \Gamma(x)$ because $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$.

λ_{SEC}-**BINOP** Then $e = e_1 \oplus e_2$ and $s = \texttt{bool}_\ell$ where $\ell = \ell_1 \sqcup \ell_2$. That each $\gamma_i(e_1 \oplus e_2)$ is well-typed and has type s follows from the Lemma 3.1.2. It thus remains to show:

$$\gamma_1(e_1 \ \oplus \ e_2) \Downarrow v_1 \ \ \wedge \ \ \gamma_2(e_1 \ \oplus \ e2)) \Downarrow v_2 \ \ \wedge \ \ v_1 \approx_\zeta v_2 : \texttt{bool}_\ell$$

But, from the definition of substitution, we have

$$\gamma_i(e_1 \ \oplus \ e_2) = \gamma_i(e_1) \ \oplus \ \gamma_i(e_2)$$

So by inversion of the typing judgment and two applications of the induction hypothesis, we obtain

$$\gamma_1(e_1) \approx_\zeta \gamma_2(e_1) : \mathcal{C}(\texttt{bool}_{\ell_1})$$

and

$$\gamma_1(e_2) \approx_\zeta \gamma_2(e_2) : \mathcal{C}(\texttt{bool}_{\ell_2})$$

Consequently, we have $\gamma_1(e_1) \Downarrow v_{11}$ and $\gamma_2(e_1) \Downarrow v_{21}$ where $v_{11} \approx_\zeta v_{21} : \texttt{bool}_{\ell_1}$. Similarly $\gamma_1(e_2) \Downarrow v_{12}$ and $\gamma_2(e_2) \Downarrow v_{22}$ where $v_{12} \approx_\zeta v_{22} : \texttt{bool}_{\ell_2}$. By Canonical Forms (Lemma 3.1.3) we have:

$$\begin{array}{lcllcl} v_{11} & = & (bv_{11})_{\ell_{11}} & v_{12} & = & (bv_{12})_{\ell_{12}} \\ v_{21} & = & (bv_{21})_{\ell_{21}} & v_{22} & = & (bv_{22})_{\ell_{22}} \end{array}$$

By two applications of rule λ_{SEC}-EVAL-BINOP we have:

$$\begin{array}{l} \gamma_1(e_1 \ \oplus \ e_2) \Downarrow (bv_{11} [\![\oplus]\!] \ bv_{12})_{(\ell_{11} \sqcup \ell_{12})} \\ \gamma_2(e_1 \ \oplus \ e_2) \Downarrow (bv_{21} [\![\oplus]\!] \ bv_{22})_{(\ell_{21} \sqcup \ell_{22})} \end{array}$$

It remains to show that

$$(bv_{11} [\![\oplus]\!] \ bv_{12})_{(\ell_{11} \sqcup \ell_{12})} \approx_\zeta (bv_{21} [\![\oplus]\!] \ bv_{22})_{(\ell_{21} \sqcup \ell_{22})} : \texttt{bool}_\ell$$

If $\ell \not\sqsubseteq \zeta$ then the result follows trivially because $\approx_\zeta$ relates all well-typed Boolean expressions. So assume that $\ell \sqsubseteq \zeta$, and from the definition of $\approx_\zeta$ relations at Boolean type, we must show that the expressions are equal. By the inequalities expressed above, it follows that both $\ell_1 \sqsubseteq \zeta$ and $\ell_2 \sqsubseteq \zeta$. Thus, $v_{11} \approx_\zeta v_{21}) : \texttt{bool}_{\ell_1}$ and $v_{12} \approx_\zeta v_{22}) : \texttt{bool}_{\ell_2}$ but then we have that $v_{11} = v_{21}$ and $v_{21} = v_{22}$. Consequently, we obtain

$$(bv_{11} [\![\oplus]\!] \ bv_{12})_{(\ell_{11} \sqcup \ell_{12})} = (bv_{21} [\![\oplus]\!] \ bv_{22})_{(\ell_{21} \sqcup \ell_{22})}$$

as required.

λ_{SEC}-**FUN** In this case, $e = (\lambda x\!:\!s'.\ e')_\ell$ and $s = (s' \to s'')_\ell$. From the bound-variable assumption, we have $\gamma_i(e) = (\lambda x\!:\!s'.\gamma_i(e'))_\ell$. Because Lemma 3.1.2 indicates that the resulting term is well-typed and these terms are already evaluated, it simply remains to show that

$$(\lambda x\!:\!s'.\ \gamma_1(e'))_\ell \approx_\zeta (\lambda x\!:\!s'.\ \gamma_2(e'))_\ell : (s' \to s'')_\ell$$

To do so, note that if $\ell \not\sqsubseteq \zeta$ then the terms are related in $\mathcal{S}$ trivially. Otherwise, we have $\ell \sqsubseteq \zeta$ and we must show that for $v_1 \approx_\zeta v_2 : s'$ that

$$((\lambda x\!:\!s'.\ \gamma_1(e'))_\ell\ v_1) \approx_\zeta ((\lambda x\!:\!s'.\ \gamma_2(e'))_\ell\ v_2) : \mathcal{C}(s'' \sqcup \ell)$$

But by the evaluation rule λ_{SEC}-EVAL-APP, these computations are related whenever

$$\gamma_1(e')\{v_1/x\} \Downarrow v_1' \ \ \wedge \ \ \gamma_2(e')\{v_2/x\} \Downarrow v_2' \ \ \wedge \ \ v_1' \approx_\zeta v_2' : (s'' \sqcup \ell)$$

By inversion of the typing rule, we have that $\Gamma, x : s' \vdash e' : s''$ and that $x \notin \mathrm{dom}(\Gamma)$. Observe that because $v_1 \approx_\zeta v_2 : s'$ it follows that

$$\Gamma, x\!:\!s' \vdash (\gamma_1\{x \mapsto v_1\}) \approx_\zeta (\gamma_2\{x \mapsto v_2\})$$

Now by the induction hypothesis it follows that

$$(\gamma_1\{x \mapsto v_1\}(e')) \approx_\zeta (\gamma_2\{x \mapsto v_2\}(e')) : \mathcal{C}(s'' \sqcup \ell)$$

But because $x \notin \mathrm{dom}(\Gamma)$ the above statement is equivalent to

$$\gamma_1(e')\{v_1/x\} \Downarrow v_1' \ \ \wedge \ \ \gamma_2(e')\{v_2/x\} \Downarrow v_2' \ \ \wedge \ \ v_1' \approx_\zeta v_2' : (s'' \sqcup \ell)$$

as required.

λ_{SEC}-**APP** In this case, $e = e_1\ e_2$ and $s = s'' \sqcup \ell$ for some appropriate s'' and ℓ. It follows that $\gamma_i(e) = \gamma_i(e_1\ e_2) = \gamma_i(e_1)\ \gamma_i(e_2)$. It follows from the induction hypothesis and the well-typing of e that $(\gamma_1(e_1)) \approx_\zeta (\gamma_2(e_1)) : \mathcal{C}(s' \to s'')_\ell$ and $(\gamma_1(e_2)) \approx_\zeta (\gamma_2(e_2)) : \mathcal{C}(s')$. It follows from the definitions that

$$\gamma_1(e_1) \Downarrow v_{11} \ \ \wedge \ \ \gamma_2(e_1) \Downarrow v_{21} \ \ \wedge \ \ v_{11} \approx_\zeta v_{21} : (s' \to s'')_\ell$$

and that

$$\gamma_1(e_2) \Downarrow v_{12} \ \ \wedge \ \ \gamma_2(e_2) \Downarrow v_{22} \ \ \wedge \ \ v_{12} \approx_\zeta v_{22} : s'$$

But now, by definition of the value $\approx_\zeta$ at function type relations, we have

$$(v_{11}\ v_{12}) \approx_\zeta (v_{21}\ v_{22}) : (s'' \sqcup \ell)$$

λ_{SEC}-**COND** In this case, $e = \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ where $\Gamma \vdash e : \texttt{bool}_\ell$ and $\Gamma \vdash e_i : s' \sqcup \ell$ and $s = s' \sqcup \ell$. We must show that

$$(\gamma_1(\texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2)) \approx_\zeta (\gamma_2(\texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2)) : \mathcal{C}(s' \sqcup \ell)$$

By definition of substitution, this is just

$$(\texttt{if}\ \gamma_1(e')\ \texttt{then}\ \gamma_1(e_1)\ \texttt{else}\ \gamma_1(e_2)) \approx_\zeta (\texttt{if}\ \gamma_2(e')\ \texttt{then}\ \gamma_2(e_1)\ \texttt{else}\ \gamma_2(e_2)) : \mathcal{C}(s' \sqcup \ell)$$

If $\ell \not\sqsubseteq \zeta$ then the two terms are related trivially, because the $\approx_\zeta$ relations relate all such well-typed terms. So assume that $\ell \sqsubseteq \zeta$. By the induction hypothesis, it follows that $\gamma_1(e') \approx_\zeta \gamma_2(e') : \mathcal{C}(\texttt{bool}_\ell)$, so by definition we have $\gamma_1(e') \Downarrow v_1$ and $\gamma_2(e') \Downarrow v_2$ and $v_1 \approx_\zeta v_2 : \texttt{bool}_\ell$. But, since $\ell \sqsubseteq \zeta$, we have that $v_1 = v_2$. Thus, either rule λ_{SEC}-EVAL-COND1 applies to both terms or λ_{SEC}-EVAL-COND2 applies to both terms; assume the former applies (the latter case is analogous). In this case,

$$\gamma_1(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \Downarrow v_{11} \qquad \text{where } \gamma_1(e_1) \Downarrow v_{11}$$

Similarly,

$$\gamma_2(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) \Downarrow v_{21} \qquad \text{where } \gamma_2(e_1) \Downarrow v_{21}$$

but by the induction hypothesis of this lemma, we already have $v_{11} \approx_\zeta v_{21} : \mathcal{C}(s' \sqcup \ell)$ as needed.

λ_{SEC}-**SUB** This follows directly from Lemma 3.1.5.

□

Finally, we obtain the noninterference proof as a simple corollary of Lemma 3.1.6.

3.2 $\lambda^{\text{REF}}_{\text{SEC}}$: a secure language with state

This section describes how to augment λ_{SEC} to include mutable state.

$\lambda^{\text{REF}}_{\text{SEC}}$ includes a new type, s `ref`, that describes mutable references that point to objects of type s. For example, if L is a memory location that stores the secure Boolean value $\texttt{t}_\bot$, then L can be given the type $\texttt{bool}_\bot$ `ref`.

The memory location L may be updated to contain the value $\texttt{f}_\bot$ by doing an assignment with the program expression $L := \texttt{f}_\bot$. The contents of L may be retrieved by the dereference operation. For example, after the assignment just described, the program $\texttt{let}\ x = !L\ \texttt{in}\ e$ binds the variable x to the value $\texttt{f}_\bot$.

It is unsafe to assign a high-security value to a low-security memory location. For example, assuming L has type $\texttt{bool}_\bot\ \texttt{ref}$ we must prevent the assignment $L := \texttt{t}_\top$ because such an assignment constitutes a direct flow from $\top$ to $\bot$. This typing restriction also prevents *aliases*, program variables that refer to the same memory location, from being used to leak information. For instance, in the following program[3], the variables `x` and `y` must both be given the type $\texttt{bool}_\bot\ \texttt{ref}$

```
let x = L in
let y = x in
```

Otherwise, the alias could be used to leak information to the low-security location L.

Because references are themselves first-class values, they must also be given security annotations. To see why, consider the following program in which L and L' are both low-security memory locations (they both contain data of type $\texttt{bool}_\bot$) and h is of type $\texttt{bool}_\top$.

```
L := t⊥;
L' := t⊥;
let x = if h then L else L' in
  x := f⊥;
  if !L then ...          h is false
        else ...          h is true
```

This program contains an information flow from $\top$ to $\bot$ because whether the variable x refers to L or L' depends on high security information. To prevent this flow, secure reference types include an additional security label, and are of the form $s\ \texttt{ref}_\ell$. Here, L and L' might be given type $\texttt{bool}_\bot\ \texttt{ref}_\bot$ but because the variable x depends on the high-security h, x must be given the type $\texttt{bool}_\bot\ \texttt{ref}_\top$. The labeled operational semantics and type system prevent the "bad" assignment $x := \texttt{f}_\bot$ in the above program requiring that the label of the reference be protected by the label of its contents. To assign to a reference of type $s\ \texttt{ref}_\ell$ it must be the case that $\ell \sqsubseteq \mathsf{label}(s)$. The example is ruled out because $\top \not\sqsubseteq \bot$.

There is one more subtlety in dealing with mutable state. There can be an implicit flow. Consider the following program, where L again has type $\texttt{bool}_\bot\ \texttt{ref}_\bot$ and h is of type $\texttt{bool}_\top$.

```
if h then L := t⊥ else L := f⊥
```

Here, the problem is that, even though the assignments to L are individually secure, which of them occurs depends on high-security data. To prevent such implicit flows,

[3]The example (and others in this chapter) uses standard syntactic sugar for `let` and sequencing operations.

the type system for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ associates a label pc with the program counter. Intuitively, the program counter label approximates the information that can be learned by observing that the program has reached a particular point during the execution. In the example above, the program counter reveals the value of h, so inside the branches of the conditional, we have $\mathsf{pc} = \top$. To prevent these implicit flows, the labeled semantics requires that $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$ whenever an assignment to a reference of type $s\ \mathtt{ref}_\ell$ occurs in the context with program counter label pc. This rules out the above example.

Another implicit information flow can arise due to the interaction between functions and state. For example, consider a function f that takes no argument and assigns the location L the constant $\mathtt{t}_\bot$. Function f can be written as:

$$f \stackrel{\mathrm{def}}{=} \lambda().\ L := \mathtt{t}_\bot$$

This function is perfectly secure and can be used in many contexts, but it can also be used to leak information. For example, consider the program below:

```
L := f⊥;
if h then f() else skip;
```

This program is insecure because f writes to the low-security memory location L. Calls to functions that have side effects (writes to memory locations) can leak information in the same way that assignment leaks information about the program counter.

To detect and rule out such implicit flows, function types in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ include an additional label; they are of the form $[\ell]s_1 \to s_2$. The label ℓ is a lower bound on the labels of any locations that might be written when calling the function. To call a function of this type in a context where the program counter has label pc, the operational semantics and type system require that $\mathsf{pc} \sqsubseteq \ell$. Thus, because f writes to a low security location, f is given the type $[\bot]\mathtt{unit}_\bot \to \mathtt{unit}_\bot$; since $\mathsf{pc} = \top$ inside the branches of the conditional guarded by h, the above program is ruled out.

With these intuitions in mind, we can now present details of $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$. Figure 3.6 contains the grammar for this new source language, called $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$.

As just described, function types now include a label pc in their syntax $[\mathsf{pc}]s \to s$. This label bounds the effects—writes to memory—that may occur when a function with this type is invoked.

To model state, $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ includes locations, ranged over by L^s, which are the names of memory cells that contain values of type s. Concrete memory locations are written using lowercase letters like $\mathtt{a}^s, \mathtt{b}^{s'}$, etc., although the type annotations will often be omitted when they are unimportant or clear from context. The type s decorating a location is used for type checking purposes.

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ provides a mechanism for allocating a new memory cell and storing a value there: The expression $\mathtt{ref}^s\ e$ first evaluates the expression e to a value v, creates a fresh

location in memory, and then stores the value into that location. The result of $\texttt{ref}^s\ e$ is the newly created location.

The expression dereference operation $!e$ evaluates e to obtain a location and then returns the value stored in the memory at that location. The form $e_1 := e_2$ updates the location indicated by e_1 to contain the value obtained by evaluating e_2 and then returns $\langle\rangle$. If the dereferenced or assigned location has not been created in memory, the program halts (crashes).

Definition 3.2.1 (Memory) *A* **memory** M *is a finite map from locations to values. Locations, ranged over by the metavariable* L *and decorated with a type* s*, are written* L^s*. The notation* $M(L^s)$ *denotes the value associated with location* L^s *in memory* M*. The notation* $M[L^s \mapsto v]$ *indicates the memory formed by replacing the contents of* L^s *by the value* v*. If* L^s *is not in* $\mathrm{dom}(M)$*, then a new binding is created.*

As an example, if M is the memory $[\mathtt{a}^{\mathtt{bool}_\ell} \mapsto \mathtt{t}_\ell]$, the expression $\mathtt{a}^{\mathtt{bool}_\ell} := \mathtt{f}_\ell$ causes the memory to be updated to $M[\mathtt{a}^{\mathtt{bool}_\ell} \mapsto \mathtt{f}_\ell] = [\mathtt{a}^{\mathtt{bool}_\ell} \mapsto \mathtt{f}_\ell]$.

An additional difference between λ_{SEC} and $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ is that $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ allows functions to be recursive. The syntax $\lambda[\mathsf{pc}]\ f\ (x : s).\ e$ describes a function named f whose body is able to assign to references that point to data with confidentiality label pc or higher. As shown below, the pc is used to rule out insecure information flows that might arise due to control flow involving calls to this function. The name f is bound within the body e; it is used to invoke the function recursively. For example, the following function, when invoked, goes into an infinite loop by calling itself immediately:

$$\lambda[\bot]\ f\ (x{:}s).\ f\ x$$

This function can be given the type $[\bot]s \to s$ for any secure type s.

3.2.1 Operational semantics

For $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ (and the other languages discussed in the remainder of this thesis), we present only the labeled syntax and nonstandard operational semantics—from them it is straightforward to define the label erasure to a standard programming model.

The operational semantics for $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ is more complex than that of λ_{SEC} because it must keep track of the effects that take place in the mutable storage. Accordingly, we augment the abstract machine configurations to include memories as defined above.

Definition 3.2.2 (Machine configuration) *A* **machine configuration** *is a triple, written* $\langle M, \mathsf{pc}, e\rangle$*, containing a memory* M*, a program counter label* $\mathsf{pc} \in \mathcal{L}$*, and an expression* e *representing the program.*

$\ell, \mathtt{pc}$	$\in$	$\mathcal{L}$	Security labels
t	$::=$	$\mathtt{unit}$	Unit type
	$\mid$	$\mathtt{bool}$	Boolean type
	$\mid$	$s\ \mathtt{ref}$	Reference type
	$\mid$	$[\mathtt{pc}]s \to s$	Function type
s	$::=$	t_ℓ	Security types
bv	$::=$	$\mathtt{t} \mid \mathtt{f}$	Boolean base values
	$\mid$	$\langle\rangle$	Unit value
	$\mid$	$\lambda[\mathtt{pc}]\ f\ (x:s).\ e$	Recursive functions
	$\mid$	L^s	Memory locations
v	$::=$	x	Variables
	$\mid$	bv_ℓ	Secure Values
e	$::=$	v	Values
	$\mid$	$e\ e$	Function applications
	$\mid$	$e\ \oplus\ e$	Primitive operations
	$\mid$	$\mathtt{ref}^s e$	Reference creations
	$\mid$	$!e$	Dereferences
	$\mid$	$e := e$	Assignments
	$\mid$	$\mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e$	Conditionals
$\oplus$	$::=$	$\wedge \mid \vee \mid \ldots$	Boolean operations
M	$::=$	$\emptyset \mid M[L^s \mapsto v]$	Machine memories
m	$::=$	$\langle M, \mathtt{pc}, e\rangle$	Machine configurations

Figure 3.6: $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ grammar

A given machine configuration $\langle M,\ \mathtt{pc},\ e\rangle$ may evaluate to a final state of the form $\langle M', v\rangle$ or it may diverge. Figure 3.7 summarizes the operational rules. Just as with λ_{SEC}, the operational semantics models function application using substitution.

The state also contains a security label pc that describes the information flows implicit in the control flow of the program. For instance, recall the following example, where h is of type $\mathtt{bool}_\top$:

$$\mathtt{if}\ h\ \mathtt{then}\ L := \mathtt{t}\ \mathtt{else}\ L := \mathtt{f}$$

This program copies the value in h into the value pointed to by reference L but returns $\langle\rangle$ no matter which branch is taken. If L represents a low-security memory location (i.e. has type $\mathtt{ref}^{\mathtt{bool}_\bot}$), this program is insecure. The problem is that the effect of writing to the location L reveals information about the program counter at which the write occurs.

In order to determine that the program above is insecure, an analysis must be aware that the assignment to L takes place in a context that reveals high-security information. That fact is captured by a label pc, the *program counter label*, that conservatively bounds the information that can be learned by observing that the program has reached that point in the code. Within the body of a conditional guarded by a high-security Boolean, the program counter label is high-security.

The operational semantics presented in Figure 3.7 includes additional checks that regulate when it is safe to store a value in a memory location. For example, the rule $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-EVAL-ASSIGN requires that $\mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(s)$: the information contained in the program counter together with the information label on the reference must be more public than the label on the contents of the location. This run-time check prevents the program above from writing to the location L if it stores low-security information.

References also have labels associated with them to prevent information flows that result from aliasing: two variables that hold references may point to the same memory location, and hence information may be leaked by assigning to one reference and detecting the change through the other. One example of such aliasing was already described. For another example, consider the following program:

```
a := t⊥;
let x = (if h then a else (ref t⊥)) in
let v1 = !x in
  a := (¬ v1);
  if (!x) = (¬ v1) then l := t⊥
                   else l := f⊥
```

Where the variables might be given the following types.

$\boxed{\langle M_1,\ \mathsf{pc},\ e\rangle \Downarrow \langle M_2,\ v\rangle}$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-VAL
$$\langle M,\ \mathsf{pc},\ v\rangle \Downarrow \langle M, v \sqcup \mathsf{pc}\rangle$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-PRIM
$$\frac{\langle M,\ \mathsf{pc},\ e_1\rangle \Downarrow \langle M', (v_1)_{\ell_1}\rangle \quad \langle M',\ \mathsf{pc},\ e_2\rangle \Downarrow \langle M'', (v_2)_{\ell_2}\rangle}{\langle M,\ \mathsf{pc},\ e_1\ \oplus\ e_2\rangle \Downarrow \langle M'', (v_1\ [\![\oplus]\!]\ v_2)_{(\ell_1 \sqcup \ell_2)}\rangle}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-APP
$$\frac{\begin{array}{c}\langle M,\ \mathsf{pc},\ e_1\rangle \Downarrow \langle M', (\lambda[\mathsf{pc}']\ f\ (x\!:\!s).\ e)_\ell\rangle \\ \langle M',\ \mathsf{pc},\ e_2\rangle \Downarrow \langle M'', v\rangle \qquad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \\ \langle M'',\ \mathsf{pc}',\ e\{v/x\}\{(\lambda[\mathsf{pc}']\ f\ (x\!:\!s).\ e)_\ell/f\}\rangle \Downarrow \langle M''', v'\rangle\end{array}}{\langle M,\ \mathsf{pc},\ e_1\ e_2\rangle \Downarrow \langle M''', v'\rangle}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-COND1
$$\frac{\langle M,\ \mathsf{pc},\ e\rangle \Downarrow \langle M', \mathtt{t}_\ell\rangle \quad \langle M',\ \mathsf{pc} \sqcup \ell,\ e_1\rangle \Downarrow \langle M'', v\rangle}{\langle M,\ \mathsf{pc},\ \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \Downarrow \langle M'', v\rangle}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-COND2
$$\frac{\langle M,\ \mathsf{pc},\ e\rangle \Downarrow \langle M', \mathtt{f}_\ell\rangle \quad \langle M',\ \mathsf{pc} \sqcup \ell,\ e_2\rangle \Downarrow \langle M'', v\rangle}{\langle M,\ \mathsf{pc},\ \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \Downarrow \langle M'', v\rangle}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-REF
$$\frac{\begin{array}{c}\mathsf{pc} \sqsubseteq \mathsf{label}(s) \\ \langle M,\ \mathsf{pc},\ e\rangle \Downarrow \langle M', v\rangle \quad L^s \notin \mathrm{dom}(M')\end{array}}{\langle M,\ \mathsf{pc},\ \mathtt{ref}^s e\rangle \Downarrow \langle M'[L^s \mapsto v], L^s_{\mathsf{pc}}\rangle}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-DEREF
$$\frac{\langle M,\ \mathsf{pc},\ e\rangle \Downarrow \langle M', L^s_\ell\rangle \quad M'(L^s) = v}{\langle M,\ \mathsf{pc},\ !e\rangle \Downarrow \langle M', v \sqcup \ell\rangle}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EVAL-ASSIGN
$$\frac{\begin{array}{cc}\mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(s) & L^s \in \mathrm{dom}(M') \\ \langle M,\ \mathsf{pc},\ e_1\rangle \Downarrow \langle M', L^s_\ell\rangle & \langle M',\ \mathsf{pc},\ e_2\rangle \Downarrow \langle M'', v\rangle\end{array}}{\langle M,\ \mathsf{pc},\ e_1 := e_2\rangle \Downarrow \langle M''[L^s \mapsto v], \langle\rangle_{\mathsf{pc}}\rangle}$$

Figure 3.7: Operational semantics for $\lambda^{\text{REF}}_{\text{SEC}}$

h : $\texttt{bool}_\top$
$\texttt{a}$: $\texttt{bool}_\bot\ \texttt{ref}_\bot$
$\texttt{x}$: $\texttt{bool}_\bot\ \texttt{ref}_\top$
$\texttt{v1}$: $\texttt{bool}_\bot$
$\texttt{l}$: $\texttt{bool}_\bot\ \texttt{ref}_\bot$

This program copies the high-security boolean h into a location `l`. It does so by conditionally creating an alias `x` to the location `a` and then testing whether in fact an alias has been created. In this case, it is not the contents of the location `a` or `x` that leak the information, it is the fact that `a` and `x` alias. (Pierce and Sangiorgi [PS99] point out that a similar problem with aliasing in ML allows a programmer to violate parametricity—aliasing provides a means for leaking type information.)

The label annotations on references rule out the program above when `v1` is a low-security Boolean because any value read through the reference high-security reference `x` becomes high-security. This program will be rejected with the types given above because `!c` has type $\texttt{bool}_\top$ by `v1` expects a $\bot$-security Boolean. If instead, `v1` were given the type $\texttt{bool}_\top$, the program would still be ruled out because of the implicit flow to the variable `l` in the branches of the second conditional. The only way for the above program to be considered secure, is when, in addition to `v1` having high security, variable `l` is a reference to high security data.

3.2.2 Type system

The source type system is reminiscent of the type system for λ_{SEC}, but also draws on the type systems found in previous work on information-flow languages such as the one proposed by Volpano, Smith and Irvine [VSI96] and Heintze and Riecke's SLam calculus [HR98]. Unlike the SLam calculus, which also performs access control checks, the source language type system is concerned only with secure information flow. The type system rules out insecure information flows and thus eliminates the need for the dynamic checks found in the operational semantics.

As with λ_{SEC}, the $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ type system lifts the ordering on the security lattice to a subtyping relation on the values of the language (Figure 3.8).

Reference types obey the expected invariant subtyping, which is already expressed by the $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$-TREFL rule. The security labels of the references themselves obey the usual covariant subtyping given by $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$-SLAB. Consequently, $s\ \texttt{ref}_\bot \leq s\ \texttt{ref}_\top$ for any s, but it is never the case that $s\ \texttt{ref}_\ell \leq s'\ \texttt{ref}_{\ell'}$ when $s \neq s'$.

The judgment $\Gamma\ [\texttt{pc}] \vdash e : s$ shows that expression e has source type s under type context Γ, assuming the program-counter label is bounded above by `pc`. Intuitively, the `pc` appearing in the judgment approximates the information that can be learned by

$\boxed{\vdash t_1 \leq t_2}$ $\boxed{\vdash s_1 \leq s_2}$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-TREFL $\quad \vdash t \leq t$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-TTRANS $\quad \dfrac{\vdash t \leq t' \quad \vdash t' \leq t''}{\vdash t \leq t''}$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-TFUNSUB $\quad \dfrac{\mathsf{pc}' \sqsubseteq \mathsf{pc} \quad \vdash s_1' \leq s_1 \quad \vdash s_2 \leq s_2'}{\vdash [\mathsf{pc}]s_1 \to s_2 \quad \leq \quad [\mathsf{pc}']s_1' \to s_2'}$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-SLAB $\quad \dfrac{\vdash t \leq t' \quad \ell \sqsubseteq \ell'}{\vdash t_\ell \leq t'_{\ell'}}$

Figure 3.8: Value subtyping in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$

observing that the program counter has reached a particular point in the program. The rules for checking program expressions appear in Figure 3.10.

Function types are labeled with their *latent effect*, a lower bound on the security level of memory locations that will be written to by that functions. A function with type $[\mathsf{pc}]s_1 \to s_2$ may be called safely only from contexts for which the program-counter label, pc' satisfies $\mathsf{pc}' \sqsubseteq \mathsf{pc}$ because the side effects within the function body may leak information visible at level pc. Rule $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-FUN in Figure 3.9 shows that the label appearing in a function's type is used to check the body of the function.

The $[\mathsf{pc}]$ component of $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ function types is contravariant, as shown in the rule $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-TFUN-SUB. This contravariance arises because the $[\mathsf{pc}]$ component is a lower bound on the side effects that may occur in the body of a function. A function that has a *higher* lower bound can exhibit *fewer* side effects, consequently such a function may be used anywhere a function that is permitted to exhibit *more* side effects is required.

It is easy to see that this type system conservatively takes into account the information flows from the context of the program to the value produced by the computation, as shown by the following lemma:

Lemma 3.2.1 *If* $\Gamma\ [\mathsf{pc}] \vdash e : s$ *then* $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$.

Proof: By a trivial induction on the derivation that e has type s, observing that in the base case (rule $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-VAL) the condition $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$ appears as an antecedent. □

Even though a well-formed source program contains no label values, memory locations may be allocated during the course of its evaluation. The $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-EVAL-DEREF

$\boxed{\Gamma \vdash v : s}$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-TRUE} \qquad \Gamma \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell$$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-FALSE} \qquad \Gamma \vdash \mathtt{f}_\ell : \mathtt{bool}_\ell$$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-UNIT} \qquad \Gamma \vdash \langle\rangle_\ell : \mathtt{unit}_\ell$$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-LOC} \qquad \Gamma \vdash L^s_\ell : s\ \mathtt{ref}_\ell$$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-VAR} \qquad \frac{\Gamma(x) = s}{\Gamma \vdash x : s}$$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-FUN} \qquad \frac{\begin{array}{c} f, x \notin \mathrm{dom}(\Gamma) \\ s' = ([\mathsf{pc}]s_1 \to s_2)_\ell \\ \Gamma, f : s', x : s_1\ [\mathsf{pc}] \vdash e : s_2 \end{array}}{\Gamma \vdash (\lambda[\mathsf{pc}]\ f\ (x : s).\ e)_\ell : s'}$$

$$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}\text{-SUB} \qquad \frac{\Gamma \vdash v : s \quad \vdash s \leq s'}{\Gamma \vdash v : s'}$$

Figure 3.9: Value typing in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$

$\boxed{\Gamma\ [\mathsf{pc}] \vdash e : s}$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-VAL
$$\frac{\Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{\Gamma\ [\mathsf{pc}] \vdash v : s}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-APP
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : ([\mathsf{pc}']s' \to s)_\ell \quad \Gamma\ [\mathsf{pc}] \vdash e' : s' \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}'}{\Gamma\ [\mathsf{pc}] \vdash e\ e' : s \sqcup \ell}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-PRIM
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e_1 : \mathtt{bool}_\ell \quad \Gamma\ [\mathsf{pc}] \vdash e_2 : \mathtt{bool}_\ell}{\Gamma\ [\mathsf{pc}] \vdash e_1\ \oplus\ e_2 : \mathtt{bool}_\ell}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-REF
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : s}{\Gamma\ [\mathsf{pc}] \vdash \mathtt{ref}^s e : s\ \mathtt{ref}_{\mathsf{pc}}}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-DEREF
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : s\ \mathtt{ref}_\ell}{\Gamma\ [\mathsf{pc}] \vdash !e : s \sqcup \ell}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-ASSN
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e_1 : s\ \mathtt{ref}_\ell \quad \Gamma \vdash e_2 : s \quad \ell \sqsubseteq \mathsf{label}(s)}{\Gamma\ [\mathsf{pc}] \vdash e_1 := e_2 : \mathtt{unit}_{\mathsf{pc}}}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-COND
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : \mathtt{bool}_\ell \quad \Gamma\ [\mathsf{pc} \sqcup \ell] \vdash e_i : s \quad i \in \{1, 2\}}{\Gamma\ [\mathsf{pc}] \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : s}$$

$\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-EXPRSUB
$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : s \quad \vdash s \leq s'}{\Gamma\ [\mathsf{pc}] \vdash e : s'}$$

Figure 3.10: Expression typing in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$

implicitly requires that the location being dereferenced be in the domain of the memory. Consequently, for the type system to rule out dereferencing of unallocated memory cells, there must be a notion of when a memory is well formed. More formally, for any program expression e, the set $Loc(e)$ consists of all location names appearing in e. The memory reference invariant says that in any well-formed machine configuration $\langle M, \mathsf{pc}, e\rangle$ it should be the case that $Loc(e) \subseteq \mathrm{dom}(M)$. Indeed this is the case, but to show that such a property holds, the type system must show that the memory M is well-formed.

Because memories may contain cyclic data structures, we must carefully formulate the notion of when a memory is well-formed. Intuitively, we want a location L^s to point to a value v of type s, but v may contain references to other locations—even L^s itself. We thus must use a kind of "assume"–"guarantee" reasoning, in which we assume that all the required locations are present and well-typed in the memory when showing that a particular memory item is well-formed.

Definition 3.2.3 (Memory well-formedness) *A memory M is well-formed, written $\vdash M$ wf if and only if*

$$\forall L^s \in \mathrm{dom}(M).\ \ Loc(M(L^s)) \in \mathrm{dom}(M) \quad \wedge \quad \vdash M(L^s) : s$$

The intention is that whenever a closed program is placed in the context of a well-formed memory that provides a meaning to all of the locations occurring in the program, the type system guarantees that there will be no illegal information flows or unexpected errors during the evaluation of the program. Formally:

Lemma 3.2.2 ($\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ Preservation) *If $\cdot\ [\mathsf{pc}] \vdash e : s$ and $\vdash M$ wf and $Loc(e) \subseteq |M|$ and $\langle M, \mathsf{pc}, e\rangle \Downarrow \langle M', v\rangle$ then $\cdot \vdash v : s$ and $\vdash M'$ wf and $Loc(v) \subseteq |M'|$.*

Proof: A standard proof on the derivation that $\langle M, \mathsf{pc}, e\rangle \Downarrow \langle M', v\rangle$. □

3.2.3 Noninterference for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$

The type system for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ is sufficient to establish a noninterference result, but doing so requires a more sophisticated proof than the logical relations argument used for λ_{SEC} in Section 3.1.4. The difficulty is that $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ has mutable state and first-class references. Consequently, the equivalence relations must be extended to account for the contents of memory.

Rather than prove noninterference for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ directly, we shall translate $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ to a language that makes the operational behavior of the programs more explicit and prove noninterference for that language instead. Correctness of the translation then implies that $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ inherits the security properties of the target language.

The next chapter proves the desired noninterference result for the target language.

3.3 Related work

Palsberg and Ørbæk studied a simply-typed lambda calculus that included facilities for distinguishing trusted and untrusted computation [PO95]. Their type system enforces an information-flow property intended to protect the integrity of data.

Volpano, Smith and Irvine [VSI96, VS97] were among the first to propose a security type system. They considered a small imperative language with `while` loops and conditionals. Memory locations are tagged with their security level and may contain only integers. This work contributed the first soundness result for a security-typed language intended to protect confidentiality.

Heintze and Riecke created the SLam (Secure Lambda) Calculus to investigate noninterference in higher-order languages [HR98]. SLam is a variant of the simply-typed lambda calculus, similar to the languages presented in this chapter. In addition, SLam includes fixed-point recursion, products, and sum type constructors.

In the purely functional case, Heintze and Riecke provide a noninterference proof using a logical-relations style argument over a denotational semantics of the program. The idea is to interpret the type-structure of the language as partial equivalence relations (PERs) over the denotations of the terms. Low-security views of high-security data are represented by complete relations, and the fact that well-formed programs must preserve all relations implies that high-security data cannot be observed by looking at low-security data. This proof of noninterference for λ_{SEC} given in this chapter is an operational semantics adaptation of their approach. They do not prove a noninterference result for the stateful version of SLam.

The idea that noninterference can be captured by using a partial equivalence relation semantics is further investigated by Abadi and others [ABHR99]. This work observes that noninterference can be framed as a dependency analysis, and that several well-known analyses including binding time, SLam's type system, and the type system of Smith and Volpano could be unified into a common theoretical framework. This model, called the Dependency Core Calculus (DCC), builds on a long thread of research to understand parametric polymorphism [Str67, Rey74, Rey83, MPS86, CGW89, AP90, MR92a].

Program slicing techniques [Tip95] also provide information about the data dependencies in a piece of software. The use of backward slices to investigate integrity and related security properties has been proposed [FL94, LWG$^+$95]. Program debugging and understanding existing software is the focus of the slicing work, and these approaches typically do not allow programmer specified annotations to guide the slicing analysis.

Sabelfeld and Sands have extended the denotational semantics approach to proving noninterference properties to the case of nondeterministic and probabilistic programs [SS99]. They confine themselves to a simple imperative programming language similar to the one studied by Volpano and Smith. The technique is essentially the same one

used in DCC: logical relations are built over a denotational semantics of the program. The new twist is that powerdomains (the domain-theoretic analog to the powerset) are used to accommodate the set of possible outputs due to nondeterminism. The interesting observation here is that the choice of powerdomain operator (there are three) gives rise to different ways of handling nonterminating programs: one powerdomain corresponds to total correctness, one to partial correctness, and one to a "mixture" of the two that yields more accurate results with respect to information flow.

Pottier and Conchon [PC00] describe a means of automatically extending an existing, sound type system so that a type inference algorithm also infers information flows within a program. This effectively separates the mechanism for discovering information flows from the policy about which flows are acceptable.

Reitman [Rei78] and Andrews [AR80] propose a Hoare-style logic for reasoning about information flows in a simple imperative language. They extend the axiomatic semantics to handle concurrency primitives, but give no correctness proofs of the logic. Their work is less concerned with providing ways to enforce security policies than with simply reasoning about the flows within the program.

Chapter 4

Noninterference in a Higher-order Language with State

This chapter proves a noninterference result for a higher-order language with state, and shows how to apply that result to the language $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$. Rather than prove noninterference for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ directly, we instead translate $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ to a new, lower-level language called $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$.

The purpose of $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ is to make the operational behavior of programs more explicit by introducing the notion of a *continuation*. A continuation is an abstraction of the (potential) future computation of a program. Like functions, continuations take arguments and encapsulate a piece of code to run when supplied a value for the argument. Unlike functions, continuations never return to the calling context. Instead, a continuation either halts the whole program or invokes another continuation.

Because continuations are themselves first-class objects that can be manipulated as data values, more structured control-flow mechanisms, like procedure call and return, can be decomposed into *continuation-passing style* (CPS) [Fis72, Rey72, Ste78, App92, DF92, FSDF93]. Continuations are thus more primitive than functions.

Compiling a structured, higher-order program to CPS exposes its control-transfer behavior. There are a number of reasons why this compilation approach to establishing noninterference is useful:

1. In contrast to $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$, $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ has a small-step operational semantics. This means that the sequence of states through which the memory passes during evaluation is more apparent in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ than in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$. Consequently, the noninterference result for $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ is stronger than the corresponding result for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ because the former says that the *sequence* of states induced must not reveal information to a low-security observer

whereas the latter says only that the *final state* reached by the program (if any) should not reveal information to a low-security observer.[1]

2. The target language $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ is more expressive than the source language in the sense that it permits certain code transformations, like tail-call optimizations, that are useful in practice.

3. Because the semantics of $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ are closer to those of an actual computer, studying how information-flow policies can be enforced at this level of abstraction opens up the potential for checking security of assembly code [MWCG99], perhaps by way of proof-carrying code [Nec97].

4. Finally, using continuations, rather than more structured control transfers like procedure calls, provides a stepping stone to the distributed computing setting. Message passing in distributed systems, like continuation invocation, is a fundamentally unidirectional operation from which more complex behaviors are defined. Understanding continuation-passing style in a sequential, single-processor setting leads to a better understanding of the problems that arise in the concurrent, distributed setting.

This chapter explores the problem of CPS compilation for security-typed languages. It shows how a type system that captures the structured operations of the source language can enforce information-flow security in low-level programming languages.

4.1 CPS and security

As we was shown in the last chapter, type systems for secrecy or integrity are concerned with tracking dependencies in programs. Recall that one difficulty is *implicit flows*, which arise from the control flow of the program. Consider the code fragment `(A)` in Figure 4.1.[2] There is an implicit flow between the value stored in `h` and the value stored in `a`, because examining the contents of `a` after the program has run gives information about the value in `h`. There is no information flow between `h` and `b`, however. Consequently, this code is secure when `h` and `a` are high-security variables and `b` is low-security.

[1] One could also formulate a small-step semantics for $\lambda_{\mathrm{SEC}}^{\mathrm{REF}}$ directly; doing so and establishing a noninterference result requires essentially the same amount of work as proving noninterference for $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$. The other benefits favor the CPS approach.

[2] The examples are written in an imperative pseudo-code in which continuations can be introduced explicitly (as in `k = (`$\lambda\langle\rangle$`. halt)`) and invoked (as in `k` $\langle\rangle$). The actual syntax of $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ is given in Section 4.2.1.

```
(A)   if h then { a := t; } else { a := f; }
      b := f; halt;

(B)   let k = (λ⟨⟩. b := f; halt) in
      if h then { a := t; k ⟨⟩; } else { a := f; k ⟨⟩; }

(C)   let k = (λ⟨⟩. b := f; halt) in
      if h then { a := t; k ⟨⟩; } else { a := f; halt; }

(D)   letlin k = (λ⟨⟩. b := f; halt) in
      if h then { a := t; k ⟨⟩; } else { a := f; k ⟨⟩; }

(E)   letlin k0 = (λ⟨⟩. halt) in
      letlin k1 = (λk. b := t; k ⟨⟩) in
      letlin k2 = (λk. b := f; k ⟨⟩) in
      if h then { letlin k = (λ⟨⟩. k1 k0) in k2 k }
           else { letlin k = (λ⟨⟩. k2 k0) in k1 k }
```

Figure 4.1: Examples of information flow in CPS

A programmer using a type system for enforcing information flow policies might assign `h` the type $\texttt{bool}_\top$ (high-security Boolean) and `b` the type $\texttt{bool}_\bot$ (low-security Boolean). If `a` were given the type $\texttt{bool}_\top$, program fragment `(A)` would type check, but if `a` were given a low-security type `(A)` would not type check due to the implicit flow from `h` to `a`. As we saw in the previous chapter, security-typed languages deal with these implicit flows by associating a security annotation with the *program counter* (which we will usually indicate by pc). In example `(A)`, the program counter at the point before the `if` statement might be labeled with L to indicate that it does not depend on high-security data. Recall that within the branches of the conditional the program counter depends on the value of `h`, and hence the pc must be $\top$—the security label of `h`. Values (such as the constants `t` and `f` of the example) pick up the security annotation of the program counter, and consequently when `a` has type $\texttt{bool}_\bot$ the assignment `a := t` is illegal—the (implicitly) high-security value `t` is being assigned to a low-security memory location.

Suppose we were to straightforwardly adapt the source rule $\lambda^{\text{REF}}_{\text{SEC}}$-APP, which type-checks function application, to account for continuation invocation. The resulting rule would look something like:

$$\textsc{Conservative}\qquad \frac{\Gamma[\mathsf{pc}] \vdash k : [\mathsf{pc}']s \to 0 \quad \Gamma[\mathsf{pc}] \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{pc}'}{\Gamma[\mathsf{pc}] \vdash k\ v}$$

Here, the type of a continuation expecting an argument of type s is written $[\mathsf{pc}']s \to \mathtt{0}$. As with function types, the label pc' is a lower bound on the security level of effects that occur inside the body of the continuation k. The $\to \mathtt{0}$ part indicates that, unlike functions, continuations never return to their calling context. Just as for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-APP, the condition $\mathsf{pc} \sqsubseteq \mathsf{pc}'$ requires that the information contained in the program counter of the caller is more public than the effects that occur once the continuation has been called.

Fragment (B) illustrates the problem with this naive rule for continuations. It shows the code from (A) after CPS translation has made control transfer explicit. The variable k is bound to the continuation of the if, and the jump is indicated by the application k $\langle\rangle$. Because the invocation of k has been lifted into the branches of the conditional, the rule CONSERVATIVE will require that the body of k not write to low-security memory locations. The value of h would apparently be observable by low-security code and program (B) would be rejected because k writes to a low-security variable, b.

However, this code *is* secure; there is no information flow between h and b in (B) because the continuation k is invoked in both branches. On the other hand, as example (C) shows, if k is *not* used in one of the branches, then information about h can be learned by observing b. Linear type systems [Abr93, Gir87, Wad90, Wad93] can express exactly the constraint that k is used in both branches. By making k's linearity explicit, the type system can use the additional information to recover the precision of the type system for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$. Fragment (D) illustrates this simple approach; in addition to a normal let construct, we include letlin for introducing linear continuations. The program (D) certifies as secure even when b is a low-security variable, whereas (C) does not.

Although linearity allows for more precise reasoning about information flow, linearity alone is insufficient for security in the presence of first-class continuations. In example (E), continuations k0, k1, and k2 are all linear, but there is an implicit flow from h to b because b lets us observe the *order* in which k1 and k2 are invoked. It is thus necessary to regulate the ordering of linear continuations. The type system presented in Section 4.2.4 requires that exactly one linear continuation be available at any point—thus eliminating the possibility of writing code like example (E). We show in Section 4.4 that these constraints are sufficient to prove a noninterference result.

It is easier to make information-flow analysis precise for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ than $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ because the structure of $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ limits control flow. For example, it is known that both branches of a conditional return to a common merge point. This knowledge is exploited by the type system to obtain less conservative analysis of implicit flows than the rule CONSERVATIVE above. Unfortunately, the standard CPS transformation loses this information by unifying all forms of control to a single mechanism. With the linearity approach, the target language still has a single underlying control transfer mechanism (examples (B) and (D) execute exactly the same code), but the type system statically distinguishes between different kinds of continuations, allowing information flow to be analyzed with the same precision as in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$.

4.1.1 Linear Continuations

Before diving into the formal definition of the secure CPS language, it is helpful to have some intuition about what a linear continuation is. Ordinary continuations represent a possible future computation of a program. How they are manipulated encapsulates the control flow aspects of a piece of code. Powerful language constructs such as `callcc` (found in the high level languages Scheme and Standard ML of New Jersey) expose the continuations to the programmer in a first-class way, allowing direct manipulation of the control flow. However, such use of continuations is far from common. As observed by Berdine et al. [BORT01], many control-flow constructs use continuations linearly (exactly once). This linearity arises from restrictions of the source language: functions return exactly once, merge-points of conditional statements are reachable in exactly one way from each branch, etc. The fact that `callcc` and other nonstandard control-flow operators discard or duplicate continuations is part of what makes reasoning about them difficult.

Combining linearity with an ordering on continuations restricts their manipulation even further. Ordered linear continuations enforce a stack discipline on control [PP00]. Introducing a linear continuation is analogous to pushing a return address onto a stack; invoking a linear continuation corresponds to popping that address and jumping to the return context. Because many constructs (function call/return, nested blocks, and merge-points of conditionals) of high-level structured programs can be implemented via a stack of activation records, ordered linear continuations are a natural fit to describing their control flow behavior.

Using ordered linear continuations in a type system divorces the description of the stack-like control constructs of a programming language from its syntax (block structure). This separation is essential for preserving control-flow information across compilation steps such as CPS transformation, because the syntactic structure of the program is altered. The main insight is that we can push information implicitly found in the structure of a program into explicit descriptions (the types) of the program.

4.2 $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$: a secure CPS calculus

This section describes the secure CPS language, its operational behavior, and its static semantics. $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ is a call-by-value, imperative language similar to those found in the work on Typed Assembly Language [CWM99, MWCG99], although its type system is inspired by previous language-based security research [HR98, Mye99, VSI96].

4.2.1 Syntax

The syntax for $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ is given in Figure 4.2.

Following the proposal for labeling data outlined in Chapter 2, we assume a lattice of security labels, $\mathcal{L}$. The $\sqsubseteq$ symbol denotes the lattice ordering. As before, the lattice join and meet operations are given by $\sqcup$ and $\sqcap$, respectively, and the least and greatest elements are written $\bot$ and $\top$. Elements of $\mathcal{L}$ are ranged over by meta-variables ℓ and pc. As in the $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ semantics, we reserve the meta-variable pc to suggest that the security label corresponds to information learned by observing the program counter.

Types fall into two main syntactic classes: security types, s, and linear types, κ. Security types are the types of ordinary values and consist of a base-type component, t, annotated with a security label, ℓ. Base types include Booleans, unit, and references. Continuation types, written $[\mathsf{pc}](s, \kappa) \rightarrow \mathtt{0}$, indicate a security level and the types of their arguments. The notation $\mathtt{0}$ describes the "void" type, and it indicates that a continuation never returns.

Corresponding to these types, base values, bv, include the Booleans $\mathtt{t}$ and $\mathtt{f}$, a unit value $\langle\rangle$, type-annotated memory locations, L^s, and continuations, $\lambda[\mathsf{pc}]f(x:s, y:\kappa).\, e$. All computation occurs over secure values, v, which are base values annotated with a security label. Variables, x, range over values.

As an example, the value $\mathtt{t}_\bot$ represents a low-security Boolean (one of type $\mathtt{bool}_\bot$) that is observable by any computation. On the other hand, the value $\mathtt{f}_\top$ represents a high-security Boolean that should be observable only by high-security computations—those computations that do not indirectly influence the low-security portions of the memory. The operational semantics will ensure that labels are propagated correctly. For instance we have $\mathtt{t}_\bot \wedge \mathtt{f}_\top = \mathtt{f}_\top$, because low-security computation, which can affect the low-security portions of memory and hence leak information to a low-security observer of the program, should be prevented from observing the result—it contains information about the high-security value $\mathtt{f}_\top$.

As in $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$, references contain two security annotations. For example, the type $\mathtt{bool}_\top\ \mathtt{ref}_\bot$ represents the type of low-security pointers to high-security Booleans, which is distinct from $\mathtt{bool}_\bot\ \mathtt{ref}_\top$, the type of high-security pointers to low-security Booleans. The data returned by a dereference operation is protected by the join of the two labels. Thus, Booleans obtained through pointers of either of these two reference types will receive a security label of $\top$.

An ordinary continuation $\lambda[\mathsf{pc}]f(x:s, y:\kappa).\, e$ is a piece of code (the expression e) that accepts a nonlinear argument of type s and a linear argument of type κ. Continuations may recursively invoke themselves using the name f, which is bound in e. The notation $[\mathsf{pc}]$ indicates that this continuation may be called only from a context in which the program counter carries information of security at most pc. To avoid unsafe implicit

ℓ, pc	$\in$	$\mathcal{L}$	Security Labels
t	::=	$\texttt{unit}$	Unit type
	$\mid$	$\texttt{bool}$	Boolean type
	$\mid$	$s\ \texttt{ref}$	Reference types
	$\mid$	$[\mathsf{pc}](s, \kappa) \to \mathsf{0}$	Ordinary continuation types
s	::=	t_ℓ	Security types
κ	::=	$s \to \mathsf{0}$	Linear continuation types
bv	::=	$\langle\rangle$	Unit value
	$\mid$	$\texttt{t} \mid \texttt{f}$	Boolean values
	$\mid$	L^s	Memory locations
	$\mid$	$\lambda[\mathsf{pc}] f(x\!:\!s, y\!:\!\kappa).\, e$	Continuation values
v	::=	x	Variables
	$\mid$	bv_ℓ	Secure Values
lv	::=	y	Linear variables
	$\mid$	$\lambda\langle\mathsf{pc}\rangle(x\!:\!s).\, e$	Linear continuations
$prim$	::=	v	Primitive value
	$\mid$	$v \oplus v$	Primitive Boolean operation
	$\mid$	$!v$	Location dereference
e	::=	$\texttt{let}\ x = prim\ \texttt{in}\ e$	Primitive value binding
	$\mid$	$\texttt{let}\ x = \texttt{ref}^s e\ \texttt{in}\ e$	Reference creation
	$\mid$	$\texttt{set}\ v := v\ \texttt{in}\ e$	Assignment
	$\mid$	$\texttt{letlin}\ y = lv\ \texttt{in}\ e$	Linear binding
	$\mid$	$\texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e$	Conditional
	$\mid$	$\texttt{goto}\ v\ v\ lv$	Ordinary continuation invocation
	$\mid$	$\texttt{lgoto}\ lv\ v$	Linear continuation invocation
	$\mid$	$\texttt{halt}^s\ v$	Program termination

Figure 4.2: Syntax for the $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ language

flows, the body of the continuation may create effects observable only by principals able to read data with label pc.

A linear value, lv, is either a variable (ranged over by y), or a linear continuation, which contains a code expression e parameterized by a nonlinear argument just as for ordinary continuations. Linear continuations may not be recursive, but they may be invoked from any calling context; hence linear types do not require any pc annotation. The syntax $\langle \mathsf{pc} \rangle$ serves to distinguish linear continuation values from nonlinear ones. As for ordinary continuations, the label pc restricts the continuation's effects, but unlike ordinary continuations, the pc is constrained only by the context at the point of their creation (as opposed to the context in which they are invoked). Intuitively, linear continuations capture the security context in which they are created and, when invoked, restore the program counter label to the one captured.

The primitive operations include binary arithmetic, $\oplus$, dereference, and a means of copying secure values. Primitive operations are side-effect free. Program expressions consist of a sequence of `let` bindings for primitive operations, reference creation, and imperative updates (via `set`). The `letlin` construct introduces a linear continuation. A straight-line code sequence is terminated by a conditional statement, a `goto` or a `lgoto`.

The expression $\texttt{halt}^s\ v$ is a program that terminates and produces the final output v of type s.

4.2.2 Operational semantics

The operational semantics (Figure 4.3) is given by a transition relation between machine configurations of the form $\langle M,\ \mathsf{pc},\ e \rangle$. The notation $e\{v/x\}$ indicates capture-avoiding substitution of value v for variable x in expression e.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ memories, M, are defined just as for $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$, except that the types annotating the memory locations are $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ types and the locations store $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ values. We use the same notational conventions for describing memory updates. A memory is *well-formed* if it is closed under the dereference operation and each value stored in the memory has the correct type. We use $\emptyset$ to denote the empty memory, and we write $Loc(e)$ for the set of location names occurring in e.

The label pc in a machine configuration represents the security level of information that could be learned by observing the location of the program counter. Instructions executed with a program-counter label of pc are restricted so that they update only memory locations with labels more secure than pc. For example, $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-SET shows that it is valid to store a value to a memory location of type s only if the security label of the data joined with the security labels of the program counter and the reference itself is lower than $\mathsf{label}(s)$, the security clearance needed to read the data stored at that location. Rules $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-COND1 and $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-COND2 show how the program-counter label changes after branching on data of security level ℓ. Observing which branch is taken

$\boxed{\langle M,\, \mathsf{pc},\, prim\rangle \Downarrow v}$	$\boxed{\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \to \langle M_2,\, \mathsf{pc}_2,\, e_2\rangle}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-PRIM	$\langle M,\, \mathsf{pc},\, bv_\ell\rangle \;\Downarrow\; bv_{\ell \sqcup \mathsf{pc}}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-BINOP	$\langle M,\, \mathsf{pc},\, n_\ell \oplus n'_{\ell'}\rangle \;\Downarrow\; (n[\![\oplus]\!]n')_{\ell \sqcup \ell' \sqcup \mathsf{pc}}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-DEREF	$\dfrac{M(L^s) = bv_{\ell'}}{\langle M,\, \mathsf{pc},\, !L^s_\ell\rangle \;\Downarrow\; bv_{\ell \sqcup \ell' \sqcup \mathsf{pc}}}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETPRIM	$\dfrac{\langle M,\, \mathsf{pc},\, prim\rangle \;\Downarrow\; v}{\langle M,\, \mathsf{pc},\, \texttt{let}\; x = prim\; \texttt{in}\; e\rangle \to \langle M,\, \mathsf{pc},\, e\{v/x\}\rangle}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETREF	$\dfrac{\ell \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(s) \quad L^s \notin \mathrm{dom}(M)}{\begin{array}{l}\langle M,\, \mathsf{pc},\, \texttt{let}\; x = \texttt{ref}^s bv_\ell\; \texttt{in}\; e\rangle \to \\ \quad \langle M[L^s \mapsto bv_{\ell \sqcup \mathsf{pc}}],\, \mathsf{pc},\, e\{L^s_{\mathsf{pc}}/x\}\rangle\end{array}}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-SET	$\dfrac{\ell \sqcup \ell' \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(s) \quad L^s \in \mathrm{dom}(M)}{\langle M,\, \mathsf{pc},\, \texttt{set}\; L^s_\ell := bv_{\ell'}\; \texttt{in}\; e\rangle \to \langle M[L^s \mapsto bv_{\ell \sqcup \ell' \sqcup \mathsf{pc}}],\, \mathsf{pc},\, e\rangle}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETLIN	$\langle M,\, \mathsf{pc},\, \texttt{letlin}\; y = lv\; \texttt{in}\; e\rangle \to \langle M,\, \mathsf{pc},\, e\{lv/y\}\rangle$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-COND1	$\langle M,\, \mathsf{pc},\, \texttt{if}\; \mathtt{t}_\ell\; \texttt{then}\; e_1\; \texttt{else}\; e_2\rangle \to \langle M,\, \mathsf{pc} \sqcup \ell,\, e_1\rangle$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-COND2	$\langle M,\, \mathsf{pc},\, \texttt{if}\; \mathtt{f}_\ell\; \texttt{then}\; e_1\; \texttt{else}\; e_2\rangle \to \langle M,\, \mathsf{pc} \sqcup \ell,\, e_2\rangle$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-GOTO	$\dfrac{\mathsf{pc} \sqsubseteq \mathsf{pc}' \qquad v = (\lambda[\mathsf{pc}']f(x\!:\!s, y\!:\!\kappa).\, e)_\ell \qquad e' = e\{v/f\}\{bv_{\ell' \sqcup \mathsf{pc}}/x\}\{lv/y\}}{\langle M,\, \mathsf{pc},\, \texttt{goto}\; (\lambda[\mathsf{pc}']f(x\!:\!s, y\!:\!\kappa).\, e)_\ell\; bv_{\ell'}\; lv\rangle \to \langle M,\, \mathsf{pc}' \sqcup \ell,\, e'\rangle}$
$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LGOTO	$\langle M,\, \mathsf{pc},\, \texttt{lgoto}\; (\lambda\langle \mathsf{pc}'\rangle(x\!:\!s).\, e)\; bv_\ell\rangle \to \langle M,\, \mathsf{pc}',\, e\{bv_{\ell \sqcup \mathsf{pc}}/x\}\rangle$

Figure 4.3: Expression evaluation

reveals information about the condition variable, so the program counter must have the higher security label $\mathsf{pc} \sqcup \ell$.

As shown in rules $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-PRIM through $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-DEREF, computed values are stamped with the pc label. The notation $[\![\oplus]\!]$ denotes the semantic counterpart to the syntactic operation $\oplus$. Run-time label checks prevent illegal information flows via direct means such as assignment. For example, $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-SET requires the following label constraint to hold:

$$\ell \sqcup \ell' \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(s)$$

This constraint says that the label on the data being assigned into the reference, joined with the label that regulates reference aliases and the current pc label should be more public than the label on the type of data the location stores. This prevents direct, alias, and indirect information leaks from occurring due to the assignment operation.

Section 4.4 shows that, for well-typed programs, *all* illegal information flows are ruled out, and hence these dynamic label checks are unnecessary.

Operationally, the rules for `goto` and `lgoto` are very similar—each causes control to be transferred to the target continuation. They differ in their treatment of the program-counter label, as seen in rules $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-GOTO and $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LGOTO. Ordinary continuations require that the program-counter label of the calling context, pc, be protected by the program-counter label of the continuation, and the computation proceeds with the label declared in the continuation. Linear continuations instead cause the program-counter label to be restored (potentially, lowered) to that of the context in which they were declared. In accordance with the label-stamping intuition, both `goto` and `lgoto` stamp the pc label of the calling context into the value passed to the continuation.

As mentioned above, well-formed programs stop when they reach the expression $\texttt{halt}^s\ v$. Consequently the "stuck" term $\texttt{halt}^s\ v$ represents a valid terminal state.

4.2.3 An example evaluation

This section gives a concrete example of the operational semantics.

Consider the evaluation shown in Figure 4.4. It shows the program fragment `(D)` from Figure 4.1 of the introduction using the syntax of our secure CPS language. In this instance, the condition variable is the high-security value $\texttt{t}_\top$, and the program-counter label is initially $\bot$, the lowest security label. The memory, M, initially maps the high-security location `a` to the value $\texttt{t}_\top$ and the low-security location `b` to the value $\texttt{t}_\bot$. (This information is summarized in the figure.)

Step (1) is a transition via $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETLIN that introduces the linear continuation, k_{impl} and binds it to the variable `k`. As indicated by the notation $\langle\bot\rangle$ in k_{impl}'s definition, when invoked, k_{impl} will set the pc label back to $\bot$. In step (2), the program

$$
\begin{array}{lcll}
 & & \langle M, & \bot, & \texttt{letlin k = } k_{impl} \texttt{ in} \\
 & & & & \texttt{if t}_\top \texttt{ then set a := t}_\bot \texttt{ in lgoto k } \langle\rangle \\
 & & & & \qquad \texttt{else set a := f}_\bot \texttt{ in lgoto k } \langle\rangle \quad \rangle \\
(1) & \rightarrow & \langle M, & \bot, & \texttt{if t}_\top \texttt{ then set a := t}_\bot \texttt{ in lgoto } k_{impl} \langle\rangle \\
 & & & & \qquad \texttt{else set a := f}_\bot \texttt{ in lgoto } k_{impl} \langle\rangle \rangle \\
(2) & \rightarrow & \langle M, & \top, & \texttt{set a := t}_\bot \texttt{ in lgoto } k_{impl} \langle\rangle \quad \rangle \\
(3) & \rightarrow & \langle M', & \top, & \texttt{lgoto } k_{impl} \langle\rangle \quad \rangle \\
(4) & \rightarrow & \langle M', & \bot, & \texttt{set b := f}_\bot \texttt{ in halt}^{\texttt{unit}_\bot} \quad \rangle \\
(5) & \rightarrow & \langle M'', & \bot, & \texttt{lgoto halt}^{\texttt{unit}_\bot} \quad \rangle
\end{array}
$$

Where

$$
\begin{array}{rcl}
M & = & \{\texttt{a} \mapsto \texttt{t}_\top, \texttt{z} \mapsto \texttt{t}_\bot\} \\
M' & = & \{\texttt{a} \mapsto \texttt{t}_\top, \texttt{z} \mapsto \texttt{t}_\bot\} \\
M'' & = & \{\texttt{a} \mapsto \texttt{t}_\top, \texttt{z} \mapsto \texttt{f}_\bot\} \\
\texttt{a} & : & \texttt{bool}_\top \texttt{ ref}_\bot \\
\texttt{b} & : & \texttt{bool}_\bot \texttt{ ref}_\bot \\
k_{impl} & = & \lambda\langle\bot\rangle(\texttt{h:unit}_\bot).\texttt{set b := f}_\bot \texttt{ in halt}^{\texttt{unit}_\bot}
\end{array}
$$

Figure 4.4: Example program evaluation

transitions via rule E5, testing the condition variable. In this case, because $\texttt{t}_\top$ is high-security, the program counter label increases to $\top = \bot \sqcup \top$, and the program takes the first branch. Next, step (3) is a transition by rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-SET, which updates the contents of memory location a. The new value stored is high-security, because, instantiating $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-SET, we have: $\ell = \bot, \ell' = \bot, \texttt{pc} = \top$ and $\ell' \sqcup \ell \sqcup \texttt{pc} = \bot \sqcup \bot \sqcup \top = \top$. This assignment succeeds because a is a location that stores high-security data; if a were a location of type $\texttt{bool}_\bot \texttt{ ref}_\bot$, the check $\ell' \sqcup \ell \sqcup \texttt{pc} \sqsubseteq \mathsf{label}(\texttt{bool}_\bot) = \bot$ would fail—however, the type system presented in the next section statically rules out such behavior, making such dynamic checks unnecessary.

The fourth step is the linear invocation, via rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LGOTO. As promised, k_{impl} resets the program counter label to $\bot$, and in addition, we substitute the actual arguments for the formal parameters in the body of the continuation. The last transition is another instance of rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-SET, this time updating the contents of b with the low-security value $\texttt{f}_\bot$.

How would this program differ if an ordinary continuation were used instead of k_{impl}? The crucial difference would appear in step (4), where instead of evaluating via rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LGOTO, we would be forced to use rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-GOTO. Note that $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-GOTO requires the pc label of the continuation to be *higher* than the one in the machine configuration. In this case, because the calling context has $\texttt{pc} = \top$,

the body of the continuation would be forced to $\top$ as well. It is not possible to write a value to the low-security location b in such circumstances, and hence we cannot write this program using an ordinary continuation in place of k_{impl} without forcing b to be a high-security location.

4.2.4 Static semantics

The type system for the secure CPS language enforces the linearity and ordering constraints on continuations and guarantees that security labels on values are respected. Together, these restrictions rule out illegal information flows and impose enough structure on the language for us to prove a noninterference property.

As in other mixed linear–nonlinear type systems [TW99], the type context is split into an ordinary, nonlinear section and a linear section. Γ is a finite partial map from nonlinear variables to security types, whereas K is an *ordered* list (with concatenation denoted by ",") mapping linear variables to their types. The order in which continuations appear in K defines the order in which they are invoked: Given $K = \cdot, (y_n : \kappa_n), \ldots, (y_1 : \kappa_1)$, the continuations will be executed in the order $y_1 \ldots y_n$. Thus, the context K spells out explicitly the stack-like behavior of ordered linear continuations. A key part of the type system is ensuring that this stack is respected by the program. The nonlinear context Γ admits the usual weakening and exchange rules (which we omit), but the linear context does not. The two parts of the context are separated by $\|$ in the judgments to make them more distinct (as in $\Gamma \parallel K$). We use $\cdot$ to denote an empty nonlinear context.

Figures 4.5 through 4.10 show the rules for type-checking. The judgment form $\Gamma \vdash v : s$ says that ordinary value v has security type s in context Γ. Linear values may mention linear variables and so have judgments of the form $\Gamma; K \vdash lv : \kappa$. Like values, primitive operations may not contain linear variables, but the security of the value produced depends on the program-counter. We thus use the judgment $\Gamma\ [\mathsf{pc}] \vdash prim : s$ to say that in context Γ where the program-counter label is bounded above by pc, $prim$ computes a value of type s. Similarly, $\Gamma; K\ [\mathsf{pc}] \vdash e$ means that expression e is type-safe and contains no illegal information flows in the type context $\Gamma \parallel K$, when the program-counter label is at most pc. Because expressions represent continuations, and hence do not return values, no type is associated with judgments $\Gamma; K\ [\mathsf{pc}] \vdash e$. Alternatively, we could write $\Gamma; K\ [\mathsf{pc}] \vdash e : \mathtt{0}$ to indicate that e does not return. But, as all expressions have type 0, we simply omit the $: \mathtt{0}$. In the latter two forms, pc is a conservative approximation to the security label of information affecting the program counter.

The rules for checking ordinary values, shown in Figure 4.5, are, for the most part, the same as for those of $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$. A value cannot contain free linear variables because

$\boxed{\Gamma \vdash v : s}$

$\lambda^{\text{CPS}}_{\text{SEC}}$-BOOL $\quad \Gamma \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell \qquad \Gamma \vdash \mathtt{f}_\ell : \mathtt{bool}_\ell$

$\lambda^{\text{CPS}}_{\text{SEC}}$-UNIT $\quad \Gamma \vdash \langle\rangle_\ell : \mathtt{unit}_\ell$

$\lambda^{\text{CPS}}_{\text{SEC}}$-LOC $\quad \Gamma \vdash L^s_\ell : s\ \mathtt{ref}_\ell$

$\lambda^{\text{CPS}}_{\text{SEC}}$-VAR $\quad \dfrac{\Gamma(x) = s}{\Gamma \vdash x : s}$

$\lambda^{\text{CPS}}_{\text{SEC}}$-CONT $\quad \dfrac{\begin{array}{c} f, x \notin \mathrm{dom}(\Gamma) \\ s' = ([\mathsf{pc}](s, \kappa) \to 0)_\ell \\ \Gamma, f : s', x : s; y : \kappa\ [\mathsf{pc}] \vdash e \end{array}}{\Gamma \vdash (\lambda[\mathsf{pc}] f(x : s, y : \kappa).\ e)_\ell : s'}$

$\lambda^{\text{CPS}}_{\text{SEC}}$-SUB $\quad \dfrac{\Gamma \vdash v : s \quad \vdash s \leq s'}{\Gamma \vdash v : s'}$

Figure 4.5: Value typing

$\boxed{\vdash t_1 \leq t_2}$ $\boxed{\vdash s_1 \leq s_2}$

$\lambda^{\text{CPS}}_{\text{SEC}}$-TREFL $\quad \vdash t \leq t$

$\lambda^{\text{CPS}}_{\text{SEC}}$-TTRANS $\quad \dfrac{\vdash t \leq t' \quad \vdash t' \leq t''}{\vdash t \leq t''}$

$\lambda^{\text{CPS}}_{\text{SEC}}$-TCONTSUB $\quad \dfrac{\mathsf{pc}' \sqsubseteq \mathsf{pc} \quad \vdash s' \leq s \quad \vdash \kappa' \leq \kappa}{\vdash [\mathsf{pc}](s, \kappa) \to 0 \quad \leq \quad [\mathsf{pc}'](s', \kappa') \to 0}$

$\lambda^{\text{CPS}}_{\text{SEC}}$-SLAB $\quad \dfrac{\vdash t \leq t' \quad \ell \sqsubseteq \ell'}{\vdash t_\ell \leq t'_{\ell'}}$

Figure 4.6: Value subtyping in $\lambda^{\text{CPS}}_{\text{SEC}}$

discarding (or copying) the value would break the linearity constraint on the variable. A continuation type contains the pc label used to check its body (rule $\lambda^{\text{CPS}}_{\text{SEC}}$-CONT).

The lattice ordering on security labels lifts to a subtyping relationship on values (shown in Figure 4.6). Continuations exhibit the expected contravariance, as shown in rule $\lambda^{\text{CPS}}_{\text{SEC}}$-TCONTSUB. References, are, as in $\lambda^{\text{REF}}_{\text{SEC}}$, invariant with respect to the data being stored but covariant with respect to their outer label.

Linear values are checked using the rules in Figures 4.7 and 4.8. Subtyping linear types is standard. As shown in 4.8, linear values may safely mention free linear variables, but the variables must not be discarded or reordered. Thus we may conclude that a linear variable is well-formed exactly when the only variable in the linear context is the variable in question (rule $\lambda^{\text{CPS}}_{\text{SEC}}$-LVAR).

In a linear continuation (rule $\lambda^{\text{CPS}}_{\text{SEC}}$-LCONT) the linear context must be used within the body of the continuation, but the non-linear argument is added to Γ.

The rules for primitive operations (in Figure 4.9) require that the calculated value have security label at least as restrictive as the current pc, reflecting the "label stamping" behavior of the operational semantics. Values read through `deref` (rule $\lambda^{\text{CPS}}_{\text{SEC}}$-DEREF) pick up the label of the reference as well, which prevents illegal information flows due to aliasing.

Figure 4.10 lists the rules for type checking expressions. Primitive operations are introduced by a `let` expression as shown in $\lambda^{\text{CPS}}_{\text{SEC}}$-PRIM. The rules for creating new references and doing reference update, rules $\lambda^{\text{CPS}}_{\text{SEC}}$-REF and $\lambda^{\text{CPS}}_{\text{SEC}}$-ASSN, require that the reference protect the security of the program counter. The condition $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$ says that any data read through the created reference may only be observed by contexts

$\boxed{\vdash \kappa_1 \leq \kappa_2}$

$$\lambda^{\text{CPS}}_{\text{SEC}}\text{-LREFL} \qquad \vdash \kappa \leq \kappa$$

$$\lambda^{\text{CPS}}_{\text{SEC}}\text{-LTRANS} \qquad \frac{\vdash \kappa \leq \kappa' \quad \vdash \kappa' \leq \kappa''}{\vdash \kappa \leq \kappa''}$$

$$\lambda^{\text{CPS}}_{\text{SEC}}\text{-LCONTSUB} \qquad \frac{\vdash s' \leq s}{\vdash s \to \mathsf{0} \quad \leq \quad s' \to \mathsf{0}}$$

Figure 4.7: Linear value subtyping in $\lambda^{\text{CPS}}_{\text{SEC}}$

$\boxed{\Gamma; K \vdash lv : \kappa}$

$$\lambda^{\text{CPS}}_{\text{SEC}}\text{-LVAR} \qquad \Gamma; y{:}\kappa \vdash y : \kappa$$

$$\lambda^{\text{CPS}}_{\text{SEC}}\text{-LCONT} \qquad \frac{\begin{array}{c} x \notin \mathrm{dom}(\Gamma) \\ \Gamma, x{:}s; K\ [\mathsf{pc}] \vdash e \end{array}}{\Gamma; K \vdash \lambda\langle \mathsf{pc} \rangle (x{:}s).\, e : s \to \mathsf{0}}$$

$$\lambda^{\text{CPS}}_{\text{SEC}}\text{-LSUB} \qquad \frac{\Gamma; K \vdash lv : \kappa \quad \vdash \kappa \leq \kappa'}{\Gamma; K \vdash lv : \kappa'}$$

Figure 4.8: Linear value typing in $\lambda^{\text{CPS}}_{\text{SEC}}$

$\boxed{\Gamma\ [\mathsf{pc}] \vdash prim : s}$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-VAL} \qquad \frac{\Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{\Gamma\ [\mathsf{pc}] \vdash v : s}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-BINOP} \qquad \frac{\Gamma \vdash v : \mathtt{bool}_\ell \quad \Gamma \vdash v' : \mathtt{bool}_\ell \quad \mathsf{pc} \sqsubseteq \ell}{\Gamma\ [\mathsf{pc}] \vdash v \oplus v' : \mathtt{bool}_\ell}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-DEREF} \qquad \frac{\Gamma \vdash v : s\ \mathtt{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s \sqcup \ell)}{\Gamma\ [\mathsf{pc}] \vdash !v : s \sqcup \ell}$$

Figure 4.9: Primitive operation typing in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$

able to observe the current program counter. The reference itself starts initially with a secrecy determined by the current pc[3] The condition $\mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(s)$ in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-ASSN prevents explicit flows in a similar way.

Rule $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-COND illustrates how the conservative bound on the information contained in the program-counter is propagated: the label used to check the branches is the label before the test, pc, joined with the label on the data being tested, ℓ. The rule for $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-GOTO restricts the program-counter label of the calling context, pc, joined with the label on the continuation itself, ℓ, to be less than the program-counter label under which the body was checked, pc'. This prevents implicit information flows from propagating into function bodies. Likewise, the values passed to a continuation (linear or not) must pick up the calling context's pc (via the constraint $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$) because they carry information about the context in which the continuation was invoked.

The rule $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-HALT requires an empty linear context, indicating that the program consumes all linear continuations before stopping. The s annotating `halt` is the type of the final output of the program; its label should be constrained by the security clearance of the user of the program.

The rule $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-LETLIN manipulates the linear context to enforce the ordering property on continuations. The top of the continuation stack, K_2, must be used in the body of the continuation being declared. The body of the declaration, e, is checked under the assumption that the new continuation, y, is available. Collectively, these manipulations amount to pushing the continuation y onto the control stack.

[3]It is possible to allow users to annotate the `ref` creation instruction with a security label, but permitting such annotation does not yield any interesting insights, so it is omitted here.

$\boxed{\Gamma; K\ [\mathsf{pc}] \vdash e}$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Prim}\quad \frac{\Gamma\ [\mathsf{pc}] \vdash prim : s \qquad \Gamma, x{:}s; K\ [\mathsf{pc}] \vdash e}{\Gamma; K\ [\mathsf{pc}] \vdash \mathtt{let}\ x = prim\ \mathtt{in}\ e}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Ref}\quad \frac{\Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s) \qquad \Gamma, x{:}s\ \mathtt{ref}_{\mathsf{pc}}; K\ [\mathsf{pc}] \vdash e}{\Gamma; K\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{ref}^s v\ \mathtt{in}\ e}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Assn}\quad \frac{\Gamma \vdash v : s\ \mathtt{ref}_\ell \quad \Gamma; K\ [\mathsf{pc}] \vdash e \qquad \Gamma \vdash v' : s \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(s)}{\Gamma; K\ [\mathsf{pc}] \vdash \mathtt{set}\ v := v'\ \mathtt{in}\ e}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Letlin}\quad \frac{\Gamma; K_2 \vdash \lambda\langle \mathsf{pc}'\rangle(x{:}s).\, e' : s \to 0 \qquad \mathsf{pc} \sqsubseteq \mathsf{pc}' \qquad \Gamma; K_1, y{:}s \to 0\ [\mathsf{pc}] \vdash e}{\Gamma; K_1, K_2\ [\mathsf{pc}] \vdash \mathtt{letlin}\ y = \lambda\langle \mathsf{pc}'\rangle(x{:}s).\, e'\ \mathtt{in}\ e}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Cond}\quad \frac{\Gamma \vdash v : \mathtt{bool}_\ell \quad \Gamma; K\ [\mathsf{pc} \sqcup \ell] \vdash e_i}{\Gamma; K\ [\mathsf{pc}] \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Goto}\quad \frac{\Gamma \vdash v : ([\mathsf{pc}'](s, \kappa) \to 0)_\ell \qquad \Gamma \vdash v' : s \qquad \Gamma; K \vdash lv : \kappa \qquad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{\Gamma; K\ [\mathsf{pc}] \vdash \mathtt{goto}\ v\ v'\ lv}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{LGoto}\quad \frac{\Gamma; K \vdash lv : s \to 0 \qquad \Gamma \vdash v : s \qquad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{\Gamma; K\ [\mathsf{pc}] \vdash \mathtt{lgoto}\ lv\ v}$$

$$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}\text{-}\textsc{Halt}\quad \frac{\Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{\Gamma; \cdot\ [\mathsf{pc}] \vdash \mathtt{halt}^s\ v}$$

Figure 4.10: Expression typing in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$

Linear continuations capture the pc (or a more restrictive label) of the context in which they are introduced, as shown in rule $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-LETLIN. Unlike the rule for goto, the rule for lgoto does not constrain the program-counter label of the target continuation, because the linear continuation *restores* the program-counter label to the one it captured.

Because linear continuations capture the pc of their introduction context, we make the mild assumption that *initial programs* introduce all linear continuation values (except variables) via letlin. This assumption rules out trivially insecure programs; during execution this constraint is not required, and programs in the image of the CPS translation of Section 4.5 satisfy this property.

4.3 Soundness of $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$

This section proves the soundness theorem for $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$. The proof is, for the most part, standard, following in the style of Wright and Felleisen [WF92, WF94]. As usual for a language with subtyping, our proofs assume that typing derivations are in a canonical form in which applications of subsumption alternate with other rules. That such canonical proofs exist follows from the reflexive and transitive nature of the subtyping relation.

We first begin by establishing a few standard properties of typed languages. A simple proposition that we shall not prove is the following, which says that if a base value is well-typed when stamped with one label, it is well-typed when stamped with any other label.

Proposition 4.3.1 (Base Value Relabeling) *If* $\Gamma \vdash bv_\ell : \tau_\ell$ *then* $\Gamma \vdash bv_{\ell'} : \tau_{\ell'}$.

We shall use the Base Value Relabeling proposition without mentioning it explicitly in the proofs below.

Next, we establish that capture-avoiding substitution of a well-typed term into another well-typed term yields a well-typed term:

Lemma 4.3.1 (Substitution I) *Assume* $\Gamma \vdash v : s$ *then*

(i) *If* $\Gamma, x{:}s \vdash v' : s'$ *then* $\Gamma \vdash v'\{v/x\} : s'$.

(ii) *If* $\Gamma, x{:}s; K \vdash lv : \kappa$ *then* $\Gamma; K \vdash lv\{v/x\} : \kappa$.

(iii) *If* $\Gamma, x{:}s\ [\mathsf{pc}] \vdash prim : s'$ *then* $\Gamma\ [\mathsf{pc}] \vdash prim\{v/x\} : s'$.

(iv) *If* $\Gamma, x{:}s; K\ [\mathsf{pc}] \vdash e$ *then* $\Gamma; K\ [\mathsf{pc}] \vdash e\{v/x\}$.

Proof: By mutual induction on the (canonical) derivations of (i)–(iv).

(i) By assumption, there exists a derivation of the form

$$\frac{\Gamma, x{:}s \vdash v' : s'' \quad \vdash s'' \leq s'}{\Gamma, x{:}s \vdash v' : s'}$$

We proceed by cases on the rule used to conclude $\Gamma, x : s \vdash v' : s''$. In cases $\lambda^{\text{CPS}}_{\text{SEC}}$-BOOL, $\lambda^{\text{CPS}}_{\text{SEC}}$-UNIT, and $\lambda^{\text{CPS}}_{\text{SEC}}$-LOC we have $v'\{v/x\} = v'$, and the result follows by Strengthening and derivation above. In the case of $\lambda^{\text{CPS}}_{\text{SEC}}$-VAR, we have either $x\{v/x\} = v$, which, by assumption, has type $s = s''$ or $x'\{v/x\} = x'$, also of type s''. In either case, this information plus the derivation above yields the desired result. The case of $\lambda^{\text{CPS}}_{\text{SEC}}$-CONT follows from induction hypothesis (iv).

(ii) This cases follows analogously to the case for (i); the rule $\lambda^{\text{CPS}}_{\text{SEC}}$-LCONT makes use of induction hypothesis (iv).

(iii) This follows directly from the induction hypothesis (i).

(iv) Follows by induction hypotheses (i)–(iv).

□

Note that neither ordinary values nor primitive operations may contain free linear variables. This means that substitution of a linear value in them has no effect. The following lemma strengthens substitution to *open* linear values and also shows that the ordering on the linear context is maintained.

Lemma 4.3.2 (Substitution II) *Assume* $\cdot; K \vdash lv : \kappa$ *then:*

(i) *If* $\Gamma; K_1, y{:}\kappa, K_2 \vdash lv' : \kappa'$ *then* $\Gamma; K_1, K, K_2 \vdash lv'\{lv/y\} : \kappa'$.

(ii) *If* $\Gamma; K_1, y{:}\kappa, K_2\ [\mathsf{pc}] \vdash e$ *then* $\Gamma; K_1, K, K_2\ [\mathsf{pc}] \vdash e\{lv/y\}$.

Proof: By mutual induction on the (canonical) typing derivations of (i) and (ii).

(i) The canonical typing derivation for lv' is:

$$\frac{\Gamma; K_1, y{:}\kappa, K_2 \vdash lv' : \kappa'' \quad \vdash \kappa'' \leq \kappa'}{\Gamma; K_1, y{:}\kappa, K_2 \vdash lv' : \kappa'}$$

We proceed by cases on the rule used to conclude $\Gamma; K_1, y{:}\kappa, K_2 \vdash lv' : \kappa''$.

$\lambda^{\text{CPS}}_{\text{SEC}}$**-LVAR** Then $lv' = y$ and $\kappa'' = \kappa$, it follows that $K_1, y : \kappa, K_2 = y : \kappa$, and hence $K_1 = K_2 = \cdot$. This implies that $K_1, K, K_2 = K$ and thus by the assumption that $lv = y\{lv/y\}$ is well-typed under K, we have $\Gamma; K_1, K, K_2 \vdash lv : \kappa$, and the result follows from the subtyping of $\kappa \leq \kappa'$.

$\lambda^{\text{CPS}}_{\text{SEC}}$-**LCONT** This case follows immediately from induction hypothesis (ii).

(ii) This part of the lemma follows almost immediately from the induction hypotheses. The interesting case is $\lambda^{\text{CPS}}_{\text{SEC}}$-LETLIN, which must ensure that the ordering on the linear context is maintained.

$\lambda^{\text{CPS}}_{\text{SEC}}$-**LETLIN** By assumption, there is a derivation of the following form:

$$\frac{\begin{array}{c}\Gamma; K_b \vdash \lambda\langle \mathsf{pc}'\rangle(x' : s').\, e' : \kappa'' \\ \kappa'' = s' \to \mathsf{0} \\ \mathsf{pc} \sqsubseteq \mathsf{pc}' \qquad \Gamma; K_a, y'' : \kappa'' \; [\mathsf{pc}] \vdash e\end{array}}{\Gamma; K_1, y : \kappa, K_2 \; [\mathsf{pc}] \vdash \mathtt{letlin}\; y'' = \lambda\langle \mathsf{pc}'\rangle(x' : s').\, e' \;\mathtt{in}\; e}$$

Where $K_1, y : \kappa, K_2 = K_a, K_b$. If y appears in K_a then $K_a = K_1, y : \kappa, K_2^-$ and $K_b = K_2^+$ where $K_2^-, K_2^+ = K_2$. In this case, y can't appear in K_b and it follows that $e'\{lv/y\} = e'$. Induction hypothesis (ii) applied to $\Gamma; K_1, y : \kappa, K_2^- \; [\mathsf{pc}] \vdash e$ yields the judgment $\Gamma; K_1, K, K_2^- \; [\mathsf{pc}] \vdash e\{lv/y\}$. Thus, an application of rule $\lambda^{\text{CPS}}_{\text{SEC}}$-LETLIN yields the desired result, as $K_1, K, K_2^-, K_2^+ = K_1, K, K_2$.

The case in which y appears in K_b is similar to the one above, except that K_1 is split into K_1^- and K_1^+.

□

Next we must establish that the syntactic form of a value is determined by its type.

Lemma 4.3.3 (Canonical Forms I) *If* $\cdot \vdash v : s$ *then*

(i) *If* $s = \mathtt{bool}_\ell$ *then* $v = \mathtt{t}_{\ell'}$ *or* $v = \mathtt{f}_{\ell'}$ *for some* $\ell' \sqsubseteq \ell$.

(ii) *If* $s = \mathtt{unit}_\ell$ *then* $v = \langle\rangle_{\ell'}$ *for some* $\ell' \sqsubseteq \ell$.

(iii) *If* $s = s'\ \mathtt{ref}_\ell$ *then* $v = L^{s'}_{\ell'}$ *and* $\ell' \sqsubseteq \ell$.

(iv) *If* $s = ([\mathsf{pc}](s', \kappa) \to \mathsf{0})_\ell$ *then* $v = (\lambda[\mathsf{pc}']\ f\ (x : s'', y : \kappa').\, e)_{\ell'}$ *where* $\ell' \sqsubseteq \ell$, $\mathsf{pc} \sqsubseteq \mathsf{pc}'$, $\vdash s' \leq s''$, *and* $\vdash \kappa \leq \kappa'$.

Proof (sketch): By inspection of the typing rules and the form of values. □

Lemma 4.3.4 (Canonical Forms II) *If* $\cdot;\cdot \vdash lv : s \to \mathsf{0}$ *then* $lv = \lambda\langle \mathsf{pc}\rangle(x : s').\, e$ *where* $\vdash s \leq s'$.

Proof (sketch): By inspection of the typing rules and the form of linear values. □

Definition 4.3.1 (Locations) *For any well-typed primitive operation, $prim$ (or program, e), let $Loc(prim)$ (respectively $Loc(e)$), be the set of all locations L^s appearing in $prim$ (respectively e).*

Recall the definition of Memory Well-formedness:

Definition 4.3.2 (Memory well formedness) *A memory M is well formed, written $\vdash M$ wf if and only if*

$$\forall L^s \in |M|.\ Loc(M(L^s)) \in |M| \quad \wedge \quad \cdot \vdash M(L^s) : s$$

Next, we show that evaluation of primitive operations preserves typing, so long as the memory contains the appropriate locations used by the primitive operation.

Lemma 4.3.5 (Primitive Evaluation) *If $\cdot\ [\mathsf{pc}] \vdash prim : s, \vdash M$ wf and $Loc(prim) \subseteq \mathrm{dom}(M)$ and $\langle M, \mathsf{pc}, prim\rangle \Downarrow v$ then $\cdot \vdash v : s$.*

Proof: By cases on the evaluation rule used.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-PRIM** By assumption, we have the canonical derivation:

$$\dfrac{\dfrac{\cdot \vdash bv_\ell : \tau_\ell \quad \vdash \tau_\ell \leq \tau'_{\ell'}}{\cdot \vdash bv_\ell : \tau'_{\ell'}} \qquad \mathsf{pc} \sqsubseteq \ell'}{\cdot\ [\mathsf{pc}] \vdash bv_\ell : \tau'_{\ell'}}$$

We need to show $\cdot \vdash bv_{\ell \sqcup \mathsf{pc}} : \tau'_{\ell'}$. It follows from the derivation above that $\ell \sqsubseteq \ell'$, and as $\mathsf{pc} \sqsubseteq \ell'$ it is the case that $\ell \sqcup \mathsf{pc} \sqsubseteq \ell'$. Thus we have the inequality $\tau_{\ell \sqcup \mathsf{pc}} \leq \tau'_{\ell'}$, and so the result follows by $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-SUB and an application of the rule $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-VAL.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-BINOP** This case is similar to the previous one.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-DEREF** In this case, we assume a derivation of the following form:

$$\dfrac{\dfrac{\cdot \vdash L^{\tau_\ell}_{\ell'} : \tau_\ell\ \mathtt{ref}_{\ell'} \quad \vdash \tau_\ell\ \mathtt{ref}_{\ell'} \leq \tau_\ell\ \mathtt{ref}_{\ell''}}{\cdot \vdash L^{\tau_\ell}_{\ell'} : \tau_\ell\ \mathtt{ref}_{\ell''}} \qquad \mathsf{pc} \sqsubseteq \ell \sqcup \ell''}{\cdot\ [\mathsf{pc}] \vdash\ !L^{\tau_\ell}_{\ell'} : \tau_{\ell \sqcup \ell''}}$$

By the well-formedness of M, $M(L^{\tau_\ell}) = bv_{\ell'''}$ and we also have $\cdot \vdash bv_{\ell'''} : \tau_\ell$. This implies that $\ell''' \sqsubseteq \ell$. We must show that $\cdot \vdash bv_{\ell' \sqcup \ell''' \sqcup \mathsf{pc}} : \tau_{\ell \sqcup \ell''}$ but this follows from the transitivity of subtyping and $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-SUB because the label inequalities in the above derivation yield:

$$\begin{array}{rcl} \ell' \sqcup \ell''' \sqcup \mathsf{pc} & \sqsubseteq & \ell'' \sqcup \ell''' \sqcup \mathsf{pc} \\ & \sqsubseteq & \ell'' \sqcup \ell \sqcup \mathsf{pc} \\ & \sqsubseteq & \ell'' \sqcup \ell \sqcup (\ell \sqcup \ell'') \\ & \sqsubseteq & \ell \sqcup \ell'' \end{array}$$

□

The next lemma is used in the proof of subject reduction in the case for evaluating conditional expressions. The idea is that because the static labels are just approximations to the actual run-time labels that appear in the operational semantics, we must have a way of accounting for the difference between what is statically inferred and what takes place during evaluation. Thus, while the static label of the variable x in the conditional $\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ may be ℓ, dynamically the value substituted for x might have any label $\ell' \sqsubseteq \ell$. Since the rules for evaluating conditionals use the dynamic label ℓ', in order to establish subject-reduction, the branches of the conditional must be well-typed under the weaker constraint.

Lemma 4.3.6 (Program Counter Variance)

If $\Gamma; K\ [\mathsf{pc} \sqcup \ell] \vdash e$ *and* $\ell' \sqsubseteq \ell$ *then* $\Gamma; K\ [\mathsf{pc} \sqcup \ell'] \vdash e$.

Proof: The proof is by induction on the derivation that e is well-typed. Note that because $\mathsf{pc} \sqcup \ell' \sqsubseteq \mathsf{pc} \sqcup \ell$ all inequalities involving $\mathsf{pc} \sqcup \ell$ on the left of $\sqsubseteq$ in the typing rules will still hold with $\mathsf{pc} \sqcup \ell'$. □

The subject reduction lemma follows almost immediately from the previous lemmas. It says that well-typing is preserved by evaluation.

Lemma 4.3.7 (Subject Reduction) *If* $\cdot; K\ [\mathsf{pc}] \vdash e$, *and* $\vdash M$ wf, *and* $Loc(e) \subseteq \mathrm{dom}(M)$ *and* $\langle M, \mathsf{pc}, e\rangle \rightarrow \langle M', \mathsf{pc}', e'\rangle$ *then* $\cdot; K\ [\mathsf{pc}'] \vdash e'$ *and* $\vdash M'$ wf, *and* $Loc(e') \subseteq \mathrm{dom}(M')$.

Proof: By cases on the transition step taken:

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**Eval-LetPrim** Because $\texttt{let}\ x = prim\ \texttt{in}\ e$ is well-typed, $prim$ is too. Thus by the Primitive Evaluation Lemma, $prim$ evaluates to a value v of the same type. Substitution I, part (iv) tells us that $e\{v/x\}$ is well-typed. Because M doesn't change it is still well-formed, and to see that $Loc(e\{v/x\}) \subseteq \mathrm{dom}(M)$ consider that the only way $Loc(e\{v/x\})$ could be larger than $Loc(\texttt{let}\ x = prim\ \texttt{in}\ e)$ is if $prim$ is a dereference operation and the memory location contains another location not in e. This case is covered by the requirement that M is well-formed and hence closed under dereference.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**Eval-LetRef** By assumption, that $\cdot; K\ [\mathsf{pc}] \vdash \texttt{let}\ x = \texttt{ref}^s bv_\ell\ \texttt{in}\ e$. Working backwards through the canonical derivation yields the following antecedents: $\cdot \vdash bv_\ell : \tau_\ell$, and $\vdash \tau_\ell \leq s$, and $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$, and $x : s\ \texttt{ref}_{\mathsf{pc}}; K\ [\mathsf{pc}] \vdash e$. From $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-Loc we have $\cdot \vdash L^s_{\ell' \sqcup \mathsf{pc}} : s\ \texttt{ref}_{\ell'}$. This fact, plus the well-typedness of e lets

us apply Substitution Lemma I, (iv) to conclude $\cdot; K\ [\mathsf{pc}] \vdash e\{L^s_{\ell' \sqcup \mathsf{pc}}/x\}$. Now, to see that the conditions on M are maintained, note that if bv contains a location, then it is contained in the set of locations of the entire let expression and thus, by assumption must occur in the domain of M. This implies that $M[L^s \mapsto bv_{\ell \sqcup \mathsf{pc}}]$ is still closed under dereference. Finally, we must check that $\cdot \vdash bv_{\ell \sqcup \mathsf{pc}} : s$, but this follows from subsumption and the facts that $\tau_\ell \leq s$ and $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-SET** This case follows similarly to the previous case.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-LETLIN** This case follows from Substitution II, (ii), and the fact that the conditions on M are satisfied after the substitution. Note that the order of K is preserved by the step.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-COND1** Assume that $\cdot; K\ [\mathsf{pc}] \vdash \texttt{if}\ \mathtt{t}_\ell\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. It follows that $\cdot \vdash 0_\ell : \texttt{bool}_{\ell'}$ and $\ell \sqsubseteq \ell'$ and that $\cdot; K\ [\mathsf{pc} \sqcup \ell'] \vdash e_1$. It follows by the Program Counter Variance Lemma that $\cdot; K\ [\mathsf{pc} \sqcup \ell] \vdash e_1$. Because M doesn't change and it initially satisfied the well-formedness conditions and the locations of e_1 are a subset of the locations of the entire conditional, M is still valid after the transition.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-COND2** This case is nearly identical to the previous one.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-GOTO** This case follows from the well-typedness of the body of the continuation being jumped to, plus two applications of Substitution I, (iv) and one application of Substitution II, (ii). The fact that $bv_{\ell' \sqcup \mathsf{pc}}$ has type s follows from subsumption and the facts that $\mathsf{pclabel}(s)$ and $\ell' \sqsubseteq \mathsf{label}(s)$.

$\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-**EVAL-LGOTO** This case is similar to the previous case.

□

The progress lemma says that the dynamic checks performed by the operational semantics are actually unnecessary: a well-typed program is either already a value, or it can perform a step of computation.

Lemma 4.3.8 (Progress) *If* $\cdot; \cdot\ [\mathsf{pc}] \vdash e$ *and* M wf *and* $Loc(e) \subseteq \mathrm{dom}(M)$*, then either* e *is of the form* $\texttt{halt}^s\ v$ *or there exist* M', A', pc'*, and* e' *such that*

$$\langle M,\ \mathsf{pc},\ e\rangle \rightarrow \langle M',\ A',\ \mathsf{pc}'\rangle e'$$

Proof (sketch): By the Canonical Forms lemmas and inspection of the rules. We must ensure that conditions such as $\ell \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(s)$ on rule $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-LETREF are met by well-typed terms. These constraints are taken from the typing derivations. For example, for $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-LETREF the fact that $\mathsf{pc} \sqsubseteq \mathsf{label}(s)$ is an immediate antecedent; we have $\ell \sqsubseteq \mathsf{label}(s)$ from the subtyping rules and the fact that the value is well formed.
□

Definition 4.3.3 *A configuration* $\langle M, \mathtt{pc}, e\rangle$ *is* stuck *if* e *is not* $\mathtt{halt}^s\ v$ *for some value* v *or no transition rule applies.*

Theorem 4.3.1 (Type Soundness) *Well-typed programs do not get stuck.*

Proof: By induction on the number of transition steps taken, using Subject Reduction and Progress. □

Note that Subject Reduction holds for terms containing free occurrences of linear variables. This fact is important for proving that the ordering on linear continuations is respected. The Progress Lemma (and hence Soundness) applies only to closed terms.

4.4 Noninterference

This section proves a noninterference result for the secure CPS language, generalizing a previous result from Smith and Volpano [SV98]. The approach is to use a preservation-style argument that shows a particular invariant related to low-security views of a well-typed program is maintained by each computation step.

In contrast to the previous work, the proof here handles the case of first-class continuations and structured memories. The Smith and Volpano language permits only simple `while`-loops and `if` statements and restricts memories locations to hold only integer values. Heintze and Riecke's SLam calculus [HR98] permits first-class functions and reference cells, but their work does not prove a noninterference result.

Since this proof was originally published [ZM01a, ZM02], other researchers have given alternative approaches to proving noninterference. Pottier and Conchon give a proof for Core ML, a language with higher-order functions, references, and `let`-polymorphism in the style of SML [MTHM97]. Their proof differs quite substantially from the one presented here. Honda and Yoshida extend the Smith and Volpano system to include general references, first-class functions, and concurrency. Their noninterference result is established by a CPS-like translation to a secure version of Milner's pi calculus [HY02], but the full proof of noninterference has not been published.

Informally, the noninterference result says that low-security computations are not able to observe high-security data. Here, the term "low-security" is relative to an arbitrary point, ζ, in the security lattice $\mathcal{L}$. Thus, ℓ is a low-security label whenever $\ell \sqsubseteq \zeta$. Similarly, "high-security" refers to those labels $\not\sqsubseteq \zeta$. The security level of a computation is indicated by the label of the program counter under which the computation is taking place. Thus, by "low-security computation", we mean a transition step in the operational semantics whose starting configuration (the one before the step) contains a $\mathtt{pc} \sqsubseteq \zeta$.

The proof shows that high-security data and computation can be arbitrarily changed without affecting the value of any computed low-security result. Furthermore, memory

locations visible to low-security observers (locations storing data labeled $\sqsubseteq \zeta$) are likewise unaffected by high-security values. This characterization reduces noninterference to the problem of showing that a given program e_1 is equivalent (from a low-security observer's point of view) to any program e_2 that differs from e_1 only in its high-security parts.

Key to the argument is a formal definition of "low-equivalence," by which we intend to capture the property that two programs' executions are indistinguishable by an observer only able to see the low-security portions of memory and machine state.

How do we show that configurations $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle$ and $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle$ behave identically from the low-security point of view? Clearly, the memories M_1 and M_2 must agree on the values contained in low-security locations. In addition, if $\mathsf{pc}_1, \mathsf{pc}_2 \sqsubseteq \zeta$, meaning that e_1 and e_2 might perform actions visible to low observers (such as modifying a low-security memory location), the programs necessarily must perform the same computation on low-security values. On the other hand, when $\mathsf{pc} \not\sqsubseteq \zeta$, the actions of e_1 and e_2 should be invisible to the low view.

This intuition guides the formal definition of low-equivalence, which we write $\approx_\zeta$. The definition builds on standard alpha-equivalence (written $\equiv_\alpha$) as a base notion of equality. We use substitutions to factor out the relevant high-security values and those linear continuations that reset the program-counter label to be $\sqsubseteq \zeta$.

Definition 4.4.1 (Substitutions) *For context Γ, let $\gamma \models \Gamma$ mean that γ is a finite map from variables to closed values such that $\mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma)$ and for every $x \in \mathrm{dom}(\gamma)$ it is the case that $\cdot \vdash \gamma(x) : \Gamma(x)$.*

Substitution application, written $\gamma(e)$, indicates capture-avoiding substitution of the value $\gamma(x)$ for free occurrences of x in e, for each x in the domain of γ.

To show ζ-equivalence between e_1 and e_2, we should find substitutions γ_1 and γ_2 containing the relevant high-security data such that $e_1 \equiv_\alpha \gamma_1(e)$ and $e_2 \equiv_\alpha \gamma_2(e)$—both e_1 and e_2 look the same as e after factoring out the high-security data.

The other important piece of the proof is that we can track the linear continuations that restore the program counter to a label that is $\not\sqsubseteq \zeta$. Here is where the stack ordering on linear continuations comes into play: The operational semantics guarantees that the program-counter label is monotonically increasing *except* when a linear continuation is invoked. If e_1 invokes a linear continuation that causes pc_1 to fall below ζ, e_2 must follow suit and call an equivalent continuation; otherwise the low-security observer might distinguish e_1 from e_2. The stack ordering on linear continuations is exactly the property that forces e_2 to invoke the same low-security continuation as e_1.

Note that only the low-security linear continuations are relevant to the ζ-equivalence of two programs—the high-security linear continuations in the programs may differ. Furthermore, our plan is to establish an invariant with respect to the operational semantics. This means we must be able to keep track of the relevant low-security continuations

as they are introduced and consumed by `letlin` and `lgoto`. There is a slight technical difficulty in doing so in the substitution-style operational semantics we have presented: We want to maintain the invariant that ζ-equivalent programs always have equivalent pending low-security continuations. Statically, the linear context names these continuations, but dynamically, these variables are substituted away—there is no way to name the "next" low-security linear continuation.

To get around this problem, our approach is to introduce auxiliary substitutions that map stacks of linear variables to low-security linear continuations. The top of stack corresponds to the next low-security linear continuation that will be invoked.

Definition 4.4.2 (Linear Continuation Stack) *Let K be an ordered list (a stack) of linear variables $y_1 : \kappa_1, \ldots, y_n : \kappa_n$ such that $n \geq 0$. We write $\Gamma \vdash k \models K$ to indicate that k is a substitution that maps each y_i to a linear value such that $\Gamma; \cdot \vdash k(y_1) : \kappa_1$ and $\Gamma; y_{i-1} : \kappa_{i-1} \vdash k(y_i) : \kappa_i$ for $i \in \{2 \ldots n\}$.*

Application of a stack substitution k to a term e is defined as:

$$k(e) = e\{k(y_n)/y_n\}\{k(y_{n-1})/y_{n-1}\} \ldots \{k(y_1)/y_1\}.$$

Note that the order of the substitutions is important because the continuation $k(y_n)$ may refer to the linear variable y_{n-1}.

Two linear continuation stacks k_1 and k_2 are equivalent if they have the same domain and map each variable to equivalent continuations. We must also ensure that the stack contains *all* of the pending low-security continuations.

Definition 4.4.3 (`letlin` Invariant) *A term satisfies the `letlin` invariant if every linear continuation expression $\lambda\langle \mathsf{pc} \rangle (x : s).\, e$ appearing in the term is either in the binding position of a `letlin` or satisfies $\mathsf{pc} \not\sqsubseteq \zeta$.*

The idea behind the `letlin` invariant is that when $k(e)$ is a closed term such that e satisfies the `letlin` invariant, any invocation of a low-security linear continuation in e must arise from the substitution k—in other words, k contains any pending low-security linear continuations.

Extending these ideas to values, memories, and machine configurations we obtain the definitions below:

Definition 4.4.4 (ζ-Equivalence)

$\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ — *If $\gamma_1, \gamma_2 \models \Gamma$ and for every $x \in \mathrm{dom}(\Gamma)$ it is the case that $\mathsf{label}(\gamma_i(x)) \not\sqsubseteq \zeta$ and $\gamma_i(x)$ satisfies the `letlin` invariant.*

$\Gamma \parallel K \vdash k_1 \approx_\zeta k_2$	*If* $\Gamma \vdash k_1, k_2 \models K$ *and for every* $y \in \mathrm{dom}(K)$ *it is the case that* $k_i(y) \equiv_\alpha \lambda\langle \mathsf{pc} \rangle (x : s).\, e$ *such that* $\mathsf{pc} \sqsubseteq \zeta$ *and* e *satisfies the* `letlin` *invariant.*
$v_1 \approx_\zeta v_2 : s$	*If there exist* Γ, γ_1, *and* γ_2 *plus terms* $v_1' \equiv_\alpha v_2'$ *such that* $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, *and* $\Gamma \vdash v_i' : s$ *and* $v_i = \gamma_i(v_i')$ *and each* v_i' *satisfies the* `letlin` *invariant.*
$M_1 \approx_\zeta M_2$	*If for all* $L^s \in \mathrm{dom}(M_1) \cup \mathrm{dom}(M_2)$, $\mathsf{label}(s) \sqsubseteq \zeta$ *implies that* $L^s \in \mathrm{dom}(M_1) \cap \mathrm{dom}(M_2)$ *and* $M_1(L^s) \approx_\zeta M_2(L^s) : s$.

Finally, we can put all of these requirements together to define the ζ-equivalence of two machine configurations, which also gives us the invariant for the noninterference proof.

Definition 4.4.5 (Noninterference Invariant) *The noninterference invariant is a predicate on machine configurations, written* $\Gamma \parallel K \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1 \rangle \approx_\zeta \langle M_2,\, \mathsf{pc}_2,\, e_2 \rangle$, *that holds if there exist substitutions* $\gamma_1, \gamma_2, k_1, k_2$ *and terms* e_1' *and* e_2' *such that the following conditions are all met:*

- **(i)** $e_1 = \gamma_1(k_1(e_1'))$ *and* $e_2 = \gamma_2(k_2(e_2'))$.
- **(ii)** $\Gamma; K\ [\mathsf{pc}_1] \vdash e_1'$ *and* $\Gamma; K\ [\mathsf{pc}_2] \vdash e_2'$
- **(iii)** *Either* (a) $\mathsf{pc}_1 = \mathsf{pc}_2 \sqsubseteq \zeta$ and $e_1' \equiv_\alpha e_2'$ or
  (b) $\mathsf{pc}_1 \not\sqsubseteq \zeta$ and $\mathsf{pc}_2 \not\sqsubseteq \zeta$.
- **(iv)** $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ *and* $\Gamma \parallel K \vdash k_1 \approx_\zeta k_2$
- **(v)** $Loc(e_1) \subseteq \mathrm{dom}(M_1)$ *and* $Loc(e_2) \subseteq \mathrm{dom}(M_2)$ *and* $M_1 \approx_\zeta M_2$.
- **(vi)** *Both* e_1' *and* e_2' *satisfy the* `letlin` *invariant.*

The main technical work of the noninterference proof is a preservation argument showing that the Noninterference Invariant holds after each transition. When the pc is low, equivalent configurations execute in lock step (modulo high-security data). After the program branches on high-security information (or jumps to a high-security continuation), the two programs may temporarily get out of sync, but during that time they may affect only high-security data. If the program counter drops low again (via a linear continuation), both computations return to lock-step execution.

We first show that ζ-equivalent configuration evaluate in lock step as long as the program counter has low security.

Lemma 4.4.1 (Low-pc Step) *Suppose*

- $\Gamma \parallel K \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1 \rangle \approx_\zeta \langle M_2,\, \mathsf{pc}_2,\, e_2 \rangle$
- $\mathsf{pc}_1 \sqsubseteq \zeta$ *and* $\mathsf{pc}_2 \sqsubseteq \zeta$

- $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \to \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle$

then $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \to \langle M_2',\, \mathsf{pc}_2',\, e_2'\rangle$ *and there exist* Γ' *and* K' *such that:*

$$\Gamma' \parallel K' \vdash \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle \approx_\zeta \langle M_2',\, \mathsf{pc}_2',\, e_2'\rangle$$

Proof: Let $e_1 = \gamma_1(k_1(e_1''))$ and $e_2 = \gamma_2(k_2(e_2''))$ where the substitutions are as described by the conditions of the Noninterference Invariant. Because $\mathsf{pc}_i \sqsubseteq \zeta$, clause (iii) implies that e_1'' and e_2'' must be α-equivalent expressions and $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc}$. Hence the only difference in their behavior arises due to the substitutions or the different memories. We proceed by cases on the transition step taken by the first program. The main technique is to reason by cases on the security level of the value used in the step—if it's low-security, by α-equivalence, both programs compute the same values, otherwise we extend the substitutions γ_1 and γ_2 to contain the high-security data. We show a few representative cases in detail to give the flavor of the argument, the remainder follow in a similar fashion.

$\lambda^{\text{CPS}}_{\text{SEC}}$-**EVAL-LETPRIM**

$$\frac{\langle M,\, \mathsf{pc},\, prim\rangle \;\Downarrow\; v}{\langle M,\, \mathsf{pc},\, \mathtt{let}\; x = prim\; \mathtt{in}\; e\rangle \to \langle M,\, \mathsf{pc},\, e\{v/x\}\rangle}$$

In this case, e_1'' and e_2'' must be of the form $\mathtt{let}\; x = prim\; \mathtt{in}\; e$, consequently e_2 must also transition via rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETPRIM. Because $M_1 = M_1'$ and $M_2 = M_2'$, and the locations found in terms e_1' and e_2' are found in e_1 and e_2 respectively, condition (v) of the Noninterference Invariant holds after the transition.

It suffices to find an e' and γ_i' such that $e_1' = \gamma_1'(k_1(e'))$ and $e_2' = \gamma_2'(k_2(e'))$. If $prim$ is a value, then take $\gamma_i' = \gamma_i$ and let $e' = e\{prim/x\}$. These choices satisfy the conditions. Otherwise, $prim$ is not a value. Consider the evaluation $\langle M_1,\, \mathsf{pc},\, \gamma_1(prim)\rangle \;\Downarrow\; bv_\ell$. There are two cases.

If $\ell \sqsubseteq \zeta$ then $prim$ cannot contain any free variables, for otherwise condition (iv) would be violated—evaluation rules $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-PRIM and $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-BINOP imply that the label of the resulting value be higher than the label of any constituent, and all the values of γ_1 have label higher than ζ. Thus, $\gamma_1(prim) = prim = \gamma_2(prim)$. $\langle M_2,\, \mathsf{pc},\, \gamma_2(prim)\rangle \;\Downarrow\; bv_{\ell'}'$ and because $M_1 \approx_\zeta M_2$ we have $bv_\ell \approx_\zeta bv_{\ell'}' : s$. Thus, there exist Γ'', γ_1'', γ_2'' and values $v_1 \equiv_\alpha v_2$ such that $\Gamma'' \vdash \gamma_1'' \approx_\zeta \gamma_2''$ and $bv_\ell = \gamma_1''(v_1)$ and $bv_{\ell'}' = \gamma_2''(v_2)$. Thus, we take $\gamma_1' = \gamma_1 \cup \gamma_1''$, $\gamma_2' = \gamma_2 \cup \gamma_2''$ and $e_i'' = e\{v_i/x\}$. Conditions (iv), (v), and (vi) hold trivially; conditions (i), (ii), and (iii) are easily verified based on the operational semantics and the fact that $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc}$.

If $\ell \not\sqsubseteq \zeta$ then $\langle M_2,\, \mathsf{pc},\, \gamma_2(prim)\rangle \;\Downarrow\; bv_{\ell'}'$ where it is also the case that $\ell' \not\sqsubseteq \zeta$. ($prim$ either contains a variable, which forces ℓ' to be high, or $prim$ contains a

value explicitly labeled with a high-label.) It follows that $bv_\ell \approx_\zeta bv'_{\ell'} : s$ and we take $\gamma'_1 = \gamma_1\{x \mapsto bv_\ell\}$ and $\gamma'_2 = \gamma_2\{x \mapsto bv'_{\ell'}\}$, and $e''_i = e$, which are easily seen to satisfy the conditions.

$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETREF

$$\frac{\ell \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(s) \quad L^s \notin \mathrm{dom}(M)}{\begin{array}{l}\langle M,\, \mathsf{pc},\, \mathtt{let}\ x = \mathtt{ref}^s bv_\ell\ \mathtt{in}\ e\rangle \\ \quad \to \langle M[L^s \mapsto bv_{\ell \sqcup \mathsf{pc}}],\, \mathsf{pc},\, e\{L^s_{\mathsf{pc}}/x\}\rangle\end{array}}$$

In this case, e''_1 and e''_2 must be of the form $\mathtt{let}\ x = \mathtt{ref}^s v\ \mathtt{in}\ e$ where $v = bv_\ell$. Note that $\gamma_1(v) \approx_\zeta \gamma_2(v) : s$ so it follows that $M'_1 = M_1[L^s \mapsto \gamma_1(v) \sqcup \mathsf{pc}]$ is ζ-equivalent to $M'_2 = M_2[L^s \mapsto \gamma_2(v) \sqcup \mathsf{pc}]$, satisfying invariant (v). Now we simply take $\gamma'_i = \gamma_i$, and note that $e'_1 = \gamma_1(e\{L^s_{\mathsf{pc}}/x\})$ and $e'_2 = \gamma_2(e\{L^s_{\mathsf{pc}}/x\})$ satisfy the required invariants.

$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-LETLIN

$$\langle M,\, \mathsf{pc},\, \mathtt{letlin}\ y' = lv\ \mathtt{in}\ e\rangle \to \langle M,\, \mathsf{pc},\, e\{lv/y'\}\rangle$$

If $lv = \mathtt{halt}^s$, then the Noninterference Invariant holds trivially after the transition. Otherwise, $lv = \lambda\langle \mathsf{pc}'\rangle(x : s).\, e'$. In this case, e''_1 and e''_2 are $\mathtt{letlin}\ y' = \lambda\langle \mathsf{pc}'\rangle(x : s).\, e'\ \mathtt{in}\ e$. If $\mathsf{pc}' \sqsubseteq \zeta$, simply take $K' = K, y' : s \to \mathtt{0}$ and choose $k'_i = k_i \cup \{y' \mapsto \lambda\langle \mathsf{pc}'\rangle(x : s).\, e'\}$, which satisfies invariant (iv) because $k_1 \approx_\zeta k_2$ and the terms e''_1 and e''_2 are well-typed. In the case that $\mathsf{pc}' \not\sqsubseteq \zeta$, we take $k'_i = k_i$ and choose each e'_i to be $e\{\lambda\langle \mathsf{pc}'\rangle(x : s).\, e'/y'\}$ which again satisfies invariant (iv) and the `letlin`-invariant, (vi). The remaining invariants are easily seen to hold because the memories and ordinary value substitutions do not change.

$\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-COND1

$$\langle M,\, \mathsf{pc},\, \mathtt{if}\ \mathtt{t}_\ell\ \mathtt{then}\ e'_a\ \mathtt{else}\ e'_b\rangle \to \langle M,\, \mathsf{pc} \sqcup \ell,\, e'_a\rangle$$

In this case, e''_1 and e''_2 must be of the form $\mathtt{if}\ v\ \mathtt{then}\ e_a\ \mathtt{else}\ e_b$. If v is not a variable, then by α-equivalence, e_2 must also transition via rule $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-COND1. Because M_1 and M_2 don't change, it is easy to establish that all of the invariants hold. When v is a variable, $\gamma_1(v) = \mathtt{t}_\ell$ for $\ell \not\sqsubseteq \zeta$. Similarly, $\gamma_2(v) = b_{\ell'}$ for $\ell' \not\sqsubseteq \zeta$. Because b could be either `t` or `f`, we don't know whether the second program transitions via $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-COND1 or $\lambda^{\text{CPS}}_{\text{SEC}}$-EVAL-COND2, but in either case it is easy to establish that the resulting configurations are $\approx_\zeta$. Clause (i) holds via the original substitutions; clause (ii) follows from the fact that the configurations are well-typed; clause (iii) holds because part (b) lets us relate *any* high-security programs; clauses (iv) through (vi) are a simple consequence of ζ-equivalence of e_1 and e_2.

$\lambda^{\text{CPS}}_{\text{SEC}}$-**EVAL-GOTO**

$$\frac{\mathsf{pc} \sqsubseteq \mathsf{pc}' \quad v = (\lambda[\mathsf{pc}']\ f\ (x\!:\!s).\ y\!:\!\kappa e)_\ell}{\langle M,\ \mathsf{pc},\ \texttt{goto}\ v\ v'\ lv\rangle \rightarrow \langle M,\ \mathsf{pc}' \sqcup \ell,\ e\{v/f\}\{v' \sqcup \mathsf{pc}/x\}\{lv/y\}\rangle}$$

In this case, each $e''_i = \texttt{goto}\ v\ v'\ lv$. It must be the case that $\gamma_1(v) = (\lambda[\mathsf{pc}']f(x : s, y : \kappa).\, e)_\ell$. If $\ell \sqsubseteq \zeta$, then $v = (\lambda[\mathsf{pc}']f(x : s, y : \kappa).\, e')_\ell$ where $e' = \gamma_1(e)$ because, by invariant (iii), the continuation could not be found in γ_1. Note that $\gamma_1(v') \approx_\zeta \gamma_2(v') : s$. There are two sub-cases, depending on whether $\gamma_1(v')$ has label $\sqsubseteq \zeta$. If so, it suffices to take $\Gamma' = \Gamma$, $K' = K$, and leave the substitutions unchanged, for we have $e'_i = \gamma_i(k_i(e\{v/f\}\{\gamma_i(v') \sqcup \mathsf{pc}/x\}\{lv/y\}))$. Otherwise, if the label of $\gamma_1(v') \not\sqsubseteq \zeta$, we take $\Gamma' = \Gamma, x : s$ and $\gamma'_i = \gamma_i\{x \mapsto \gamma_i(v') \sqcup \mathsf{pc}\}$. The necessary constraints are then met by $e'_i = \gamma'_i(k_i(e\{v/f\}\{lv/y\}))$.

The other case is that $\ell \not\sqsubseteq \zeta$, and hence the label of $\gamma_2(v)$ is also $\not\sqsubseteq \zeta$. Thus, $\mathsf{pc}'_1 = \mathsf{pc} \sqcup \ell \not\sqsubseteq \zeta$ and $\mathsf{pc}'_2 \not\sqsubseteq \zeta$. The resulting configurations satisfy part (b) of clause (iii). The bodies of the continuations are irrelevant, as long as the other invariants are satisfied, but this follows if we build the new value substitutions as in the previous paragraph

$\lambda^{\text{CPS}}_{\text{SEC}}$-**EVAL-LGOTO** This follows analogously to the previous case, except that the stronger constraints that relate linear contexts imply that the continuations being invoked are actually α-equivalent.

□

Next, we prove that linear continuations do indeed get called in the order described by the linear context. The proof follows directly from Subject Reduction and the linearity built into the type system.

Lemma 4.4.2 (Linear Continuation Ordering) *Assume $K = y_n : \kappa_n, \ldots, y_1 : \kappa_1$ for some $n \geq 1$, each κ_i is a linear continuation type, and $\cdot; K\ [\mathsf{pc}] \vdash e$. If $\cdot \vdash k \models K$, then in the evaluation starting from any well-formed configuration $\langle M,\ \mathsf{pc},\ k(e)\rangle$, the continuation $k(y_1)$ will be invoked before any other $k(y_i)$.*

Proof: The operational semantics and Subject Reduction are valid for open terms. Progress, however, does not hold for open terms. Consider the evaluation of the open term e in the configuration $\langle M,\ \mathsf{pc},\ e\rangle$. If the computation diverges, none of the y_i's ever reach an active position, and hence are not invoked. Otherwise, the computation must get stuck (it can't halt because Subject Reduction implies that all configurations are well-typed; the `halt` expression requires an empty linear context). The stuck term must be of the form $\texttt{lgoto}\ y_i\ v$, and because it is well-typed, rules $\lambda^{\text{CPS}}_{\text{SEC}}$-LVAR and $\lambda^{\text{CPS}}_{\text{SEC}}$-LGOTO together imply that $y_i = y_1$. □

We use the stack ordering property of linear continuations, as made explicit in the Progress Lemma, to prove that equivalent high-security configurations eventually return to equivalent low-security configurations.

Lemma 4.4.3 (High-pc Step) *Suppose*

- $\Gamma \parallel K \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \approx_\zeta \langle M_2,\, \mathsf{pc}_2,\, e_2\rangle$
- $\mathsf{pc}_1 \not\sqsubseteq \zeta$ *and* $\mathsf{pc}_2 \not\sqsubseteq \zeta$
- $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \rightarrow \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle$

then either e_2 diverges or $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \longmapsto^* \langle M_2',\, \mathsf{pc}_2',\, e_2'\rangle$ and there exist Γ' and K' such that $\Gamma' \parallel K' \vdash \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle \approx_\zeta \langle M_2',\, \mathsf{pc}_2',\, e_2\rangle$.

Proof: By cases on the transition step of the first configuration. Because $\mathsf{pc}_1 \not\sqsubseteq \zeta$ and all rules except $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-LGOTO increase the program-counter label, we may choose zero steps for e_2 and still show that $\approx_\zeta$ is preserved. Condition (iii) holds via part (b). The other invariants follow because all values computed and memory locations written to must have labels higher than pc_1 (and hence $\not\sqsubseteq \zeta$). Thus, the only memory locations affected are high-security: $M_1' \approx_\zeta M_2 = M_2'$. Similarly, $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-LETLIN forces linear continuations introduced by e_1 to have $\mathsf{pc} \not\sqsubseteq \zeta$. Substituting them in e_1 maintains clause (v) of the invariant.

Now consider $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-EVAL-LGOTO. Let $e_1 = \gamma_1(k_1(e_1''))$, then $e_1'' = \texttt{lgoto}\ lv\ v_1$ for some lv. If lv is not a variable, clause (vi) ensures that the program counter in lv's body is $\not\sqsubseteq \zeta$. Pick 0 steps for the second configuration as above, and it easily follows that the resulting configurations are $\approx_\zeta$ under Γ and K. Otherwise, lv is the variable y. By assumption, $k_1(y) = \lambda\langle \mathsf{pc}\rangle(x : s).\, e$, where $\mathsf{pc} \sqsubseteq \zeta$. Assume e_2 does not diverge. By the Progress Lemma $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \longmapsto^* \langle M_2',\, \mathsf{pc}_2',\, \texttt{lgoto}\ k_2(y)\ v_2\rangle$ (by assumption, it can't diverge). Simple induction on the length of this transition sequence shows that $M_2 \approx_\zeta M_2'$, because the program counter may not become $\sqsubseteq \zeta$. Thus, $M_1' = M_1 \approx_\zeta M_2 \approx_\zeta M_2'$. By invariant (iv), $k_2(y) \equiv_\alpha k_1(y)$. Furthermore, $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-LGOTO requires that $\mathsf{label}(s) \not\sqsubseteq \zeta$. Let $\Gamma' = \Gamma, x : s$, $\gamma_1' = \gamma_1\{x \mapsto \gamma_1(v_1) \sqcup \mathsf{pc}_1\}$, $\gamma_2' = \gamma_2\{x \mapsto \gamma_2(v_2) \sqcup \mathsf{pc}_2\}$; take k_1' and k_2' to be the restrictions of k_1 and k_2 to the domain of the tail of K, which we choose for K'. Finally, let $e_1' = \gamma_1'(k_1'(e))$ and $e_2' = \gamma_2'(k_2'(e))$. All of the necessary conditions are satisfied as is easily verified via the operational semantics. □

Finally, we use the above lemmas to prove noninterference. Assume a program that computes a low-security Boolean has access to high-security data. Arbitrarily changing the high-security data does not affect the program's result.

First, some convenient notation for the initial linear continuation: Let

$$stop(s) \stackrel{\mathrm{def}}{=} \lambda\langle\bot\rangle(x : s).\, \texttt{halt}^s\ x$$

It has type $\kappa_{stop(s)} = s \rightarrow 0$.

Theorem 4.4.1 (Noninterference) *Suppose*

- $x\!:\!s; y\!:\!\texttt{bool}_\zeta \to \texttt{0}\ [\bot] \vdash e$ *for some initial program* e*.*
- $\mathsf{label}(s) \not\sqsubseteq \zeta$
- $\cdot \vdash v_1, v_2 : s$

Then $\langle \emptyset,\ \bot,\ e\{v_1/x\}\{\mathit{stop}(\texttt{bool}_\zeta)/y\}\rangle \longmapsto^* \langle M_1,\ \zeta,\ \texttt{halt}^{\texttt{bool}_\zeta}\ n_{\ell_1}\rangle$ *and* $\langle \emptyset,\ \bot,\ e\{v_2/x\}\{\mathit{stop}(\texttt{bool}_\zeta)/y\}\rangle \longmapsto^* \langle M_2,\ \zeta,\ \texttt{halt}^{\texttt{bool}_\zeta}\ m_{\ell_2}\rangle$
imply that $M_1 \approx_\zeta M_2$ *and* $n = m$*.*

Proof: Let e_1 be the term $e\{v_1/x\}$ and let e_2 be the term $e\{v_2/x\}$. It is easy to verify that

$$x\!:\!s \parallel y\!:\!\texttt{bool}_\zeta \to \texttt{0} \vdash \langle \emptyset,\ \bot,\ e_1\rangle \approx_\zeta \langle \emptyset,\ \bot,\ e_2\rangle$$

by letting $\gamma_1 = \{x \mapsto v_1\}$, $\gamma_2 = \{x \mapsto v_2\}$, and $k_1 = k_2 = \{y \mapsto y\}$. Induction on the length of the first expression's evaluation sequence, using the Low- and High-pc Step lemmas plus the fact that the second evaluation sequence terminates, implies that

$$\Gamma \parallel K \vdash \langle M_1,\ \zeta,\ \texttt{halt}^{\texttt{bool}_\zeta}\ n_{\ell_1}\rangle \approx_\zeta \langle M_2,\ \zeta,\ \texttt{halt}^{\texttt{bool}_\zeta}\ m_{\ell_2}\rangle$$

Clause (v) of the Noninterference Invariant implies that $M_1 \approx_\zeta M_2$. Soundness implies that $\ell_1 \sqsubseteq \zeta$ and $\ell_2 \sqsubseteq \zeta$. This means, because of clause (iv), that neither n_{ℓ_1} nor m_{ℓ_2} are in the range of γ_i'. Thus, the Booleans present in the `halt` expressions do not arise from substitution. Because $\zeta \sqsubseteq \zeta$, clause (iii) implies that $\texttt{halt}^{\texttt{bool}_\zeta}\ n_{\ell_1} \equiv_\alpha \texttt{halt}^{\texttt{bool}_\zeta}\ m_{\ell_2}$, from which we obtain $n = m$ as desired.

□

4.5 Translation

The source types of $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ are like those of $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$, except that instead of continuations, $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ has functions. The type translation from $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ types to $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ types, following previous work on typed CPS conversion [HL93], is given in terms of three mutually recursive functions: $(-)^*$, for base types, $(-)^+$ for security types, and $(-)^-$ to linear continuation types:

$$\begin{array}{rcl}
\texttt{unit}^* & = & \texttt{unit} \\
\texttt{bool}^* & = & \texttt{bool} \\
(s\ \texttt{ref})^* & = & s^+\ \texttt{ref} \\
([\ell]s_1 \to s_2)^* & = & [\ell](s_1^+, s_2^-) \to \texttt{0} \\
t^+_\ell & = & (t^*)_\ell \\
s^- & = & s^+ \to \texttt{0}
\end{array}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-VAL

$$\frac{\Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{[\![\Gamma\ [\mathsf{pc}] \vdash v : s]\!]y} \rightsquigarrow \texttt{lgoto}\ y\ [\![v]\!]$$

$\lambda^{\text{REF}}_{\text{SEC}}$-APP

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : ([\mathsf{pc}']s' \to s)_\ell \quad \Gamma\ [\mathsf{pc}] \vdash e' : s' \quad \mathsf{pc} \sqsubseteq \mathsf{pc}'}{[\![\Gamma\ [\mathsf{pc}] \vdash e\ e' : s \sqcup \ell]\!]y} \rightsquigarrow$$

$$\begin{array}{l}\texttt{letlin}\ k_1 = \lambda\langle \mathsf{pc}\rangle(f : ([\mathsf{pc}']s' \to s)^+_\ell).\\ \qquad\qquad \texttt{letlin}\ k_2 = \lambda\langle \mathsf{pc}\rangle(x : s'^+).\ \texttt{goto}\ f\ x\ y\\ \qquad\qquad \texttt{in}\ [\![\Gamma\ [\mathsf{pc}] \vdash e' : s']\!]k_2\\ \texttt{in}\ [\![\Gamma\ [\mathsf{pc}] \vdash e : ([\mathsf{pc}']s' \to s)_\ell]\!]k_1\end{array}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-PRIM

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e_1 : \texttt{bool}_\ell \quad \Gamma\ [\mathsf{pc}] \vdash e_2 : \texttt{bool}_\ell}{[\![\Gamma\ [\mathsf{pc}] \vdash e_1 \oplus e_2 : \texttt{bool}_\ell]\!]y} \rightsquigarrow$$

$$\begin{array}{l}\texttt{letlin}\ k_1 = \lambda\langle \mathsf{pc}\rangle(x_1 : \texttt{bool}^+_\ell).\\ \qquad\qquad \texttt{letlin}\ k_2 = \lambda\langle \mathsf{pc}\rangle(x_2 : \texttt{bool}^+_\ell).\\ \qquad\qquad\qquad \texttt{let}\ x = x_1 \oplus x_2\ \texttt{in lgoto}\ y\ x\\ \qquad\qquad \texttt{in}\ [\![\Gamma\ [\mathsf{pc}] \vdash e_2 : \texttt{bool}_\ell]\!]k_2\\ \texttt{in}\ [\![\Gamma\ [\mathsf{pc}] \vdash e_1 : \texttt{bool}_\ell]\!]k_1\end{array}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-REF

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : s}{[\![\Gamma\ [\mathsf{pc}] \vdash \texttt{ref}^s e : s\ \texttt{ref}_{\mathsf{pc}}]\!]y} \rightsquigarrow$$

$$\begin{array}{l}\texttt{letlin}\ k = \lambda\langle \mathsf{pc}\rangle(x : s^+).\\ \qquad\qquad \texttt{let}\ r = \texttt{ref}^s\ x\ \texttt{in lgoto}\ y\ r\\ \texttt{in}\ [\![\Gamma\ [\mathsf{pc}] \vdash e : s]\!]k\end{array}$$

Figure 4.11: CPS translation

$\lambda^{\text{REF}}_{\text{SEC}}$-DEREF

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : s\ \mathtt{ref}_\ell}{[\![\Gamma\ [\mathsf{pc}] \vdash\ !e : s \sqcup \ell]\!]y}$$

$$\rightsquigarrow$$

$$\begin{array}{l}\mathtt{letlin}\ k\ =\ \lambda\langle \mathsf{pc}\rangle(r : s\ \mathtt{ref}^+_\ell).\\ \qquad\qquad \mathtt{let}\ x\ =\ !r\ \mathtt{in}\ \mathtt{lgoto}\ y\ x\\ \mathtt{in}\ [\![\Gamma\ \ [\mathsf{pc}] \vdash e : s\ \mathtt{ref}_\ell]\!]k\end{array}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-ASSN

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e_1 : s\ \mathtt{ref}_\ell \quad \Gamma \vdash e_2 : s \quad \ell \sqsubseteq \mathsf{label}(s)}{[\![\Gamma\ [\mathsf{pc}] \vdash e_1 := e_2 : \mathtt{unit}_{\mathsf{pc}}]\!]y}$$

$$\rightsquigarrow$$

$$\begin{array}{l}\mathtt{letlin}\ k_1\ =\ \lambda\langle \mathsf{pc}\rangle(x_1 : s\ \mathtt{ref}^+_\ell).\\ \qquad\qquad \mathtt{letlin}\ k_2\ =\ \lambda\langle \mathsf{pc}\rangle(x_2 : s^+).\\ \qquad\qquad\quad \mathtt{let}\ x_1 := x_2\ \mathtt{in}\ \mathtt{lgoto}\ y\ \langle\rangle_{\mathsf{pc}}\\ \qquad\qquad \mathtt{in}\ [\![\Gamma \vdash e_2 : s]\!]k_2\\ \mathtt{in}\ [\![\Gamma\ \ [\mathsf{pc}] \vdash e_1 : s\ \mathtt{ref}_\ell]\!]k_1\end{array}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-COND

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : \mathtt{bool}_\ell \quad \Gamma\ [\mathsf{pc} \sqcup \ell] \vdash e_i : s \quad i \in \{1, 2\}}{[\![\Gamma\ [\mathsf{pc}] \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : s]\!]y}$$

$$\rightsquigarrow$$

$$\begin{array}{l}\mathtt{letlin}\ k\ =\ \lambda\langle \mathsf{pc}\rangle(x : \mathtt{bool}^+_\ell).\\ \qquad\qquad \mathtt{if}\ x\ \mathtt{then}\ [\![\Gamma\ \ [\mathsf{pc} \sqcup \ell] \vdash e_1 : s]\!]y\\ \qquad\qquad\quad\ \mathtt{else}\ [\![\Gamma\ \ [\mathsf{pc} \sqcup \ell] \vdash e_1 : s]\!]y\\ \mathtt{in}\ [\![\Gamma\ \ [\mathsf{pc}] \vdash e : \mathtt{bool}_\ell]\!]k\end{array}$$

$\lambda^{\text{REF}}_{\text{SEC}}$-EXPRSUB

$$\frac{\Gamma\ [\mathsf{pc}] \vdash e : s \quad \vdash s \leq s'}{[\![\Gamma\ [\mathsf{pc}] \vdash e : s']\!]y}$$

$$\rightsquigarrow$$

$$[\![\Gamma\ [\mathsf{pc}] \vdash e : s]\!]y$$

Figure 4.12: CPS translation (continued)

Figures 4.11 and 4.12 show the term translation as a type-directed map from source typing derivations to target terms. The rules are of the form:

$$\frac{\mathcal{P}_1 \ldots \mathcal{P}_n}{[\![\Gamma\ [\mathsf{pc}] \vdash e_{source} : s]\!]y} \rightsquigarrow e_{target}$$

Here, the $\mathcal{P}_i$'s represent premises of the inference rule in the source type system. The semantic brackets around the conclusion of the inference rule indicate the typing context and the source expression to be translated, assuming that the result of the computation is to be passed to the continuation stored in the $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$-variable y. The translation of e_{source} is the term e_{target}, which may mention the translations of types appearing in the inference rule. Recursive translation of a subexpressions of e_{source}, as directed by the premise $\mathcal{P}$ of the inference rule, are indicated in e_{target} as $[\![\mathcal{P}]\!]$. For instance, the rule for translating $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$-PRIM depends on translating the two subexpressions.

Because $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ and $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ agree on the values for `unit` and `bool`, the translation of those values or variables is just the identity. Functions are the only values whose translation is not the identity, their translation is mutually recursive with the translation for program expressions:

$$\begin{array}{rcl}
[\![x]\!] & = & x \\
[\![\langle\rangle_\ell]\!] & = & \langle\rangle_\ell \\
[\![b_\ell]\!] & = & b_\ell \quad (b \in \{\mathtt{t}, \mathtt{f}\}) \\
[\![L^s]\!] & = & L^{s^+} \\
[\![(\lambda[\mathsf{pc}]\ f\ (x\!:\!s).\ e)_\ell]\!] & = & (\lambda[\mathsf{pc}] f(x\!:\!s_1^+, y\!:\!s_2^-).\ [\![\Gamma, f\!:\!s', x\!:\!s_1\ [\mathsf{pc}] \vdash e : s_2]\!]y)_\ell \\
 & & \text{where } s' = ([\mathsf{pc}]s_1 \to s_2)_\ell
\end{array}$$

For simplicity, we present an un-optimizing CPS translation, although we expect that first-class linear continuations will support more sophisticated translations, such as tail-call optimization [DF92]. To obtain the full translation of a closed term e of type s, we pass in the initial continuation instantiated at the correct type:

$$[\![e]\!] = \mathtt{letlin}\ y\ \mathtt{=}\ stop(s^+)\ \mathtt{in}\ [\![\cdot[\mathsf{pc}] \vdash e : s]\!]y$$

As expected, linear continuations are introduced by the translation at points that correspond (via the structure of the source program) to pushing an activation record on to the stack, and `lgotos` are introduced where pops occur. The linear variable y represents the current "top of stack" continuation; invoking it will cause the activation stack to be popped, after executing the body of the continuation y. Note that *all* of the implicit control flow of the source language is expressed by ordered linear continuations;

ordinary continuations are used only to express source-level functions, which, because they might be copied or never invoked, are inherently nonlinear. However, the unique return continuation of a function is represented by a linear continuation.

The basic lemma for establishing type correctness of the translation is proved by induction on the typing derivation of the source term. This result also shows that the CPS language is at least as precise as the source.

Lemma 4.5.1 (Type Translation)
If $\Gamma[\mathsf{pc}] \vdash e : s$ *then* $\Gamma^+; y : s^- \ [\mathsf{pc}] \vdash [\![\Gamma[\mathsf{pc}] \vdash e : s]\!] y$.

Proof: Straightforward. The necessary lattice inequalities to give a target derivation are established by simply following the source derivation. □

To show that this CPS translation is operationally correct, we would like to establish the following lemma. Giving a complete proof is beyond the scope of this thesis. However, the result should follow by a simulation argument similar to those used by Danvy and Filinski [DF92]. There are some subtleties with the simulation that arise because of the presence of mutable state. Because this is an unoptimizing version of CPS translation, it is also necessary to allow *administrative redexes* to remain in the resulting terms [DF92]. Doing so complicates the exact strengthening of the simulation relations, but it can be done [Plo75, Rie89].

Lemma 4.5.2 (Operational Correctness) *Suppose* e *is a* $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ *term such that* $\Gamma\ [\bot] \vdash e : s$ *and that* M *is a* $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$ *memory such that* $Loc(e) \subseteq |M|$. *Then*

$$\langle M,\, \bot,\, e\rangle \Downarrow \langle M', v\rangle \Leftrightarrow \langle [\![M]\!],\, \bot,\, [\![e]\!]\rangle \longmapsto^* \langle [\![M']\!],\, \bot,\, \mathtt{halt}^{s^+}\ [\![v]\!]\rangle$$

Proof (sketch): The proof goes by strengthening the induction hypothesis to handle the case where the linear continuation isn't known to be $stop(s^*)$, the program counter may be different from $\bot$, and the programs are related up to administrative redexes. Additional interesting points include:

1. Binary operations are compatible in both languages.

2. There is a caveat that the memory allocation must be somehow coherent—that is, the locations that are created "fresh" in the source can be mapped isomorphically onto those created "fresh" in the target.

3. It's important that the target can take multiple evaluation steps to simulate a source evaluation rule, because there can be multiple `lgoto` operations involved in threading the control through the target.

□

4.6 Related work

Pottier and Simonet [PS02] give a proof technique for proving noninterference in a security-typed version of core ML, which is similar to $\lambda^{\mathrm{REF}}_{\mathrm{SEC}}$. Their approach differs from the one presented here by using a nonstandard operational semantics in which two program evaluations can be represented simultaneously. Their construction reduces the noninterference result for the source language to the subject-reduction theorem for the nonstandard language.

The constraints imposed by linearity can be seen as a form of resource management [Gir87], in this case limiting the set of possible future computations.

There are many similarities between the stack-like nature of ordered linear continuations and other type systems for regulating resource consumption. For example, the capabilities calculus uses linearity constraints to manage regions of memory [CWM99]; the regions obey a stack discipline. Linearity has been widely used in the context of memory management [Abr93, CWM99, Wad90, Wad93].

Linear continuations have been studied in terms of their category-theoretic semantics [Fil92] and for their role in a computational interpretation of classical logic [Bie99]. Polakow and Pfenning have investigated the connections between ordered linear-logic, stack-based abstract machines, and CPS [PP00]. Berdine, et al., have also studied a number of situations in which continuations are used linearly [BORT01].

CPS translation has been studied in the context of program analysis [FB93, Nie82]. Sabry and Felleisen observed that increased precision in some CPS data flow analyses is due to duplication of analysis along different execution paths [SF94]. They also note that some analyses "confuse continuations" when applied to CPS programs. Our type system distinguishes linear from nonlinear continuations to avoid confusing "calls" with "returns." More recently, Damian and Danvy showed that CPS translation can improve binding-time analysis in the λ-calculus [DD00], suggesting that the connection between binding-time analysis and security [ABHR99] warrants more investigation.

Linear continuations appear to be a higher-order analog to *post-dominators* in a control-flow graph. Algorithms for determining post-dominators in control-flow graphs (see Muchnick's text [Muc97]) might yield inference techniques for linear continuation types. Conversely, linear continuations might yield a type-theoretic basis for correctness proofs of optimizations based on post-dominators.

Linearity also plays a role in security types for concurrent process calculi such as the π-calculus [HVY00, HY02]. Because the usual translation of the λ-calculus into the π-calculus can be seen as a form of CPS translation, it is enlightening to investigate the connections between security in process calculi and low-level code. These connections are made more explicit in the next chapter.

Chapter 5

Secure Concurrent Programs

Concurrent programs—programs in which multiple threads of control are permitted to evaluate simultaneously—are invaluable for building systems that must react to or control ongoing activities in their environments [Sch97]. Typical examples of concurrent programs include operating systems, databases, web servers, and user interfaces.

Clearly, there are information security concerns when designing such programs, yet the problem of checking information flow properties in concurrent programming languages has not been solved satisfactorily. This chapter considers this problem, and proposes a new language, called $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ whose type system and accompanying definition of information security avoid some of the limitations of previous work.

Languages for concurrent programming vary widely in the details of the features they provide, but they share some essential traits: support for multiple threads of execution and the ability to describe interthread communication. One key difference between concurrent languages and sequential languages is that they are *nondeterministic*: a single machine configuration m may transition to two different machine configurations m_1 and m_2, reflecting a choice of which of multiple threads of execution to run.

The nondeterminism in the operational semantics is important, because it allows the behavior of the thread scheduler to depend on details of run-time context that are not under the programmer's control: for instance, the operating system, the available resources, or the presence of other threads managed by the scheduler. However, the nondeterministic behavior of concurrent programs can also permit multiple threads to interact in unwanted (and often unpredictable) ways. Practical languages for concurrent programs provide synchronization mechanisms to help avoid this problem.

Concurrency adds several new difficulties for regulating information flows:

1. How nondeterminism is resolved in an implementation of the language plays an important role in the information flows of a concurrent program. An observer with knowledge of the scheduler implementation might be able to deduce more infor-

mation from the behavior of the program. These information leaks are sometimes called *refinement attacks* [Ros95].

2. Communication between concurrently running threads also creates information flows. More problematically, race conditions can arise that might leak information if the outcome of a race is influenced by high-security data. Such race conditions often involve *timing flows*, covert channels that have long been considered difficult to control [Lam73].

3. Synchronization itself exchanges information between threads and hence creates information flows. However, synchronization may also eliminate race conditions and other forms of resource contention—thereby preventing potentially insecure information flows.

Security-typed concurrent languages must address all of these issues. However, dealing with them in a way that permits useful programs and a tractable program analysis is challenging. Previous work has approached the problem by either using simple `while`-based programs with shared-memory thread communication [SV98, SV00, VS00, Smi01, Sab01] or by using the pi calculus [HY02, Pot02], a widely studied process calculus [MPW92]. The language $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ introduced in this chapter is based on Fournet's join calculus [Fou98], a process calculus whose syntax more closely resembles that of $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$. $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ differs from previous approaches in the way that it treats synchronization and timing.

Despite their importance for practical concurrent programming and their impact on information-flow properties, synchronization mechanisms have only recently been studied in the context of information security [Sab01, HY02, Pot02]. $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ incorporates a controlled form of synchronization that uses linearity constraints similar to those on the linear continuations of Chapter 4. Linear synchronization provides structure that allows the type system to more accurately describe information flows that arise due to synchronization. Although not complete—there are secure synchronization behaviors that are not expressible in this framework—$\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ provides the ability to write programs with useful synchronization behavior.

In contrast to many previous security-typed language approaches for concurrent programs [SV98, SV00, VS00, Smi01, Sab01, HY02, Pot02, BC02], the definition of information-flow presented in this chapter permits high-security information to affect the termination behavior and external timing behavior of the program. Instead, the proposal here controls information flows by eliminating certain race conditions between threads. This approach draws a distinction between the internally and externally observable timing behavior of programs. External observations are those made by an observer outside the system, timing the program with mechanisms external to the computing system. Internal observations are those made by other programs running on the same

system. In principle, internal observations are more easily controlled because other programs can be subjected to a validation process before being allowed to run. Typically, internal observations also offer greater potential for high-bandwidth information transfer, so they may also be more important to control. In this work the focus is on controlling information flows that are internal to the system. Because there are many external flows, some of which are difficult to control (e.g., conversation), and other techniques that can be applied to controlling them (e.g., auditing or dynamically padding total execution time), this decision seems reasonable.

The $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ security model is based on observational determinism. Suppose that we have two well-formed initial configurations m and m' that differ only in some high-security inputs ($m \approx_\zeta m'$), where evaluating m may result in a trace of machine states T and evaluating m' may result in a trace T'. We require $T \approx_\zeta T'$. Two traces are considered equivalent if they are equivalent up to stuttering and prefixing at every memory location, considered *independently* of other memory locations.

To be more precise, let M, a memory, be a finite map from locations L to values v. Then $M(L)$ is the contents of location L. Let $T(L)$ be the projection of the trace T onto a single memory location L; that is, if $T = [M_0, M_1, M_2, \ldots]$ then $T(L) = [M_0(L), M_1(L), M_2(L), \ldots]$. A sequence of values $[v_0, v_1, v_2, \ldots]$ is equivalent to another sequence of values $[v'_0, v'_1, v'_2, \ldots]$ if $v_i \approx_\zeta v'_i$ for all i up to the length of the shorter sequence. Then $T \approx_\zeta T'$ if for all locations L, $T(L)$ is equivalent up to stuttering to $T'(L)$, or vice versa.

This security condition allows high-security information to affect the external termination and timing behavior of a program, while preventing any effect on internally observable termination and timing behavior. Two aspects of this definition merit discussion. First, allowing one sequence to be a prefix of the other permits an *external* nontermination channel that leaks one bit of information, but removes the obligation to prove program termination. Second, considering each memory location independently is justified because internal observations of the two locations can only observe the relative ordering of their updates by using code that contains a race—and programs containing races are disallowed. By requiring only per-location ordering of memory operations, this definition avoids the restrictiveness incurred when timing may not depend on high-security data.

The next section discusses the issues of information-flow, concurrency, and synchronization more thoroughly, motivating the design of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ and informally introducing its syntax. Section 5.2 presents the concrete language, its type operational semantics and type system. Section 5.3 gives a proof of subject reduction. Section 5.4 formally defines noninterference and race freedom and proves that the $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ type system provides noninterference. This chapter concludes with further comparison to related work.

5.1 Thread communication, races, and synchronization

This section describes information flows that arise in concurrent languages and informally introduces the syntax for $\lambda_{\text{SEC}}^{\text{CONCUR}}$.

The syntax for running two processes concurrently is $P_1 \mid P_2$. A process[1] itself may consist of multiple concurrent subprocesses. For example, P_1 might be the process $Q_1 \mid Q_2$. The order in which the threads are written is unimportant: the program $P_1 \mid P_2$ is equivalent to the program $P_2 \mid P_1$. Similarly, the $\mid$ operator is associative, so the programs $(P_1 \mid P_2) \mid P_3$ and $P_1 \mid (P_2 \mid P_3)$ are equivalent.

The operational semantics for $\lambda_{\text{SEC}}^{\text{CONCUR}}$ allow any possible interleaving of the evaluations of two parallel processes. Informally[2], there are two rules for evaluating $P_1 \mid P_2$:

$$\frac{P_1 \rightarrow P_1'}{P_1 \mid P_2 \rightarrow P_1' \mid P_2} \qquad \frac{P_2 \rightarrow P_2'}{P_1 \mid P_2 \rightarrow P_1 \mid P_2'}$$

These two rules directly exhibit the nondeterminism of the operational semantics.

5.1.1 Shared memory and races

One basic means by which threads can communicate is *shared memory*. In shared memory models, one thread can write to a shared location and a second thread can read from it, thereby transferring information from the first thread to the second.

Due to concurrency, two threads might try to write to the same location simultaneously or one thread might attempt to write to a location concurrently with a read by a second thread, as illustrated below in examples (1) and (2) below:

```
(1)  (l := t) | (l := f)
(2)  (l := t) | (y := !l)
```

Both programs might behave nondeterministically. In example (1), the final value stored in location `l` depends on the order in which the two threads are executed. Similarly, in example (2), the final contents of `y` might depend on whether the read of location `l` is scheduled before the write.

These *write–write* and *write–read* races can be used to leak confidential information due to the relative timing behavior of the two threads. For example, the following program sets up a race to assign to location `l`. The amount of time it takes the first thread to reach the assignment depends on high-security data `h`. As an example, consider the following program that might leak information about `h` via a write–read race:

[1] The terms *thread* and *process* are used interchangeably.

[2] The actual operational semantics of $\lambda_{\text{SEC}}^{\text{CONCUR}}$ is given in Figures 5.4 and 5.5

(3)
```
x := t;
  (if h then delay(100) else skip; x := f)
| (delay(50); l := x;  ...)        l likely to equal h
```

This program initializes variable `x` to be true and then spawns two threads. The first thread assigns `x` the value false either immediately or after a long delay. The second thread waits long enough so that the assignment to `x` is likely to have occurred when `h` is false; this second thread then assigns `l` the value of `x`. Depending on the thread-scheduler implementation, this program can reliably copy the contents of `h` into the variable `l`—a potentially insecure information leak.

Termination behavior also plays a role in the information flows possible in concurrent programs. For example, the following program copies the value of `h` to `l`, assuming that `LOOP()` is a function that diverges:

(4)
```
  (if h then LOOP() else skip; l := f)
| (if (not h) then LOOP() else skip; l := t)
```

In this example, exactly one of the threads will terminate and only one assignment to `l` will take place. Some third thread that can read from `l` could poll its value to learn the contents of `h`. This example also depends on the assumption of a fair scheduler—an unfair scheduler could run the `LOOP()` code forever, never giving the other thread a chance to make its assignment.

Suppose that the variable `h` used in the examples contains high-security data and `l` is a low-security variable. Whether to consider these programs secure is dependent on what the low-security observer knows about the thread scheduler implementation. We have already observed that if the scheduler is *fair*, meaning that each thread is given eventually given a chance to run, then example (4) is not secure—it copies the contents of `h` into `l`.

Similarly, if it is known that the thread scheduler alternates between threads (starting with the first) and runs each for exactly 200 operational steps before switching, example (3) has deterministic behavior with respect to the variable `l`—by the time the second thread tests `x` it is always false. In this case, the program should be considered secure. In contrast, if the scheduler runs each thread for fewer than 100 steps before switching, example 3 does leak `h` to `l` and should be considered insecure.

As these examples have shown, whether a concurrent program is secure depends on what information about the thread scheduler is available to the low-level observer of the program. Examples (1) and (2) could be considered insecure if the low-level observer knows that the schedule chosen depends on high-security information—even though neither example mentions high-security data.

It might be reasonable to assume some characteristics about the thread scheduler, for instance, that it is fair, or that it schedules processes uniformly at random. However, it is

still difficult to rule out programs that are insecure due to races. The existing type systems that follow this approach lead to extremely restrictive models of programming. For example, Smith and Volpano consider only a fixed number of threads [SV98, Smi01]. Both their definition of noninterference and most other definitions [SS01, MS01, Sab01, BC02, HY02, Pot02, SM02] forbid low-security computation from depending on the timing or termination behavior that is influenced by high-security data. These approaches correctly rule out examples (3) and (4) as insecure while permitting programs (1) and (2). However, due to the restrictions on high-security data, those type systems also rule out the following single-threaded programs because the assignment to `l` sequentially follows computation whose *external* timing or termination behavior depends on high-security data.

(5) `while (h < 100) do { h := h + 1 }; l := 1;`
(6) `if h then LOOP() else skip; l := 1;`

Type systems that rule out these programs implicitly assume that there may be other unknown threads running concurrently with the program being analyzed and that those other threads have access to the memory locations used in the program in question. This makes the noninterference analysis very compositional, but quite restrictive—examples (5) and (6) are considered insecure.

A different approach is to observe that examples (1)–(4) share a simple common feature: there is a race on a low-security variable.[3] As we have seen, whether such races leak high-security information depends both on the threads that are running and on what the low-security observer knows about the thread scheduler. Therefore, a natural way to eliminate the information flows that arise from concurrency is to eliminate races to low-security memory locations. Such race freedom requires that the sequence of values stored in every memory location is deterministic. The definition of noninterference in concurrent languages used here is based on low-security observational determinism. A similar proposal has been made by Roscoe in the context of a labeled-transition semantics for CSP [Ros95]. One benefit of this approach is that, under the caveat that the variable `l` is local to the program being analyzed, examples (5) and (6) are considered secure. As with previous work, examples (3) and (4) are considered insecure. However, with this definition of noninterference examples (1) and (2) are *not* considered secure. Another benefit of this approach is that it yields a very strong kind of scheduler-independence: determinism of the low-security memory writes implies that the same

[3] Whether there is a race in example (4) is debatable because we know ahead of time that at most one assignment to `l` will ever take place. Because detecting races is, in general, undecidable (replace `h` with an undecidable predicate), any static analysis sophisticated enough to determine that there is no race in programs like this one could detect the information flow from `h` to `l` and hence rule this program out as insecure.

sequence of reads and writes will be seen regardless of what decisions are made by the thread scheduler.

We have seen that noninterference in the presence of concurrency relies crucially on the absence of races. Of course, eliminating race conditions can itself be difficult, particularly due to aliasing. For instance, we must reject the following program that has a write–write race on the memory location aliased by `l` and `l'`:

```
let l' = l in
  (l' := 1) | (l := 0)
```

The simplest way to rule out race conditions on shared memory is to eliminate shared memory altogether. A less severe, yet sufficient, approach is to restrict the memory shared between two threads to be read-only, thereby eliminating write–write and write–read races. This strategy does not prohibit writable memory locations; it requires that mutable state be local to a single thread.

Alias analysis plays a crucial role in establishing thread locality of memory resources, and hence whether a program is race free. However, developing such an analysis is beyond the scope of this thesis. Instead, we assume the existence of such an analysis for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ programs. The abstract requirements of a race-freedom analysis and some potential implementations in terms of alias analysis are described in Section 5.2.3.

Because we do not make any assumptions about the thread scheduler, the condition of race-freedom and the lack of synchronization together rule out interthread communication via shared memory. However, for concurrency to be useful, the threads still need some means of exchanging data. Therefore $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ allows a form of message passing to compensate for the lack of general shared memory. The message-passing constructs have a built-in synchronization mechanism that can be used to prevent race conditions and to encode secure forms of shared-memory communication.

5.1.2 Message passing

In message-passing models (see Schneider's text [Sch97] for an overview), one thread may *send* a message on a *channel*; a second thread may then *receive* the message from the channel. *Synchronous message passing* requires that the sending thread block until the receiving thread has received the message, and, symmetrically, the receiving thread must block until a message is sent.

Asynchronous message passing allows the sender of the message to continue before the message has necessarily been received; a symmetric construct allows the receiver to accept a message if one has been sent, or continue processing otherwise. $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ provides asynchronous message passing; synchronous message passing can be encoding using the synchronization mechanisms, as shown below.

The $\lambda_{\text{SEC}}^{\text{CONCUR}}$ notation for sending a value v on a channel `c` is `c(`v`)`. Messages may contain no data, in which case the send is written `c()`, or they may contain multiple values, in which case the send is written `c(`v_1`,...,`v_n`)`.

The way a $\lambda_{\text{SEC}}^{\text{CONCUR}}$ process reacts when it receives a message on a channel is described by a *message handler* (or simply *handler*). The syntax for a handler is illustrated below:

```
c(x) ▷ l := x
```

This handler waits for a message on channel `c`. When it receives such a message the handler creates a new thread that executes the program `l := `x with the contents of the message bound to the variable x. For example, when the handler receives the message `c(t)`, it creates a new thread that performs the assignment `l := t`. The message is consumed by the handler.

Handlers definitions are introduced via `let`-syntax, just as the continuations of $\lambda_{\text{SEC}}^{\text{CPS}}$ or the functions of $\lambda_{\text{SEC}}^{\text{REF}}$ are introduced. The channel name defined in the handler is in scope within the body of the let. As an example, a $\lambda_{\text{SEC}}^{\text{CONCUR}}$ program that sends the message as described above is:

```
let c(x) ▷ l := x in
  c(t)
```

In full generality, multiple threads might attempt to send messages on a channel concurrently. This situation is similar to the write–write race condition possible with shared memory; information implicit in the thread scheduler might affect the outcome of the communication. Such *send contention* occurs when multiple threads try to send concurrently on the same channel, as illustrated in this example:

```
let c(x) ▷ l := x in
  c(t) | c(f)
```

This example also shows how send contention can lead to a race condition. The handler above will react to the two messages by spawning a new thread to handle each; therefore this example effectively evaluates to this program, which clearly exhibits a race:

```
let c(x) ▷ l := x in
 (l := t) | (l := f)
```

Handlers like the ones above remain permanently in the program environment once declared. They are thus able to react to any number of messages sent on the channels they define. Note that lexical scoping of channel names makes it impossible to introduce *read contention*, which multiple handlers vie for a message. In the program below, the second handler definition for the channel `c` shadows the first:

```
let c(x) ▷ l := x in
let c(x) ▷ l := t in
  c(f)
```

$\lambda_{\text{SEC}}^{\text{CONCUR}}$ permits channels to be used in a first-class way. This means that they may be passed as values in the messages sent on other channels. For example, the channel `double` sends two (empty) messages on any channel it is given as an argument:

```
let double(c) ▷ c() | c() in
let d() ▷ P in
  double(d)
```

Just as recursive functions may mention the name of the function inside its body, handler definitions may mention the channels they define. This makes it possible for $\lambda_{\text{SEC}}^{\text{CONCUR}}$ handler definitions to encode recursive functions. For example, the following program creates an infinite loop:

```
let c(x) ▷ c(x) in
  c(t)
```

To further illustrate the connection between handlers and functions, compare the handler definition to a similar recursive function written in $\lambda_{\text{SEC}}^{\text{REF}}$:

$$\lambda[\bot]\ f\ (x : s).\ f\ x$$

The handler for a channel is like a continuation: both accept a value and then perform some computation based on that value. The ordered linear continuations of Chapter 4 imposes enough structure on sequential programs to establish a noninterference result. Similarly, $\lambda_{\text{SEC}}^{\text{CONCUR}}$ provides *linear channels*, on which exactly one message must be sent. The symbol $\multimap$ is used in place of $\triangleright$ to indicate that a message handler is linear.

For example, the following program declares a linear channel `k` that must be used exactly once along each execution path:

```
let k(x) —○ l := x in
  if h then k(t) else k(f)
```

The following programs are ill-formed, because they violate linearity of the channel `k`. The first uses `k` in two separate threads, the second discards `k` in the second branch of the conditional:

```
let k(x) —○ l := x in
  k(t) | k(f)
```

```
let k(x) —∘ l := x in
  if h then k(t) else (l := f)
```

Importantly, linear handlers can never be used to create nondeterminism because, in contrast to nonlinear handlers, there is never any send contention. This observation justifies a weaker program-counter label constraint for using linear channels in the $\lambda_{\text{SEC}}^{\text{CONCUR}}$ type system, just as the $\lambda_{\text{SEC}}^{\text{CPS}}$ rules for linear continuations introduce weaker constraints than the rules for nonlinear continuations.

Channels may be seen as generalizing the abstraction of memory locations. A memory location is a channel that accepts messages that are pairs of the form `(set,` v`)` or `(get,` c`)` where v is the value assigned by a `set` operation and c is itself the name of a channel on which to return the contents of the memory cell.

Channels fundamentally exhibit the same difficulties that shared references do, so why consider adding both to a concurrent language? Including both mechanisms provides a separation of concerns: references are used to structure the data of a computation, whereas channels (and message passing) are used to structure the behavior of a computation.

5.1.3 Synchronization

$\lambda_{\text{SEC}}^{\text{CONCUR}}$ provides a synchronization mechanism that allows a single handler definition to block waiting for messages on multiple channels.

A handler definition defines a *set* of channels that all share the same continuation. As a simple example, consider this handler definition:

```
input₁(x) | input₂(y) ▷ let z = x ∧ y in output(z)
```

It declares two channel names $\texttt{input}_1$ and $\texttt{input}_2$ that block, each waiting to receive a message. After both messages have been received, the handler triggers. In this case, the data in the message received on $\texttt{input}_1$ is bound to the variable x and the data in the message received on $\texttt{input}_2$ is bound to the variable y. The body of this handler computes $x \wedge y$ and then sends the result on a third channel, `output`.

As another example, the handler definition below declares two channels `c` and `d`. Channel `c` accepts messages containing a single value and channel `d` accepts messages that contain no data:

```
let c(x) | d() ▷ P(x) in Q
```

If the process Q sends messages on both channels `c` and `d`, the handler will react by creating a new thread P in which the value sent on `c` is bound to the variable x. If Q never sends a message on one of the two channels, the handler will never be triggered.

For example, the following program sends memory location `l` on channel `c`, but only sends a message on channel `d` if the value of h is true:

```
let l = ref f in
let c(x) | d() ▷ x := t in
  c(l) | (if h then d() else 0)
```

This construct provides synchronous message passing between two threads. Suppose thread P_1 wants to send the message `m` synchronously to thread Q_1 after which it continues as P_2. Thread Q_1 blocks waiting for the message `m` and then continues as thread $Q_2(\mathtt{m})$. In a traditional message-passing notation, this situation might be expressed by the following program:

$$\mathtt{cobegin}\ (P_1;\mathtt{send}_c(\mathtt{m});P_2)\ |\ (Q_1;\mathtt{recv}_c(x);Q_2(x))\ \mathtt{coend}$$

Here, $\mathtt{send}_c(\mathtt{m})$ sends the message `m` on channel `c`, blocking until the message has been received. Symmetrically, $\mathtt{recv}_c(x)$ blocks until channel `c` has been sent the message `m`, which it binds to variable x.

Using the syntactic sugar $P_1; P_2$ to denote sequential composition of (sequential) processes P_1 and P_2, this example can be written using $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$'s synchronization mechanisms:

$$\begin{array}{l}\mathtt{let}\ \mathtt{send}_c(x)\ |\ \mathtt{recv}_c()\ \triangleright\ P_2\ |\ Q_2(x)\\ \mathtt{in}\\ \quad P_1;\mathtt{send}_c(\mathtt{m})\ |\ Q_1;\mathtt{recv}_c()\end{array}$$

$\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ also allows synchronization on linear channels. For example, the program below declares a handler for two linear channels $\mathtt{k}_0$ and $\mathtt{k}_1$:

$$\mathtt{let}\ \mathtt{k}_0()\ |\ \mathtt{k}_1(x)\ \multimap\ P\ \mathtt{in}\ Q$$

As with the linear continuations of $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$, channels $\mathtt{k}_0$ and $\mathtt{k}_1$ must be used exactly once in each possible execution path of the process Q.

The channel synchronization mechanism in $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ provides a flexible way of structuring inter-thread communication.

The left half of Figure 5.1 illustrates a nested synchronization structure possible for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ programs. The wavy arrows in this picture denote sequential threads of execution. Forks in the computation are indicated by two solid arrows leaving a thread; synchronization between threads is indicated by two solid arrows joining together—the lines are labeled with the (linear) channels on which the synchronization takes place . The corresponding program is:

$$\begin{array}{l}\mathtt{let}\ \mathtt{k}_0()\ |\ \mathtt{k}_1()\ \multimap\ S\ \mathtt{in}\\ \quad P;(\ Q_1;(\mathtt{let}\ \mathtt{k}_2()\ |\ \mathtt{k}_3()\ \multimap\ \mathtt{k}_1()\ \mathtt{in}\\ \qquad\qquad (R_1;\ \mathtt{k}_3()\ |\ R_2;\ \mathtt{k}_2()))\\ \qquad |\ Q_2;\ \mathtt{k}_0()\ )\end{array}$$

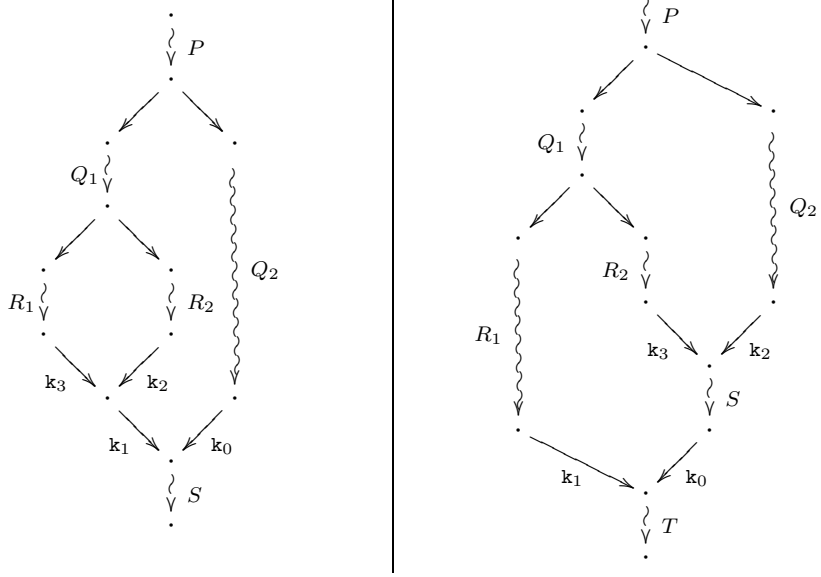


Figure 5.1: Synchronization structures

More complex synchronization behavior is also possible. For instance, the following program has the synchronization structure pictured in the right part of Figure 5.1.

```
let k0() | k1() —∘ T in
let k2() | k3() —∘ S; k0() in
  P;( Q1;( R1; k1()
         | R2; k3())
    | Q2; k2())
```

Note that this program uses channel `k0` inside the body of the handler defining channels `k2` and `k3`.

5.2 $\lambda_{\text{SEC}}^{\text{CONCUR}}$: a secure concurrent calculus

This section introduces the formal semantics for $\lambda_{\text{SEC}}^{\text{CONCUR}}$, including its syntax, operational semantics. and type system.

5.2.1 Syntax and operational semantics

Figure 5.2 shows the syntax for $\lambda_{\text{SEC}}^{\text{CONCUR}}$ programs.

Base values in $\lambda_{\text{SEC}}^{\text{CONCUR}}$ are channel names c, memory locations L, or Booleans `t` and `f`. The metavariable f ranges over channels and variables that have channel or linear channel type. Primitive operations are defined exactly as in $\lambda_{\text{SEC}}^{\text{CPS}}$.

A *process*[4], P consists of a sequence of `let`-declarations and primitive operations followed by either `0`, the terminal process, an `if` expression, or the concurrent composition of two processes, written P_1 `|` P_2. The terminal process `0` is analogous to the `halt` instruction of $\lambda_{\text{SEC}}^{\text{CPS}}$, except that it does not "return" any final output.

If `c` is a channel, then the syntax `c`$(\vec{v})$ denotes a message with contents $(\vec{v})$ sent on channel `c`. Message sends are asynchronous; but message handlers are blocking.

A *join pattern* $J = f_1(\vec{x_1})$ `|` $\ldots$ `|` $f_n(\vec{x_n})$ is a generalization of the binding construct $\lambda f(x)$ found in $\lambda_{\text{SEC}}^{\text{CPS}}$[5]. A join pattern declares a set of channels (or channel variables) $f_1 \ldots f_n$. Each channel f_i accepts a vector of arguments that will be bound to the vector variables $\vec{x_i}$.

There are two kinds of join patterns. *Nonlinear* join patterns may bind linear variables y (although they are not required to), and thus can include channel declarations

[4]The words *thread* and *process* are used interchangeably Because $\lambda_{\text{SEC}}^{\text{CONCUR}}$ does not have explicit thread (or process) identifiers, the concept is somewhat nebulous.

[5]The word "join" here comes from "fork" and "join" terminology of multithreading, not the "join" of the security lattice.

x, f	$\in$	$\mathcal{V}$	Variable names
bv	$::=$	c	Channel value
	$\mid$	L	Reference value
	$\mid$	$\texttt{t} \mid \texttt{f}$	Boolean values
v	$::=$	x	Variables
	$\mid$	bv_ℓ	Secure values
lv	$::=$	$y \mid c$	Linear values
$prim$	$::=$	v	Values
	$\mid$	$v \oplus v$	Boolean operations
	$\mid$	$!v$	Dereference
f	$::=$	$x \mid y \mid c$	Variables or channels
J	$::=$	$f(\vec{x}, y)$	Nonlinear channel
	$\mid$	$f(\vec{x})$	Linear channel
	$\mid$	$J \mid J$	Join pattern
P	$::=$	$\texttt{let}\ x = prim\ \texttt{in}\ P$	Primitive operation
	$\mid$	$\texttt{let}\ x = \texttt{ref}\ v\ \texttt{in}\ P$	Reference creation
	$\mid$	$\texttt{set}\ v := v\ \texttt{in}\ P$	Assignment
	$\mid$	$\texttt{let}\ J \triangleright P\ \texttt{in}\ P$	Handler definition
	$\mid$	$\texttt{let}\ J \multimap P\ \texttt{in}\ P$	Linear handler definition
	$\mid$	$v\,(\vec{v}, lv^{\mathsf{opt}})$	Message send
	$\mid$	$lv\,(\vec{v})$	Linear message send
	$\mid$	$\texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ P$	Conditional
	$\mid$	$(P \mid P)$	Parallel processes
	$\mid$	$\texttt{0}$	Inert process

Figure 5.2: Process syntax

of the form $f(\vec{x}, y)$. *Linear* join patterns never bind linear variables—they contain only channel declarations of the form $f(\vec{x})$.

The restriction that linear channels are not permitted to carry other linear channels prevents sequencing problems like the ones encountered for continuations in $\lambda_{\text{SEC}}^{\text{CPS}}$. For example, the $\lambda_{\text{SEC}}^{\text{CPS}}$ program `(D)` of Figure 4.1 can be encoded straightforwardly into $\lambda_{\text{SEC}}^{\text{CONCUR}}$. Join patterns introduce another means by which deterministic ordering of linear continuations can be violated. For example, if linear channels were permitted to carry other linear channels, the following program would contain an information flow from `h` to `l` because the two assignments to `l` occur in different orders depending on the value of `h`.

```
letlin k0() —∘ 0 in
letlin k1(y1) —∘ set l := t in y1() in
letlin k2(y2) —∘ set l := f in y2() in
letlin k3(y3) | k4(y4) —∘
   (letlin k() —∘ y3(k0) in y4(k))
in
  if h then k3(k1) | k4(k2)
       else k3(k2) | k4(k1)
```

Although it would be possible to generalize the ordering constraints on the linear continuations of $\lambda_{\text{SEC}}^{\text{CPS}}$ to the linear channels of $\lambda_{\text{SEC}}^{\text{CONCUR}}$, doing so complicates the type system substantially. Instead, the race-freedom requirement implies an appropriate ordering on the usage of linear channels.

As described informally in the previous section, a *handler definition* is the $\lambda_{\text{SEC}}^{\text{CONCUR}}$ generalization of a continuation. Formally, it consists of a join pattern and a process P called the *body* of the handler. The syntax for nonlinear handlers is $J \rhd P$. The syntax for linear handlers is $J \multimap P$.

Just like nonlinear continuations, nonlinear channels may be duplicated and freely used; however to prevent race conditions from arising, nonlinear handlers must be reentrant in a sense described below. Each linear channel must be used exactly once in all possible future paths of computation.

It is helpful for writing examples to define a sequential subset of $\lambda_{\text{SEC}}^{\text{CONCUR}}$.

Definition 5.2.1 (Sequential processes) *A $\lambda_{\text{SEC}}^{\text{CONCUR}}$ process P is* **sequential** *if it does not contain the* `|` *symbol.*

If $P(y)$ is a sequential process that contains one free linear channel variable y, the process P; Q is defined as: `let` $y()$ $\multimap$ Q `in` $P(y)$.

Note that if P and Q are both sequential processes, then P; Q is also a sequential process. Also observe that the sequential sublanguage of $\lambda_{\text{SEC}}^{\text{CONCUR}}$ is the same (modulo syntax) as the language $\lambda_{\text{SEC}}^{\text{CPS}}$.

M	$::=$	$M[L \mapsto v]$	Memory location L storing v
	$\mid$	$M[\mathsf{pc} : J \rhd P]$	Message handler
	$\mid$	$\cdot$	
S	$::=$	$S[\mathsf{pc} : J \multimap P]$	Linear handler
	$\mid$	$\cdot$	
N	$::=$	$\cdot \mid N \mid [\mathsf{pc} : P]$	Network
m	$=$	$\langle M,\ S,\ N \rangle$	Machine configuration

Figure 5.3: Dynamic state syntax

Figure 5.3 shows the syntax for $\lambda_{\text{SEC}}^{\text{CONCUR}}$ memories, synchronization environments, networks and machine configurations. These structures make up the dynamic state of a $\lambda_{\text{SEC}}^{\text{CONCUR}}$ program.

Unlike $\lambda_{\text{SEC}}^{\text{CPS}}$, the memories of $\lambda_{\text{SEC}}^{\text{CONCUR}}$ contain channel handler definitions in addition to the ordinary mapping between locations and their values. The new syntax for a memory M is the binding $[\mathsf{pc} : J \rhd P]$. We generalize the domain of a memory M to include the join patterns J it provides definitions to.

The memory M of the $\lambda_{\text{SEC}}^{\text{CONCUR}}$ abstract machine consists of a collection of bindings of the form $[L \mapsto v]$ and $[\mathsf{pc} : J \rhd P]$, as shown in Figure 5.2. The domain of M, written $\text{dom}(M)$, is the set of locations L and join patterns J that appear in M. If $L \in \text{dom}(M)$ then we write $M(L)$ for the value v such that $[L \mapsto v] \in M$. Similarly, if $J \in \text{dom}(M)$, we write $M(J)$ for the (open) process P.

A *synchronization environment*, denoted by S, stores the linear handler definitions that have been declared by the program. Notation similar to that of memories is used to describe the domain of a synchronization environment.

In order to track the information flows through a concurrent program, each thread must have its own pc label. $\lambda_{\text{SEC}}^{\text{CONCUR}}$ associates a process P with its pc label using the syntax $[\mathsf{pc} : P]$. A collection of such threads running concurrently is called a *network*. This terminology is chosen in anticipation of the developments of Chapter 7, where threads may be running on distinct hosts—for $\lambda_{\text{SEC}}^{\text{CONCUR}}$, a network can be thought of as a pool of threads running on a single host. The syntax for a network N is shown near the bottom of Figure 5.2.

In $\lambda_{\text{SEC}}^{\text{CONCUR}}$, a *machine configuration* m is a triple $\langle M,\ S,\ N \rangle$, where M is a memory, S is a synchronization environment, and N is a network.

$\boxed{M, \mathsf{pc} \models prim \Downarrow v}$ $\boxed{\langle M_1,\, S_1,\, N_1\rangle \to \langle M_2,\, S_2,\, N_2\rangle}$

$\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-PRIM $\quad M,\ \mathsf{pc} \models v \Downarrow v \sqcup \mathsf{pc}$

$\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-BINOP $\quad M,\ \mathsf{pc} \models n_\ell \oplus n'_{\ell'} \Downarrow (n[\![\oplus]\!]n')_{\ell \sqcup \ell'} \sqcup \mathsf{pc}$

$\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-DEREF $\quad \dfrac{M(L) = v}{M,\ \mathsf{pc} \models !L \Downarrow v \sqcup \mathsf{pc}}$

$\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-LETPRIM

$$\frac{M,\ \mathsf{pc} \models prim \Downarrow v}{\langle M,\, S,\, (N \mid [\mathsf{pc} : \mathtt{let}\ x = prim\ \mathtt{in}\ e])\rangle \to \langle M,\, S,\, (N \mid [\mathsf{pc} : e\{v/x\}])\rangle}$$

$\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-LETREF

$$\begin{array}{lll} & \langle M,\, S,\, (N \mid [\mathsf{pc} : \mathtt{let}\ x = \mathtt{ref}\ v\ \mathtt{in}\ P])\rangle & \\ \to & \langle M[L \mapsto v],\, S,\, (N \mid [\mathsf{pc} : P\{L_{\mathsf{pc}}/x\}])\rangle & (L \text{ fresh}) \end{array}$$

$\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-SET

$$\begin{array}{ll} & \langle M,\, S,\, (N \mid [\mathsf{pc} : \mathtt{set}\ L\ @\ h := v'\ \mathtt{in}\ P])\rangle \\ \to & \langle M[L \mapsto v' \sqcup \ell \sqcup \mathsf{pc}],\, S,\, (N \mid [\mathsf{pc} : P])\rangle \end{array}$$

Figure 5.4: $\lambda^{\text{CONCUR}}_{\text{SEC}}$ operational semantics

Figures 5.4 and 5.5 contain the operational semantics for $\lambda^{\text{CONCUR}}_{\text{SEC}}$. The rules define a transition relation $m_1 \to m_2$ between machine configurations. Evaluation of primitive operations, reference creation, assignment, and conditionals is essentially the same as in $\lambda^{\text{CPS}}_{\text{SEC}}$; such evaluation is defined by the relation $M, \mathsf{pc} \models prim \Downarrow v$. The description below concentrates on the new features of $\lambda^{\text{CONCUR}}_{\text{SEC}}$, shown in Figure 5.5.

The rules $\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-HANDLER and $\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-LINHANDLER allocate fresh channel names for each variable f_i in the join pattern of the handler. The channels declared in nonlinear handlers may be used recursively (inside the body of the handler), so the fresh channel names are substituted in the body. Nonlinear handlers are placed in the memory of the machine configuration. Linear handlers are put into the synchronization environment. Both kinds of handlers capture the pc label of the introduction context and record it in the machine state.

Rule $\lambda^{\text{CONCUR}}_{\text{SEC}}$-EVAL-SEND describes how a nonlinear handler is invoked. It uses the syntactic abbreviation $\mid_i P_i \stackrel{\text{def}}{=} \mathtt{0} \mid P_1 \mid \ldots \mid P_n$. Suppose the handler

$$[\mathsf{pc} : c_1(\vec{x_1}) \mid \ldots \mid c_n(\vec{x_n}) \rhd P]$$

$\boxed{\langle M_1,\, S_1,\, N_1\rangle \to \langle M_2,\, S_2,\, N_2\rangle}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-HANDLER

$\langle M,\, S,\, (N \mid [\mathsf{pc} : \texttt{let}\ f_1(\vec{x}_1) \mid \ \ldots\ \mid f_n(\vec{x}_n) \rhd P_1\ \texttt{in}\ P_2])\rangle$
$\to\quad \langle M[c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \rhd P_1\{(c_i)_{\mathsf{pc}}/f_i\}],\, S,\, (N \mid [\mathsf{pc} : P_2\{(c_i)_{\mathsf{pc}}/f_i\}])\rangle$
where the c_i are fresh

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LINHANDLER

$\langle M,\, S,\, (N \mid [\mathsf{pc} : \texttt{let}\ f_1(\vec{x}_1) \mid \ \ldots\ \mid f_n(\vec{x}_n) \multimap P_1\ \texttt{in}\ P_2])\rangle$
$\to\quad \langle M,\, S[\mathsf{pc} : c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \multimap P_1],\, (N \mid [\mathsf{pc} : P_2\{c_i/f_i\}])\rangle$
where the c_i are fresh

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-COND1

$\langle M,\, S,\, (N \mid [\mathsf{pc} : \texttt{if}\ \mathtt{t}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2])\rangle$
$\to\quad \langle M,\, S,\, (N \mid [\mathsf{pc} \sqcup \ell : P_1])\rangle$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-COND2

$\langle M,\, S,\, (N \mid [\mathsf{pc} : \texttt{if}\ \mathtt{f}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2])\rangle$
$\to\quad \langle M,\, S,\, (N \mid [\mathsf{pc} \sqcup \ell : P_2])\rangle$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-SEND

$\langle M[c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \rhd P],\, S,\, (N \mid_i [\mathsf{pc}_i : c_{i\ell_i}(\vec{v}_i, lv_i^{\mathsf{opt}})])\rangle$
$\to\quad \langle M[c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \rhd P],\, S,\, (N \mid [\ell : P\{\vec{v}_i \sqcup \mathsf{pc}_i/\vec{x}_i\}\{lv_i/y_i\}^{\mathsf{opt}}])\rangle$
where $\ell = \bigsqcup_i \mathsf{pc}_i \sqcup \ell_i$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LINSEND

$\langle M,\, S[\mathsf{pc} : c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \multimap P],\, (N \mid_i [\mathsf{pc}_i : c_i(\vec{v}_i)])\rangle$
$\to\quad \langle M,\, S,\, (N \mid [\mathsf{pc} : P\{\vec{v}_i \sqcup \mathsf{pc}_i/\vec{x}_i\}])\rangle$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-FORK

$\langle M,\, S,\, (N \mid [\mathsf{pc} : P \mid Q])\rangle$
$\to\quad \langle M,\, S,\, (N \mid [\mathsf{pc} : P] \mid [\mathsf{pc} : Q])\rangle$

Figure 5.5: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ operational semantics (continued)

$$
\begin{array}{lrcl}
\textsc{ProcUnit} & P \mid \mathtt{0} & \equiv & P \\
\textsc{ProcComm} & P_1 \mid P_2 & \equiv & P_2 \mid P_1 \\
\textsc{ProcAssoc} & (P_1 \mid P_2) \mid P_3 & \equiv & P_1 \mid (P_2 \mid P_3)
\end{array}
$$

Figure 5.6: Process structural equivalence

is in the memory. If there are messages $c_i(\vec{v_i})$ waiting at each of the channels c_i, the handler triggers, causing process P to execute with the message contents substituted for the formal parameters. The program counter label of the new process is the join of the pc labels that were present when the messages were sent. Joining the pc's prevents implicit flows due to synchronization. Importantly, the nonlinear handler remains in the memory after being triggered—it can be invoked many times (or never).

The rule for invoking linear handlers, $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LINSEND, is similar to the rule for nonlinear handlers, except for two differences. First, the program counter label of the new process is the same as the one when the handler was declared. This rule is analogous to one used for linear continuations in $\lambda_{\text{SEC}}^{\text{CPS}}$—see rule $\lambda_{\text{SEC}}^{\text{CPS}}$-EVAL-LGOTO of Figure 4.3. Second, the linear handler is removed from the synchronization environment after it is used, so no further invocations of it are possible.

The last rule, $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-FORK, says that a forked process inherits the pc label of the point at which it was spawned.

The operational semantics in the figure are too rigid: they require the right-most processes to take part in the computational step. Because thread scheduling should ignore the syntactic ordering of networks and processes running concurrently, we introduce the notion of structural equivalence. The process *structural equivalence* relation says that the syntactic order of concurrent processes is irrelevant. Network structural equivalence says that the syntactic order of the processes in a network is irrelevant and that there is no distinction between the syntax for a halted process, 0, and an empty network.

Structural equivalence is an instance of a *congruence* relation on program terms. Congruence relations respect the structure of the syntax of the language—if two terms are congruent, then placing them in identical program contexts yields two congruent terms.

Let Proc be the set of all process terms. Let Net be the set of all network terms.

Definition 5.2.2 (Network congruence) *A relation $\mathcal{R} \subseteq \mathsf{Net} \times \mathsf{Net}$ is a* **congruence on networks** *if it contains the $=_\alpha$ relation, and, furthermore, $N_1 \mathcal{R} N_2$ implies that for any $N \in \mathsf{Net}$ it is the case that*

$$(N_1 \mid N)\mathcal{R}(N_2 \mid N) \qquad \textit{and} \qquad (N \mid N_1)\mathcal{R}(N \mid N_2)$$

NETUNIT	$N \mid \cdot$	$\equiv$	N	
NETCOMM	$N_1 \mid N_2$	$\equiv$	$N_2 \mid N_1$	
NETASSOC	$(N_1 \mid N_2) \mid N_3$	$\equiv$	$N_1 \mid (N_2 \mid N_3)$	
NETPROC	$[\mathsf{pc} : P_1]$	$\equiv$	$[\mathsf{pc} : P_2]$	(if $P_1 \equiv P_2$)
NETZERO	$[\mathsf{pc} : 0]$	$\equiv$	$\cdot$	

Figure 5.7: Network structural equivalence

Definition 5.2.3 (Process congruence) *Let $t[\cdot]$ be a term with a single 'hole' in it obtained from the grammar in Figure 5.2 by extending the set of processes to include $P ::= \ldots \mid [\cdot]$. For any process P, let $t[P]$ be the ordinary process term obtained from $t[\cdot]$ by filling the single instance of a hole in $t[\cdot]$ with the process P. A relation $\mathcal{R} \subseteq \mathsf{Proc} \times \mathsf{Proc}$ is a* **congruence on processes** *if it contains the $=_\alpha$ relation, and, furthermore, $P_1 \mathcal{R} P_2$ implies that for any $t[\cdot]$ it is the case that $(t[P_1])\mathcal{R}(t[P_2])$.*

Definition 5.2.4 (Structural equivalence) *Structural equivalence on processes, written $\equiv$, is the least symmetric, transitive process congruence satisfying the axioms given in Figure 5.6. Structural equivalence on networks, also written $\equiv$, is the least symmetric, transitive network congruence satisfying the axioms given in Figure 5.7.*

The structural equivalence on networks extends to a structural equivalence on machine configurations. In addition, we allow machine configurations in which the names of locations or channels are systematically changed to be considered equivalent.

Definition 5.2.5 (Configuration structural equivalence) *Two machine configurations $\langle M_1,\, S_1,\, N_1\rangle$ and $\langle M_2,\, S_2,\, N_2\rangle$ are structurally equivalent, written $\langle M_1,\, S_1,\, N_1\rangle \equiv \langle M_2,\, S_2,\, N_2\rangle$, if they are α-equivalent (where locations and channel definitions in a memory or synchronization environment are considered binding occurrences of locations and channels) and $N_1 \equiv N_2$.*

The syntax-independent operational semantics is given by the transition relation $\Rrightarrow$ defined from the $\rightarrow$ relation and structural equivalence as shown in the following rule.

$$\lambda^{\text{CONCUR}}_{\text{SEC}}\text{-EQSTEP} \qquad \frac{\begin{array}{l}\langle M_1,\, S_1,\, N_1\rangle \equiv \langle M_1',\, S_1',\, N_1'\rangle \\ \langle M_1',\, S_1',\, N_1'\rangle \rightarrow \langle M_2',\, S_2',\, N_2'\rangle \\ \langle M_2',\, S_2',\, N_2'\rangle \equiv \langle M_2,\, S_2,\, N_2\rangle\end{array}}{\langle M_1,\, S_1,\, N_1\rangle \Rrightarrow \langle M_2,\, S_2,\, N_2\rangle}$$

5.2.2 $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ type system

Figure 5.8 shows the types for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ programs. As with $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$, types are divided into nonlinear security types and linear types. Base types, t, consist of Booleans, channel types, and references.

The channel type $[\mathsf{pc}](\vec{s}, k^{\mathsf{opt}})$ has any number of nonlinear arguments and at most one linear argument. The $[\mathsf{pc}]$ component of a channel type is, as in $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$, a lower bound on the security level of memory locations that might be written to if a message is sent on this channel. Note that the channel type $[\mathsf{pc}](s, k)$ corresponds precisely to the nonlinear continuation type $[\mathsf{pc}](s, k) \to \mathsf{0}$ used in $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$.

The linear types are channels $(\vec{s})$ that accept some number of nonlinear arguments. Sending a message on a linear channel does not itself reveal information about the sending context (although the message contents might), so linear channel types do not require the $[\mathsf{pc}]$ component. The linear channel type (s) corresponds precisely to the linear continuation type $(s) \to \mathsf{0}$. The security lattice is lifted to a subtyping relation on $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ types, as shown in Figure 5.9.

A type context Γ is a finite map from nonlinear variables to their types. If Γ_1 and Γ_2 are type contexts, the notation Γ_1, Γ_2 forms there disjoint union: $\Gamma_1, \Gamma_2 = \Gamma_1 \cup \Gamma_2$ whenever $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. Linear type contexts K are finite maps from linear variables to linear types. Disjoint union of linear contexts K_1, K_2 is defined similarly to the case for nonlinear type contexts.

A memory type, H (for *heap*), is a mapping from locations and channels to their types. Memory types were implicit in $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ because each location value was tagged with its associated type. For $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$, because memories also store handler definitions, it is syntactically less cumbersome to factor the type information for locations and channels into these explicit memory types. A synchronization state type T similarly maps linear channels to their linear types.

The type system for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ is shown in Figure 5.13. These judgments make use of auxiliary judgments that ensure values and linear values are well-typed (Figures 5.10 and 5.11), primitive operations are well-typed (Figure 5.12), and that memories and synchronization environments are well-formed (Figures 5.16 and 5.17).

The type system is designed to guarantee the following properties:

1. Explicit and implicit insecure information flows are ruled out.

2. Channel names introduced in a linear handler are used linearly.

$\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ typing judgments have the form $H; \Gamma; T; K\ [\mathsf{pc}] \vdash P$. This judgment asserts that process P is well-typed. The contexts H and Γ map locations, nonlinear channels, and nonlinear variables to their types as described above. T is a linear context that maps linear channels to their types and K maps linear variables to their types. Unlike $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ the linear contexts are unordered—the necessary ordering on linear channels

pc, ℓ	$\in$	$\mathcal{L}$	Security labels
s	$::=$	t_ℓ	Security types
t	$::=$	$\texttt{bool}$	Booleans
	$\mid$	$[\mathsf{pc}](\vec{s}, k^{\mathsf{opt}})$	Channel types
	$\mid$	$s\ \texttt{ref}$	Reference types
k	$::=$	$(\vec{s})$	Linear channel types
Γ	$::=$	$\cdot \mid \Gamma, x : s$	Type contexts
H	$::=$	$\cdot$	Empty memory type
	$\mid$	$H, [L : s]$	Location type
	$\mid$	$H, [c : s]$	Channel definition type
K	$::=$	$\cdot \mid K, y : k$	Linear type contexts
T	$::=$	$\cdot \mid T, c : k$	Synchronization state types

Figure 5.8: Process types

$\boxed{\vdash t_1 \leq t_2}$ $\boxed{\vdash s_1 \leq s_2}$ $\boxed{\vdash k_1 \leq k_2}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-TREFL $\quad \vdash t \leq t$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-TTRANS $\quad \dfrac{\vdash t \leq t' \quad \vdash t' \leq t''}{\vdash t \leq t''}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-TCHANSUB $\quad \dfrac{\mathsf{pc}' \sqsubseteq \mathsf{pc} \quad \vdash s_i' \leq s_i \quad (\vdash \kappa' \leq \kappa)^{\mathsf{opt}}}{\vdash [\mathsf{pc}](\vec{s}, \kappa^{\mathsf{opt}}) \quad \leq \quad [\mathsf{pc}'](\vec{s'}, \kappa'^{\mathsf{opt}})}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-SLAB $\quad \dfrac{\vdash t \leq t' \quad \ell \sqsubseteq \ell'}{\vdash t_\ell \leq t'_{\ell'}}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-TLINSUB $\quad \dfrac{\vdash s_i' \leq s_i}{\vdash (\vec{s}) \leq (\vec{s'})}$

Figure 5.9: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ subtyping

$\boxed{H; \Gamma \vdash v : s}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-VAR $\quad H; \Gamma \vdash x : \Gamma(x)$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-TRUE $\quad H; \Gamma \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-FALSE $\quad H; \Gamma \vdash \mathtt{f}_\ell : \mathtt{bool}_\ell$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-LOC $\quad H; \Gamma \vdash L_\ell : H(L) \sqcup \ell$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-CHAN $\quad H; \Gamma \vdash c_\ell : H(c) \sqcup \ell$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-SUB $\quad \dfrac{\vdash s_1 \leq s_2 \quad H; \Gamma \vdash v : s_1}{H; \Gamma \vdash v : s_2}$

Figure 5.10: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ value typing

$\boxed{T; K \vdash lv : k}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-LINVAR $\quad \cdot; y{:}k \vdash y : k$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-LINCHAN $\quad c{:}k; \cdot \vdash c : k$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-LINSUB $\quad \dfrac{\vdash k_1 \leq k_2 \quad T; K \vdash lv : k_1}{T; K \vdash lv : k_2}$

Figure 5.11: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ linear value types

$\boxed{H; \Gamma\ [\mathsf{pc}] \vdash prim : s}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-VAL $\quad \dfrac{H; \Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{H; \Gamma\ [\mathsf{pc}] \vdash v : s}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-BINOP $\quad \dfrac{H; \Gamma \vdash v : \mathtt{bool}_\ell \quad H; \Gamma \vdash v' : \mathtt{bool}_\ell \quad \mathsf{pc} \sqsubseteq \ell}{H; \Gamma\ [\mathsf{pc}] \vdash v \oplus v' : \mathtt{bool}_\ell}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-DEREF $\quad \dfrac{H; \Gamma \vdash v : s\ \mathtt{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s \sqcup \ell)}{H; \Gamma\ [\mathsf{pc}] \vdash !v : s \sqcup \ell}$

Figure 5.12: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ primitive operation types

$\boxed{H; \Gamma; T; K\ [\mathsf{pc}] \vdash P}$

$$\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}\text{-}\textsc{Prim} \quad \frac{H; \Gamma\ [\mathsf{pc}]\ \vdash\ prim\ :\ s \qquad H; \Gamma, x{:}s; T; K\ [\mathsf{pc}] \vdash P}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \mathtt{let}\ x = prim\ \mathtt{in}\ P}$$

$$\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}\text{-}\textsc{Ref} \quad \frac{H; \Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s) \qquad H; \Gamma, x{:}s\ \mathtt{ref}_{\mathsf{pc}}; T; K\ [\mathsf{pc}] \vdash P}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{ref} v\ \mathtt{in}\ P}$$

$$\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}\text{-}\textsc{Assn} \quad \frac{H; \Gamma \vdash v : s\ \mathtt{ref}_{\ell} \quad H; \Gamma; T; K\ [\mathsf{pc}] \vdash P \qquad H; \Gamma \vdash v' : s \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(s)}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \mathtt{set}\ v := v'\ \mathtt{in}\ P}$$

$$\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}\text{-}\textsc{If} \quad \frac{H; \Gamma\ [\mathsf{pc}] \vdash v : \mathtt{bool}_{\ell} \qquad H; \Gamma; T; K\ [\mathsf{pc} \sqcup \ell] \vdash P_i \quad (i \in \{1, 2\})}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \mathtt{if}\ v\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2}$$

$$\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}\text{-}\textsc{Zero} \quad H; \Gamma; \cdot, \cdot\ [\mathsf{pc}] \vdash \mathtt{0}$$

Figure 5.13: Process typing

is induced by the race-freedom assumption. As in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$, nonlinear contexts Γ permit weakening and contraction, whereas linear contexts K do not.

The type system, like that of $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$, uses the pc label to approximate the information that can learned by seeing that the program execution has reached a particular point. The $[\mathsf{pc}]$ component is the program counter security label that is a lower bound on the label of memory locations that may be written to by the process P. The critical rule is $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$-IF, which checks that after the program has branched on ℓ-level data there are no writes to memory locations lower than ℓ.

The typing rules $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$-PRIM, $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$-REF, and $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$-ASSN introduce the same label constraints that primitive operations and references do in $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$. These constraints are sufficient to prevent unwanted information flows due to reference creation or assignment, and they approximate the information flows due to binary operations.

Rule $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$-IF propagates the label of the conditional into the pc labels for checking the branches. Because the linear resources must be used no matter which branch is taken, both branches have access to all of the linear context.

$\boxed{H; \Gamma; T; K\ [\mathsf{pc}] \vdash P}$

$$\lambda^{\text{CONCUR}}_{\text{SEC}}\text{-PAR} \qquad \frac{H; \Gamma; T_i; K_i\ [\mathsf{pc}] \vdash P_i \qquad (i \in \{1, 2\})}{H; \Gamma; T_1, T_2; K_1, K_2\ [\mathsf{pc}] \vdash P_1 \mid P_2}$$

$$\lambda^{\text{CONCUR}}_{\text{SEC}}\text{-LET} \qquad \frac{\begin{array}{l} J \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_f;\ \Gamma_{args};\ K_{args} \rangle \\ H; \Gamma, \Gamma_f, \Gamma_{args}; \cdot, K_{args}\ [\mathsf{pc}] \vdash P_1 \\ H; \Gamma, \Gamma_f; T, K\ [\mathsf{pc}] \vdash P_2 \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \texttt{let}\ J \rhd P_1\ \texttt{in}\ P_2}$$

$$\lambda^{\text{CONCUR}}_{\text{SEC}}\text{-LETLIN} \qquad \frac{\begin{array}{l} J \rightsquigarrow \langle K_f;\ \Gamma_{args} \rangle \\ H; \Gamma, \Gamma_{args}; T_1; K_2\ [\mathsf{pc}] \vdash P_1 \\ H; \Gamma; T_2; K_1, K_f\ [\mathsf{pc}] \vdash P_2 \end{array}}{H; \Gamma; T_1, T_2; K_1, K_2\ [\mathsf{pc}] \vdash \texttt{let}\ J \multimap P_1\ \texttt{in}\ P_2}$$

$$\lambda^{\text{CONCUR}}_{\text{SEC}}\text{-SEND} \qquad \frac{\begin{array}{l} H; \Gamma \vdash v : [\mathsf{pc}'](\vec{s}, k^{\mathsf{opt}})_\ell \\ H; \Gamma\ [\mathsf{pc}] \vdash v_i : s_i \\ T; K \vdash lv^{\mathsf{opt}} : k^{\mathsf{opt}} \\ \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash v(\vec{v}, lv^{\mathsf{opt}})}$$

$$\lambda^{\text{CONCUR}}_{\text{SEC}}\text{-LINSEND} \qquad \frac{\begin{array}{l} T; K \vdash lv : (\vec{s}) \\ H; \Gamma\ [\mathsf{pc}] \vdash v_i : s_i \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash lv(\vec{v})}$$

Figure 5.14: Process typing (continued)

$\boxed{J \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_f; \Gamma_{args}; K \rangle}$ $\boxed{J \rightsquigarrow \langle K; \Gamma_{args} \rangle}$

$$f(\vec{x}) \rightsquigarrow_{\mathsf{pc}} \langle f : [\mathsf{pc}](\vec{s}),\ \vec{x} : \vec{s},\ \emptyset \rangle$$

$$f(\vec{x}, y) \rightsquigarrow_{\mathsf{pc}} \langle f : [\mathsf{pc}](\vec{s}, k),\ \vec{x} : \vec{s},\ y : k \rangle$$

$$\frac{\begin{array}{c} J_1 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f1};\ \Gamma_{args1};\ K_1 \rangle \\ J_2 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f2};\ \Gamma_{args2};\ K_2 \rangle \end{array}}{J_1 \mid J_2 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f1}, \Gamma_{f2};\ \Gamma_{args1}, \Gamma_{args2};\ K_1, K_2 \rangle}$$

$$f(\vec{x}) \rightsquigarrow \langle f : (\vec{s}),\ \vec{x} : \vec{s} \rangle$$

$$\frac{J_1 \rightsquigarrow \langle K_1;\ \Gamma_{args1} \rangle \quad J_2 \rightsquigarrow \langle K_2;\ \Gamma_{args2} \rangle}{J_1 \mid J_2 \rightsquigarrow \langle K_1, K_2;\ \Gamma_{args1}, \Gamma_{args2} \rangle}$$

Figure 5.15: Join pattern bindings

Rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-ZERO says that the null process type-checks only if all of the linear resources have been used.

Concurrent processes $P_1 \mid P_2$ are checked using the program-counter label of the parent process, as shown in rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-PAR of Figure 5.14. The two processes have access to the same nonlinear resources, but the linear resources must be partitioned between them.

The typing rules $\lambda_{\text{SEC}}^{\text{CONCUR}}$-LET and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-LETLIN make use of auxiliary operations that extract variable binding information from handler definitions. A join pattern J yields a collection Γ_f of channels it defines and a set of variables bound in the body of the handler definition Γ_{args}. For nonlinear join patterns, the linear variables form a synchronization context K. The operation $J \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_f;\ \Gamma_{args};\ K \rangle$, defined in Figure 5.15 collects these channel names and variables for nonlinear join patterns and assigns them types. A similar operation $J \rightsquigarrow \langle K;\ \Gamma_{args} \rangle$ defined for linear join patterns extracts the synchronization point K and the context for the handler body, Γ_{args}.

Rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-LET is analogous to $\lambda_{\text{SEC}}^{\text{CPS}}$-CONT from $\lambda_{\text{SEC}}^{\text{CPS}}$ (see Figure 4.5). It checks the body of the handler under the assumption that the arguments bound by the join pattern have the appropriate types. Nonlinear handlers cannot capture free linear values or channels, because that would potentially violate their linearity. Consequently, the only linear resources available inside the body P_1 are those explicitly passed to the handler: K_{args}. Note that the channels defined by the nonlinear handler (Γ_f) are

$\boxed{H \vdash M}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-EMPTY
$$H \vdash \cdot$$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-LOC
$$\frac{H \vdash M \quad H; \cdot \vdash v : H(L)}{H \vdash M[L \mapsto v]}$$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-HANDLER
$$\frac{\begin{array}{l} H \vdash M \\ H(c_i) = [\mathsf{pc}](\vec{s_i}, k_i^{\mathsf{opt}}) \\ H; \vec{x_1} : \vec{s_1} \ldots \vec{x_n} : \vec{s_n}; y_1^{\mathsf{opt}} : k_1^{\mathsf{opt}} \ldots y_n^{\mathsf{opt}} : k_n^{\mathsf{opt}}\ [\mathsf{pc}] \vdash P \end{array}}{H \vdash M[c_1(\vec{x_1}, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x_n}, y_n^{\mathsf{opt}}) \rhd P]}$$

Figure 5.16: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ heap types

available inside the handler body, which allows recursion. The process P_2 has access to the newly defined channels (in Γ_f) and to the previously available resources.

Rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-LETLIN shows how linear resources are manipulated when a linear handler is declared. The join pattern J defines a collection of linear channels K_f and arguments Γ_{args}. The arguments (Γ_{args}), as well as some of previously defined linear channels (T_1 and K_1), are available in the linear handler's body (P_1). The rest of the linear resources (T_2 and K_2), plus the newly defined linear channels, are available in the process P_2.

The rule for type-checking messages sends on nonlinear channels requires that the channel type and the types of the values passed in the message agree. Also, the program counter at the point of the send must be protected by the label of message handler; this constraint rules out implicit information flows. Rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-SEND shows how these constraints are required.

Sending a message on a linear channel does not impose any constraints on the pc label at the point of the send, reflecting that fact that there is no information revealed by the fact that a message is sent on a linear channel. Note that the contents of the messages are labeled with the pc label—the message sent on a linear channel might contain information about the program counter.

A memory M is a well-formed with heap type H when $H \vdash M$ can be derived according to the rules in Figure 5.16. Furthermore, no channel name should be defined in more than one handler definition appearing in M: for all distinct join patterns J_1 and J_2 in $\mathrm{dom}(M)$ if $J_1 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f_1}; -; - \rangle$ and $J_2 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f_2}; -; - \rangle$, then

$$\mathrm{dom}(\Gamma_{f_1}) \cap \mathrm{dom}(\Gamma_{f_2}) = \emptyset$$

$\boxed{H; T_1 \vdash S; T_2}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-S-EMPTY $\quad H; T \vdash \cdot; \cdot$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-S-HANDLER
$$\frac{\begin{array}{c} H; T \vdash S; T_1 \\ T(c_i) = (\vec{s_i}) \quad T_2 \subseteq T \\ H; \vec{x_1} : \vec{s_1} \ldots \vec{x_n} : \vec{s_n}; T_2, \cdot\, [\mathsf{pc}] \vdash P \end{array}}{H; T \vdash S[\mathsf{pc} : c_1(\vec{x_1}) \mid \ldots c_n(\vec{x_n}) \multimap P]; T_1, T_2}$$

Figure 5.17: $\lambda_{\text{SEC}}^{\text{CONCUR}}$ synchronization environment types

$\boxed{H; \Gamma; T; K \vdash N}$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-EMPTY $\quad H; \Gamma; \cdot; \cdot \vdash \cdot$

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC
$$\frac{\begin{array}{c} H; \Gamma; T_1; K_1 \vdash N \\ H; \Gamma; T_2; K_2\, [\mathsf{pc}] \vdash P \end{array}}{H; \Gamma; T_1, T_2; K_1, K_2 \vdash N \mid [\mathsf{pc} : P]}$$

Figure 5.18: Network typing rules

Note that the heap type H may contain more bindings than are present in M.

A synchronization environment S has type T when no channel name in T is defined in more than one handler, $\mathrm{dom}(S) = \mathrm{dom}(T)$, and $H; T \vdash S; T'$ can be derived according to the rules in Figure 5.17. Any linear channel in T must be used in at most one of the handlers present in the synchronization environment. The type T' records which channels have been used in S. The channels in $T \setminus T'$ must be used within the program accompanying the synchronization environment, as required by the rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-CONFIG (given below).

A network of processes is well-typed whenever each of the processes in the network is well-typed and the linear resources are completely used, as shown in Figure 5.18. Finally, a configuration $\langle M,\, S,\, N \rangle$ is well-typed if its components are and all of the linear channels defined in S are used exactly once in either S or N:

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-CONFIG
$$\frac{H \vdash M \quad H; T_1, T_2 \vdash S; T_1 \quad H; \cdot; T_2; \cdot \vdash N}{H; T_1, T_2 \vdash \langle M,\, S,\, N \rangle}$$

5.2.3 Race prevention and alias analysis

As we have seen, two concurrently running threads might leak confidential information if they have write–write or read–write races. This section discusses how to formalize race freedom and how program analysis techniques can potentially be used to establish that a program is race free.

Consider example (1) from Section 5.1.1. In the $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ syntax, it can be written as the program P_1:

$$P_1 \stackrel{\text{def}}{=} \texttt{(set } L_1 \texttt{ := t in 0) | (set } L_1 \texttt{ := f in 0)}$$

The possible evaluation sequences of this program, starting from the memory $[L_1 \mapsto \texttt{t}]$ are shown in the next diagram.[6]

$$\langle [L_1 \mapsto \texttt{t}],\, \cdot,\, P_1 \rangle \;\begin{cases} \Rrightarrow \langle [L_1 \mapsto \texttt{t}],\, \cdot,\, \texttt{set } L_1 \texttt{ := f in 0} \rangle \;\Rrightarrow\; \langle [L_1 \mapsto \texttt{f}],\, \cdot,\, \texttt{0} \rangle \\ \Rrightarrow \langle [L_1 \mapsto \texttt{f}],\, \cdot,\, \texttt{set } L_1 \texttt{ := t in 0} \rangle \;\Rrightarrow\; \langle [L_1 \mapsto \texttt{t}],\, \cdot,\, \texttt{0} \rangle \end{cases}$$

The evaluation is nondeterministic because of the concurrency. Once the choice of which thread to run first is made, the evaluations become distinct —the nondeterministic choice becomes apparent from the program state. Importantly, the nondeterminism is visible solely by watching the contents of location L_1. Example (2) from Section 5.1.1 exhibits similar properties.

Contrast those programs with the following race-free program, P_2:

$$P_2 \stackrel{\text{def}}{=} \texttt{(set } L_1 \texttt{ := t in 0) | (set } L_2 \texttt{ := f in 0)}$$

The possible evaluation sequences of this program starting from the memory $M_1 = [L_1 \mapsto \texttt{f}][L_2 \mapsto \texttt{f}]$ are shown below:

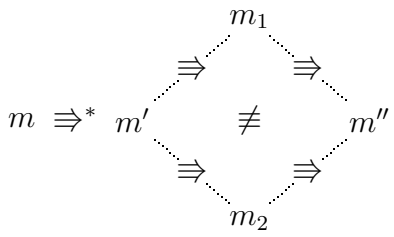
$$\langle M_1,\, \cdot,\, P_1 \rangle \;\begin{cases} \Rrightarrow \langle M_1[L_1 \mapsto \texttt{t}],\, \cdot,\, \texttt{set } L_2 \texttt{ := f in 0} \rangle \Rrightarrow \\ \Rrightarrow \langle M_1[L_2 \mapsto \texttt{f}],\, \cdot,\, \texttt{set } L_1 \texttt{ := t in 0} \rangle \Rrightarrow \end{cases} \langle [L_1 \mapsto \texttt{t}][L_2 \mapsto \texttt{f}],\, \cdot,\, \texttt{0} \rangle$$

In this case, even though there is still nondeterminism in the evaluation, the evaluation of one thread does not disrupt the computation of the other thread. Evaluation of the two threads *commutes*, which implies that the scheduler is free to choose any ordering—the eventual outcome of the program is unaffected.

These observations lead to the following definition of race freedom, which requires that any configuration reachable from the starting configuration satisfy the commutativity property.

[6] For the sake of clarity, the network syntax $[\texttt{pc} : P_1]$ has been omitted from this description, because the program counter label is irrelevant.

Definition 5.2.6 *A configuration m is* **race free** *whenever $m \Rrightarrow^* m'$ and $m' \Rrightarrow m_1$ and $m' \Rrightarrow m_2$ and $m_1 \not\equiv m_2$ imply that there exists an m'' such that $m_1 \Rrightarrow m''$ and $m_2 \Rrightarrow m''$. Pictorially, this can be seen as:*



An open term is race free whenever all of its closed instances are race free. Formally, N is race free if $\cdot; \Gamma; \cdot; \cdot \vdash N$ and for every substitution γ such that $\cdot \models \gamma : \Gamma$ the configuration $\langle \cdot, \cdot, \gamma(N) \rangle$ is race free.

This is a strong notion of race freedom. For example, under this definition the following program has a race:

```
let h(x) | g() = let a = x in 0 in
  h(t) | h(f) | g()
```

Here, the program evolves into one of two distinct possible configurations that have networks containing either the (unhandled) message send `h(t)` or the (unhandled) message send `h(f)` —such nondeterminism corresponds to send contention.

This definition of race freedom is certainly sufficient to rule out the timing leaks that may occur between threads. However, it is stronger than necessary if the only externally observable aspect of the machine configuration is the memory (and not the program counter or channel states). It is possible to weaken the definition of race freedom to consider harmful only nondeterminism apparent from the memory, in which case side-effect free programs (like the one above) are permitted to exhibit nondeterminism.

However, even with a weakened definition of race freedom, the nondeterminism on nonlinear channels can cause races. For example, the following program nondeterministically stores either `t` or `f` into the reference `l` because there is a race between the two sends on channel `h`.

```
let h(x) | g() = set l := x in 0 in
  h(t) | h(f) | g()
```

The channels involved in the race need not carry values (as in the program above). Instead, the handler itself may contain state that introduces the nondeterminism, as shown here:

```
let h() ▷ let z = !a in set a := ¬z in 0 in
  h() | h()
```

As a last example, the following program exhibits a race to the assignment of the location `a`. It shows a slightly more subtle way to create a race to the send on channel `h`:

```
let h(x) ▷ set a := x in 0 in
let g(h1, h2) ▷ h1(t) | h2(f) in
  g(h, h)
```

Observe that all of the examples of races involve two aliases of either a reference or a channel used concurrently. Consequently, preventing races relies on detecting possible aliasing of references and channels and disallowing aliases to be used by multiple threads. Rather than formulate a specific alias analysis for fine-grained control over the resources available to each thread, this thesis instead assumes that the concurrent program is race-free.

There are a number of ways that race freedom can be established. One useful approach is to use alias analysis to (soundly) approximate the set of locations and channels written to (or sent messages) by a thread. Call this set by $\mathsf{write}(P)$. By determining which locations are potentially read by P (a set $\mathsf{read}(P)$), an analysis can prevent races by requiring that, for any subprograms P_1 and P_2 that might occur during evaluation:

$$\begin{array}{rl} & \mathsf{write}(P_1) \cap (\mathsf{read}(P_2) \cup \mathsf{write}(P_2)) = \emptyset \\ \wedge & \mathsf{write}(P_2) \cap (\mathsf{read}(P_1) \cup \mathsf{write}(P_1)) = \emptyset \end{array}$$

Alias analyses construct finite models of the dynamic behavior of a program so that which references are dynamically instantiated with which memory locations can be statically approximated. The more closely the abstract model agrees with the true behavior of the system, the more accurate the aliasing information can be. An abstract interpretation is *sound* if it faithfully models the behavior of the system—it does not give answers that disagree with the actual behavior of the system. Here, the alias analysis can be used to approximate the sets $\mathsf{read}(-)$ and $\mathsf{write}(-)$ sufficient to establish race freedom.[7]

As an extreme, one sound analysis is to simply assume that any process might read or write any reference. Such a rough approximation to the actual aliasing would require that the program be sequential. Another possibility is to approximate alias information using types: a reference of type `bool ref` can never alias a reference of type `int ref`, for example. This scheme would allow concurrent threads to mutate parts of the heap that contain different types of data.

A second possibility is to ensure that references and nonlinear channels do not cause races is to require them to be used sequentially. Simple syntactic constraints similar to

[7] Although it is beyond the scope of this thesis, it should be possible to prove that this application of alias analysis implies the semantic definition of race freedom.

linearity that can force a handler to be used sequentially. (See, for example, Reynolds' original work on syntactic control of interference [Rey78].) Consider the handler declaration `let J` $\triangleright$ `P in Q`. If the channel names defined in J are used *affinely* (at most once statically) in P and affinely in Q, then the body of the handler (P) will never execute concurrently with another instance of itself—the handler must be used sequentially. One can formulate sequentiality in the type system (as shown by Honda et al. [HVY00, HY02]), but doing so is rather complex.

Another possibility is to track aliasing directly in the type system [SWM00, WM00]. Such an approach would potentially permit very fine-grained control of concurrency. More generally, pointer or shape analysis can be used to approximate the $\mathsf{read}(-)$ and $\mathsf{write}(-)$ sets. Most relevant to this thesis are interprocedural analyses [LR92, Deu94], analyses that deal with function pointers [EGH94], and the work on pointer analysis for multithreaded programs [RR99].

There are also a number of type systems that regulate locking protocols to prevent race conditions [FA99b, FA99a, FF00]. Instead of using aliasing information directly, these analyses ensure that an appropriate lock is held before a memory location is accessed; they introduce ordering constraints on locks to serialize the memory accesses. Although not intended to guarantee the strong race-freedom requirement needed here (these analyses still permit nondeterministic ordering of memory writes), it might be possible to use them as a starting point.

It is worth noting that linear channels may never be aliased because they cannot be duplicated or stored in the memory. Linear type information can therefore be used by the alias analysis. However, we have not yet addressed the connection between the ordered linear continuations of $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ and the linear channels of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$.

The type system presented in Figures 5.13 and 5.14 does not explicitly enforce any ordering constraints on the use linear channels. Instead, the ordering is induced by the race-freedom requirement because it restricts how linear channels may be mentioned in the program.

To see how linear channels, sequentiality, and race prevention are related, consider the following program.

```
let h(x) ▷ P in
  h(t) | h(f)
```

The channel `h` itself is nonlinear and hence can be used multiple times. It may step to a configuration containing the process $P\{\mathtt{3/x}\}$ or $P\{\mathtt{4/x}\}$ nondeterministically. Any sequencing between invocations of `h` must be encoded explicitly using linear channels; the linear message acknowledges that the channel has received one message and is waiting for another. To ensure that the message `h(f)` is consumed before `h(t)`, the above example would be changed to:

```
let h(x,k) ▷ P;k() in
let k1() —∘ h(t, k0) in
  h(f, k1)
```

Here, the linear channel $\mathtt{k}_1$ expresses the causal connection between the send of `f` on `h` and the send of `t` on `h`. Note that $\mathtt{k}_0$ is unspecified: It expresses the causal relation between the send of `f` on `h` and any future sends. Importantly, $\mathtt{k}_0$ is *free* in the body of the handler for $\mathtt{k}_1$.

The linearity of the acknowledgment channels is crucial: duplicating the acknowledgment messages allows multiple "futures" to take place, destroying sequentiality:

```
let h(x,k) ▷ P;k() in
let k1() ▷ h(t, k0) in
  h(f, k1) | h(t, k1)
```

Note that because of synchronization between linear channels, the causal ordering can be more complex. Consider the following example.

```
let k1() | k2() —∘ P in
let k3() —∘ Q;k1() in
  k2() | R;k3()
```

In this program, threads R, Q, and P must be evaluated in that order, even though there is a message sent on channel $\mathtt{k}_2$ (potentially) before the send on channel $\mathtt{k}_3$. The sequentiality is guaranteed because the handler body for $\mathtt{k}_3$ sends the message on $\mathtt{k}_1$—the message on $\mathtt{k}_3$ must take place before the synchronization on $\mathtt{k}_1$ and $\mathtt{k}_2$.

Race freedom rules out the following program because there is no causal ordering between the handlers for $\mathtt{k}_1$ and $\mathtt{k}_2$, even though they are used linearly:

```
let k1() —∘ set a := t in 0 in
let k2() —∘ set a := f in 0 in
  k1() | k2()
```

The following similar program does establish a total ordering between $\mathtt{k}_1$ and $\mathtt{k}_2$, so it is permitted by the race-freedom condition.

```
let k1() —∘ set a := t in 0 in
let k2() —∘ set a := f in k1() in
  k2()
```

Race freedom implies that there is a *causal ordering* relationship between the linear synchronization handlers and that any two handlers that interfere are totally ordered. This constraint ensures that interfering linear handlers are used sequentially, which implies that updates to memory locations mentioned in them are deterministic.

5.3 Subject reduction for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$

This section establishes a subject reduction theorem for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$. It follows the same general outline as the subject reduction proof for $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$. As with $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ the subject reduction property is key to establishing noninterference for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$.

We first establish the standard lemmas.

Lemma 5.3.1 (Substitution 1) *Suppose* $H; \Gamma, x : s; T, K\ [\mathsf{pc}] \vdash P$ *and* $H; \cdot \vdash v : s$ *then* $H; \Gamma; T, K\ [\mathsf{pc}] \vdash P\{v/x\}$.

Proof (sketch): The proof first strengthens the hypothesis to allow substitution in all of the syntactic classes of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$. The result then follows from mutual induction on the structure of the typing derivation. The argument is similar to the proof of Lemma 4.3.1. □

Lemma 5.3.2 (Substitution 2) *Suppose* $H; \Gamma; T_1, K_1, y : k\ [\mathsf{pc}] \vdash P$ *and* $T_2; K_2 \vdash lv : k$ *then* $H; \Gamma; T_1, T_2, K_1, K_2\ [\mathsf{pc}] \vdash P\{lv/y\}$.

Proof (sketch): The proof first strengthens the hypothesis to allow substitution in all of the syntactic classes of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$. The result then follows from mutual induction on the structure of the derivation that P is well-typed. The base cases follow directly from the rules in Figure 5.11. □

Lemma 5.3.3 (Program counter variance) *If* $H; \Gamma; T, K\ [\mathsf{pc}] \vdash P$ *and* $\mathsf{pc}' \sqsubseteq \mathsf{pc}$ *then* $H; \Gamma; T, K\ [\mathsf{pc}'] \vdash P$.

Proof: By induction on the derivation that P is well-typed under $[\mathsf{pc}]$. Note that all lattice inequalities involving the program counter appear to the left of $\sqsubseteq$. Thus, replacing pc by pc' in a typing derivation will still yield a valid derivation. □

Lemma 5.3.4 (Primitive evaluation) *If* $H; \cdot\ [\mathsf{pc}] \vdash prim : s$ *and* $M, \mathsf{pc} \models prim \Downarrow v$ *then* $H; \cdot \vdash v : s$.

Proof (sketch): The proof is nearly identical to that of Lemma 4.3.5. □

Lemma 5.3.5 (Heap weakening) *If* H' *extends* H *and* H *appears to the left of* $\vdash$ *in the conclusion of a typing derivation, then replacing* H *by* H' *in the conclusion yields a valid typing judgment.*

Proof (sketch): By induction on the typing derivations. The base case follows from the fact that H' extends H (and hence agrees with H on their common domain), so rules $\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-LOC and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-HANDLER hold with H' instead of H.
□

Because $\lambda_{\text{SEC}}^{\text{CONCUR}}$ has a more complicated operational semantics, we also need to show that the additional components of the machine configuration are well-typed.

Lemma 5.3.6 (Synchronization environment weakening) *If $H; T_1 \vdash S; T_2$ and T' is any synchronization state type such that $\text{dom}(T') \cap \text{dom}(T_1) = \emptyset$ then $H; T_1, T' \vdash S; T_2$*

Proof: By induction on the typing derivation. The base case follows immediately from the rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-S-EMPTY. □

The structural equivalence of machine configurations allows us to disregard the syntactic order of processes in a program. The following lemma establishes, as expected, that reordering the processes in a configuration does not affect its typing properties.

Lemma 5.3.7 (Equivalence preserves typing) *If $H; T \vdash \langle M, S, N\rangle$ and configuration $\langle M, S, N\rangle \equiv \langle M', S', N'\rangle$ then there exists a renaming H' of H and a renaming T' of T such that $H'; T' \vdash \langle M', S', N'\rangle$.*

Proof (sketch): This lemma follows from several observations:

1. The rules $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-EMPTY and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-ZERO agree on the linear portion of the context.

2. Partitioning of linear contexts is commutative and associative (and hence agrees with the requirements of rules $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-PAR).

3. Consistent renaming of memory locations or channel names does not change their associated types.

□

Using the above lemmas, we prove the following lemma.

Lemma 5.3.8 (Subject reduction) *Suppose $H; T \vdash \langle M, S, N\rangle$ and*

$$\langle M, S, N\rangle \Rrightarrow \langle M', S', N'\rangle$$

Then there exists H' and T' such that $H'; T' \vdash \langle M', S', N'\rangle$.

Proof: This proof follows as a corollary of the strengthened version below, using Lemma 5.3.7 twice to recover the result for $\Rrightarrow$ from the case for $\rightarrow$ and the structural evaluation rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EQSTEP. □

Lemma 5.3.9 (Strengthened subject reduction) *Suppose* $H; T \vdash \langle M,\, S,\, N\rangle$ *and*

$$\langle M,\, S,\, N\rangle \to \langle M',\, S',\, N'\rangle$$

Then there exists H' *and* T' *such that* $H'; T' \vdash \langle M',\, S',\, N'\rangle$*. Furthermore,* H' *extends* H *and* T *and* T' *agree on the channels in their intersection.*

Proof: The proof is by cases on the evaluation step used.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$**-EVAL-LETPRIM** This case follows immediately from Lemmas 5.3.4 and 5.3.1. Note that neither the memory nor the synchronization environment change.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$**-EVAL-LETREF** It must be that $N = N'' \mid [\mathsf{pc} : \texttt{let}\ x = \texttt{ref}\ v\ \texttt{in}\ P]$ and $N' = N'' \mid [\mathsf{pc} : P\{L/x\}]$ and $M' = M[L \mapsto v]$. Let $H' = H[L : s\ \texttt{ref}]$. By the rule's side condition, the location L does not occur in the domain of H or M. Therefore, H' extends H. Because N is well-typed, it follows that $H; \cdot; T_1; \cdot \vdash N''$ and $H; x : s\ \texttt{ref}_{\mathsf{pc}}; T_2, \cdot\ [\mathsf{pc}] \vdash P$ where $T = T_1, T_2$. Furthermore, it must be the case that $H; \cdot \vdash v : s$. By Lemma 5.3.5, we then have $H'; \cdot; T_1; \cdot \vdash N''$ and $H'; x : s\ \texttt{ref}; T_2, \cdot\ [\mathsf{pc}] \vdash P$ and $H'; \cdot \vdash v : s$. Note that $H' \vdash M'$ follows from Lemma 5.3.5 and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-LOC. The fact that N' is well-typed follows from Lemma 5.3.1 and rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$**-EVAL-SET** This case follows almost exactly as the previous one.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$**-EVAL-HANDLER** This case is like the one for $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LETREF except that weakening (Lemma 5.3.5) and rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-HANDLER are used to establish that substitution applies. Note that substitution must be performed on the handler body. This case also relies on the operation $\rightsquigarrow_{\mathsf{pc}}$ agreeing with the requirements of rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-HANDLER. The freshness of the channel names is needed to establish that H' extends H.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$**-EVAL-LINHANDLER** It must be the case that $N = N'' \mid P$ where $P = [\mathsf{pc} : \texttt{let}\ J \multimap P_1\ \texttt{in}\ P_2]$ and $J \rightsquigarrow \langle K_f; \Gamma_{args}\rangle$. Suppose that

$$K_f = \{f_1 : (\vec{s_1}), \ldots, f_n : (\vec{s_n})\}$$

Because the configuration is well-typed under H and T, we also have $H \vdash M$ and $H; T \vdash S; T_1$ and $H; \cdot; T_2; \cdot \vdash N$, where $T = T_1, T_2$. Inversion of rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC yields $H; \cdot; T_3; \cdot \vdash N''$ and $H; \cdot; T_4, \cdot\ [\mathsf{pc}] \vdash P$, where $T_2 = T_3, T_4$. Inversion of rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-LETLIN yields $H; \Gamma_{args}; T_5, \cdot\ [\mathsf{pc}] \vdash P_1$ and $H; \cdot; T_6, K_f\ [\mathsf{pc}] \vdash P_2$ where $T_4 = T_5, T_6$. Let $T_i' = \{c_i : K_f(f_i)\}$ for $1 \le i \le n$ and the c_i chosen according to the transition rule and let $T_7 = \bigcup_i^n T_i'$. Then

by rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-LINCHAN we have $T_i'; \cdot \vdash c_i : K_f(f_i)$. By n applications of Lemma 5.3.2 we obtain $H; \cdot; T_6, T_7, \cdot\, [\mathsf{pc}] \vdash P_2\{\vec{c_i}/f_i\}$.

Applying rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC to the antecedents

$$H; \cdot; T_3; \cdot \vdash N'' \quad \text{and} \quad H; \cdot; T_6, T_7, \cdot\, [\mathsf{pc}] \vdash P_2\{\vec{c_i}/f_i\}$$

we obtain

$$H; \cdot; T_3, T_6, T_7; \cdot \vdash N'' \mid [\mathsf{pc} : P_2\{\vec{c_i}/f_i\}]$$

We have already shown that $H; \Gamma_{args}; T_5, \cdot\, [\mathsf{pc}] \vdash P_1$ and $H; T \vdash S; T_1$ hold. By weakening (Lemma 5.3.6) it follows that $H; T, T_7 \vdash S; T_1$. Note that $T_5 \subseteq T_4 \subseteq T_2 \subseteq T \subseteq T, T_7$ and that, by construction $(T, T_7)(c_i) = K_f(x_i)$. Therefore, we may apply rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-S-HANDLER to obtain

$$H; T, T_7 \vdash S[\mathsf{pc} : J \multimap P_1]; T_1, T_5$$

Now we must account for the linearity of the channels:

$$T, T_7 = T_1, T_2, T_7 = T_1, T_3, T_4, T_7 = T_1, T_3, T_5, T_6, T_7$$

Resources T_1 and T_5 are used in $S' = S[\mathsf{pc} : J \multimap P_1]$ and resources T_3, T_6, and T_7 are used in $N' = N'' \mid [\mathsf{pc} : P_2\{\vec{c_i}/f_i\}]$, so we may apply $\lambda_{\text{SEC}}^{\text{CONCUR}}$-CONFIG to conclude $H; T, T_7 \vdash \langle M,\, S',\, N' \rangle$, as required.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-**EVAL-COND1** We have $N = N'' \mid [\mathsf{pc} : \texttt{if}\ \mathtt{t}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2]$. Because the configuration is well-typed under H and T, we also have $H \vdash M$ and $H; T \vdash S; T_1$ and $H; \cdot; T_2; \cdot \vdash N$, where $T = T_1, T_2$. Inversion of rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC yields $H; \cdot; T_3; \cdot \vdash N''$ and

$$H; \cdot; T_4, \cdot\, [\mathsf{pc}] \vdash \texttt{if}\ \mathtt{t}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2$$

where $T_2 = T_3, T_4$. From $\lambda_{\text{SEC}}^{\text{CONCUR}}$-IF, it follows that $H; \cdot \vdash \mathtt{t}_\ell : \mathtt{bool}_{\ell'}$ where $\ell \sqsubseteq \ell'$ and that $H; \cdot; T_4, \cdot\, [\mathsf{pc} \sqcup \ell'] \vdash P_1$. By program counter variance (Lemma 5.3.3) it follows that $H; \cdot; T_4, \cdot\, [\mathsf{pc} \sqcup \ell] \vdash P_1$. Applying rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC yields $H; \cdot; T_3, T_4; \cdot \vdash N'' \mid [\mathsf{pc} \sqcup \ell : P_1]$, from which we can use $\lambda_{\text{SEC}}^{\text{CONCUR}}$-CONFIG to conclude $H; T_1, T_3, T_4 \vdash \langle M,\, S,\, N' \rangle$, where $N' = N'' \mid [\mathsf{pc} \sqcup \ell : P_1]$. Recall that $T_3, T_4 = T_2$ and that $T_1, T_2 = T$, so we have the desired result.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-**EVAL-COND2** This case is nearly identical to the previous case.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-**EVAL-SEND** We have $N = N'' \mid_i [\mathsf{pc}_i : c_{i\ell_i}(\vec{v}_i, lv_i^{\mathsf{opt}})]$ and it is also the case that $M = M''[c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \rhd P]$. Because the configuration is well-typed under H and T, we also have $H \vdash M$ and $H; T \vdash S; T_1$ and $H; \cdot; T_2; \cdot \vdash N$, where $T = T_1, T_2$. Inversion of rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC n times yields $H; \cdot; T_3; \cdot \vdash N''$ and $H; \cdot; T_i', \cdot\,[\mathsf{pc}_i] \vdash c_{i\ell_i}(\vec{v_i}, lv_i^{\mathsf{opt}})$ where $T_2 = T_3, T_4$ and $T_4 = T_1', \ldots, T_n'$. For each $i \in \{1, \ldots, n\}$ the rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-SEND yields $H; \cdot \vdash c_{i\ell_i} : [\mathsf{pc}_i'](\vec{s_i}, k_i^{\mathsf{opt}})_{\ell_i'}$ where $\mathsf{pc}_i \sqcup \ell_i' \sqsubseteq \mathsf{pc}_i'$ and $\ell_i \sqsubseteq \ell_i'$. Furthermore, we also have $H; \cdot \vdash v_{ij} : s_{ij}$ where $\mathsf{pc}_i \sqsubseteq \mathsf{label}(s_{ij})$ and $T_i'; \cdot \vdash lv_i^{\mathsf{opt}} : k_i^{\mathsf{opt}}$.

Because $\mathsf{pc}_i \sqsubseteq \mathsf{label}(s_{ij})$, we have $H; \cdot \vdash v_{ij} \sqcup \mathsf{pc}_i : s_{ij}$.

From $H \vdash M$ and rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-HEAP-HANDLER we have

$$H; \vec{x_1} : \vec{s_1}, \ldots, \vec{x_n} : \vec{s_n}; \cdot; y_1^{\mathsf{opt}} : k_1^{\mathsf{opt}}, \ldots, y_n^{\mathsf{opt}} : k_n^{\mathsf{opt}}\ [\mathsf{pc}] \vdash P$$

Applying the substitution Lemmas 5.3.1 and 5.3.2 we obtain

$$H; \cdot; T_4, \cdot\,[\mathsf{pc}] \vdash P\{\vec{v_i} \sqcup \mathsf{pc}_i / \vec{x_i}\}\{lv_i / y_i\}^{\mathsf{opt}}$$

Note that rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-SEND requires that $\mathsf{pc}_i' \sqsubseteq \mathsf{pc}$ and because $\mathsf{pc}_i \sqcup \ell_i \sqsubseteq \mathsf{pc}_i'$ we have

$$\ell = \bigsqcup_i^n (\mathsf{pc}_i \sqcup \ell_i) \sqsubseteq \bigsqcup_i^n \mathsf{pc}_i' \sqsubseteq \mathsf{pc}$$

By Lemma 5.3.3 it follows that $H; \cdot; T_4, \cdot\,[\ell] \vdash P\{\vec{v_i} \sqcup \mathsf{pc}_i / \vec{x_i}\}\{lv_i / y_i\}^{\mathsf{opt}}$.

Applying $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC, we obtain

$$H; \cdot; T_3, T_4; \cdot \vdash N'' \mid [\ell : P\{\vec{v_i} \sqcup \mathsf{pc}_i / \vec{x_i}\}\{lv_i / y_i\}^{\mathsf{opt}}]$$

Lastly, we observe that $T_2 = T_3, T_4$ so rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-CONFIG yields the desired result.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-**EVAL-LINSEND** This case is similar to the previous case, except that we note that the T' is T minus the linear channels used in the step.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-**EVAL-FORK** This case follows straightforwardly from the two typing rules $\lambda_{\text{SEC}}^{\text{CONCUR}}$-NET-PROC and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-PAR and the fact that set union (for linear contexts) is associative.

□

Note that progress, as it is usually stated, does not hold for $\lambda_{\text{SEC}}^{\text{CONCUR}}$. A nonlinear channel may never be sent a message, causing the corresponding handler to block forever waiting. For example, the following program either runs the subprocess P or gets stuck, depending on the value of `l`, but it should be typable:

```
let h() | g() ▷ P in
  h() | (if l then 0 else g())
```

There is a weaker analog to progress in this setting. A program may not get stuck if there are still outstanding linear channels that have not yet received communication. One could formulate type-soundness without using a progress lemma, for instance, by adding an explicit bad state and then showing that the type system prevents a well-typed program from stepping to the bad state. Such a result is unimportant for the discussion of noninterference in the next section; subject reduction is the key lemma.

5.4 Noninterference for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$

This section establishes that $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ programs that are well typed and satisfy a race-freedom condition obey noninterference. Recall that the ultimate goal of the noninterference result is to establish that altering the high-security parts of the program does not affect the deterministic behavior of the low-security parts of the store. Because this definition of noninterference ignores external timing channels and internal timing channels are prevented by eliminating races, it suffices to show that the low-security memory access behavior of the program can be simulated by another deterministic program that differs in its high-security parts.

A program meets race-freedom requirement if all of its low-security simulations are race free. Intuitively, a low-security simulation of a $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ program P is another $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ program that differs from P in high-security values; the low-security simulation also forces high-security computation to terminate in exactly one step. A low-security simulation of P reflects the possible behavior of P as seen by a low-security observer. Because high-security computation in the simulation terminates after one step, the resulting security condition is timing and termination insensitive.

Importantly, because the simulation of a program P is itself a $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ program, establishing that the simulation is race free is no harder than establishing that a $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ program is race free. Any of the techniques discussed in Section 5.2.3 can be used to ensure the race-freedom requirement. Also, because the simulations of a program can be treated as abstract interpretations of the program, it is plausible that *all* of the simulations of a program can be determined to be race free simultaneously.

In order to state the determinism property formally, we must first build some additional technical machinery. For now, we state the noninterference result informally.

Theorem 5.4.1 ((Informal) Noninterference for $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$) *If N is well-typed and ζ-equivalent simulations are race free, then for any low-security memory location L and any two high-security values v_1 and v_2 the sequence of values stored in memory location*

L during the evaluation of $\langle \cdot, \cdot, N\{v_1/x\}\rangle$ is a prefix of the sequence of values stored in L by $\langle \cdot, \cdot, N\{v_2/x\}\rangle$ (or vice-versa).

At a high level, the proof strategy is similar to that used in $\lambda_{\text{SEC}}^{\text{CPS}}$. We first establish an appropriate notion of ζ-equivalence that equates values that should not be distinguishable to an observer with security clearance less-than ζ. We then use this equivalence to construct a faithful low-security simulation of the process; the relevant lemmas are the $\lambda_{\text{SEC}}^{\text{CONCUR}}$ analog of 4.4.1 and 4.4.3. Using these results, it we establish the noninterference result by induction on the evaluation sequence of the original program, using the fact that secure programs are race free.

5.4.1 ζ-equivalence for $\lambda_{\text{SEC}}^{\text{CONCUR}}$

We first define ζ-equivalence, which indicates when two values (or substitutions) look the same to a ζ-level observer.

Definition 5.4.1 (ζ-equivalence) *Let ζ-equivalence (written $\approx_\zeta$) be the family of symmetric binary relations inductively defined as follows.*

- *For values:*

$$H; \Gamma \models v_1 \approx_\zeta v_2 : t_\ell \quad \Leftrightarrow \quad H; \Gamma \vdash v_i : t_\ell \ \wedge\ (\ell \sqsubseteq \zeta \Rightarrow v_1 = v_2)$$

- *For linear values:*

$$T \models c_1 \approx_\zeta c_2 : k \quad \Leftrightarrow \quad T(c_i) = k \ \wedge\ c_1 = c_2$$

- *For nonlinear substitutions: $H \models \gamma_1 \approx_\zeta \gamma_2 : \Gamma$ iff*

$$H; \cdot \vdash \gamma_i \models \Gamma \ \wedge\ \forall x \in \mathrm{dom}(\Gamma).\ H; \cdot \models \gamma_1(x) \approx_\zeta \gamma_2(x) : \Gamma(x)$$

- *For linear substitutions: $T \models \sigma_1 \approx_\zeta \sigma_2 : K$ iff*

$$T \vdash \sigma_i \models K \ \wedge\ \forall y \in \mathrm{dom}(K).\ T \models \sigma_1(y) \approx_\zeta \sigma_2(y) : K(y)$$

Generalizing ζ-equivalence to processes is more involved for $\lambda_{\text{SEC}}^{\text{CONCUR}}$ than for $\lambda_{\text{SEC}}^{\text{CPS}}$. The problem is that because of concurrency, there can be both high-security and low-security computations running simultaneously, so relating corresponding parts of subprograms is more complicated. In particular, the approach used for $\lambda_{\text{SEC}}^{\text{CPS}}$, which required lock-step bisimulation for low transitions, is no longer appropriate in a concurrent setting. Rather than giving a bisimulation directly, we instead give a simulation $\lesssim_\zeta$

relation. Two programs are then ζ-equivalent if they can both be simulated by the same machine.

The simulation relation is induced by the typing structure of a source machine configuration. Intuitively, if $H; T \vdash m \ \lesssim_\zeta \ m'$ then configuration m' can simulate the low-security behavior of m while ignoring both the timing and termination behavior of the high-security computation in m.

Definition 5.4.2 (ζ-simulation) *Let ζ-approximation, written $\lesssim_\zeta$, be the relation (mutually) inductively defined as shown in Figures 5.19,5.20, 5.21, 5.22 and in the rules below.*

For configurations:

$$\textsc{Sim-Config}\quad \frac{\begin{array}{l} H \vdash M_1 \ \lesssim_\zeta \ M_2 \\ H; T_1, T_2 \vdash S_1 \ \lesssim_\zeta \ S_2, T_2 \\ H; \cdot; T_2; \cdot \vdash N_1 \ \lesssim_\zeta \ N_2 \end{array}}{H; T_1, T_2 \vdash \langle M_1,\, S_1,\, N_1 \rangle \ \lesssim_\zeta \ \langle M_2,\, S_2,\, N_2 \rangle}$$

For processes with $\mathsf{pc} \not\sqsubseteq \zeta$*:*

$$\textsc{Sim-High-Proc}\quad \frac{\mathsf{pc} \not\sqsubseteq \zeta \quad T(c_i) = (\vec{s_i}) \quad H; \Gamma \vdash \vec{v_i} : \vec{s_i}}{H; \Gamma; T; \cdot\, [\mathsf{pc}] \vdash P \ \lesssim_\zeta \ |_i \, c_i(\vec{v_i})}$$

For processes that are well-typed with a program counter $\sqsubseteq \zeta$, the $\lesssim_\zeta$ relationship acts homomorphically on the typing rule of the term, replacing the judgment $H; \Gamma \vdash v : s$ with the equivalence rule $H; \Gamma \vdash v_1 \approx_\zeta v_2 : s$ (and similarly for primitive operations). For example, the simulation for conditionals is derived from the typing rule $\lambda^{\text{CONCUR}}_{\text{SEC}}$-IF:

$$\textsc{Sim-If}\quad \frac{\begin{array}{l} \mathsf{pc} \sqsubseteq \zeta \\ H; \Gamma[\mathsf{pc}] \vdash v_1 \lesssim_\zeta v_2 : \mathtt{bool}_\ell \\ H; \Gamma; T; \cdot\, [\mathsf{pc} \sqcup \ell] \vdash P_{1i} \ \lesssim_\zeta \ P_{2i} \quad i \in \{1, 2\} \end{array}}{H; \Gamma; T; \cdot\, [\mathsf{pc}] \vdash \mathtt{if}\ v_1\ \mathtt{then}\ P_{11}\ \mathtt{else}\ P_{12} \ \lesssim_\zeta \ \mathtt{if}\ v_2\ \mathtt{then}\ P_{21}\ \mathtt{else}\ P_{22}}$$

The most important part of the $\lesssim_\zeta$ relation is rule SIM-HIGH-PROC. This rule says that any process that is well typed with a pc label not protected by ζ can be simulated by the process that just sends a response on each of the linear channels. Intuitively, this simulation bypasses all of the potential high-security computation performed in P and simply returns via the linear-channel invocations. Importantly, for a high-security process P such that $P \lesssim_\zeta P'$ the simulation P' always terminates, even if P does not. The simulation ignores the termination behavior of P.

Observe that all of the values returned from a high-security context via linear channels must themselves be high-security. (See the premise of rule $\lambda^{\text{CONCUR}}_{\text{SEC}}$-LINSEND

$\boxed{H; \Gamma[\mathsf{pc}] \vdash prim_1 \lesssim_\zeta prim_2 : s}$

$$\textsc{Sim-Val}\quad \frac{H; \Gamma \models v_1 \approx_\zeta v_2 : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{H; \Gamma[\mathsf{pc}] \vdash v_1 \lesssim_\zeta v_2 : s}$$

$$\textsc{Sim-Binop}\quad \frac{H; \Gamma \models v_{11} \approx_\zeta v_{12} : \mathtt{bool}_\ell \quad H; \Gamma \models v_{21} \approx_\zeta v_{22} : \mathtt{bool}_\ell \quad \mathsf{pc} \sqsubseteq \ell}{H; \Gamma[\mathsf{pc}] \vdash v_{11} \oplus v_{12} \lesssim_\zeta v_{21} \oplus v_{22} : \mathtt{bool}_\ell}$$

$$\textsc{Sim-Deref}\quad \frac{H; \Gamma \models v_1 \approx_\zeta v_2 : s\ \mathtt{ref}_\ell \qquad \mathsf{pc} \sqsubseteq \mathsf{label}(s \sqcup \ell)}{H; \Gamma[\mathsf{pc}] \vdash\ !v_1 \lesssim_\zeta\ !v_2 : s \sqcup \ell}$$

Figure 5.19: Primitive operation simulation relation

in Figure 5.14, which uses the value rule from Figure 5.12. $\lambda_{\text{SEC}}^{\text{CONCUR}}$-VAL requires a lower bound of pc for the label of the value.) Therefore, it does not matter what values are returned in the ζ-simulation because these values are not observable anyway.

Also note that if there are no linear channels in the context, SIM-HIGH-PROC says that $P \lesssim_\zeta \mathtt{0}$—the high-security process P has no way of affecting low-security memory locations, so from the low-security view, P may as well not exist.

In this setting the $\lesssim_\zeta$ relation is more fundamental and easier to work with than $\approx_\zeta$. However, two machine configurations are ζ-equivalent if they are both simulated by the same configuration.

Definition 5.4.3 (ζ-equivalence for configurations) *Configurations m_1 and m_2 are ζ-equivalent, written $H; T \models m_1 \approx_\zeta m_2$, if and only if there exists a configuration m such that $H; T \vdash m_1 \lesssim_\zeta m$ and $H; T \vdash m_2 \lesssim_\zeta m$.*

To prove that $\lambda_{\text{SEC}}^{\text{CONCUR}}$ satisfies noninterference, we follow a similar strategy to the proof for $\lambda_{\text{SEC}}^{\text{CPS}}$. We first establish some basic properties of the simulation relations and show that the simulations are faithful. Next, we prove the analogs of Lemmas 4.4.1 and 4.4.3, which show that the low-security parts of the computation are simulated correctly. Lastly, we combine these lemmas to show that altering the high-security inputs to the program does not affect its low-security deterministic behavior.

Lemma 5.4.1 (Simulation preserves typing) *If $H; T \vdash m$ and $H; T \vdash m \lesssim_\zeta m'$ then $H; T \vdash m'$.*

Proof: By induction on the derivation of $H; T \vdash m$; the inductive hypothesis must be extended to the other judgment forms. The one interesting case is SIM-HIGH-PROC,

$\boxed{H \vdash M_1 \lesssim_\zeta M_2}$

$$\textsc{Sim-Heap-Empty}\quad \frac{\begin{array}{l} H \vdash M \\ \forall L \in \mathrm{dom}(M).\mathsf{label}(H(L)) \not\sqsubseteq \zeta \\ \forall c \in \mathrm{dom}(M).\mathsf{label}(H(c)) \not\sqsubseteq \zeta \end{array}}{H \vdash M \lesssim_\zeta \cdot}$$

$$\textsc{Sim-Heap-Loc}\quad \frac{H \vdash M_1 \lesssim_\zeta M_2 \quad H \models v_1 \approx_\zeta v_2 : H(L)}{H \vdash M_1[L \mapsto v_1] \lesssim_\zeta M_2[L \mapsto v_2]}$$

$$\textsc{Sim-Heap-Handler}\quad \frac{\begin{array}{l} H \vdash M_1 \lesssim_\zeta M_2 \\ J \rightsquigarrow_{\mathsf{pc}} \langle \{c_i : H(c_i)\};\ \{\vec{x_i} : \vec{s_i}\};\ \{y_i^{\mathsf{opt}} : k_i^{\mathsf{opt}}\}\rangle \\ H(c_i) = [\mathsf{pc}](\vec{s_i}, k_i^{\mathsf{opt}}) \\ H; \vec{x_i} : \vec{s_i}; \cdot; \{y_i^{\mathsf{opt}} : k_i^{\mathsf{opt}}\}\ [\mathsf{pc}] \vdash P_1 \lesssim_\zeta P_2 \end{array}}{H \vdash M_1[J \rhd P_1] \lesssim_\zeta M_2[J \rhd P_2]}$$

Figure 5.20: Memory simulation relation

$\boxed{H; T_1 \vdash S_1 \lesssim_\zeta S_2, T_2}$

$$\textsc{Sim-S-Empty}\quad \frac{\begin{array}{l} H; T \vdash S; T' \\ \forall [\mathsf{pc} : J \multimap P] \in S.\mathsf{pc} \not\sqsubseteq \zeta \end{array}}{H; T \vdash S \lesssim_\zeta \cdot, T'}$$

$$\textsc{Sim-S-Handler}\quad \frac{\begin{array}{l} H; T \vdash S_1 \lesssim_\zeta S_2, T_1 \\ J \rightsquigarrow \langle \{c_i : T(c_i)\};\ \{\vec{x_i} : \vec{s_i}\}\rangle \\ T(c_i) = (\vec{s_i}) \quad T_2 \subseteq T \\ H; \vec{x_i} : \vec{s_i}; T_2; \cdot\ [\mathsf{pc}] \vdash P_1 \lesssim_\zeta P_2 \end{array}}{H; T \vdash S_1[\mathsf{pc} : J \multimap P_1] \lesssim_\zeta S_2[\mathsf{pc} : J \multimap P_2], T_1, T_2}$$

Figure 5.21: Synchronization environment simulation relation

$\boxed{H; \Gamma; T; K \vdash N_1 \ \lesssim_\zeta \ N_2}$

SIM-NET-EMPTY
$$H; \Gamma; \cdot; \cdot \vdash \cdot \ \lesssim_\zeta \ \cdot$$

SIM-NET-PROC
$$\frac{H; \Gamma; T_1; K_1 \vdash N \ \lesssim_\zeta \ N' \quad H; \Gamma; T_2; K_2 \ [\mathsf{pc}] \vdash P \ \lesssim_\zeta \ P'}{H; \Gamma; T_1, T_2; K_1, K_2 \vdash N \mid [\mathsf{pc} : P] \ \lesssim_\zeta \ N' \mid [\mathsf{pc} : P']}$$

SIM-NET-EQUIV
$$\frac{N_1 \equiv N_2 \quad H; \Gamma; T; K \vdash N_2 \ \lesssim_\zeta \ N_3 \quad N_3 \equiv N_4}{H; \Gamma; T; K \vdash N_1 \ \lesssim_\zeta \ N_4}$$

Figure 5.22: Network simulation relation

which holds because each free linear channel c_i mentioned in P is used exactly once in the simulation $\mid_i c_i(\vec{v_i})$. □

The following lemmas show that substitution of ζ-equivalent values preserves the simulation relation.

Lemma 5.4.2 ($\lesssim_\zeta$-Substitution) *If $H; \Gamma; T; K \ [\mathsf{pc}] \vdash P_1 \ \lesssim_\zeta \ P_2$ and $H \models \gamma_1 \approx_\zeta \gamma_2 : \Gamma$ then $H; \cdot; T; K \ [\mathsf{pc}] \vdash \gamma_1(P_1) \ \lesssim_\zeta \ \gamma_2(P_2)$.*

Proof: Easy induction on the derivation of the simulation relation. □

Lemma 5.4.3 ($\lesssim_\zeta$-Linear-Substitution) *Suppose that*

$$H; \Gamma; T_1; K \ [\mathsf{pc}] \vdash P_1 \ \lesssim_\zeta \ P_2 \quad \textit{and} \quad H; T_2 \models \sigma_1 \approx_\zeta \sigma_2 : K$$

Then it is the case that $H; \Gamma; T_1, T_2; \cdot \ [\mathsf{pc}] \vdash \sigma_1(P_1) \ \lesssim_\zeta \ \sigma_2(P_2)$.

Proof: Easy induction on the derivation of the simulation relation. □

The next lemma shows that evaluation of a primitive operation in two related configurations yields related results.

Lemma 5.4.4 (Primitive Simulation) *Suppose that*

$$H \vdash M_1 \ \lesssim_\zeta \ M_2 \textit{ and } H; \cdot [\mathsf{pc}] \vdash prim_1 \lesssim_\zeta prim_2 : s \textit{ and } M_i, \mathsf{pc} \models prim_i \Downarrow v_i$$

Then it is the case that $H; \cdot [\mathsf{pc}] \vdash v_1 \lesssim_\zeta v_2 : s$

Proof: By cases on the primitive operations involved. □

Now we establish that the simulation respects the operational semantics.

Lemma 5.4.5 ($\Rrightarrow$-Simulation) *Suppose that $H;T \models m_1 \lesssim_\zeta m_2$ and $m_1 \Rrightarrow m_1'$. Then either $m_1' \lesssim_\zeta m_2$ or there exists H', T', and m_2' such that $m_2 \Rrightarrow m_2'$ and, furthermore, $H', T' \models m_1' \lesssim_\zeta m_2'$.*

Proof: This result follows directly from the $\rightarrow$-Simulation lemma below and the rule SIM-NET-EQUIV. □

Because we want to show that high-security computation does not affect the low-security behavior, we need a way of distinguishing the high-security transition steps from the low-security ones.

Definition 5.4.4 (ζ-low and ζ-high evaluation steps) *Let configuration $m \rightarrow m'$ and let $[\mathsf{pc}_i : P_i]$ be the processes reduced during the single step, i.e.*

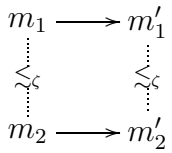
$$m \equiv \langle M,\, S,\, N \mid \mid_i [\mathsf{pc}_i : P_i] \rangle$$

where N and the $[\mathsf{pc}_i : P_i]$ processes define a redex according to Figure 5.4 or Figure 5.5.

*Then m is said to take a ζ-***low evaluation step** *if $(\bigsqcup_i \mathsf{pc}_i) \sqsubseteq \zeta$. Otherwise, m is said to take a ζ-* **high evaluation step**.

We next prove the analog of Lemma 4.4.1. It says that any ζ-low evaluation step performed by one configuration can also be performed by its simulation.

Lemma 5.4.6 (ζ-low simulation) *If $H;T \models m_1 \lesssim_\zeta m_2$ and $m_1 \rightarrow m_1'$ via a ζ-low evaluation step then there exists H', T', and m_2' such that $m_2 \rightarrow m_2'$ and, furthermore, $H', T' \models m_1' \lesssim_\zeta m_2'$. Pictorially, this requirement is:*



Proof: By cases on the evaluation step taken by m_1. Because the pc of each process involved in the redex is $\sqsubseteq \zeta$, the reduced processes in m_1 are homomorphic to some process in m_2. Therefore, most of the cases follow from straightforwardly unwinding the definitions and applying either primitive simulation (Lemma 5.4.4) or one of the substitution lemmas (Lemma 5.4.2 or 5.4.3).

The interesting cases are when the process resulting from the redex has $\mathsf{pc} \not\sqsubseteq \zeta$, which can occur when evaluating a conditional or invoking a nonlinear handler. A representative case considered in detail below; similar arguments holds for evaluation via rules $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-COND2 or $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-SEND.

$\lambda_{\text{SEC}}^{\text{CONCUR}}$-**EVAL-COND1** Then $m_1 = \langle M_1,\, S_1,\, N_1 \mid [\mathsf{pc} : \texttt{if}\ \texttt{t}_\ell\ \texttt{then}\ P_{11}\ \texttt{else}\ P_{12}]\rangle$. Because $\mathsf{pc} \sqsubseteq \zeta$, and $H; T \vdash m_1 \lesssim_\zeta m_2$, we must have

$$m_2 = \langle M_2,\, S_2,\, N_2 \mid [\mathsf{pc} : \texttt{if}\ v\ \texttt{then}\ P_{21}\ \texttt{else}\ P_{22}]\rangle$$

Furthermore, $m_1' = \langle M_1,\, S_1,\, N_1 \mid [\mathsf{pc} \sqcup \ell : P_{11}]\rangle$. If $\ell \sqsubseteq \zeta$, then $H; \cdot[\mathsf{pc}] \vdash \texttt{t}_\ell \lesssim_\zeta v : \texttt{bool}_\ell$ implies that $v = \texttt{t}_{\ell'}$ for some $\ell' \sqsubseteq \zeta$, but then m_2 transitions via rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-COND1 to $\langle M_2,\, S_2,\, N_2 \mid [\mathsf{pc} \sqcup \ell : P_{21}]\rangle$. Because $\mathsf{pc} \sqcup \ell \sqsubseteq \zeta$, the definition of $\lesssim_\zeta$ yields $H; \cdot; T_1; \cdot\ [\mathsf{pc} \sqcup \ell] \vdash P_{11} \lesssim_\zeta P_{21}$, so the result follows from SIM-NET-PROC and SIM-CONFIG.

The other case is when $\ell \not\sqsubseteq \zeta$. It follows from SIM-IF and SIM-HIGH-PROC that

$$H; \cdot; T_1; \cdot\ [\mathsf{pc} \sqcup \ell] \vdash P_j \lesssim_\zeta \mid_i c_i(\vec{v_i})$$

for $j \in \{1, 2\}$. Therefore even though m_2 may transition either via the rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-COND1 or via the rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-COND2, both transitions yield the same configuration

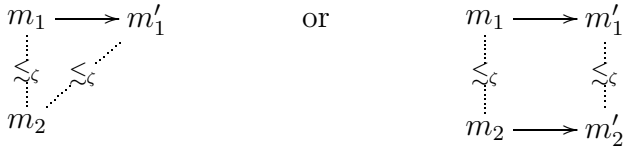
$$m_2' = \langle M_2,\, S_2,\, N_2 \mid [\mathsf{pc} \sqcup \ell : \mid_i c_i(\vec{v_i})]\rangle$$

In both cases, the resulting configurations satisfy $H; T \vdash m_1' \lesssim_\zeta m_2'$ as required.

□

The analog of Lemma 4.4.3 says that if a configuration transitions by a ζ-high step, then the transition can be simulated by zero or one steps. Rather than using some analog of the linear continuation ordering lemma, the SIM-HIGH-PROC rule builds the appropriate linear channel invocation into the simulation.

Lemma 5.4.7 (ζ-high simulation) *If $H; T \models m_1 \lesssim_\zeta m_2$ and $m_1 \to m_1'$ via a ζ-high evaluation step then either $m_1' \lesssim_\zeta m_2$ or there exists H', T', and m_2' such that $m_2 \to m_2'$ and $H', T' \models m_1' \lesssim_\zeta m_2'$. Pictorially, these requirements are:*



Proof: By cases on the transition step. In all cases except $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-SEND and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LINSEND there is only one process involved in the redex. In those cases, $m_1 = \langle M_1,\, S_1,\, N_1 \mid [\mathsf{pc} : P]\rangle$ and from the definition of SIM-NET-PROC we have that $m_2 \equiv \langle M_1,\, S_1,\, N_2 \mid [\mathsf{pc} : \mid_i c_i(\vec{v_i})]\rangle$ where the c_i are the free linear channels

occurring in P. It must be the case that $m_1' = \langle M_1', S_1', N_1 \mid [\mathsf{pc} : P'] \rangle$ for P' defined according to the transition relation. Observe that because the configuration is well-typed, $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-REF and $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-ASSN imply that M_1' and M_1 differ in only locations that have label $\not\sqsubseteq \zeta$. Similarly, S_1' may differ from S_1 only on linear handlers with $\mathsf{pc} \not\sqsubseteq \zeta$. Therefore, it is easy to establish that there is a H' and T' such that $H' \vdash M_1 \lesssim_\zeta M_1'$ and $H'; T' \vdash S_1 \lesssim_\zeta S_1', .$ Furthermore, subject reduction implies that the free linear channels of P' are the same as the free linear channels of P, so $H'; \cdot; T'; \cdot \; [\mathsf{pc}] \vdash P' \lesssim_\zeta \mid_i c_i(\vec{v_i})$ by SIM-HIGH-PROC. This is enough to establish that $H'; T' \vdash m_1' \lesssim_\zeta m_2$, as required.

The case for $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-SEND follows because the body of the handler is well-typed with $\mathsf{pc} \not\sqsubseteq \zeta$. Lemma 5.4.3 implies that the body is simulated by the corresponding sends on the linear channels. Thus, as above, we have $H; T \vdash m_1' \lesssim_\zeta m_2$ as needed.

Finally, consider the case for $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LINSEND. If the pc label of the target handler is $\not\sqsubseteq \zeta$, then the result follows exactly as above. Otherwise, it is the case that the program counter is being reset to some label $\sqsubseteq \zeta$. Note that because the synchronization environments satisfy $H; T \vdash S_1 \lesssim_\zeta S_2, T'$ it is possible for m_2 to step via $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-LINSEND. The resulting configuration m_2' satisfies $H; T' \vdash m_1' \lesssim_\zeta m_2'$ by construction of the $\lesssim_\zeta$ relation and an application of Lemma 5.4.2. □

Lemma 5.4.8 (→-simulation) *If $H; T \models m_1 \lesssim_\zeta m_2$ and $m_1 \to m_1'$ then either $m_1' \lesssim_\zeta m_2$ or there exists H', T', and m_2' such that $m_2 \to m_2'$ and $H', T' \models m_1' \lesssim_\zeta m_2'$.*

Proof: Follows immediately from the ζ-low and ζ-high simulation lemmas. □

Next, we formally specify the determinism condition. Intuitively, we are interested in establishing that during the evaluation of a secure program, the sequence of updates to a particular memory location L is deterministic.

First we define some useful notation that makes it easier to isolate memory updates to particular locations. For any configuration $m = \langle M, S, N \rangle$ and location $L \in \text{dom}(M)$ let $m(L) = M(L)$. Two configurations that are well-typed under memory type H are ζ-equivalent at L, written $m_1 \overset{L}{\sim}_\zeta m_2$, if they agree on the value stored at location L:

$$m_1 \overset{L}{\sim}_\zeta m_2 \quad \Leftrightarrow \quad H; \cdot \vdash m_1(L) \approx_\zeta m_2(L) : H(L)$$

Because we are interested in the determinism of writes to the low-security memory locations, it is helpful to separate the transition relations into those that affect a particular location and those that do not. Therefore we partition the $\Rrightarrow$ relation into two cases: transition $m \overset{L}{\Rrightarrow}_\zeta m'$ informally says that configuration m writes to location L; a transition $m \overset{\overline{L}}{\Rrightarrow}_\zeta m'$ says that the transition does not affect location L. Formally, we define

these transition relations using the $\approx_\zeta$ relation. Treating these relations as sets of pairs, we have:

$$\overset{L}{\Rrightarrow}_\zeta \overset{\text{def}}{=} \Rrightarrow \cap \overset{L}{\not\sim}_\zeta$$
$$\overset{\overline{L}}{\Rrightarrow}_\zeta \overset{\text{def}}{=} (\Rrightarrow \cap \overset{L}{\sim}_\zeta) \cup \equiv$$

Now we establish some properties relating the $\overset{L}{\Rrightarrow}_\zeta$ relation to race conditions.

Lemma 5.4.9 (L-transitions write to L) *If $m_1 \overset{L}{\Rrightarrow}_\zeta m_2$ then $m_1 \equiv \langle M,\, S,\, N \mid [\mathtt{pc} : \mathtt{set}\ L := v\ \mathtt{in}\ P] \rangle$.*

Proof: By definition, the memory location L must change. By inspection of the operational semantics, we see that m_1 must contain an assignment to location L. □

Lemma 5.4.10 (Two writes) *If $m_1 \overset{L}{\Rrightarrow}_\zeta m_2$ and $m_1 \overset{L}{\Rrightarrow}_\zeta m_3$ and $m_2 \overset{L}{\not\sim}_\zeta m_3$ then*

$$m_1 \equiv \langle M,\, S,\, N \mid [\mathtt{pc} : \mathtt{set}\ L := v\ \mathtt{in}\ P] \mid [\mathtt{pc} : \mathtt{set}\ L := v'\ \mathtt{in}\ P'] \rangle$$

where $H; \cdot \vdash v \not\approx_\zeta v'$.

Proof: By two applications of the lemma above, we see that m_1 must contain two distinct assignments. By inspection of the syntax and the definition of $\equiv$, the only way this may occur is if m_1 has the form required. □

The following lemma says that race freedom implies that there are no write–write conflicts—the next update of any location L is deterministic.

Lemma 5.4.11 (Write–write race) *If m is race free then it is not the case that there exists m', m_1, m_2 such that $m \Rrightarrow^* m'$ and $m' \Rrightarrow m_1$ and $m' \Rrightarrow m_2$ and $m_1 \overset{L}{\not\sim}_\zeta m_2$.*

Proof: Note that because $m_1 \overset{L}{\not\sim}_\zeta m_2$ it must be the case that $m_1 \not\equiv m_2$. Hence the definition of race free implies that there exists m_3 such that $m_1 \Rrightarrow m_3$ and $m_2 \Rrightarrow m_3$. Observe that because $m_3 \overset{L}{\sim}_\zeta m_3$ it must be the case that the transitions from m_1 and m_2 both assign ζ-equivalent values to L. Therefore, there exist $v_1 \approx_\zeta v_2$ such that:

$$m_1 \equiv \langle M_1,\, S_1,\, N_1 \mid [\mathtt{pc}_1 : \mathtt{set}\ L := v_1\ \mathtt{in}\ P_1] \rangle$$
$$m_2 \equiv \langle M_2,\, S_2,\, N_2 \mid [\mathtt{pc}_2 : \mathtt{set}\ L := v_2\ \mathtt{in}\ P_2] \rangle$$

By applying Lemma 5.4.10 to configuration m', we also have that there exist $v_1' \not\approx_\zeta v_2'$ such that:

$$m' \equiv \langle M',\, S',\, N' \mid [\mathtt{pc} : \mathtt{set}\ L := v_1'\ \mathtt{in}\ Q_1] \mid [\mathtt{pc} : \mathtt{set}\ L := v_2'\ \mathtt{in}\ Q_2] \rangle$$

Because the evaluation rule $\lambda_{\text{SEC}}^{\text{CONCUR}}$-EVAL-SET involves the reduction of only one network term, it must be the case that N_1 contains the assignment $\texttt{set}\ L := v_2'\ \texttt{in}\ Q_2$ and N_2 contains the assignment $\texttt{set}\ L := v_1'\ \texttt{in}\ Q_1$. It follows that $N_1 \not\equiv N_2$ because they both evolved from the same configuration. But this yields a contradiction, because $m_3 \equiv \langle M_3,\, S_3,\, N_1 \mid P_1 \rangle$ and $m_3 \equiv \langle M_3,\, S_3,\, N_2 \mid P_2 \rangle$, which is impossible when $N_1 \not\equiv N_2$ (even if $P_1 \equiv P_2$). □

Next we establish two key technical lemmas that show how $\overset{L}{\Rightarrow}_\zeta$ and $\overset{\overline{L}}{\Rightarrow}_\zeta$ transitions interact.

Lemma 5.4.12 ($\overline{L}$–$\overline{L}$ square) *Suppose that* $m \overset{\overline{L}}{\Rightarrow}_\zeta m_1$ *and* $m \overset{\overline{L}}{\Rightarrow}_\zeta m_2$ *with* $m_1 \not\equiv m_2$. *If there exists* m_3 *such that* $m_1 \Rrightarrow m_3$ *and* $m_2 \Rrightarrow m_3$ *then* $m_1 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$ *and* $m_2 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$. *Pictorially, this situation is:*

Proof: Because $m_1 \not\equiv m_2$, the configurations must differ in some respect. Note that since $m \overset{L}{\sim}_\zeta m$ and m_1 and m_2 are reached from m by a transition that does not affect location L, it must be the case that $m \overset{L}{\sim}_\zeta m_1$ and $m \overset{L}{\sim}_\zeta m_2$. It is not possible that $m_1 \overset{L}{\Rightarrow}_\zeta m_3$ and $m_2 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$ because then, by definition of $\overset{L}{\Rightarrow}_\zeta$ and $\overset{\overline{L}}{\Rightarrow}_\zeta$, we would have this contradiction:

$$m \overset{L}{\sim}_\zeta m_1 \overset{L}{\not\sim}_\zeta m_3 \overset{L}{\sim}_\zeta m_2 \overset{L}{\sim}_\zeta m$$

Similarly, it is not possible that $m_1 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$ and $m_2 \overset{L}{\Rightarrow}_\zeta m_3$. Finally, reasoning similar to that used in Lemma 5.4.11 shows that $m_1 \overset{L}{\Rightarrow}_\zeta m_3'$ and $m_2 \overset{L}{\Rightarrow}_\zeta m_3''$ implies $m_3' \not\equiv m_3''$. Therefore, the only remaining possibility is for $m_1 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$ and $m_2 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$, as required. □

Lemma 5.4.13 (L–$\overline{L}$ square) *Suppose that* $m \overset{L}{\Rightarrow}_\zeta m_1$ *and* $m \overset{\overline{L}}{\Rightarrow}_\zeta m_2$ *with* $m_1 \not\equiv m_2$. *If there exists* m_3 *such that* $m_1 \Rrightarrow m_3$ *and* $m_2 \Rrightarrow m_3$ *then* $m_1 \overset{\overline{L}}{\Rightarrow}_\zeta m_3$ *and* $m_2 \overset{L}{\Rightarrow}_\zeta m_3$. *Pictorially, this situation is:*

Proof: This proof is similar to that of 5.4.12, so we sketch it only briefly. Note that by definition $m \overset{L}{\not\sim}_\zeta m_1$ and $m \overset{L}{\sim}_\zeta m_2$. Therefore, to reach a common m_3, at least one of m_1 and m_2 must perform a write to L. It can't be m_1 because then reasoning similar to that in 5.4.11 yields a contradiction. Therefore m_2 must perform a write to L. □

Lemma 5.4.14 *If $m_{(0,0)}$ is race free and*

$$m_{(0,0)} \ (\overset{\overline{L}}{\Rightarrow}_\zeta)^i \ m_{(i,0)} \ \overset{L}{\Rightarrow}_\zeta \ m_{(i+1,0)}$$

and $m_{(0,0)} \overset{\overline{L}}{\Rightarrow}_\zeta m_{(0,1)}$ then there exists a sequence of configurations $m_{(j,1)}$ for $0 \leq j \leq i+1$ such that for all $0 \leq k < i$ either $m_{(k,1)} \equiv m_{(k+1,1)}$ or $m_{(k,1)} \overset{\overline{L}}{\Rightarrow}_\zeta m_{(k+1,1)}$ and $m_{(i,1)} \overset{L}{\Rightarrow}_\zeta m_{(i+1,1)}$, where $m_{(i+1,1)} \overset{L}{\sim}_\zeta m(i+1,0)$. Pictorially, this lemma says that the following diagram:

$$\begin{array}{l} m_{(0,0)} \cdots\overset{\overline{L}}{\Rightarrow}_\zeta \cdots m_{(1,0)} \cdots(\overset{\overline{L}}{\Rightarrow}_\zeta)^{(i-1)}\cdots m_{(i,0)} \cdots\overset{L}{\Rightarrow}_\zeta \cdots m_{(i+1,0)} \\ \quad\vdots \\ \quad\overset{\overline{L}}{\Rightarrow}_\zeta \\ \quad\vdots \\ m_{(0,1)} \end{array}$$

can be completed to a diagram or the form:

$$\begin{array}{ccccccccc} m_{(0,0)} & \cdots\overset{\overline{L}}{\Rightarrow}_\zeta \cdots & m_{(1,0)} & \cdots(\overset{\overline{L}}{\Rightarrow}_\zeta)^{(i-1)}\cdots & m_{(i,0)} & \cdots\overset{L}{\Rightarrow}_\zeta \cdots & m_{(i+1,0)} \\ \vdots & & \vdots & & \vdots & & \vdots \\ \overset{\overline{L}}{\Rightarrow}_\zeta & & \overset{\overline{L}}{\Rightarrow}_\zeta & & \overset{\overline{L}}{\Rightarrow}_\zeta & & \overset{\overline{L}}{\Rightarrow}_\zeta \\ \vdots & & \vdots & & \vdots & & \vdots \\ m_{(0,1)} & \cdots\overset{\overline{L}}{\Rightarrow}_\zeta \cdots & m_{(1,1)} & \cdots(\overset{\overline{L}}{\Rightarrow}_\zeta)^{(i-1)}\cdots & m_{(i,1)} & \cdots\overset{L}{\Rightarrow}_\zeta \cdots & m_{(i+1,1)} \end{array}$$

Proof: *By induction on i. The base case, for $i = 0$ follows directly from Lemma 5.4.13. The induction step follows because $m_{(0,0)}$ is race free, using Lemma 5.4.12 to complete the first square in the diagram.* □

The following lemma is crucial to establishing the determinism of secure programs.

Lemma 5.4.15 (Race freedom implies determinism) *Suppose that both*

$$m_{(0,0)} \ (\overset{\overline{L}}{\Rightarrow}_\zeta)^i \ m_{(i,0)} \ \overset{L}{\Rightarrow}_\zeta \ m_{(i+1,0)}$$

and

$$m_{(0,0)} \ (\overset{\overline{L}}{\Rightarrow}_\zeta)^j \ m_{(0,j)} \ \overset{L}{\Rightarrow}_\zeta \ m_{(0,j+1)}$$

then $m_{(i+1,0)} \overset{L}{\sim}_\zeta m_{(0,j+1)}$.

Proof: By induction on (i, j). For the base case, $(i, j) = (0, 0)$ we have the following diagram:

$$\begin{array}{ccc} m_{(0,0)} & \overset{L}{\Rightarrow_\zeta} & m_{(1,0)} \\ \overset{L}{\Rightarrow_\zeta} & & \\ m_{(0,1)} & & \end{array}$$

Applying Lemma 5.4.11, we obtain $m_{(1,0)} \overset{L}{\sim_\zeta} m_{(0,1)}$ as desired.

For the induction step, we have $i > 0$ or $j > 0$. Without loss of generality, assume that $i > 0$ (the case for $j > 0$ is symmetric). In the case that $j = 0$ we have the following diagram:

$$\begin{array}{ccccccc} m_{(0,0)} & \overset{\overline{L}}{\Rightarrow_\zeta} & m_{(1,0)} & \overset{\overline{L}}{(\Rightarrow_\zeta)^{(i-1)}} & m_{(i,0)} & \overset{L}{\Rightarrow_\zeta} & m_{(i+1,0)} \\ \overset{L}{\Rightarrow_\zeta} & & & & & & \\ m_{(0,1)} & & & & & & \end{array}$$

It must be the case that $m_{(1,0)} \overset{L}{\not\sim_\zeta} m_{(0,1)}$, which implies that $m_{(1,0)} \not\equiv m_{(0,1)}$. Therefore, because $m_{(0,0)}$ is race free, there must exist a configuration $m_{(1,1)}$ such that $m_{(1,0)} \Rrightarrow m_{(1,1)}$ and $m_{(0,1)} \Rrightarrow m_(1, 1)$. So, by Lemma 5.4.13 we can complete the diagram above to:

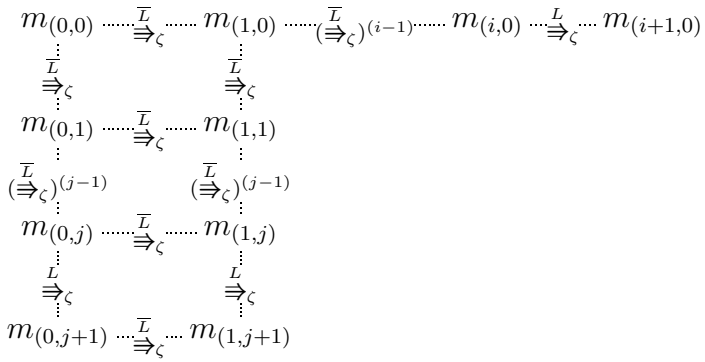
$$\begin{array}{ccccccc} m_{(0,0)} & \overset{\overline{L}}{\Rightarrow_\zeta} & m_{(1,0)} & \overset{\overline{L}}{(\Rightarrow_\zeta)^{(i-1)}} & m_{(i,0)} & \overset{L}{\Rightarrow_\zeta} & m_{(i+1,0)} \\ \overset{L}{\Rightarrow_\zeta} & & \overset{L}{\Rightarrow_\zeta} & & & & \\ m_{(0,1)} & \overset{\overline{L}}{\Rightarrow_\zeta} & m_{(1,1)} & & & & \end{array}$$

Note that $m_{(0,1)} \overset{L}{\sim_\zeta} m_{(1,1)}$. Finally, the induction hypothesis applies to the configuration $m_{(1,0)}$ because it is race free. Therefore, we obtain $m_{(0,1)} \overset{L}{\sim_\zeta} m_{(1,1)} \overset{L}{\sim_\zeta} m_{(i+1,0)}$ as required.

Now suppose that $j > 0$. Then we have the following diagram:

$$\begin{array}{ccccccc} m_{(0,0)} & \overset{\overline{L}}{\Rightarrow_\zeta} & m_{(1,0)} & \overset{\overline{L}}{(\Rightarrow_\zeta)^{(i-1)}} & m_{(i,0)} & \overset{L}{\Rightarrow_\zeta} & m_{(i+1,0)} \\ \overset{\overline{L}}{\Rightarrow_\zeta} & & & & & & \\ m_{(0,1)} & & & & & & \\ \overset{\overline{L}}{(\Rightarrow_\zeta)^{(j-1)}} & & & & & & \\ m_{(0,j)} & & & & & & \\ \overset{L}{\Rightarrow_\zeta} & & & & & & \\ m_{(0,j+1)} & & & & & & \end{array}$$

By Lemma 5.4.14 we can complete the diagram above to:



Note that $m_{(0,j+1)} \overset{L}{\sim}_\zeta m_{(1,j+1)}$ and that $m_{(1,0)}$ is race free. Therefore, we use the induction hypothesis applied to $m_{(1,0)}$ to obtain that $m_{(1,j+1)} \overset{L}{\sim}_\zeta m_{(i+1,0)}$ from which we conclude $m_{(0,j+1)} \overset{L}{\sim}_\zeta m_{(i+1,0)}$ as needed.

□

The following lemma says that starting from a configuration with an open network and closing the network under similar substitutions yields $\approx_\zeta$ configurations. It lets us establish that two programs that differ only in their high-security inputs can be simulated by the *same* low-simulation. The noninterference theorem, proved next, uses the shared simulation, together with the race-freedom requirement to show that related programs update memory deterministically and identically.

Lemma 5.4.16 (Simulations and High-substitution) *Suppose the following:*

- $H \vdash M$
- $H; T_1, T_2 \vdash S; T_1$
- $H; \Gamma; T_2; \cdot \vdash N$
- $H \models \gamma_1 \approx_\zeta \gamma_2 : \Gamma$
- $\forall x \in \mathrm{dom}(\Gamma).\ \mathsf{label}(\Gamma(x)) \not\sqsubseteq \zeta$

then for any configuration m, $H; T \models \langle M, S, \gamma_1(N)\rangle \lesssim_\zeta m$ implies that $H; T \models \langle M, S, \gamma_2(N)\rangle \lesssim_\zeta m$.

Proof: By an easy induction on the typing derivation for network N. The base case for values follows from rule SIM-VAL and the requirement that $\mathsf{label}(\Gamma(x)) \not\sqsubseteq \zeta$. The case for primitive operations follows from the fact that if the primitive operation involves a

variable in Γ, then its result is $\not\sqsubseteq \zeta$. The inductive cases follow from the construction of the $\lesssim_\zeta$ relation. $\square$

Finally, we can prove noninterference for $\lambda_{\text{SEC}}^{\text{CONCUR}}$.

Theorem 5.4.2 (Noninterference for $\lambda_{\text{SEC}}^{\text{CONCUR}}$) *Let ζ be an arbitrary label in the security lattice. Suppose that $H; x : s; \cdot; \cdot \vdash N$ is derivable. Let an initial memory M be given such that $H \vdash M$ and suppose that whenever $H; \cdot \vdash \langle M, \cdot, N\rangle \lesssim_\zeta m$ the simulation m is race free. Let location $L \in \text{dom}(H)$ be given such that $\mathsf{label}(H(L)) \sqsubseteq \zeta$. Further suppose that $\mathsf{label}(s) \not\sqsubseteq \zeta$. Then for any two values v_1 and v_2 such that $H; \cdot \vdash v_i : s$ the sequence of values stored in memory location L during the evaluation of $\langle M, \cdot, N\{v_1/x\}\rangle$ is a prefix of the sequence of values stored in L by $\langle M, \cdot, N\{v_2/x\}\rangle$ (or vice-versa).*

Proof: Let $m_{(1,0)} = \langle M, \cdot, N\{v_1/x\}\rangle$ and $m_{(2,0)} = \langle M, \cdot, N\{v_2/x\}\rangle$. Note that by Lemma 5.4.16 there exists a configuration m' such that $H; \cdot \vdash m_1 \lesssim_\zeta m'$ and $H; \cdot \vdash m_2 \lesssim_\zeta m'$. Furthermore, note that m' must be race free. Suppose, for the sake of contradiction, that the evaluations disagree on the $n + 1^{st}$ update to the location L.

We derive a contradiction by induction on n. For the base case, $n = 0$ and there must exist evaluation sequences:

$$m_{(1,0)} \ (\overset{\overline{L}}{\Rrightarrow}_\zeta)^i \ m_{(1,i)} \ \overset{L}{\Rrightarrow}_\zeta \ m_{(1,i+1)}$$

$$m_{(2,0)} \ (\overset{\overline{L}}{\Rrightarrow}_\zeta)^j \ m_{(2,i)} \ \overset{L}{\Rrightarrow}_\zeta \ m_{(2,j+1)}$$

where $m_{(1,i+1)} \overset{L}{\not\sim}_\zeta m_{(2,j+1)}$. However, by induction on the lengths of these sequences applying Lemma 5.4.5, we obtain the following simulations:

$$m' \equiv m'_{(1,0)} \ (\overset{\overline{L}}{\Rrightarrow}_\zeta)^i \ m'_{(1,i)} \ \overset{L}{\Rrightarrow}_\zeta \ m'_{(1,i+1)}$$

$$m' \equiv m'_{(2,0)} \ (\overset{\overline{L}}{\Rrightarrow}_\zeta)^j \ m'_{(2,i)} \ \overset{L}{\Rrightarrow}_\zeta \ m'_{(2,j+1)}$$

Because the location L is visible at ζ, it must be that $m_{(1,i+1)} \overset{L}{\sim}_\zeta m'_{(1,i+1)}$ and $m_{(2,j+1)} \overset{L}{\sim}_\zeta m'_{(2,j+1)}$. We may also apply Lemma 5.4.15 to conclude that $m'_{(1,i+1)} \overset{L}{\sim}_\zeta m'_{(2,j+1)}$, but that is a contradiction.

The inductive step follows similarly to the inductive step of Lemma 5.4.5. $\square$

5.5 Related work

A few researchers have investigated noninterference-based type systems for concurrent languages and process calculi. Smith and Volpano have studied multithreaded programs, although they assumed a fixed number of threads and a uniform thread scheduler [SV98]. Their subsequent work refines the original type system to account for probabilistic thread scheduling and to relax the constraints due to timing channels [SV00, VS00, Smi01].

Roscoe [Ros95] was the first to propose a determinism-based definition of noninterference for labeled-transition systems. This approach has not been used previously in type systems for programming languages.

Focardi and Gorrieri [FG97] have implemented a flow-checker for a variant of Milner's calculus of concurrent systems (CCS). Honda, Vasoncelos, and Yoshida have proposed a similar system for the π-calculus in which they can faithfully encode Smith and Volpano's language [HVY00, HY02]. Their work relies on a sophisticated type system that distinguishes between linear channels, affine channels, and nonlinear channels while also tracking information about stateful computations. Both of these approaches use techniques of bisimulation to prove noninterference properties. A similar approach is taken by Abadi and Gordon to prove the correctness of cryptographic protocols in the Secure Pi Calculus [AG99], but the security policies they enforce are not information-flow policies.

Hennessy and Riely consider information-flow properties in the asynchronous pi-calculus [Hen00, HR00]. Their may-testing definition of noninterference is quite similar to the definition used here, because it is timing and termination insensitive. However, their language does not support synchronous communication or refinement of the information-flow analysis via linearity constraints.

Pottier [Pot02] gives an elementary proof of noninterference for a variant of the pi-calculus, but its type system is quite restrictive because it does not make a distinction between linear and nonlinear channel usage. Pottier observes that bisimulation-based definitions of noninterference give stronger security guarantees than those based on may-testing (i.e. [Hen00]) in the presence of race conditions. However, as described in this thesis, it is not clear that permitting races is desirable, so the additional constraints imposed by a type system that permits races may not be warranted.

Sabelfeld and Sands have considered concurrent languages in a probabilistic setting [SS00]. They use a novel probabilistic bisimulation approach to specify noninterference properties, and have used the techniques to prove correct Agat's program transformation for eliminating timing channels [Aga00]. Mantel and Sabelfeld have also considered type systems for multithreaded secure languages [MS01].

Reitman was among the earliest to consider message-passing primitives and their impact on information flows [Rei78]. However, there were no correctness proofs es-

tablished for his security logic. Banâtre, Bryce, and Le Metáyer [BBL84] give a static analysis for discovering information flows in a nondeterministic language, but their approach appears to be unsound (see the discussion in [VSI96]).

The design of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ was inspired by concurrent process calculi such as the pi calculus [MPW92] and especially the join calculus [FG96].

Chapter 6

Downgrading

Security properties based on information flow, such as the noninterference policy considered to this point in this thesis, provide strong guarantees that confidentiality and integrity are maintained. However, programs often need to leak some amount of confidential information in order to serve their intended purpose. Consider these examples:

- A secure e-mail reader might release encrypted confidential mail.
- The password-checking function of an operating system operates on confidential passwords, but granting or denying access leaks some information about the correct password.
- An on-line auction program might release the value of the winning bid after all secret bids have been made.

Similarly, programs often need to assert the integrity of a piece of data. For instance, after verifying a check sum or a digital signature a program might wish to consider a piece of data to be more trustworthy.

Consequently, realistic systems include a means of *downgrading*—allowing the security label of the data to be shifted downwards in the security lattice. For confidentiality, this process is called *declassification*; for integrity, it is called *endorsement*. The ability to escape from the strict confines of noninterference is both essential and dangerous: unregulated use of downgrading can easily result in unexpected release of confidential information or in corruption of supposedly trustworthy data.

To see the problem, first consider the simplest way to add a declassification operation to a security-typed language. We extend the syntax:

$$e ::= \ldots \mid \texttt{let}\ x = \texttt{declassify}(v, \ell)\ \texttt{in}\ e$$

and add the following typing judgment

$$\textsc{Bad-Declassify} \qquad \frac{\Gamma \vdash v : t_{\ell'} \quad \Gamma, x{:}t_\ell\ [\mathsf{pc}] \vdash e}{\Gamma\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{declassify}(v, \ell)\ \mathtt{in}\ e}$$

This judgment says that a value v with an arbitrary label can be given any other arbitrary label by declassification. This clearly breaks noninterference because high-security data can now be made low-security. Declassification is intended for this purpose, but this rule is too permissive—it can be used at any point to release confidential information. Consequently, adding such a rule to the languages studied in this thesis completely invalidates their noninterference theorems. We get no guarantees about the security of programs that use declassification, and the program may as well have been written without security types.

Because it is potentially dangerous, downgrading should only be used in certain, well-defined ways. One could imagine generalizing information-flow security policies to include specifications of exactly under what circumstances declassification or endorsement may occur. The problem with such an approach is that establishing that a given program meets the specifications of the security policy can be extremely difficult: It is the problem of proving that a program meets an arbitrary specification. Moreover, even stating these formal specifications of security policies is hard.

The noninterference policies specified using the lattice model for labels approximate the information flows in the program to avoid the difficulty of doing full-scale program verification. The next section describes Myers' and Liskov's *decentralized label model*, a particular security lattice designed to help govern the use of declassification operations. It avoids the full verification problem by introducing the notion of *authority*, which allows coarse-grained control over where declassifications may be used.

There is still a problem with regulating downgrading, even with the authority model. The last section of this chapter describes the problem and proposes a solution in which downgrading operations tie together integrity and confidentiality constraints.

6.1 The decentralized label model

The *decentralized label model* (DLM) proposed by Myers and Liskov [ML00] adds additional structure to the security lattice in order to regulate how declassification is used by a program.

Central to the model is the notion of a *principal*, which is an entity (e.g., user, process, party) that can have a confidentiality or integrity concern with respect to data. Principals can be named in information-flow policies and are also used to define the *authority* possessed by the running program. The authority A at a point in the program is a set of principals that are assumed to authorize any action taken by the program at

that point—in particular, principals may authorize declassifications of data. Different program points may have different authority, which must be explicitly granted by the principals in question.

A simple confidentiality label in this model is written $\{\mathtt{o}:\mathtt{r}_1,\mathtt{r}_2,\ldots,\mathtt{r}_n\}$, meaning that the labeled data is owned by principal o, and that o permits the data to be read by principals $\mathtt{r}_1$ through $\mathtt{r}_n$ (and, implicitly, o).

Data may have multiple owners, each controlling a different component of its label. For example, the label $\{\mathtt{o}_1:\mathtt{r}_1,\mathtt{r}_2;\ \mathtt{o}_2:\mathtt{r}_1,\mathtt{r}_3\}$, contains two components and says that owner $\mathtt{o}_1$ allows readers $\mathtt{r}_1$ and $\mathtt{r}_2$ and owner $\mathtt{o}_2$ allows readers $\mathtt{r}_1$ and $\mathtt{r}_3$. The interpretation is that *all* of the policies described by a label must be obeyed, only $\mathtt{r}_1$ will be able to read data with this annotation. Such composite labels arise naturally in collaborative computations: for example, if x has label $\{\mathtt{o}_1:\mathtt{r}_1,\mathtt{r}_2\}$ and y has label $\{\mathtt{o}_2:\mathtt{r}_1,\mathtt{r}_3\}$, then the sum x + y has the composite label $\mathtt{int}\{\mathtt{o}_1:\mathtt{r}_1,\mathtt{r}_2;\ \mathtt{o}_2:\mathtt{r}_1,\mathtt{r}_3\}$, which expresses the conservative requirement that the sum is subject to both the policy on x and the policy on y.

In the lattice, $\ell_1 \sqsubseteq \ell_2$ if the label ℓ_1 is less restrictive than the label ℓ_2. Intuitively, data with label ℓ_1 is less confidential than data with label ℓ_2—more principals are permitted to see the data, and, consequently, there are fewer restrictions on how data with label ℓ_1 may be used. For example, $\{\mathtt{o}:\mathtt{r}\} \sqsubseteq \{\mathtt{o}:\}$ holds because the left label allows both o and r to read the data, whereas the right label admits only o as a reader.

The formal definition of $\sqsubseteq$ for the decentralized label model is given in Myers' thesis [Mye99]. His thesis also shows that the relation $\sqsubseteq$ is a pre-order whose equivalence classes form a distributive lattice. The label join operation combines the restrictions on how data may be used. As an example, if x has label $\{\mathtt{o}:\mathtt{r}_1,\mathtt{r}_2\}$ and y has label $\{\mathtt{o}:\mathtt{r}_1,\mathtt{r}_3\}$, the sum x + y has label $\{\mathtt{o}:\mathtt{r}_1\}$, which includes the restrictions of both.

In this thesis, the decentralized label model is extended with label components that specify simple integrity constraints. The label $\{\mathtt{?}:\mathtt{p}_1,\ldots,\mathtt{p}_n\}$ specifies that principals $\mathtt{p}_1$ through $\mathtt{p}_n$ *trust* the data—they believe the data to be computed by the program as written. (Because integrity policies have no owner, a question mark is used in its place.) Note that the integrity label $\{\mathtt{?}:\}$ specifies a piece of data trusted by no principals; it is the label of completely untrusted data.

This is a weak notion of trust; its purpose is to protect security-critical information from damage by subverted hosts. Labels combining integrity and confidentiality components also arise naturally.

For any DLM label ℓ, the functions $C(\ell)$ and $I(\ell)$ extract the confidentiality and integrity parts of L, respectively. Because confidentiality and integrity are duals (see the discussion in Section 2.1), if $\ell_1 \sqsubseteq \ell_2$, then ℓ_2 must specify at least as much confidentiality and at *most* as much integrity as ℓ_1. This interpretation is consistent with the idea that labels represent restrictions on how data may be used; data with higher integrity has *fewer* restrictions on its use.

To emphasize that security labels are drawn from the decentralized label model, as opposed to some unspecified lattice, we shall sometimes refer to them as *DLM labels*.

We can now consider how the concept of authority helps control declassification in the decentralized label model. Consider the expression `declassify(e, ℓ)`. It allows a program acting with sufficient authority to declassify the expression `e` to label ℓ. A principal `p`'s authority is needed to perform declassifications of data owned by `p`. For example, owner `o` can add a reader `r` to a piece of data `x` by declassifying its label from `{o:}` to `{o:r}` using the expression `declassify(x, {o:r})`.

The rule for declassification in the decentralized label model is:

$$\text{DLM-DECLASSIFY} \qquad \frac{A, \Gamma \vdash v : t_{\ell'} \quad A, \Gamma, x{:}t_\ell\ [\mathsf{pc}] \vdash e \quad \mathsf{auth}(\ell, \ell') \subseteq A}{A, \Gamma\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{declassify}(v, \ell)\ \mathtt{in}\ e}$$

Here, the function $\mathsf{auth}(\ell, \ell')$ returns the set of principals whose policies are weakened by moving from label ℓ' down to label ℓ in the lattice. For example:

$$\mathsf{auth}(\{\mathtt{o:}\}, \{\mathtt{o:r}\}) = \{\mathtt{o}\}$$

The authority A is a set of principals associated with this point of the program, and is introduced at function boundaries. Therefore, the typing rule for functions is:

$$\text{DLM-FUN} \qquad \frac{A', \Gamma, x{:}s'\ [\mathsf{pc}'] \vdash e : s}{A, \Gamma\ [\mathsf{pc}] \vdash \lambda[A', \mathsf{pc}'](x{:}s').\ e : [A', \mathsf{pc}']s' \to s}$$

The function type $[A, \mathsf{pc}']s' \to s$ indicates that the function has authority A and so may perform declassifications on behalf of the principals in A. The programmer can delimit where declassifications may take place by constraining the authority available to a given function.

The caller of a function must establish that it has the authority necessary to carry out any of the declassifications that might occur inside the function call. This requirement is reflected in the function application rule:

$$\text{DLM-APP} \qquad \frac{\begin{array}{l} A, \Gamma[\mathsf{pc}] \vdash e : [A', \mathsf{pc}']s' \to s \\ A, \Gamma, [\mathsf{pc}] \vdash e' : s' \\ A' \subseteq A \quad \mathsf{pc} \sqsubseteq \mathsf{pc}' \end{array}}{A', \Gamma\ [\mathsf{pc}] \vdash e\ e' : s}$$

6.2 Robust declassification

Despite the increased control of downgrading offered by the decentralized label model, there is a weakness in its simple, authority-based approach. The problem is illustrated in Figure 6.1.

```
int{root:} secret = ...;   // compute the secret
let check = λ[{root},⊥](x:int{?:}).
   if x then print(declassify(secret, {}));
        else skip;
int x{?:} = Network.read();
  check(x)
```

Figure 6.1: The need for robust declassification

The program in the figure contains a function named `check` that tests its argument `x` and uses its value to decide whether to release the secret, using the authority of the principal `root`. Note that the privacy component of the label is declassified from `{root:}` to `{}`—the most public DLM label. The problem is that the value `x`, used to regulate the declassification, is completely untrusted by the `root` principal. This situation is exacerbated in the distributed setting discussed in the next section, because the computation that determines whether declassification should take place (the `if x` part) can potentially reside on a different host than the actual declassification itself.

Rather than give authority to the entire function body, it seems more natural to associate the required authority with the *decision* to perform the declassification. The program counter at the point of a `declassify` expression is already a model of the information used to reach the declassification. Therefore, to enforce that the decision to do the declassification is sufficiently trusted, we simply require that the program counter have high enough integrity.

These intuitions are captured in the following rule for declassification:

$$\textsc{Robust-Declassify}\qquad \frac{\Gamma \vdash v : t_{\ell'} \quad \Gamma, x{:}t_\ell\,[\mathsf{pc}] \vdash e \quad I(\mathsf{pc}) \sqsubseteq \{\mathtt{?:}\mathsf{auth}(\ell,\ell')\}}{\Gamma\,[\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{declassify}(v,\ell)\ \mathtt{in}\ e}$$

This approach equates the authority of a piece of code with the integrity of the program counter at the start of the code, simultaneously simplifying the typing rules—no authority context A is needed—and strengthening the restrictions on where declassification is permitted. This version of declassification rules out the program in Figure 6.1.

The benefit of tying downgrading to integrity is that the noninterference proofs given for the security-typed language say something meaningful for programs that include declassification. Note that the declassification operation does not change the integrity of the data being declassified. Projecting the noninterference result onto the integrity sublattice yields the following lemma as a corollary.

Lemma 6.2.1 (Robust Declassification) *Suppose that* $x : s[\bot] \vdash e : s'$ *and the integrity labels satisfy* $I(\mathsf{label}(s)) \not\sqsubseteq I(\mathsf{label}(s'))$. *Then for any values* v_1 *and* v_2 *such that* $\vdash v_i : s$ *it is the case that* $e\{v_1/x\} \Downarrow v \;\Leftrightarrow\; e\{v_2/x\} \Downarrow v$.

This lemma holds regardless of whether e contains declassification operations. It is a weak guarantee: Intuitively, low-integrity data cannot interfere with what data is declassified. This lemma does not say anything about what high-security information might be declassified. Nevertheless, it is better than giving up all security properties when declassifications are used.

One could generalize robust declassification by associating with each distinct declassification expression in the program a separate principal d and requiring that $I(\mathtt{pc}) \sqsubseteq \mathtt{\{?:d\}}$ in the declassification typing judgment. This constraint allows the programmer to name particular declassifications in security policies so that, for instance a value with integrity label $\mathtt{\{?:d_1,\ d_2\}}$ could possibly be declassified at points $\mathtt{d_1}$ and $\mathtt{d_2}$ but not at a declassification associated with point $\mathtt{d_3}$ in the program.

Whether such a generalization would be useful in practice, and how to precisely characterize the confidentiality properties of the resulting programs remains for future work.

6.3 Related work

The simplest and most standard approach to declassification is to restrict its uses to those performed by a trusted subject, similar to the DLM requirement that a function possess the proper authority. This approach does not address the question of whether an information channel is created. Many systems have incorporated a more limited form of declassification. Ferrari et. al [FSBJ97] augment information flow controls in an object-oriented system with a form of dynamically-checked declassification called *waivers*. However, these efforts provide only limited characterization of the safety of the declassification process.

The interplay between authority and declassification is similar to Java's stack inspection security model [WF98, WAF00, FG02]. In Java, privileged operations (like declassification) can require that they be invoked only in the context of some authorization clause, and, that, dynamically, no untrusted methods are between the authorization and the use of the privileged operation on the call stack. These constraints on the runtime stack are similar to the authority constraints used in the decentralized label model, but weaker than the robust declassification mechanism proposed here. The difference is that the stack-inspection approach does not track the integrity of data returned by an untrusted piece of code, so untrusted data might still influence privileged operations.

Using untrusted data to regulate privileged operations is related to an extremely common bug found in the C libraries. The string formatting utilities assume that strings are

properly delimited and do not check their bounds. Programs that use the libraries without the appropriate checks are vulnerable to attacks that occur when strings are read from an untrusted source such as the network. The analysis that is able to find such format string vulnerabilities in C [STFW01] is quite similar to an integrity-only information-flow analysis.

Intransitive noninterference policies [Rus92, Pin95, RG99] generalize noninterference to describe systems that contain restricted downgrading mechanisms. The work by Bevier et al. on *controlled interference* [BCY95] is most similar to this work in allowing the specification of policies for information released to a set of *agents*. The idea of robust declassification has been formalized in an abstract, state-machine model [ZM01a].

Chapter 7

Distribution and Heterogeneous Trust

So far, the security-typed languages in this thesis have addressed information-flow security in systems executed on a single, trusted host. This assumption is unrealistic, particularly in scenarios for which information-flow policies are most desirable—when multiple principals need to cooperate but do not entirely trust one another. Simple examples of such scenarios abound: email services, web-based shopping and financial planning, business-to-business transactions, and joint military information systems. Such sophisticated, collaborative, inter-organizational computation is becoming increasingly common; some way is needed to ensure that data confidentiality is protected.

The general problem with these collaborative computations is ensuring that the security policies of all the participants are enforced. When participants do not fully trust each others' hosts, it is necessary to distribute the data and computational work among the hosts. This distribution creates a new threat to security: the hosts used for computation might cause security violations—either directly, by leaking information, or indirectly, by carrying out computations in a way that causes other hosts to leak information.

Of course, the program itself might also cause security violations. Because the existing single-host techniques developed in this thesis address this problem, this chapter focuses on the new threat, untrusted hosts.

The goal of this chapter is to extend $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ to account for the problem of information-flow security in distributed systems with mutually untrusting hosts. $\lambda_{\mathrm{SEC}}^{\mathrm{DIST}}$ is a core language for studying information-flow security properties in distributed systems. As with $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$, it abstracts away from many of the details of actual computation and instead focuses on the key aspects of distributed computing: message passing, synchronization, and the notion of hosts with distinct identities levels of trust.

The defining characteristic of a distributed system is the absence of shared memory [Sch97]. In $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$, the requirements imposed by information-flow considerations have already restricted interthread communication to a message-passing style similar to that needed in a distributed setting. However, there are still a number of differences

between a the universally trusted, single-platform setting and a heterogeneously trusted, distributed setting:

- The lack of shared memory in a distributed setting implies that reading from a remote memory location involves sending a message to the host that is storing the data. The distributed computing model must make all interhost communication explicit, which, in contrast to a single-platform model, introduces *read channels*: implicit flows that arise due to the additional communication necessary to read a remote location.

- The failure mode in a heterogeneously-trusted system is different from that of a single trusted platform. In the universally-trusted host setting, once the platform has been compromised or subverted, the confidentiality or integrity of any data in the system might be endangered. In the heterogeneous setting, the failure of one host should affect only those principals who have declared some degree of trust in that host.

- Confidential information can be leaked based on which of two hosts sends a message to a receiver, even if the message contents are otherwise identical.

- There are more abstraction-violation attacks in a distributed setting. For example, an attacker might learn some information by watching the network traffic generated by a distributed program. Such an attack is an external channel according to the classification of information flows in the introduction of this thesis because the global state of the links on a network is not available at the programming language level of abstraction.

The next section describes the heterogeneous trust model for the distributed setting; it is based on the decentralized labels presented in the previous chapter and permits an appropriate strengthening of noninterference. The following section sketches the modifications to the type system of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$ needed to achieve such security.

7.1 Heterogeneous trust model

This section presents one model network environment and shows how the decentralized label model can describe the trust relationship between principals and hosts. There are many plausible models for network communication and its security properties; this model was chosen because it fits well with the operational semantics of $\lambda_{\mathrm{SEC}}^{\mathrm{CONCUR}}$, it seems reasonable to implement, and it is sufficient to describe additional information flows that might arise in a distributed setting.

Let H be a set of *known hosts*, among which the system is to be distributed. Pairwise communication between two members of H is assumed to be reliable: messages cannot be lost. Communication is assumed to be asynchronous: sending hosts do not block waiting for the destination host to receive the message, nor is there a bound on the amount of time it takes for a message to be delivered.

This model also assumes that messages cannot be intercepted by hosts outside H or by the other members of H. Protection against interception can be achieved efficiently through well-known encryption techniques (e.g, [SNS88, Ylo96]); for example, each pair of hosts can use symmetric encryption to exchange information, with key exchange via public-key encryption. that the same encryption mechanisms permit each member of H to authenticate messages sent and received by one another.

In addition to the underlying communication model, for security purposes, it is necessary to relate the available hosts to the principals on whose behalf the system is supposed to run. This relation is called a *trust configuration*, and it consists of two labels associated with each host.

- A *confidentiality* label $\mathtt{C}_h$ that is an upper bound on the confidentiality of information that can be sent securely (either explicitly or implicitly) to host h.
- An *integrity* label $\mathtt{I}_h$ describing an upper bound on the integrity of data that may be received from h.

Intuitively, the confidentiality label specifies which principals trust host h not to leak their confidential data, and the integrity label specifies which principals trust h not to corrupt their data.

As an example, consider a host A owned by Alice but untrusted by Bob, and a host B owned by Bob and untrusted by Alice. A reasonable trust configuration might be:

$$\begin{array}{ll} \mathtt{C}_A = \{\mathtt{Alice:}\} & \mathtt{I}_A = \{\mathtt{?:Alice}\} \\ \mathtt{C}_B = \{\mathtt{Bob:}\} & \mathtt{I}_B = \{\mathtt{?:Bob}\} \end{array}$$

Because `Bob` does not appear as an owner in the label $\mathtt{C}_A$, this description acknowledges that Bob is unwilling to send his private data to host A. Similarly, Bob does not trust information received from A because `Bob` does not appear in $\mathtt{I}_A$. The situation is symmetric with respect to Alice and Bob's host.

Next, consider hosts T and S that are partially trusted by Alice and Bob:

$$\begin{array}{ll} \mathtt{C}_T = \{\mathtt{Alice:;Bob:}\} & \mathtt{I}_T = \{\mathtt{?:Alice}\} \\ \mathtt{C}_S = \{\mathtt{Alice:;Bob:}\} & \mathtt{I}_S = \{\mathtt{?:}\} \end{array}$$

Alice and Bob both trust T not to divulge their data incorrectly; on the other hand, Bob believes that T might corrupt data—he does not trust the integrity of data received

from T. Host S is also trusted with confidential data, but neither Alice nor Bob trust data generated by S.

These trust declarations are public knowledge—that is, they are available on all known hosts—and are signed by the principals involved. Consequently, this heterogeneous trust model assumes the existence of a public-key infrastructure that makes such digital signatures feasible. (See for example, the work on public-key encryption [SNS88] and certification authorities [Zho01, ZSv00].)

The goal is to ensure that the threats to a principal's confidential data are not increased by the failure or subversion of an untrusted host.

The security of a principal is endangered only if one or more of the hosts that the principal trusts is bad. Suppose the host h is bad and let $\mathtt{L}_e$ be the label of an expression in the program. The confidentiality of the expression's value is endangered only if $C(\mathtt{L}_e) \sqsubseteq \mathtt{C}_h$; correspondingly, the expression's integrity may have been corrupted only if $\mathtt{I}_h \sqsubseteq I(\mathtt{L}_e)$.

If Alice's machine A from above is compromised, only data owned by Alice may be leaked, and only data she trusts may be corrupted. Bob's privacy and integrity are protected. By contrast, if the semi-trusted machine T malfunctions or is subverted, both Alice and Bob's data may be leaked, but only Alice's data may be corrupted because only she trusts the integrity of the machine.

When there are multiple bad machines, they might cooperate to leak or corrupt more data. The type system described in the next section enforces the following property in addition to the standard noninterference:

Definition 7.1.1 (Distributed Security Assurance) *The confidentiality of a program expression e is not threatened by a set H_{bad} of bad hosts unless $C(\mathtt{L}_e) \sqsubseteq \bigsqcup_{h \in H_{bad}} \mathtt{C}_h$; its integrity is not threatened unless $\bigsqcap_{h \in h_{bad}} \mathtt{I}_h \sqsubseteq I(\mathtt{L}_e)$.*

Note that in the case that there are no compromised hosts ($H_{bad} = \emptyset$), this definition degenerates to standard noninterference.

Providing this level of assurance involves two challenges: (1) Data with a confidentiality label (strictly) higher than $\mathtt{C}_h$ should never be sent (explicitly or implicitly) to h, and data with an integrity label lower than $\mathtt{I}_h$ should never be accepted from h. (2) Bad hosts should not be able to exploit the downgrading abilities of more privileged hosts, causing them to violate the security policy of the source program.

7.2 $\lambda^{\mathrm{DIST}}_{\mathrm{SEC}}$: a secure distributed calculus

The $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$ language guarantees noninterference for concurrent processes running on a single, universally trusted platform. This section describes an extension to $\lambda^{\mathrm{CONCUR}}_{\mathrm{SEC}}$, called $\lambda^{\mathrm{DIST}}_{\mathrm{SEC}}$, that enforces the distributed security assurance property.

7.2.1 Syntax

Programs in $\lambda_{\text{SEC}}^{\text{DIST}}$ consist of a collection of $\lambda_{\text{SEC}}^{\text{CONCUR}}$ processes, each process executing at a particular host. To describe this situation, we need to make only a few modifications to the $\lambda_{\text{SEC}}^{\text{CONCUR}}$ syntax. A $\lambda_{\text{SEC}}^{\text{CONCUR}}$ network is:

$$N ::= \cdot \mid N \mid h[\mathsf{pc} : P]$$

Here, the syntax $h[\mathsf{pc} : P]$ denotes a host h running a process P whose current program counter label is pc.

Processes, P are identical to those of $\lambda_{\text{SEC}}^{\text{CONCUR}}$, but $\lambda_{\text{SEC}}^{\text{DIST}}$ must also keep track of the host that allocates each memory location or synchronization handler definition. Consequently, locations and channel values are tagged with the host that created them:

$$bv ::= \mathtt{t} \mid \mathtt{f} \mid c \mathbin{@} h \mid L \mathbin{@} h$$

In $\lambda_{\text{SEC}}^{\text{DIST}}$, each host has a local store. The global state of the distributed system is therefore defined just like the memories used in $\lambda_{\text{SEC}}^{\text{CONCUR}}$, except that the memory contents are tagged by their hosts:

$$M ::= M, h[L \mapsto v] \mid M, h[\mathsf{pc} : J \rhd P] \mid \cdot$$

7.2.2 Operational semantics

The operational semantics of $\lambda_{\text{SEC}}^{\text{DIST}}$ is given in Figures 7.1 and 7.2. It is, for the most part, identical to the semantics for $\lambda_{\text{SEC}}^{\text{CPS}}$. Primitive operations must take place on a specific host, as indicated by the $@\ h$ to the left of the symbol $\models$. Rule $\lambda_{\text{SEC}}^{\text{DIST}}$-EVAL-DEREF requires that the host performing the dereference be the same as the host that allocated the reference—hosts have local state.

Whenever data leaves a host h its security label is stamped with the integrity label $\mathtt{I}_h$, reflecting the fact that only those principals that trust h not to corrupt data are willing to trust any value h generates. The rules $\lambda_{\text{SEC}}^{\text{DIST}}$-EVAL-SEND and $\lambda_{\text{SEC}}^{\text{DIST}}$-EVAL-LINSEND show this behavior.

7.2.3 Type system

The type system for $\lambda_{\text{SEC}}^{\text{DIST}}$ modifies the type system for $\lambda_{\text{SEC}}^{\text{CONCUR}}$ to add additional constraints reflecting the hosts' levels of trust.

The intuition is that whenever data *leaves* a host h, it is constrained so that its integrity is no more than $\mathtt{I}_h$. Similarly, when data *arrives* at a host h, it must have confidentiality no higher than $\mathtt{C}_h$.

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-PRIM
$$M,\ \mathsf{pc}\ @\ h \models v \Downarrow v \sqcup \mathsf{pc}$$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-BINOP
$$M,\ \mathsf{pc}\ @\ h \models n_\ell \oplus n'_{\ell'} \Downarrow (n[\![\oplus]\!]n')_{\ell \sqcup \ell'} \sqcup \mathsf{pc}$$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-DEREF
$$\frac{M = M', h[L \mapsto v]}{M,\ \mathsf{pc}\ @\ h \models !L\ @\ h \Downarrow v \sqcup \mathsf{pc}}$$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-LETPRIM
$$\frac{M,\ \mathsf{pc}\ @\ h \models prim \Downarrow v}{\langle M,\ S,\ (N \mid h[\mathsf{pc} : \mathtt{let}\ x = prim\ \mathtt{in}\ e])\rangle \rightarrow \langle M,\ S,\ (N \mid h[\mathsf{pc} : e\{v/x\}])\rangle}$$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-LETREF
$$\begin{array}{lll} & \langle M,\ S,\ (N \mid h[\mathsf{pc} : \mathtt{let}\ x = \mathtt{ref}\ v\ \mathtt{in}\ P])\rangle & \\ \rightarrow & \langle M, h[L \mapsto v],\ S,\ (N \mid h[\mathsf{pc} : P\{L_{\mathsf{pc}}\ @\ h/x\}])\rangle & (L \text{ fresh}) \end{array}$$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-SET
$$\begin{array}{ll} & \langle M, h[L \mapsto v],\ S,\ (N \mid h[\mathsf{pc} : \mathtt{set}\ L_\ell\ @\ h := v'\ \mathtt{in}\ P])\rangle \\ \rightarrow & \langle M, h[L \mapsto v' \sqcup \ell \sqcup \mathsf{pc}],\ S,\ (N \mid h[\mathsf{pc} : P])\rangle \end{array}$$

Figure 7.1: $\lambda^{\text{DIST}}_{\text{SEC}}$ operational semantics

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-HANDLER

$\langle M,\, S,\, (N \mid h[\![\mathsf{pc} : \texttt{let}\ f_1(\vec{x}_1) \mid \ \ldots\ \mid f_n(\vec{x}_n) \rhd P_1\ \texttt{in}\ P_2]\!])\rangle$

$\rightarrow\ \langle M, h[\![\mathsf{pc} : c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \rhd P_1\{(c_i)_{\mathsf{pc}}/f_i\}]\!],\ S,$
$(N \mid h[\![\mathsf{pc} : P_2\{(c_i)_{\mathsf{pc}}/f_i\}]\!])\rangle$

where the c_i are fresh

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-LINHANDLER

$\langle M,\, S,\, (N \mid h[\![\mathsf{pc} : \texttt{let}\ f_1(\vec{x}_1) \mid \ \ldots\ \mid f_n(\vec{x}_n) \multimap P_1\ \texttt{in}\ P_2]\!])\rangle$

$\rightarrow\ \langle M,\, S, h[\![\mathsf{pc} : c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \multimap P_1]\!],\ (N \mid h[\![\mathsf{pc} : P_2\{c_i/f_i\}]\!])\rangle$

where the c_i are fresh

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-COND1

$\langle M,\, S,\, (N \mid h[\![\mathsf{pc} : \texttt{if}\ \mathsf{t}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2]\!])\rangle$

$\rightarrow\ \langle M,\, S,\, (N \mid h[\![\mathsf{pc} \sqcup \ell : P_1]\!])\rangle$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-COND2

$\langle M,\, S,\, (N \mid h[\![\mathsf{pc} : \texttt{if}\ \mathsf{f}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2]\!])\rangle$

$\rightarrow\ \langle M,\, S,\, (N \mid h[\![\mathsf{pc} \sqcup \ell : P_2]\!])\rangle$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-SEND

$\langle M, h[\![\mathsf{pc} : c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \rhd P]\!],\ S,\ (N \mid_i h_i[\![\mathsf{pc}_i : c_{i\ell_i}(\vec{v}_i, lv_i^{\mathsf{opt}})]\!])\rangle$

$\rightarrow\ \langle M, h[\![\mathsf{pc} : c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \rhd P]\!],\ S,$
$(N \mid h[\![\ell : P\{\vec{v}_i \sqcup \mathsf{pc}_i \sqcup \mathtt{I}_{h_i}/\vec{x}_i\}\{lv_i/y_i\}^{\mathsf{opt}}]\!])\rangle$

where $\ell = \bigsqcup_i \mathsf{pc}_i \sqcup \ell_i$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-LINSEND

$\langle M,\, S, h[\![\mathsf{pc} : c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \multimap P]\!],\ (N \mid_i h_i[\![\mathsf{pc}_i : c_i(\vec{v}_i)]\!])\rangle$

$\rightarrow\ \langle M,\, S,\, (N \mid h[\![\mathsf{pc} : P\{\vec{v}_i \sqcup \mathsf{pc}_i \sqcup \mathtt{I}_{h_i}/\vec{x}_i\}]\!])\rangle$

$\lambda^{\text{DIST}}_{\text{SEC}}$-EVAL-FORK

$\langle M,\, S,\, (N \mid h[\![\mathsf{pc} : P \mid Q]\!])\rangle \rightarrow \langle M,\, S,\, (N \mid h[\![\mathsf{pc} : P]\!] \mid h[\![\mathsf{pc} : Q]\!])\rangle$

Figure 7.2: $\lambda^{\text{DIST}}_{\text{SEC}}$ operational semantics continued

$$\lambda^{\text{DIST}}_{\text{SEC}}\text{-SEND}\quad \frac{\begin{array}{l} H; \Gamma \vdash_h v : [\mathsf{pc}'](\vec{s}, k^{\mathsf{opt}}) \\ H; \Gamma \vdash_h v_i : s_i \\ T; K \vdash_h l v^{\mathsf{opt}} : k^{\mathsf{opt}} \\ \mathtt{I}_h \sqcup \mathsf{pc} \sqsubseteq \mathsf{pc}' \qquad \mathtt{I}_h \sqsubseteq \mathsf{label}(s_i) \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash_h v(\vec{v}, l v^{\mathsf{opt}})}$$

$$\lambda^{\text{DIST}}_{\text{SEC}}\text{-LET}\quad \frac{\begin{array}{l} J \rightsquigarrow_{\mathsf{pc}'} \langle \Gamma_f;\ \Gamma_{args};\ K \rangle \\ H; \Gamma, \Gamma_f, \Gamma_{args}; \cdot, K\ [\mathsf{pc}'] \vdash_h P_1 \\ H; \Gamma, \Gamma_f; T, K\ [\mathsf{pc}] \vdash_h P_2 \\ \mathsf{pc}' \sqsubseteq \mathtt{C}_h \qquad \forall x \in \mathrm{dom}(\Gamma_{args}).\ \Gamma_{args}(x) \sqsubseteq \mathtt{C}_h \end{array}}{H; \Gamma; T, K\ [\mathsf{pc}] \vdash_h \mathtt{let}\ J \rhd P_1\ \mathtt{in}\ P_2}$$

Figure 7.3: $\lambda^{\text{DIST}}_{\text{SEC}}$ typing rules for message passing

$$\lambda^{\text{DIST}}_{\text{SEC}}\text{-VAL}\quad \frac{H; \Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s) \sqsubseteq \mathtt{C}_h}{H; \Gamma\ [\mathsf{pc}] \vdash_h v : s}$$

$$\lambda^{\text{DIST}}_{\text{SEC}}\text{-BINOP}\quad \frac{H; \Gamma \vdash v : \mathtt{bool}_\ell \quad H; \Gamma \vdash v' : \mathtt{bool}_\ell \quad \mathsf{pc} \sqsubseteq \ell \sqsubseteq \mathtt{C}_h}{H; \Gamma\ [\mathsf{pc}] \vdash_h v \oplus v' : \mathtt{bool}_\ell}$$

$$\lambda^{\text{DIST}}_{\text{SEC}}\text{-DEREF}\quad \frac{H; \Gamma \vdash v : s\ \mathtt{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s \sqcup \ell) \sqsubseteq \mathtt{C}_h}{H; \Gamma\ [\mathsf{pc}] \vdash_h\ !v : s \sqcup \ell}$$

Figure 7.4: $\lambda^{\text{DIST}}_{\text{SEC}}$ typing rules for primitive operations

The rules for sending messages and declaring message handlers are shown in Figure 7.3. (The rule for linear handlers is modified analogously.)

It is also necessary to rule out high-security data from being explicitly located at a low-security host. Accordingly, the rules for typechecking primitive operations require that the host can handle the confidentiality of the data. These rules are shown in Figure 7.4.

The type system for $\lambda_{\mathrm{SEC}}^{\mathrm{DIST}}$ satisfies subject reduction; the proof is a straightforward modification of the one given for lemma 5.3.8. Although a proof is beyond the scope of this thesis, the type system given here is also intended to provide the distributed security assurance property.

7.3 Related Work

The distributed trust model presented in this chapter, was originally developed in a technical report [ZM00]. This model was further refined for the program-partitioning work described in the next chapter [ZZNM02].

$\lambda_{\mathrm{SEC}}^{\mathrm{DIST}}$'s notion of explicit hosts is similar to those in the Dπ calculus of Riely and Hennessy [RH99]. More sophisticated ambient calculi [CG00] have a dynamic notion of host and also permit mobile computing. Some security considerations have been studied in the ambient scenarios [MH02], but information-flow policies have not yet been considered.

Mantel and Sabelfeld [MS01, SM02] consider a distributed model in which multi-threaded programs communicate via synchronous and asynchronous message passing. They propose a definition of information security based on Mantel's security framework [Man00] and show how a type system can establish noninterference. Their definition of noninterference is both external-timing and termination sensitive, and hence rules out single-threaded programs like examples (5) and (6) of Section 5.1.

In addition to high- and low-security channels, the Mantel and Sabelfeld language provides encrypted channels. An encrypted channel reveals to a low-security observer only the number of messages that have been sent on it. Consequently, high-security data is not permitted to influence the number of messages sent on an encrypted channel—this restriction is quite similar to the linearity constraints considered here (where it is statically known that exactly one message will be sent on the channel). Further investigation of this possible connection is warranted.

Chapter 8

Jif/split

This chapter presents *secure program partitioning*, a way to protect the confidentiality of data for computations that manipulate data with differing confidentiality needs in the heterogeneously trusted hosts model of the previous chapter. Figure 8.1 illustrates the key insight: The security policy can be used to guide the automatic splitting of a security-typed program into communicating subprograms, each running on a different host. Collectively, the subprograms perform the same computation as the original; in addition, they satisfy all the participants' security policies without requiring a single universally trusted host.

The prototype implementation, called Jif/split, is primarily designed to enforce confidentiality policies, but due to declassification, the system must also enforce simple integrity policies as well (see Chapter 6).

As Figure 8.1 shows, Jif/split receives two inputs: the program, including its confidentiality and integrity policy annotations, and also a set of signed trust declarations stating each principal's trust in hosts and other principals, in accordance with the heterogeneous trust model presented in the last chapter. The goal of secure program partitioning is to ensure that if a host h is subverted, the only data whose confidentiality or integrity is threatened is data owned by principals that have declared they trust h. Also, note that to avoid the need for undue trust in the splitting process itself, the production of the subprogram for host h can be performed on any host that is at least as trustworthy as h—such as h itself.

It is useful to contrast this approach with the usual development of secure distributed systems, which involves the careful design of protocols for exchanging data among hosts in the system. By contrast, the splitting approach provides the following benefits:

- **Stronger security:** Secure program partitioning can be applied to information-flow policies; most distributed systems make no attempt to control information flow. It can also be applied to access control policies, which are comparatively simple to enforce with this technique.

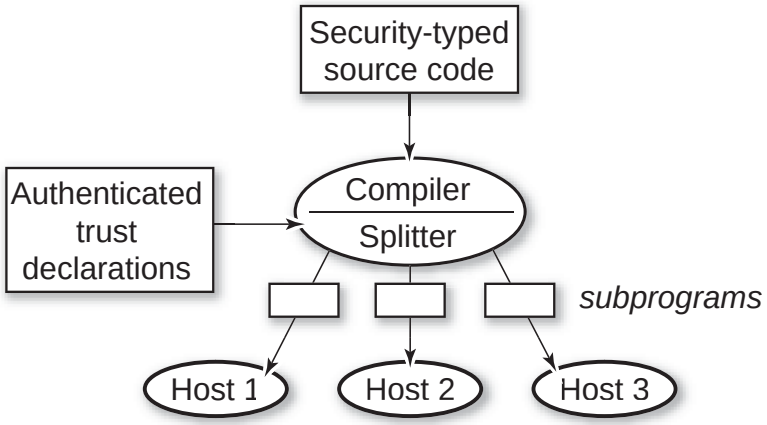


Figure 8.1: Secure program partitioning

- **Decentralization:** Collaborative computations can be carried out despite incomplete trust. In addition, for many computations, there is no need for a universally trusted host. Each participant can independently ensure that its security policies are enforced.

- **Automation:** Large computing systems with many participating parties contain complex, interacting security policies that evolve over time; automated enforcement is becoming a necessity. Secure program partitioning permits a computation to be described as a single program independent of its distributed implementation. The partitioning process then *automatically* generates a secure protocol for data exchange among the hosts.

Secure program partitioning has the most value when strong protection of confidentiality is needed by one or more principals, the computing platform consists of differently trusted hosts, there is a generally agreed-upon computation to be performed, and security, performance, or functionality considerations prevent the entire computation from being executed on a single host. One example of a possible application is an integrated medical information system that stores patient and physician records, raw test data, and employee records, and supports information exchange with other medical institutions. Another example is an automated business-to-business procurement system, in which profitable negotiation by the buyer and supplier depends on keeping some data confidential.

The goal of Jif/split is to enforce the distributed security assurance discussed in Section 7.1. It ensures that the threats to a principal's confidential data are not increased by the failure or subversion of an untrusted host that is being used for execution. Bad hosts—hosts that fail or are subverted—have full access to the part of the program ex-

ecuting on them, can freely fabricate apparently authentic messages from bad hosts, and can share information with other bad hosts. Bad hosts may execute concurrently with good hosts, whereas good hosts preserve the sequential execution of the source language—there is only one good host executing at a time. However, we assume that bad hosts are not able to forge messages from good hosts, nor can they generate certain capabilities to be described later.

The rest of this chapter describes Jif/split, an implementation of secure program partitioning, which includes a static checker, program splitter, and run-time support for the distributed subprograms. It also presents simple examples of applying this approach and some performance results that indicate its practicality.

As with the other security-typed languages in this thesis, Jif/split does not attempt to control certain classes of information flows: external timing and termination channels, or attacks based on network traffic analysis.

8.1 Jif: a security-typed variant of Java

The Jif/split program splitter extends the compiler for Jif [Mye99, MNZZ01], a security-typed extension to Java [GJS96] that uses labels from the decentralized label model (as described in 6.1).

Types in Jif are labeled, allowing the programmer to declare variables and fields that include security annotations. For example, a value with type `int{o:r}` is an integer owned by principal `o` and readable by `r`. When unlabeled Java types are written in a program, the label component is automatically inferred.

As with the other security-typed languages described in this thesis, every Jif program expression has a labeled type that indicates an upper bound (with respect to the $\sqsubseteq$ order) of the security of the data represented by the expression. Jif also uses a program counter label to track the side-effects that may be created by a method or other piece of code. Using the labels provided by the programmer and the inferred `pc` label, the Jif compiler is able to statically verify that all of the information flows apparent in the program text satisfy the label constraints that prevent illegal information flows from occurring. If the program does not satisfy the security policy, it is rejected.

In addition to these changes to the Java type system, Jif adds a number of constructs for creating secure programs. The following are germane to this dissertation:

- Declassification and endorse expressions, that follow the robust-declassification rule as described in Section 8.2.3.

- An optional `authority` clause on method declarations describes the authority available in the body of the method. Code containing such a clause can be added to the system only with the permission of the principals named in it.

- Optional label bounds on the initial pc label of a method.

For example, the method signature

$$\texttt{int\{}\ell_1\texttt{\} m\{pc\}(int\{}\ell_2\texttt{\} x) where authority \{Alice\}}$$

is translated into the notation used in this thesis as

$$[\{\texttt{Alice}\}, \texttt{pc}]\texttt{int}_{\ell_2} \rightarrow \texttt{int}_{\ell_1}$$

This type indicates that the method m can only be called when the program counter label is $\sqsubseteq$ pc. It takes an integer x with label ℓ_2 and returns an integer labeled ℓ_1. The initial label bound plays exactly the same role for Jif methods as the $[A, \texttt{pc}]$ component on function types presented in Chapter 6.

Jif also introduces some limitations to Java, which apply to Jif/split as well. The most important is that programs are assumed to be sequential: the Thread class is not available. The current implementation of Jif does not support threaded computation as proposed in Chapter 5, partially because Java's model of threads and synchronization is significantly more complex than that of $\lambda_{\text{SEC}}^{\text{CONCUR}}$. Determining how to integrate the results of Chapter 5 into Jif is left for future work.

8.1.1 Oblivious Transfer Example

Figure 8.2 shows a sample program that is used as a running example. It is based on the well-known Oblivious Transfer Problem [EGL83, Rab81], in which the principal Alice has two values (here represented by fields m1 and m2), and Bob may request exactly one of the two values. However, Bob does not want Alice to learn which of the two values was requested.

Even this short example has interesting security issues. For instance, it is well-known that a trusted third party is needed for a secure distributed implementation under the assumptions of perfect security (no information leakage) [DKS99].[1]

Alice's secret data is represented by integer fields m1 and m2, with security label {Alice:; ?:Alice}. This type indicates that these fields are owned by Alice, that she lets no one else read them, and that she trusts their contents. The boolean isAccessed records whether Bob has requested a value yet.

Lines 6 through 18 define a method transfer that encapsulates the oblivious transfer protocol. It takes a request, n, owned by Bob, and returns either m1 or m2 depending

[1] Probabilistic solutions using two hosts exist, but these algorithms leak small amounts of information. Because Jif's type system is geared to possibilistic information flows, these probabilistic algorithms are rejected as potentially insecure. Ongoing research [GS92, VS00, SS00] attempts to address probabilistic security.

```
1 public class OTExample {
2   int{Alice:; ?:Alice} m1;
3   int{Alice:; ?:Alice} m2;
4   boolean{Alice:; ?:Alice} isAccessed;
5
6   int{Bob:} transfer{?:Alice} (int{Bob:} n)
7   where authority(Alice) {
8     int tmp1 = m1;
9     int tmp2 = m2;
10     if (!isAccessed) {
11       isAccessed = true;
12       if (endorse(n, {?:Alice}) == 1)
13         return declassify(tmp1, {Bob:});
14       else
15         return declassify(tmp2, {Bob:});
16     }
17     else return 0;
18   }
19 }
```

Figure 8.2: Oblivious transfer example in Jif

on `n`'s value. Note that because Alice owns `m1` and `m2`, releasing the data requires declassification (lines 13 and 15). Her authority, needed to perform this declassification, is granted by the `authority` clause on line 7.

Ignoring for now the temporary variables `tmp1` and `tmp2` and the `endorse` statement, the body of the `transfer` method is straightforward: Line 10 checks whether Bob has made a request already. If not, line 11 records the request, and lines 12 through 15 return the appropriate field after declassifying them to be visible by Bob. If Bob has already made a request, `transfer` simply returns `0`.

The simplicity of this program is deceptive. For example, the `pc` label at the start of the `transfer` method must be bounded above by the label `{?:Alice}`, as indicated on line 6. The reason is that line 11 assigns `true` into the field `isAccessed`, which requires Alice's integrity. If the program counter at the point of assignment does not also have Alice's trust, the integrity of `isAccessed` is compromised.

These observations illustrate one benefit of programming in a security-typed language: the compiler can catch many subtle security holes even though the code is written in a style that contains no specification of how the code is to be distributed.

The interactions between confidentiality, integrity, and declassifications described in Chapter 6 explain the need for the temporary variables and endorsement. The details of this example are described in the rest of this chapter, as we consider its security in a distributed environment.

8.2 Static Security Constraints

Jif/split uses the model for heterogeneously trusted distributed systems given in Chapter 7. To securely partition a program for such an environment, the splitter must know the trust relationships between the participating principals and the hosts H. Recall that these trust configurations are provided by a confidentiality and integrity label for each host h. These labels are $\mathtt{C}_h$ and $\mathtt{I}_h$, respectively.

At a high level, the partitioning process can be seen as a constraint satisfaction problem. Given a source program and the trust relationships between principals and hosts, the splitter must assign a host in H to each field, method, and program statement in the program. This fine-grained partitioning of the code is important so that a single method may access data of differing confidentiality and integrity. The primary concern when assigning hosts is to enforce the confidentiality and integrity requirements on data; efficiency, discussed in Section 8.5, is secondary.

This section describes the static constraints on host selection, they derive from the considerations of $\lambda^{\mathrm{DIST}}_{\mathrm{SEC}}$.

8.2.1 Field and Statement Host Selection

Consider the field `m1` of the oblivious transfer example. It has label `{Alice:;?:Alice}`, which says that Alice owns and trusts this data. Only certain hosts are suitable to store this field: hosts that Alice trusts to protect both her confidentiality and integrity. If the field were stored elsewhere, the untrusted host could violate Alice's policy, contradicting the security assurance of Section 7.1. The host requirements can be expressed using labels: $\mathtt{\{Alice:\}} \sqsubseteq \mathtt{C}_h$ and $\mathtt{I}_h \sqsubseteq \mathtt{\{?:Alice\}}$. The first inequality says that Alice allows her data to flow to h, and the second says that Alice trusts the data she receives from h. In general, for a field f with label $\mathtt{L}_f$ we require

$$C(\mathtt{L}_f) \sqsubseteq \mathtt{C}_h \quad \text{and} \quad \mathtt{I}_h \sqsubseteq I(\mathtt{L}_f).$$

This same reasoning further generalizes to the constraints for locating an arbitrary program statement, S. Let $U(S)$ be the set of values *used* in the computation of S and

let $D(S)$ be the set of locations S *defines*. Suppose that the label of the value v is $\mathtt{L}_v$ and that the label of a location l is $\mathtt{L}_l$. Let

$$\mathtt{L}_{in} = \bigsqcup_{v \in U(S)} \mathtt{L}_v \quad \text{and} \quad \mathtt{L}_{out} = \bigsqcap_{l \in D(S)} \mathtt{L}_l$$

A host h can execute the statement S securely, subject to constraints similar to those for fields.

$$C(\mathtt{L}_{in}) \sqsubseteq \mathtt{C}_h \quad \text{and} \quad \mathtt{I}_h \sqsubseteq I(\mathtt{L}_{out})$$

8.2.2 Preventing Read Channels

The rules for host selection for fields in the previous section are necessary but not sufficient in the distributed environment. Because bad hosts in the running system may be able to observe read requests from good hosts, a new kind of implicit flow is introduced: a *read channel* in which the request to read a field from a remote host itself communicates information.

For example, a naive implementation of the oblivious transfer example of Figure 8.2 exhibits a read channel. Suppose that in implementing the method `transfer`, the `declassify` expressions on lines 13 and 15 directly declassified the fields `m1` and `m2`, respectively, instead of the variables `tmp1` and `tmp2`. According to Bob, the value of the variable `n` is private and not to be revealed to Alice. However, if `m1` and `m2` are stored on Alice's machine, Alice can improperly learn the value of `n` from the read request.

The problem is that Alice can use read requests to reason about the location of the program counter. Therefore, the program counter at the point of a read operation must not contain information that the field's host is not allowed to see. With each field f, the static checker associates a confidentiality label $\mathtt{Loc}_f$ that bounds the security level of implicit flows at each point where f is read. For each read of the field f, the label $\mathtt{Loc}_f$ must satisfy the constraint $C(\mathsf{pc}) \sqsubseteq \mathtt{Loc}_f$. Using this label $\mathtt{Loc}_f$, the confidentiality constraint on host selection for the field is:

$$C(\mathtt{L}_f) \sqcup \mathtt{Loc}_f \sqsubseteq \mathtt{C}_h$$

To eliminate the read channel in the example while preventing Bob from seeing both `m1` and `m2`, a trusted third party is needed. The programmer discovers this problem during development when the naive approach fails to split in a configuration with just the hosts A and B as described in Section 4.2. The error pinpoints the read channel introduced: arriving at line 13 depends on the value of `n`, so performing a request for `m1` there leaks `n` to Alice. The splitter automatically detects this problem when the field constraint above is checked.

If the more trusted host T is added to the set of known hosts, the splitter is able to solve the problem, even with the naive code, by allocating `m1` and `m2` on T, which

prevents Alice from observing the read request. If S is used in place of T, the naive code again fails to split—even though S has enough privacy to hold Alice's data, fields `m1` and `m2` can't be located there because Alice doesn't trust S not to corrupt her data. Again, the programmer is warned of the read channel, but this time a different solution is possible: adding `tmp1` and `tmp2` as in the example code give the splitter enough flexibility to *copy* the data to S rather than locating the fields there. Whether S or T is the right model for the trusted host depends on the scenario; what is important is that the security policy is automatically verified in each case.

8.2.3 Declassification Constraints

Consider the oblivious transfer example from Alice's point of view. She has two private pieces of data, and she is willing to release exactly one of the two to Bob. Her decision to declassify the data is dependent on Bob not having requested the data previously. In the example program, this policy is made explicit in two ways. First, the method `transfer` explicitly declares that it uses her authority, which is needed to perform the declassification. Second, the program itself tests (in line 10) whether `transfer` has been invoked previously—presumably Alice would not have given her authority to this program without this check to enforce her policy.

This example shows that it is not enough simply to require that any `declassify` performed on Alice's behalf executes on a host she trusts to hold the data. Alice also must be confident that the decision to perform the declassification, that is, the program execution leading to the `declassify`, is performed correctly.

The program counter label summarizes the information dependencies of the decision to arrive at the corresponding program point. Thus, a `declassify` operation using the authority of a set of principals P introduces the integrity constraint: $I(\mathtt{pc}) \sqsubseteq \mathtt{I}_P$ where $\mathtt{I}_P$ is the label $\mathtt{\{?:p}_1, \ldots, \mathtt{p}_n\mathtt{\}}$ for $\mathtt{p}_i \in P$. This constraint says that each principal `p` whose authority is needed to perform the declassification must trust that the program has reached the `declassify` correctly.

Returning to the oblivious transfer example, we can now explain the need to use the `endorse` operation. Alice's authority is needed for the declassification, but, as described above, she must also be sure of the integrity of the program counter when the program does the declassification. Omitting the `endorse` when testing `n` on line 12 would lower the integrity of the program counter within the branches—Alice doesn't trust that `n` was computed correctly, as indicated by its (lack of an) integrity label on line 6. She must add her endorsement to `n`, making explicit her agreement with Bob that she doesn't need to know `n` to enforce her security policy.

Using the static constraints just described, the splitter finds a set of possible hosts for each field and statement. This process may yield many solutions, or none at all—for

```
Val  getField(HostID h, Obj o, FieldID f)
Val  setField(HostID h, Obj o, FieldID f, Val v)
void forward(HostID h, FrameID f, VarID var, Val v)
void rgoto(HostID h, FrameID f, EntryPt e, Token t)
void lgoto(Token t)
Token sync(HostID h, FrameID f, EntryPt e, Token t)
```

Figure 8.3: Run-time interface

instance, if the program manipulates data too confidential for any known host. When no solution exists, the splitter gives an error indicating which constraint is not satisfiable. We have found that the static program analysis is remarkably useful in identifying problems with apparently secure programs. When more than one solution exists, the splitter chooses hosts to optimize performance of the distributed system, as described in Section 8.5.

8.3 Dynamic Enforcement

In the possible presence of bad hosts that can fabricate messages, run-time checks are required to ensure security. For example, access to an object field on a remote host must be authenticated to prevent illegal data transfers from occurring. Thus, the information-flow policy is enforced by a combination of static constraints (controlling how the program is split) and dynamic checks to ensure that running program obeys the static constraints.

When a program is partitioned, the resulting partitions contain both ordinary code to perform local computation and calls to a special run-time interface that supports host communication. Figure 8.3 shows the interface to the distributed run-time system.[2] There are three operations that transfer data between hosts: `getField`, `setField`, and `forward`; and three operations that transfer control between hosts: `rgoto`, `lgoto`, and `sync`. These operations define building blocks for a protocol that exchanges information among the hosts running partitions.

The `rgoto` and `lgoto` control operations are primitive constructs for transferring control from one program point to another that is located on a different host. In general a program partition comprises a set of code fragments that offer entry points to which `rgoto` and `lgoto` transfer control. These two kinds of goto operations are taken directly from the work on $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$ in Chapter 4.

[2]This interface is simplified for clarity; for instance, the actual implementation provides direct support for array manipulation.

The run-time interface describes all the ways that hosts can interact. To show that bad hosts cannot violate the security assurance provided by the system, it is therefore necessary to consider each of the run-time operations in turn and determine what checks are needed to enforce the assurance condition given in Section 7.1.

8.3.1 Access Control

The simplest operations provided by the run-time interface are `getField` and `setField`, which perform remote field reads and writes. Both operations take a handle to the remote host, the object that contains the field, and an identifier for the field itself. The `setField` operation also takes the value to be written.

These requests are dispatched by the run-time system to the appropriate host. Suppose h_1 sends a field access request to h_2. Host h_2 must perform an access control check to determine whether to satisfy the request or simply ignore it, while perhaps logging any improper request for auditing purposes. A read request for a field f labeled $\mathtt{L}_f$ is legal only if $C(\mathtt{L}_f) \sqsubseteq \mathtt{C}_{h_1}$, which says that h_1 is trusted enough to hold the data stored in f. Similarly, when h_1 tries to update a field labeled $\mathtt{L}_f$, h_2 checks the integrity constraint $\mathtt{I}_{h_1} \sqsubseteq I(\mathtt{L}_f)$, which says that the principals who trust f also trust h_1. These requirements are the dynamic counterpart to those used for host selection (see Section 8.2.1).

Note that because field and host labels are known at compile time, an access control list can be generated for each field, and thus label comparisons can be optimized into a single lookup per request. There is no need to manipulate labels at run time.

8.3.2 Data Forwarding

Another difficulty with moving to a distributed setting is that the run-time system must provide a mechanism to pass data between hosts without violating any of the confidentiality policies attached to the data. The problem is most easily seen when there are three hosts and the control flow $h_1 \longrightarrow l \longrightarrow h_2$: execution starts on h_1, transfers to l, and then completes on h_2. Hosts h_1 and h_2 must access confidential data d (and are trusted to do so), whereas l is not allowed to see d. The question is how to make d securely available to h_2. Clearly it is not secure to transfer d in plaintext between the trusted hosts via l.

There are essentially two solutions to this problem: pass d via l in encrypted form, or forward d directly to h_2. Jif/split implements the second solution. After hosts have been assigned, the splitter infers statically where the data forwarding should occur, using a standard definition-use dataflow analysis. The run-time interface provides an operation `forward` that permits a local variable to be forwarded to a particular stack frame on a remote host. The same mechanism is used to transmit a return value to a remote

host. Data forwarding requires that the recipient validate the sender's integrity, as with `setField`.

8.3.3 Control Transfer Integrity

So far, Jif/split has not addressed concurrency, which is inherently a concern for security in distributed systems. While it would be possible to make use of the structured synchronization mechanisms presented in Chapter 5, the Jif/split prototype was not been designed to take advantage of concurrency in the source program—this is a limitation inherited from Jif. Instead, Jif/split takes advantage of the single-threaded nature of the source program by using simpler ordered linear continuations of Chapter refch:cps.

Consider a scenario with three hosts: h_1 and h_2 have high integrity, and l has relatively lower integrity (that is, its integrity is not equal to or greater than that of h_1 or h_2). Because the program has been partitioned into code fragments, each host is prepared to accept control transfers at multiple entry points, each of which begins a different code fragment. Some of the code fragments on h_1 and h_2 make use of the greater privilege available due to higher integrity (e.g., the ability to declassify certain data).

Suppose the source program control flow indicates control transfer in the sequence $h_1 \longrightarrow l \longrightarrow h_2$. A potential attack is for l to improperly invoke a privileged code fragment residing on h_2, therefore violating the behavior of the original program and possibly corrupting or leaking some data. Hosts h_1 and h_2 can prevent these attacks by simply denying l the right to invoke entry points that correspond to privileged code, but this strategy prevents h_2 from using its higher privileges after control has passed through l—even if this control transfer was supposed to occur according to the source program.

The mechanism to prevent these illegal control transfers is based on a stack discipline for manipulating capabilities. Each capability represents a linear continuation, and the stack keeps track of the ordering between them.

The intuition is that the block structure and sequential behavior of the source program, which are embodied at run-time by the stack of activation records, induce a similar LIFO property on linear continuations (and the pcintegrity). The deeper the stack, the more data the program counter depends on, and consequently, the lower its integrity.

In Jif, the correspondence between stack frames and linear continuations is not perfect because the pc label need not decrease in lock step with every stack frame. A single stack frame may be used by a block of code that is partitioned across several hosts of differing integrity, for example. To distinguish between the stack of activation records (whose elements are represented by `FrameID` objects) and the stack of continuation tokens, the latter is called the ICS—integrity control stack. The ICS can be thought of as an implementation of the linear context K used in the typechecking rules for $\lambda^{\mathrm{CPS}}_{\mathrm{SEC}}$.

Informally, in the scenario above, the first control transfer (from h_1 to l) pushes a capability (a continuation) for return to h_2 onto the ICS, after which computation is more

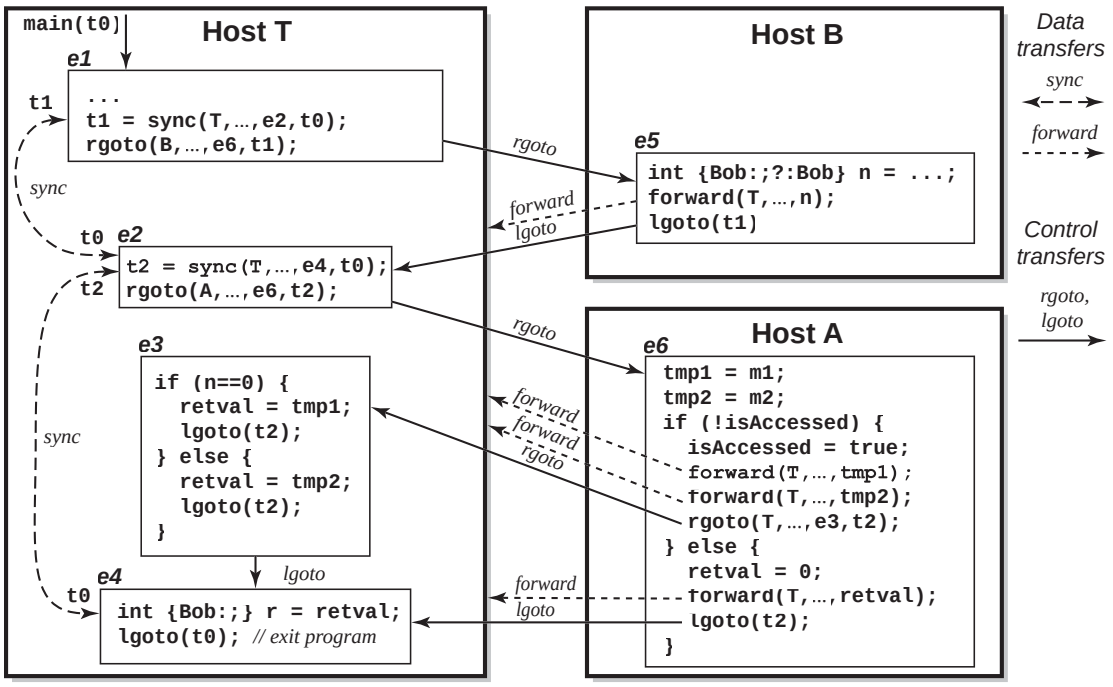


Figure 8.4: Control flow graph of the oblivious transfer program

restricted (and hence may reside on a less trusted machine). The second control transfer (from l to h_2) consumes the capability and pops it off the ICS, allowing h_2 to regain its full privileges. The idea is that before transferring control to l, trusted machines h_1 and h_2 agree that the only valid, privileged entry point between them is the one on h_2. Together, they generate a capability for the entry point that h_1 passes to l on the first control transfer. Host l must present this capability before being granted access to the more privileged code. Illegal attempts to transfer control from l to h_1 or to h_2 are rejected because h_1 and h_2 can validate the (unique) capability.

8.3.4 Example Control Flow Graph

Figure 8.3 shows the signatures for the three control transfer facilities: `rgoto` (for "regular" control transfers that do not affect the ICS), `lgoto` (for "linear" transfers—ICS pops), and `sync` (for generating capabilities—ICS pushes; these correspond to `letlin`). The continuation capabilities are represented as `Token` objects. In addition to the code fragment to be jumped to (given by the `EntryPt` argument), control transfer is to a specific stack frame (given by `FrameID`) on a particular host.

The next section describes in detail the operation of these mechanisms, but first it is helpful to see an example of their use.

Figure 8.4 shows the control-flow graph of a possible splitting of the oblivious transfer example in a host environment that contains Alice's machine A, Bob's machine B and the partially trusted server, T from Section 4.2. For completeness, the following describes the unoptimized behavior; optimizations that affect the partitioning process and run-time performance are discussed in Sections 8.5 and 8.6.

The figure shows only a fragment of the `main`[3] method. Host T initially has control and possesses a single capability `t0`, which is on top of the ICS. Bob's host is needed to initialize `n`—his choice of Alice's two fields. Recall that `{?:Bob}` $\not\sqsubseteq$ `{?:Alice}`, which means that B is relatively less trusted than T. Before transferring control to B, T `sync`'s to a suitable return point (entry `e2`), which pushes a new capability, `t1`, onto the ICS (hiding `t0`). The `sync` operation then returns this fresh capability token, `t1`, to `e1`.

Next, T passes `t1` to entry point `e5` on B via `rgoto`. There, Bob's host computes the value of `n` and returns control to T via `lgoto`, which requires the capability `t1` to return to a host with relatively higher integrity. Upon receiving this valid capability, T pops `t1`, restoring `t0` as the top of the ICS. If instead B maliciously attempts to invoke any entry point on either T or A via `rgoto`, the access control checks deny the operation. The only valid way to transfer control back to T is by invoking `lgoto` with one-time capability `t1`. Note that this prevents Bob from initiating a race to the assignment on line `11` of the example, which might allow two of his transfer requests (one for `m1` and one for `m2`) to be granted and thus violate Alice's declassification policy.

Alice's machine must check the `isAccessed` field, so after B returns control, T next `sync`s with the return point of `transfer` (the entry point `e4`), which again pushes new token `t2` onto the ICS. T then transfers control to `e6` on A, passing `t2`. The entry point `e6` corresponds to the beginning of the `transfer` method.

Alice's machine performs the comparison, and either denies access to Bob by returning to `e4` with `lgoto` using `t2`, or forwards the values of `m1` and `m2` to T and hands back control via `rgoto` to `e3`, passing the token `t2`. If Bob has not already made a request, T is able to check `n` and assign the appropriate value of `tmp1` and `tmp2` to `retval`, then jump to `e4` via `t2`. The final block shows T exiting the program by invoking the capability `t0`.

8.3.5 Control Transfer Mechanisms

This section describes how `rgoto`, `lgoto`, and `sync` manipulate the ICS, which is itself distributed among the hosts, and defines the dynamic checks that must occur to maintain the desired integrity invariant. The implementation details given here are one way of implementing the ordered synchronization handlers described in Chapters 5 and 7.

[3]The `main` method and constructors are omitted from Figure 8.2 to simplify the presentation; they contain simple initialization code. This description also omits the details of `FrameID` objects, which are unimportant for this example.

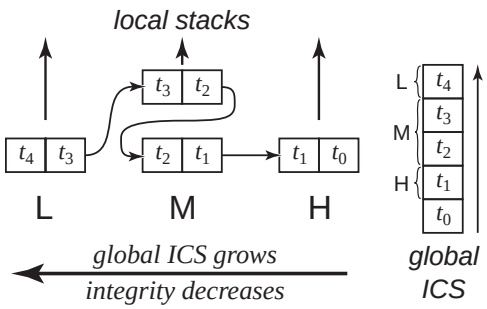


Figure 8.5: Distributed implementation of the global stack

A capability token `t` is a tuple $\{h, f, e\}_{k_h}$ containing a `HostID`, a `FrameID`, and an `EntryPt`. It represents a linear continuation—or, equivalently, a synchronization channel—that expects no arguments and whose body contains the code e. To prevent forgery and ensure uniqueness, the tuple is appended to its hash with h's private key and a nonce.

The global ICS is represented by a collection of local stacks, as shown in Figure 8.5. Host h's local stack, s_h, contains pairs of tokens $(\mathtt{t}, \mathtt{t}')$ as shown. The intended invariant is that when the top of h's stack, $\mathtt{top}(s_h)$, is $(\mathtt{t}, \mathtt{t}')$, then `t` is the token most recently issued by h. Furthermore, the only valid `lgoto` request that h will serve must present the capability `t`. The other token, $\mathtt{t}'$, represents the capability for the next item on the global stack; it is effectively a pointer to the tail of the global ICS.

To show that these distributed stacks enforce a global stack ordering on the capabilities, the proof, given in Section 8.4, establishes a stronger invariant of the protocol operations. Whenever control is transferred to low-integrity hosts, there is a unique reentry point on high-security hosts that permits high-integrity computation. This uniqueness ensures that if a low-integrity host is bad, it can only jeopardize the security of low-integrity computation.

The recipients of control transfer requests enforce the ordering protocol. Assume the recipient is the host h, and the initiator of the request is i. The table in Figure 8.6 specifies h's action for each type of request. The notation $e(f, \mathtt{t})$ is a local invocation of the code identified by entry point e in stack frame f, passing the token `t` as an additional argument.

This approach forces a stack discipline on the integrity of the control flow: `rgoto` may be used to transfer control to an entry point that requires lesser or equal integrity; `lgoto` may transfer control to a higher-integrity entry point—provided that the higher-integrity host previously published a capability to that entry point. These capabilities can be used at most once: upon receiving an `lgoto` request using the valid capability `t`, h pops its local capability stack, thereby invalidating `t` for future uses. Calls to `sync`

and `lgoto` thus come in pairs, with each `lgoto` consuming the capability produced by the corresponding `sync`.

Just as the run-time system must dynamically prevent malicious hosts from improperly accessing remote fields, it also must ensure that bad hosts cannot improperly invoke remote code. Otherwise, malicious hosts could indirectly violate the integrity of data affected by the code. Each entry point e has an associated dynamic access control label $\mathtt{I}_e$ that regulates the integrity of machines that may remotely invoke e. The receiver of an `rgoto` or `sync` request checks the integrity of the requesting host against $\mathtt{I}_e$ as shown in Figure 8.6. The label $\mathtt{I}_e$ is given by $(\bigsqcap_{v \in D(e)} \mathtt{L}_v) \sqcap \mathtt{I}_P$, where $D(e)$ is the set of variables and fields written to by the code in e and $\mathtt{I}_P$ is the integrity label of the principals, P, whose authority is needed to perform any declassifications in e.

The translation phase described in the next section inserts control transfers into the source program. To prevent confidentiality and integrity policies from being violated by the communications of the transfer mechanisms themselves, there are constraints on where `rgoto` and `sync` may be added.

Suppose a source program entry point e is assigned to host i, but doing so requires inserting an `rgoto` or `sync` to another entry point e' on host h. The necessary constraints are:

$$C(\mathsf{pc}) \sqsubseteq \mathtt{C}_h \qquad \mathtt{I}_i \sqsubseteq \mathtt{I}_{e'} \qquad \mathtt{I}_e \sqsubseteq \mathtt{I}_{e'}.$$

The first inequality says that i cannot leak information to h by performing this operation. The second inequality says that host i has enough integrity to request this control transfer. This constraint implies that the dynamic integrity checks performed by h are guaranteed to succeed for this legal transfer—the dynamic checks are there to catch malicious machines, not well-behaved ones. Finally, the third constraint says that the code of the entry point e itself has enough integrity to transfer the control to e'. Furthermore, because `sync` passes a capability to h, it requires the additional constraint that $I_h \sqsubseteq I(\mathsf{pc})$, which limits the damage h can do by invoking the capability too early, thus bypassing the intervening computation.

These enforcement mechanisms do not attempt to prevent denial of service attacks, as such attacks do not affect confidentiality or integrity. These measures *are* sufficient to prevent a bad low-integrity host from launching race-condition attacks against the higher integrity ones: hosts process requests sequentially, and each capability offers one-shot access to the higher integrity hosts.

While our restrictive stack-based control transfer mechanism is *sufficient* to provide the security property of Section 7.1, it is not *necessary*; there exist secure systems that lie outside the behaviors expressible by the ICS. However, following the stack discipline is sufficient to express many interesting protocols that move the thread of control from trusted hosts to untrusted hosts and back. Moreover, the splitter determines when a source program can obey the stack ordering and generates the protocol automatically.

Request for h	Description	h's Action

`rgoto(`h, f, e, t`)`	Transfers control to the entry point e in frame f on the host h. Host i's current capability `t` is passed to h.	`if (`$\mathtt{I}_i \sqsubseteq \mathtt{I}_e$`) {`
		$e(f, t)$
		`} else ignore`

`lgoto(`t`)`
where
$(t = \{h, f, e\}_{k_h})$
	Pops h's local control stack after verifying the capability t; control moves to entry point e in frame f on host h, restoring privileges.	`if (`*top*(s_h) `= (`t`,`t'`)) {`
		pop(s_h);
		$e(f, t')$;
		`} else ignore`

`sync(`h, f, e, t`)`	Host h checks i's integrity; if sufficient, h returns to i a new capability (nt) for entry point e in frame f.	`if (`$\mathtt{I}_i \sqsubseteq \mathtt{I}_e$`) {`
		$nt = \{h, f, e\}_{k_h}$;
		push$(s_h, (nt, t))$;
		send$(h)_i(f, nt)$;
		`} else ignore`

Figure 8.6: Host h's reaction to transfer requests from host i

8.4 Proof of Protocol Correctness

This section proves that the control-transfer protocols generated by Jif/split protect the integrity of the program counter. The purpose of these protocols is to ensure that at any point in time, the set of (relatively) low-integrity hosts has access to at most one capability that grants access to high-integrity (more privileged) code. This prevents the low-integrity hosts from having a choice about how privileges are restored to the computation, which means that they cannot inappropriately invoke declassifications that reside on more trusted machines. Thus, untrusted hosts can jeopardize only low-integrity computation—the control behavior of the high-integrity parts of the split code is the same as in the original, single-host source program. The *Stack Integrity Theorem*, described below, proves that the distributed systems generated by Jif/split satisfy this security invariant.

To arrive at this result, we need to first model the behavior of the system at an appropriate level of detail. There are two parts to this model: First, Jif/split statically produces code fragments to distribute among the hosts. These code fragments obey static constraints imposed by the compiler and splitter, but they also have a run-time effect on the behavior of the system—for instance, a code fragment may terminate in a

control transfer to a different host. Second, the run-time system of each host manipulates stacks of capability tokens that are used for dynamic checking. The combination of static constraints on the partitions created by the splitter and dynamic checks performed by the run-time system protects the control-transfer integrity.

8.4.1 Hosts

Let H be a set of known hosts $\{h_1, \ldots, h_n\}$. We assume that each host h has an associated integrity label $\mathtt{I}_h$. Fix an integrity label ι, used to define the relative trust levels between hosts. Let $H_G = \{h \mid \mathtt{I}_h \sqsubseteq \iota\}$ be the set of *good hosts*, and let $H_B = \{h \mid \mathtt{I}_h \not\sqsubseteq \iota\}$ be the set of *bad hosts*. For example, with respect to a single principal, p, we might choose $\iota = \{\mathtt{?:p}\}$. In this case, H_G is the set of hosts trusted by p and H_B is the set of hosts not trusted by p. Note that $H_G \cup H_B = H$. Throughout this section, we call objects with label $\not\sqsubseteq \iota$ *bad* and objects with label $\sqsubseteq \iota$ *good*. [4]

We assume that good hosts follow the protocols and that bad hosts might not. In particular, bad hosts may attempt to duplicate or otherwise misuse the capability tokens; they may also generate spurious messages that contain tokens previously seen by any bad host.

The run-time system provided also ensures that good hosts execute requests atomically. In particular, a host h that is executing the sequential code fragment corresponding to an entry point e_h will not be executing code for any other entry point e'_h on h. This assumption justifies the state-transition approach described below, because we show that the local processing on each host, if performed atomically, preserves a global invariant.

One subtle point is that despite the distinction between good and bad hosts, not all low-integrity computation takes place at bad hosts. Intuitively, running low-integrity computation on high-integrity hosts is allowed because the integrity constraints inferred for the source program are lower bounds on the integrity required by the hosts. It is therefore safe to use a more secure host than necessary. Consequently, high-integrity hosts may accept requests to run low-integrity parts of the program from bad hosts. In general, several good hosts may be executing concurrently—perhaps because they are responding to low-integrity requests generated by bad hosts. However, the intended effect is that such concurrency does not affect high-integrity computation. The Stack Integrity Theorem establishes that high-integrity computation is still single-threaded, despite this possible concurrency introduced by bad hosts.

[4] Recall that in the integrity lattice, labels representing more integrity are lower in the $\sqsubseteq$ order.

8.4.2 Modeling Code Partitions

To capture the static constraints on the behavior of good hosts, we define the notion of an *entry point*: an entry point e is the name of a code partition generated by the splitter—it can be thought of as a remote reference to a single-threaded piece of program that resides entirely on one host. An entry point names a program point to which control may be transferred from a remote host. Each entry point e has an associated integrity label $\mathtt{I}_e$ as described in Section 8.3. Note that a low-integrity entry point may be located at a high-integrity machine. Let E be the set of entry points generated by a given program, and let E_h be the set of entry points located on host h.

Because our proof is concerned with the control transfers between hosts, we can ignore the details of the sequential code named by an entry point. Consequently, an entry point e on a good host $h \in H_G$ can be thought of as a function that takes a frame f and a token t and produces an *action*, which is a pair (h, r) of the host h and an operation request r, in one of the following forms:

1. $e(f,t) = (h, \mathtt{rgoto}(h', f', e', t))$: Host h transfers control to entry e' on h' in frame f'.

2. $e(f,t) = (h, \mathtt{sync}(h', f', e', t))$: Host h requests a sync with entry e' on h' and frame f'; h blocks until it receives a reply.

3. $e(f,t) = (h, \mathtt{lgoto}(t))$: Host h transfers control to entry e' on h' in frame f' if $t = \{h', f', e'\}_{k_{h'}}$.

The recipient, or *destination host* of an action is the host h' in the requests described above. If r is a request, the notation $dest(r)$ indicates the destination host.

The metavariable t indicates a capability token, which is a tuple $\{h, f, e\}_{k_h}$ consisting of a host identifier, a frame identifier, and an entry point identifier. To prevent forgery and ensure uniqueness of such tokens, the tuple is appended to its hash with h's private key and a nonce. Thus a token of this form can be generated only by host h, but its validity can be checked by any host with h's public key. (In the text below, we use t, t', u, u', etc., to range over tokens.)

The `sync` operation is the only control transfer mechanism that involves an exchange of messages between hosts. (Both `rgoto` and `lgoto` unconditionally transfer control to the recipient.) Because the initiator (host h in the above description) of a `sync` request expects the recipient (h') to respond with a freshly generated token, h blocks until it gets h''s reply. For the purposes of analysis, we treat h's behavior after issuing a `sync` request to h' as an entry point *send*$(h')_h$ on h to which h' may return control. The integrity of the entry point *send*$(h')_h$ is the integrity $\mathtt{I}_e$ of the entry point containing the `sync` operation.

Just like any other entry point, *send*$(h')_h$ is a function that takes a frame and token and returns an action. In the case that h' is also a good host, and hence follows the appropriate protocol, h' will generate a fresh token t' and return control to h causing the entry point *send*$(h')_h(f, t')$ to be invoked.

For example, the code fragment below, located on a host h, includes two entry points, e and *send*$(h')_h$:

e : `x = x+1;`
`z = x-2;`
`sync`(h', f', e', t)`;`
send$(h')_h$: `y = y +2;`
`rgoto`(h'', f'', e'', t')`;`

These entry points are modeled as functions

$$e(f, t) = \texttt{sync}(h', f', e', t)$$

and

$$\textit{send}(h')_h(f, t') = \texttt{rgoto}(h'', f'', e'', t')$$

When e is invoked, the local computations are performed and then host h sends a `sync` request to h', causing h' to block (waiting for the response to be sent by h' to the entry point *send*$(h')_h$).

Note that this description of entry points is valid only for good hosts—bad hosts may do anything they like with the tokens they receive. For good hosts, the code generated by our splitter satisfies the above abstract specification by construction. For instance, the splitter always terminates the thread of control on a host by inserting a single `rgoto` or an `lgoto`—the splitter never generates several control transfers in a row from the same host (which would cause multiple remote hosts to start executing concurrently). As discussed in Section 8.3, the splitter also follows some constraints about where `rgoto` and `sync` can be inserted. In order for host h to perform an `rgoto` or `sync` at an entry point e to entry point e' on host h', the following static constraint must be satisfied: $\texttt{I}_h \sqcup \texttt{I}_e \sqsubseteq \texttt{I}_{e'}$. The `sync` operation requires an additional constraint $\texttt{I}_{h'} \sqsubseteq I(\texttt{pc})$, which limits the damage h' can do by invoking the capability too early, bypassing the intervening computation. Since we assume good hosts are not compromised, `sync`s or `rgoto`s issued by good hosts satisfy these label inequalities.

8.4.3 Modeling the Run-time Behavior

To capture the dynamic behavior of the hosts, we need to model the state manipulated by the run-time system. Let T be the set of all possible tokens. For the purposes of this proof, we assume that a host may generate a fresh, unforgeable token not yet used

anywhere in the system. In practice, this is accomplished by using nonces. The run-time system's local state on a host h includes a *token stack*, which is a list of pairs of tokens

$$(t_1, t_1') : (t_2, t_2') : \ldots : (t_n, t_n').$$

Stacks grow to the left, so pushing a new pair (t_0, t_0') onto this stack yields:

$$(t_0, t_0') : (t_1, t_1') : (t_2, t_2') : \ldots : (t_n, t_n').$$

Because only good hosts are trusted to follow the protocol, only good hosts necessarily have token stacks. For each $h \in H_G$ we write s_h for the local token stack associated with the good host h.

Let s_h be the stack $(t_1, t_1') : \ldots : (t_n, t_n')$. We use the notation *top*(s_h) to denote the top of the stack s_h: the pair (t_1, t_1'). If s_h is empty (that is, $n = 0$), then *top*(s_h) is undefined. For the pair (t, t'), let *fst*$(t, t') = t$ be the first projection and *snd*$(t, t') = t'$ be the second projection. We overload the meaning of *fst* and *snd* to include entire stacks: *fst*$(s_h) = \{t_1, t_2, \ldots, t_n\}$ and *snd*$(s_h) = \{t_1', t_2', \ldots, t_n'\}$.

When a good host h receives a request from initiator i, h reacts to the message according to the table in Figure 8.6. (We can't say anything about what a bad host does upon receiving a message, except observe that the bad host may gain access to new tokens.) The *pop* and *push* operations manipulate h's local stack.

Note that h will react to a `lgoto` request only if the token used to generate the request is on top of its local stack. The point of our protocol is to protect against any bad host (or set of bad hosts) causing more than one these "active tokens" to be used at time.

The global state of the system at any point in time is captured by a *configuration*, which is a tuple $\langle s,\ R,\ T_R \rangle$. Here, s is the mapping from good hosts to their local stacks, R is a predicate on E indicating which entry points are running, and $T_R \subseteq T$ is the set of tokens released to bad hosts or generated by them. The intention is that T_R contains all tokens that have been passed to the bad hosts or to low integrity entry points before reaching this system configuration during the computation.

As always, we cannot assume anything about what code a bad host is running. Thus, we use the predicate R only for those entry points located on good hosts. The notation $R[e \mapsto x]$ for $x \in \{\mathtt{t}, \mathtt{f}\}$ stands for the predicate on entry points that agrees with R everywhere except e, for which $R[e \mapsto x](e) = x$.

The behavior of a Jif/split system can now be seen as a *labeled transition system* in which the states are system configurations and the transitions are actions. The notation

$$\langle s,\ R,\ T_R \rangle \xrightarrow{(h,r)} \langle s',\ R',\ T_R' \rangle$$

indicates that the left configuration transitions via the action (h, r) to yield the right configuration. The effects of the transition on the configuration depend on the action.

For example, a successful `lgoto` request will cause a good host h to pop its local token stack. Thus, for that transition $s'_h = \mathit{pop}(s_h)$. Other effects, such as passing a token to a bad host, relate T_R and T'_R. The proof cases in Section 8.4.5 describe the effects of each transition.

Not every transition sequence is possible—some are ruled out by the static constraints, while some are ruled out by the dynamic checks of good hosts. Thus, we must make some further *modeling assumptions* about the valid transition systems.

- If, during a computation, the configuration S transitions to a configuration S' via an action performed by a good host, then that action is actually the result of evaluating an entry point on that host:

$$\langle s,\, R,\, T_R\rangle \xrightarrow{(h,r)} \langle s',\, R',\, T'_R\rangle \wedge h \in H_G \Rightarrow \exists e \in E_h, f, t\ .\ (e(f,t) = (h,r)) \wedge R(e)$$

- If the initiator of an action is a bad host or an entry point in E_B, then any tokens appearing in the action are available to the bad hosts (they are in the set T_R).

- Communication between bad hosts does not change the relevant parts of the global state:

$$\langle s,\, R,\, T_R\rangle \xrightarrow{(h,r)} \langle s',\, R',\, T'_R\rangle \ \wedge\ h \in H_B \ \wedge\ \mathit{dest}(r) \in H_B \Rightarrow (s = s', R = R', T_R = T'_R)$$

8.4.4 The stack integrity invariant

This section defines the stack integrity invariant, which will establish the correctness of control-transfer protocol. First, we define some useful notation for describing the relevant properties of the system configurations.

Each token $t = \{h, f, e\}_{k_h}$ generated by a good host h corresponds to the entry point e. Because tokens are hashed with a private key, it is possible to distinguish tokens generated by good hosts from tokens generated by bad hosts. For any token t, let $\mathit{creator}(t)$ be the host that signed the token (in this case, host h). Using these pieces of information, we can define the integrity level of a token as:

$$\mathtt{I}_t = \begin{cases} \mathtt{I}_e & h = \mathit{creator}(t) \in H_G \\ \mathtt{I}_h & h = \mathit{creator}(t) \in H_B \end{cases}$$

Define a *good token* to be any token t for which $\mathtt{I}_t \sqsubseteq \iota$. Let T_G be the set of all good tokens and $T_B = T \setminus T_G$ be the set of bad tokens.

Just as we have partitioned hosts and tokens into good and bad sets, we define *good entry points* and *bad entry points*. The set of low integrity entry points can be defined

as $E_G = \{e \mid e \in E \wedge \mathtt{I}_e \not\sqsubseteq \iota\}$. Correspondingly, let $E_B = E \setminus E_G$ be the set of high-integrity entry points.

Recall from Section 8.3 that the local stacks are intended to simulate a global integrity control stack (ICS) that corresponds to the program counter of the source program. Due to the presence of bad hosts, which may request `sync` operations with low-integrity entry points located at good hosts, the global structure described by the composition of the local stacks may not be a stack. To see why, consider a bad host that obtains a good token t and then uses the good token in `sync` requests to bad entry points on two different good hosts, h_1 and h_2. The resulting configuration of local stacks contains $s_{h_1} = \ldots : (t_1, t)$ and $s_{h_2} = \ldots : (t_2, t)$. Thus the global ICS isn't a stack, but a directed, acyclic graph. However, the crucial part of the invariant is that the global ICS is a stack with respect to good tokens.

The key to defining the invariant is to relax the notion of "stack" with respect to bad tokens. Observe that the global structure on T induced by the local stacks is captured by the directed graph whose nodes are tokens in T and whose edges are given by $\{(t, t') \mid \exists h \in H_G.(t, t') \in s_h\}$. If this structure truly described a stack there would be a single component:

$$t_1 \rightarrow t_2 \rightarrow \ldots t_{n-1} \rightarrow t_n$$

with no other edges present. (Here t_n is the bottom of the stack.) Instead, we show that the graph looks like:

$$B \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots t_{n-1} \rightarrow t_n$$

where B is an arbitrary dag whose nodes are only bad tokens and t_1 through t_n are good tokens.

We formalize the 'B' part of the graph by observing that the edges in B and the ones between B and t_1 originate from a bad token. Because the good hosts' local stacks change during a run of the system, it is convenient to define some notation that lets us talk about this property in a given configuration. Let S be a system configuration, $\langle s,\, R,\, T_R \rangle$. Define:

$$t \prec_S t' \;\; \Leftrightarrow \;\; \exists h \in H_G.(t, t') \in s_h \;\wedge\; t \in T_B$$

The relation $t \prec_S t'$ says that the bad token t appears immediately before the token t' in the graph above. Its transitive closure $t \prec_S^* t'$ says that there are bad tokens u_1 through u_n such that

$$t \rightarrow u_1 \rightarrow \ldots \rightarrow u_n \rightarrow t'$$

appears as part of the ICS—these bad tokens are a subgraph of the dag B. Note that t' may be either a good or a bad token. Property (iv) of the invariant (see below) says that there is at most one good token reachable through the relation $\prec_S^*$—that is, at most one good token can be a successor of B, the bad portion of the ICS.

If we conservatively assume that all bad tokens are available to the bad hosts, then $t \prec_S^* t'$ says that the bad hosts can "get to" the token t' by doing an appropriate series of `lgoto` operations (each of which will pop a t_i off the ICS).

We next define some auxiliary concepts needed to state the stack invariant:

$$\begin{array}{rcl} \mathit{Tok}_S & = & \{t \mid \exists h \in H_G.t \in \mathit{fst}(s_h)\} \\ \mathit{Tok}_S(h) & = & \mathit{fst}(s_h) \text{ whenever } h \in H_G \\ \mathit{Active}_S(t) & \Leftrightarrow & t \in \mathit{Tok}_S \wedge \exists t' \in T_R.(t = t') \vee (t' \prec_S^* t) \end{array}$$

The set Tok_S is just the set of tokens appearing in the left part of any good host's local stack—this is a set of tokens for which some good host might grant an `lgoto` request (it is conservative because it includes all the left-hand-side tokens of the stack, not just the top of the stack). The set $\mathit{Tok}_S(h)$ is the restriction of Tok_S to a particular good host h. $\mathit{Tok}_S(h)$ is exactly the set of tokens issued by h that have not been consumed by a valid `lgoto` request. Finally, the predicate $\mathit{Active}_S(t)$ determines the subset of Tok_S "reachable" from the tokens available to bad hosts.

Definition 8.4.1 *A configuration S satisfies the Stack Integrity Invariant (SII) if and only if:*

(i) $\forall t, t' \in T_G.\mathit{Active}_S(t) \wedge \mathit{Active}_S(t') \Rightarrow t = t'$

Uniqueness of exposed, good tokens.

(ii) $\exists e \in E_G.\ R(e) \Rightarrow \neg\exists t \in T_G.\mathit{Active}_S(t)$

When good code has control, there are no good, active tokens.

(iii) $\forall e, e' \in E_G.\ R(e) \wedge R(e') \Rightarrow e = e'$

Good code is single threaded.

(iv) $\forall t_1, t_1', t_2, t_2'.\ (t_1 \prec_S^* t_1') \wedge (t_2 \prec_S^* t_2') \wedge (t_1', t_2' \in T_G) \Rightarrow t_1' = t_2'$

Unique good successor token.

(v) $\forall h_1, h_2 \in H_G.\ h_1 \neq h_2 \Rightarrow \mathit{Tok}_S(h_1) \cap \mathit{Tok}_S(h_2) = \emptyset$.

No two good hosts generate identical tokens.

(vi) $\forall h \in H_G.\ s_h = (t_1, t_1') : \ldots (t_n, t_n') \Rightarrow$ t_1 *through* t_n *are pairwise distinct.*

Stack Integrity Theorem *If S is a configuration satisfying the SII and S transitions to S', then S' satisfies the SII.*

Note that condition (i) of the SII implies that if t is a good token on the top of some good host's local stack and t has been handed out as a capability to the bad hosts

($t \in T_R$), then t is unique—there are no other such capabilities available to the bad hosts. Because only good hosts can create such tokens, and they do so only by following the source program's control flow, the bad hosts cannot subvert the control-flow of high-integrity computation.

8.4.5 Proof of the stack integrity theorem

Suppose $S = \langle s,\ R,\ T_R \rangle$ is a configuration satisfying the Stack Integrity Invariant (SII). To show that our system preserves the SII, we reason by cases on all the possible actions in the system. In other words, we want to show that after any possible action, the resulting system configuration $S' = \langle s',\ R',\ T'_R \rangle$ still satisfies SII. Note that, by assumption, any communication between bad hosts does not change the state of the configuration, so we may safely eliminate those cases. We first make a few observations:

1. If $s' = s$ then invariants (iv), (v), and (vi) hold trivially because they depend only on the state of the local stacks.

2. If $s' = s$ and $T'_R = T_R$ then $\mathit{Active}_{S'} = \mathit{Active}_S$ and invariant (i) is satisfied trivially.

3. Changing the running predicate of a bad entry point does not affect invariants (ii) or (iii)—changing the running predicate on good entries, or changing Active_S may affect (ii) and (iii).

Case I. S transitions via $(h_1, \texttt{rgoto}(h_2, f_2, e_2, t))$.

(a) $h_1 \in H_B$, $h_2 \in H_G$, and $I_{h_1} \not\sqsubseteq I_{e_2}$.

In this case, because h_2 is a good host, the dynamic check on `rgoto` prevents the system configuration from changing. Thus $S' = S$, and the invariant is immediate.

(b) $h_1 \in H_B$, $h_2 \in H_G$, and $I_{h_1} \sqsubseteq I_{e_2}$.

Because $h_1 \in H_B$, we have $I_{h_1} \not\sqsubseteq \iota$ and thus $I_{e_2} \not\sqsubseteq \iota$. Consequently, $e_2 \in E_B$. Thus $T'_R = T_R \cup \{t\} = T_R$, $R' = R[e_2 \mapsto \texttt{t}]$, and $s' = s$. Observations 1, 2, and 3 show that all of the invariants but (ii) hold for S'. Invariant (ii) follows from the fact that $\mathit{Active}_{S'} = \mathit{Active}_S$ and the fact that invariant (ii) holds in S.

(c) $h_1 \in H_G$ and $h_2 \in H_B$.

By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \texttt{rgoto}(h_2, f_2, e_2, t))$$

for some f_1 and, furthermore, $R(e_1)$ holds. In the new configuration, $T'_R = T_R \cup \{t\}$, $R' = R[e_1 \mapsto \mathtt{f}]$, and $s' = s$. Observation 1 shows that invariants (iv), (v) and (vi) hold trivially because $s' = s$. Note that we also have $\prec^*_S = \prec^*_{S'}$.

If $e_1 \in E_B$ then $t \in T_R$, so $T'_R = T_R$ and observations 2 and 3 imply that S' satisfies the SII.

Otherwise $e_1 \in E_G$ and by invariant (iii) of state S no other good entry points are running. Thus, in S' we have $\forall e \in E_G.\neg R(e)$ and it follows that invariants (ii) and (iii) hold in S'. Now we must show that invariant (i) holds. Consider an arbitrary good token $u \in T_G$. Because $T'_R = T_R \cup \{t\}$ we have

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R \cup \{t\}.(u = u') \vee (u' \prec^*_S u) \\ & \Leftrightarrow & [t \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)] \vee \mathit{Active}_S(u) \end{array}$$

By invariant (ii) of configuration S, we have $u \in T_G \Rightarrow \neg \mathit{Active}_S(u)$ and so $u \in T_G \wedge \mathit{Active}_{S'}(u)$ implies $t \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)$. If $t \in T_G$ then by the definition of $\prec^*_S$ we have $\neg\exists u.t \prec^*_S u$ and consequently $u \in T_G \wedge \mathit{Active}_{S'}(u) \Rightarrow u = t$, from which invariant (i) follows immediately. Otherwise, $t \in T_B$ and we have that $u_1, u_2 \in T_G \wedge \mathit{Active}_{S'}(u_1) \wedge \mathit{Active}_{S'}(u_2) \Rightarrow t \prec^*_S u_1 \wedge t \prec^*_S u_2$, but now invariant (iv) of configuration S implies that $u_1 = u_2$ as needed. Thus invariant (i) holds in S'.

(d) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \not\sqsubseteq \mathtt{I}_{e_2}$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \mathtt{rgoto}(h_2, f_2, e_2, t))$$

for some frame f_1 and, furthermore, $R(e_1)$ holds. After the transition, we have $R' = R[e_1 \mapsto \mathtt{f}]$, and, due to the run-time check performed by h_2, we also have $T'_R = T_R$, and $s' = s$. Invariants (i), (iv), (v) and (vi) follow immediately from observations 1 and 2. If $e_1 \in E_B$ then invariants (ii) and (iii) are a direct consequence of the assumption that they hold in S'. To see that (ii) and (iii) hold when $e_1 \in E_G$, note that because $R(e_1)$ we have $\forall e \in E_G.R(e) \Rightarrow e = e_1$ (from invariant (iii)). Because R' agrees with R everywhere but e_1, (iii) holds of R' too. The same reasoning shows that (ii) holds.

(e) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \sqsubseteq \mathtt{I}_{e_2}$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \mathtt{rgoto}(h_2, f_2, e_2, t))$$

for some frame f_1 and, furthermore, $R(e_1)$ holds. After the transition, we have

$$R' = R[e_1 \mapsto \texttt{f}][e_2 \mapsto \texttt{t}]$$

and $s' = s$. This latter fact and observation 1 implies that invariants (iv), (v), and (vi) hold in S'. Note also that $\prec^*_{S'} = \prec^*_S$.

If $e_1 \in E_B$, the modeling assumption tells us that $T'_R = T_R \cup \{t\} = T_R$ because $t \in T_R$. Note that because h_1 is a good host, the static constraint on `rgoto` implies that $\mathtt{I}_{e_1} \sqsubseteq \mathtt{I}_{e_2}$, which in turn implies that $\mathtt{I}_{e_2} \not\sqsubseteq \iota$ and thus $e_2 \in E_B$. Invariants (i), (ii), and (iii) follow immediately from observations 2 and 3, plus the fact that R' agrees with R on all good entry points.

Otherwise, $e_2 \in E_G$. If $e_1 \in E_G$ then $T'_R = T_R$ because t is not passed to a bad entry point. Consequently, $\mathit{Active}_{S'} = \mathit{Active}_S$ and invariant (i) follows immediately. Because $R(e_1) \wedge e_1 \in E_G$, invariant (iii) of S implies that no other good entry points are running in predicate R. Thus, because $R' = R[e_1 \mapsto \texttt{f}][e_2 \mapsto \texttt{t}]$ it is trivial to show that $R'(e) \wedge e \in E_G \Rightarrow e = e_2$, as required. Furthermore, $R(e_1)$ implies that $\neg\exists t \in T_G.\mathit{Active}_S(t)$ and so invariant (ii) holds in configuration S' too.

The last case is when $e_1 \in E_G$ and $e_2 \in E_B$, but this case follows exactly as in the last paragraph of case I(c).

Case II. S transitions via $(h_1, \texttt{lgoto}(t))$ where $t = \{h_2, f_2, e_2\}_{k_{h_2}}$.

(a) $h_1 \in H_B, h_2 \in H_G$ and $\mathit{top}(s_{h_2}) \neq (t, t')$.

In this case, because h_2 is a good host, the dynamic check on `lgoto` prevents the system configuration from changing. Thus $S' = S$, and the invariant is immediate.

(b) $h_1 \in H_B, h_2 \in H_G$ and $\mathit{top}(s_{h_2}) = (t, t')$ for some token t'.

Note that $t \in \mathit{Tok}_S$, and by the modeling assumption, $t \in T_R$ and, consequently, we have $\mathit{Active}_S(t)$. Because h_2 pops its local stack, invariants (v) and (vi) of configuration S imply that $\mathit{Tok}_{S'}(h_2) = \mathit{Tok}_S(h_2) \setminus \{t\}$ and thus $\mathit{Tok}_{S'} = \mathit{Tok}_S \setminus \{t\}$. Also note that because of the stack pop $\prec^*_{S'} \subseteq \prec^*_S$, which implies that SII(iv) holds in configurations S'. Invariants (v) and (vi) hold in S' directly because they hold in S. There are two final cases to consider:

1. $t \in T_G$

It follows that $e_2 \in E_G$, and thus $T'_R = T_R$. Furthermore, $R' = R[e_2 \mapsto \texttt{t}]$. We now show that $\mathit{Active}_{S'}(u) \Rightarrow \mathit{Active}_S(u)$:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in (\mathit{Tok}_S \setminus \{t\}) \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Rightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u) \\ & \Leftrightarrow & \mathit{Active}_S(u) \end{array}$$

We show that in S' it is the case that $\forall u \in T_G.\neg \mathit{Active}_{S'}(u)$, from which invariants (i) and (ii) follow directly. Suppose for the sake of contradiction that $\mathit{Active}_{S'}(u)$ for some $u \in T_G$. Then, by the implication above, we have $\mathit{Active}_S(u)$. Recall that $\mathit{Active}_S(t)$, and so by invariant (ii) of the configuration S, we have $u = t$. But, $\mathit{Active}_{S'}(u) \Rightarrow u \in \mathit{Tok}_{S'} = \mathit{Tok}_S \setminus \{t\}$, which implies that $u \neq t$, a contradiction.

Lastly, we must show that SII(iii) holds in configuration S'. We know that $R'(e_2) = \texttt{t}$. Suppose $e \in E_G$ and assume $e \neq e_2$. We must show that $\neg R'(e)$. But, $R'(e) = R[e_2 \mapsto \texttt{t}](e) = R(e)$. Recalling once more that $\mathit{Active}_S(t) \wedge t \in T_G$, the contrapositive of SII(ii) for configuration S implies that $\neg R(e)$ as desired.

2. $t \in T_B$
   It follows that $e_2 \in E_B$, and thus $T'_R = T_R \cup \{t'\}$. Furthermore, $R' = R[e_2 \mapsto \texttt{t}]$ and we immediately obtain invariant (iii) via observation 3. First note that $t \prec_S t'$ because $(t, t') \in s_{h_2}$, and, consequently, if $t' \in T_B$ we have $t \prec^*_S u \wedge u \neq t' \Rightarrow t' \prec^*_S u$ for any u. We need this fact to derive the implication marked $\star$ below:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in (\mathit{Tok}_S \setminus \{t\}) \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_{S'} u) \\ \star & \Rightarrow & u \in \mathit{Tok}_S \wedge ((u = t') \vee \exists u' \in T_R.(u = u') \vee (u' \prec^*_{S'} u)) \\ & \Leftrightarrow & (u \in \mathit{Tok}_S \wedge u = t') \vee \mathit{Active}_S(u) \end{array}$$

If $t' \in \mathit{Tok}_S$, then by definition, we have $\mathit{Active}_S(t')$; otherwise in the left conjunct above we have $(u \in \mathit{Tok}_S \wedge u = t' \wedge t' \notin \mathit{Tok}_S) = \texttt{f}$. Thus, in either case, the expression above reduces to $\mathit{Active}_S(u)$ and we have $\mathit{Active}_{S'}(u) \Rightarrow \mathit{Active}_S(u)$. Invariant (i) in S' follows directly from invariant (i) of S; similarly because R' agrees with R on good entry points, invariant (ii) in S' follows directly from invariant (ii) of S.

(c) $h_1 \in H_G$ and $h_2 \in H_G$, and $\mathit{top}(s_{h_2}) \neq (t, t')$.

By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \texttt{lgoto}(t))$$

for some f_1 and, furthermore, $R(e_1)$ holds. Because $h_1 \in H_G$, we have $R' = R[e_1 \mapsto \texttt{f}]$, but the static checks performed by good host h_2 imply that $s' = s$ and $T'_R = T_R$. Invariant (ii) follows from the facts that $R'(e) \Rightarrow R(e)$ and $\mathit{Active}_{S'} = \mathit{Active}_S$. The rest of the invariants follow directly from observations 1,2,and 3.

(d) $h_1 \in H_G$ and $h_2 \in H_G$, and $\mathit{top}(s_{h_2}) = (t, t')$.
By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \texttt{lgoto}(t))$$

for some f_1 and, furthermore, $R(e_1)$ holds. Because $h_1 \in H_G$, we have $R' = R[e_1 \mapsto \texttt{f}][e_2 \mapsto \texttt{t}]$. Note that invariants (iv), (v) and (vi) for S' follow directly from the same invariants of S; popping s_{h_2} implies that $\prec^*_{S'} \subseteq \prec^*_S$.
If $e_1 \in E_B$ then $t \in T_R$ and we use the same reasoning as in Case II.(b).
Otherwise, $e_1 \in E_G$. Note that invariant (ii) of configuration S implies that $\neg\exists u \in T_G.\mathit{Active}_S(u)$ and invariant (iii) implies that $e \in E_G \wedge R(e) \Rightarrow e = e_1$.

1. $t \in T_G$.
   In this case, $e_2 \in E_G$. We first show that invariant (iii) holds in S'. We know that $R'(e_2) = \texttt{t}$, so let $e \in E_G$ be given. We must show that $R'(e) \Rightarrow e = e_2$. Suppose for the sake of contradiction that $R'(e)$ and $e \neq e_2$ then

   $$R'(e) = R[e_1 \mapsto \texttt{f}][e_2 \mapsto \texttt{t}](e) = R[e_1 \mapsto \texttt{f}](e) \wedge R'(e) \Rightarrow e \neq e_1$$

   But this contradicts invariant (iii) of configuration S which says that $e \in E_G \wedge R(e) \wedge R(e_1) \Rightarrow e = e_1$. We conclude that $e = e_2$ as desired.

   Next, we show that $\mathit{Active}_{S'}(u) \Rightarrow \mathit{Active}_S$:

   $$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in (\mathit{Tok}_S \setminus \{t\}) \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Rightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u) \\ & \Leftrightarrow & \mathit{Active}_S(u) \end{array}$$

   From this implication and the fact that $R(e_1)$ holds, we use invariant (ii) to conclude that $\neg\exists t \in T_G.\mathit{Active}_{S'}(t)$. Consequently, S' satisfies invariants (i) and (ii) as required.
2. $t \in T_B$.
   In this case, $e_2 \in E_B$ and it follows that $e_2 \neq e_1$. We show that there are no good, running entry points in S': Let $e \in E_G$ be given. We immediately have that $e \neq e_2$. If $e = e_1$, then as required:

   $$R'(e) = R[e_1 \mapsto \texttt{f}][e_2 \mapsto \texttt{t}](e) = R[e_1 \mapsto \texttt{f}](e) = \texttt{f}.$$

Assuming $e \neq e_1$ we have $R'(e) = R(e)$, and by invariant (iii) of configuration S it follows that $R(e) = \texttt{f}$. Thus, invariants (ii) and (iii) of configurations S' hold trivially.

To show invariant (i), note that $T'_R = T_R \cup \{t'\}$.

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in \mathit{Tok}_S \setminus \{t\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Rightarrow & u \in \mathit{Tok}_S \wedge ((u = t') \vee (t' \prec^*_{S'} u) \\ & & \qquad \vee \exists u' \in T_R.(u = u') \vee (u' \prec^*_S u)) \\ & \Rightarrow & (u \in \mathit{Tok}_S \wedge ((u = t') \vee (t' \prec^*_{S'} u))) \vee \mathit{Active}_S(u) \end{array}$$

Let $u, u' \in T_G$ be given and suppose $\mathit{Active}_{S'}(u) \wedge \mathit{Active}_{S'}(u')$. Note that invariant (ii) of configuration S implies that $\neg\exists u \in T_G.\mathit{Active}_S(u)$, thus we have $\mathit{Active}_{S'}(u) \Rightarrow (u \in \mathit{Tok}_S \wedge (u = t') \vee (t' \prec^*_{S'} u))$ and similarly, $\mathit{Active}_{S'}(u') \Rightarrow (u' \in \mathit{Tok}_S \wedge (u' = t') \vee (t' \prec^*_{S'} u'))$. Suppose $u = t'$. Then $t' \in T_G$ and from the definition of $\prec^*_{S'}$ it follows that $\neg(t' \prec^*_{S'} u')$ which implies that $u' = t' = u$ as required. Otherwise, we have $t' \prec^*_{S'} u$, which means that $t' \in T_B$ and it follows that $t' \prec^*_{S'} u'$. But this implies $t' \prec^*_S u \wedge t' \prec^*_S u'$, so by invariant (iv) of configuration S, we have $u = u'$.

(e) $h_1 \in H_G$ and $h_2 \in H_B$.

By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \texttt{lgoto}(t))$$

for some f_1 and, furthermore, $R(e_1)$ holds. Because $h_1 \in H_G$, we have $R' = R[e_1 \mapsto \texttt{f}]$. Because host $h_2 \in H_B$ we have $s' = s$ and $T'_R = T_R$. Invariant (ii) follows from the facts that $R'(e) \Rightarrow R(e)$ and $\mathit{Active}_{S'} = \mathit{Active}_S$. The rest of the invariants follow directly from observations 1,2,and 3.

Case III. S transitions via $(h_1, \texttt{sync}(h_2, f_2, e_2, t))$.

(a) $h_1 \in H_B$ and $h_2 \in H_G$ and $\texttt{I}_{h_1} \not\sqsubseteq \texttt{I}_{e_2}$.

In this case, because h_2 is a good host, the dynamic check on `rgoto` prevents the system configuration from changing. Thus $S' = S$, and the invariant is immediate.

(b) $h_1 \in H_B$ and $h_2 \in H_G$ and $\texttt{I}_{h_1} \sqsubseteq \texttt{I}_{e_2}$.

Because $h_2 \in H_G$, we have $s'_{h_2} = s_{h_2} : (t', t)$ where $t' = \{h_2, f_2, e_2\}_{k_{h_2}}$ is a fresh token. Invariants (v) and (vi) hold in S' because they hold in S and t' is fresh. Furthermore, because $\texttt{I}_{h_1} \sqsubseteq \texttt{I}_{e_2} \wedge h_1 \in H_B$ it follows that $\texttt{I}_{e_2} \not\sqsubseteq \iota$, and consequently $t' \in T_B$. $R' = R$ because no good host begins running after this transition; invariant (iii) follows directly.

We next show that invariant (iv) is met. Observe that $\prec_{S'} = \prec_S \cup \{(t', t)\}$. In particular, $\neg\exists u.u \prec_{S'} t'$ and so we have

$$u \prec_{S'}^* u' \Leftrightarrow (u \prec_S^* u') \vee (u = t' \wedge t \prec_S^* u')$$

Let u_1, u_1', u_2, u_2' be given and suppose that $(u_1 \prec_{S'}^* u_1') \wedge (u_2 \prec_{S'}^* u_2') \wedge (u_1', u_2' \in T_G)$. From the definition of $\prec_{S'}^*$ we obtain:

$$[(u_1 \prec_S^* u_1') \vee (u_1 = t' \wedge t \prec_S^* u_1')] \ \wedge \ [(u_2 \prec_S^* u_2') \vee (u_2 = t' \wedge t \prec_S^* u_2')]$$

But for each of the four possible alternatives described above, invariant (iv) of configuration S implies that $u_1' = u_2'$ as needed. For example, if $(u_1 \prec_S^* u_1') \wedge (t \prec_S^* u_2')$ then instantiating (iv) with $t_1 = u_1, t_1' = u_1', t_2 = t, t_2' = u_2'$ yields $u_1' = u_2'$. The other cases are similar.

Next we show that invariants (i) and (ii) are maintained. First, note that $T_R' = T_R \cup \{t'\}$ because h_2 sends the fresh token to h_1. Also observe that $\mathit{Tok}_{S'} = \mathit{Tok}_S \cup \{t'\}$ because h_2 has pushed (t', t) onto its local stack. We use the fact that $t \in T_R$ in the derivation marked $\star$ below:

$$\begin{array}{rcl}
\mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T_R'.(u = u') \vee (u' \prec_{S'}^* u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec_{S'}^* u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \\
& & \qquad \vee (u' \prec_S^* u \vee (u' = t' \wedge t \prec_S^* u)) \\
\star & \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u = u') \vee (u' \prec_S^* u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge (u = t' \vee \exists u' \in T_R.(u = u') \vee (u' \prec_S^* u)) \\
& \Leftrightarrow & u = t' \vee \mathit{Active}_S(u)
\end{array}$$

Note that, because $t' \in T_B$, we have $\mathit{Active}_{S'}(u) \wedge u \in T_G \Rightarrow \mathit{Active}_S(u)$. Consequently, invariants (i) and (ii) hold in S' because they hold in S.

(c) $h_1 \in H_G$ and $h_2 \in H_B$.

By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

$$e_1(f_1, t) = (h_1, \texttt{sync}(h_2, f_2, e_2, t))$$

for some frame f_1. Furthermore, $R(e_1)$ holds. After this transition, $s' = s$, and $T_R' = T_R \cup \{t\}$ because t has been passed to a bad host. Observation 1 shows that invariants (iv), (v) and (vi) hold immediately. The new running predicate is:

$$R' = R[e_1 \mapsto \texttt{f}][\mathit{send}(h_2)_{h_1} \mapsto x]$$

Where x can be either `t` or `f`, depending on whether the bad host h_2 replies with a token to h_1. However, because h_1 is a good host, the static constraints on inserting

sync's imply that $\mathtt{I}_{h_2} \sqsubseteq I(\mathsf{pc})$. But then, because $h_2 \in H_B$, it follows that $\mathtt{I}_{h_2} \not\sqsubseteq \iota \Rightarrow I(\mathsf{pc}) \not\sqsubseteq \iota$. Furthermore, because the integrity label on the $\mathit{send}(h_2)_{h_1}$ entry point is just $I(\mathsf{pc})$, we have that $\mathit{send}(h_2)_{h_1} \in E_B$. Thus, whether $\mathit{send}(h_2)_{h_1}$ is running does not affect invariants (ii) and (iii).

Next we calculate the predicate $\mathit{Active}_{S'}$, recalling that $T'_R = T_R \cup \{t\}$:

$$\begin{array}{rcl} \mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\ & \Leftrightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R \cup \{t\}.(u = u') \vee (u' \prec^*_S u) \\ & \Leftrightarrow & (u \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)) \vee \mathit{Active}_S(u) \end{array}$$

1. $e_1 \in E_G$.
   In this case, because $R(e_1)$ holds in configuration S, invariant (ii) tells us that when $u \in T_G$ and $\mathit{Active}_{S'}(u)$ it is the case that $(u \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u))$. To show that invariant (i) holds in S', suppose $u, u' \in T_G \wedge \mathit{Active}_{S'}(u) \wedge \mathit{Active}_{S'}(u')$. Then we have

   $$[u \in \mathit{Tok}_S \wedge (u = t \vee t \prec^*_S u)] \wedge [u' \in \mathit{Tok}_S \wedge (u' = t \vee t \prec^*_S u')]$$

   Now suppose $t \in T_G$. Then by definition $\neg\exists t'.t \prec^*_S t'$, so the above condition on u and u' becomes $u, u' \in \mathit{Tok}_S \wedge u = t \wedge u' = t$ as required. Otherwise, if $t \in T_B$, it follows that $u \neq t$ and $u' \neq t$ and we have $t \prec^*_S u$ and $t \prec^*_S u'$. But then invariant (iv) of configuration S implies that $u = u'$ in this case as well.

   To show that invariants (ii) and (iii) hold, we prove that $\neg\exists e \in E_G.R'(e)$. Suppose for contradiction that there was such an e. By the definition of R', we conclude that $e \neq e_1$, but then $R'(e) = R(e)$. From the modeling assumption, we have $R(e_1)$, and yet invariant (iii) of configuration S implies that $e = e_1$, a contradiction.
2. $e_1 \in E_B$.
   In this case, the modeling assumption tells us that $t \in T_R$, so $T'_R = T_R$. This fact immediately yields that $\mathit{Active}_{S'} = \mathit{Active}_S$, and observations 2 and 3, imply (i) and (iii) hold in S'. Invariant (ii) in S' also follows directly from invariant (ii) in S.

(d) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \not\sqsubseteq \mathtt{I}_{e_2}$.
   This case is identical to I.(d).

(e) $h_1 \in H_G$ and $h_2 \in H_G$ and $\mathtt{I}_{h_1} \sqsubseteq \mathtt{I}_{e_2}$.
   By the modeling assumptions, there exists an $e_1 \in E_{h_1}$ such that

   $$e_1(f_1, t) = (h_1, \mathtt{sync}(h_2, f_2, e_2, t))$$

for some frame f_1 and, furthermore, $R(e_1)$ holds. Because $h_2 \in H_G$, we have $s'_{h_2} = s_{h_2} : (t', t)$ where $t' = \{h_2, f_2, e_2\}_{k_{h_2}}$ is a fresh token. Invariants (v) and (vi) hold in S' because they hold in S and t' is fresh. After the transition, we have

$$R' = R[e_1 \mapsto \mathtt{f}][\mathit{send}(h_2)_{h_1} \mapsto \mathtt{t}]$$

1. $e_2 \in E_G$
   In this case, $t' \in T_G$. It follows that $\neg\exists u.t' \prec_{S'} u$, and consequently $\prec^*_{S'}=\prec^*_S$, from which we conclude that invariant (iv) holds in S'. Because $h_1 \in H_G$ the static constraints on `sync` guarantee that $\mathtt{I}_{e_1} \sqsubseteq \mathtt{I}_{e_2}$, which implies that $e_1 \in E_G$.

   If $\mathit{send}(h_1)_{h_2} \in E_G$ then $T'_R = T_R$. We now show that $\forall e \in E_G.R'(e) \Rightarrow e = \mathit{send}(h_1)_{h_2}$, from which we may immediately derive that invariant (iii) holds in S'. Let $e \in E_G$ be given and suppose for the sake of contradiction that $R'(e) \wedge e \neq \mathit{send}(h_1)_{h_2}$. From the definition of R' we have $R'(e) = R[e_1 \mapsto \mathtt{f}](e)$, and because $R'(e)$ holds, we have that $e \neq e_1$. Thus $R'(e) = R(e)$. But now invariant (iii) and the assumption that $R(e_1)$ holds in S imply that $e = e_1$, a contradiction. Note that $T'_R = T_R \wedge s' = s \Rightarrow \mathit{Active}_{S'} = \mathit{Active}_S$. Invariants (i) and (ii) follow directly from the fact that they hold in configuration S.

   Otherwise, $\mathit{send}(h_1)_{h_2} \in E_B$ and $T'_R = T_R \cup \{t'\}$. We first show that $\forall e \in E_G.\neg R'(e)$, from which invariants (ii) and (iii) follow immediately. Let $e \in E_G$ be given. We know that $e \neq \mathit{send}(h_1)_{h_2}$ because $\mathit{send}(h_1)_{h_2} \in E_B$; thus from the definition of R' we obtain $R'(e) = R[e_1 \mapsto \mathtt{f}](e)$. If $e_1 = e$ we are done. So we have that $e_1 \neq e$. Now, however, $R(e) \wedge (e \in E_G)$ implies via invariant (iii) of configuration S that $e = e_1$, a contradiction. Thus we conclude that $\neg R(e)$ as desired.
   It remains to show that invariant (i) holds in S', so we calculate the predicate $\mathit{Active}_{S'}$. The step marked $\star$ below uses the fact that $t' \in T_G$ (which, from the definition of $\prec_{S'}$ implies that $\neg\exists u.t' \prec^*_{S'} u$):

$$\begin{array}{rcl}
\mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u' = u) \vee (u' \prec^*_{S'} u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R \cup \{t'\}.(u' = u) \vee (u' \prec^*_S u) \\
\star & \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge [(u = t') \vee \exists u' \in T_R.(u' = u) \vee \\
& & \qquad (u' \prec^*_S u)] \\
& \Rightarrow & (u = t') \vee \mathit{Active}_S(u)
\end{array}$$

   Observe that invariant (ii) of configuration S implies that there does not exist $u \in T_G$ such that $\mathit{Active}_S(u)$. Thus, $\mathit{Active}_{S'}(u) \wedge u \in T_G \Rightarrow u = t'$, and invariant (i) of configuration S' follows directly.

2. $e_2 \in E_B$

   In this case, $t' \in T_B$. We use exactly the same reasoning as in Case III(b). to prove that invariant (iv) holds. Note that because $h_1 \in H_G$, the static constraints on `sync` imply that $e_1 \in E_G \Leftrightarrow \mathit{send}(h_1)_{h_2} \in E_G$.

   In the case that $e_1 \in E_B$, we reason as in Case III(b). to show that invariants (i), (ii), and (iii) hold.

   The last possibility is that $e_1 \in E_G$. Here, we have $T'_R = T_R$ because t' has not been passed to a bad entry point. Thus we can calculate $\mathit{Active}_{S'}$ as follows:

$$\begin{array}{rcl}
\mathit{Active}_{S'}(u) & \Leftrightarrow & u \in \mathit{Tok}_{S'} \wedge \exists u' \in T'_R.(u = u') \vee (u' \prec^*_{S'} u) \\
& \Leftrightarrow & u \in \mathit{Tok}_S \cup \{t'\} \wedge \exists u' \in T_R.(u = u') \\
& & \qquad \vee [(u' = t' \wedge t' \prec^*_S u) \vee u' \prec^*_S u] \\
& \star \Leftrightarrow & u \in \mathit{Tok}_S \wedge \exists u' \in T_R.(u = u') \vee u' \prec^*_S u \\
& \Leftrightarrow & \mathit{Active}_S(u)
\end{array}$$

   In the reasoning above, the step marked $\star$ uses the fact that

$$t' \notin T_R \wedge \neg \exists u'.u' \prec^*_{S'} t'$$

   Invariant (i) follows directly from the fact that (i) holds in configuration S and the equivalence of $\mathit{Active}_{S'}$ and Active_S. To see that (ii) holds, note that $R(e_1) \Rightarrow \neg \exists t \in T_G.\mathit{Active}_S(t)$, but this implies $\neg \exists t \in T_G.\mathit{Active}'_S(t)$ as required. To establish invariant (iii), let $e \in E_G$ be given and suppose $R(e)$. We show that $e = \mathit{send}(h_1)_{h_2}$. Suppose by way of contradiction that $e \neq \mathit{send}(h_1)_{h_2}$. Then from the definition of R' we have $R'(e) = R[e_1 \mapsto \mathtt{f}](e)$. By assumption $R'(e) = \mathtt{t}$ and it follows that $e \neq e_1$. But now $R'(e) = R(e)$ and invariant (iii) shows that $e = e_1$, a contradiction.

8.5 Translation

Given a program and host configuration, the splitting translation is responsible for assigning a host to each field and statement. The Jif/split compiler takes as input the annotated source program and a description of the known hosts. It produces as output a set of Java files that yield the final split program when compiled against the run-time interface. There are several steps to this process.

In addition to the usual typechecking performed by an ordinary Java compiler, the Jif/split front end collects security label information from the annotations in the program, performing label inference when annotations are omitted. This process results in a set of label constraints that capture the information flows within the program. Next,

the compiler computes a set of possible hosts for each statement and field, subject to the security constraints described in Section 8.2. If no host can be found for a field or statement, the splitter conservatively rejects the program as being insecure.

There may also be many valid host assignments for each field or statement, in which case performance drives the host selection process. The splitter uses dynamic programming to synthesize a good solution by attempting to minimize the number of remote control transfers and field accesses, two operations that dominate run-time overhead. The algorithm works on a weighted control-flow graph of the program; the weight on an edge represents an approximation to the run-time cost of traversing that edge.

This approach also has the advantage that principals may indicate a preference for their data to stay on one of severally equally trusted machines (perhaps for performance reasons) by specifying a lower cost for the preferred machine. For example, to obtain the example partition shown in Figure 8.4, Alice also specifies a preference for her data to reside on host A, causing fields `m1`, `m2`, and `isAccessed` to be located on host A. Without the preference declaration, the optimizer determines that fewer network communications are needed if these fields are located at T instead. This alternative assignment is secure because Alice trusts the server equally to her own machine.

After host selection, the splitter inserts the proper calls to the runtime, subject to the constraints described in Section 8.3. An `lgoto` must be inserted exactly once on every control flow path out of the corresponding `sync`, and the `sync`–`lgoto` pairs must be well nested to guarantee the stack discipline of the resulting communication protocol. The splitter also uses standard dataflow analysis techniques to infer where to introduce the appropriate data forwarding.

Finally, the splitter produces Java files that contain the final program fragments. Each source Jif class `C` translates to a set of classes $\texttt{C\$Host}_i$, one for each known host $h_i \in H$. In addition to the translated code fragments, each such class contains the information used by the runtime system for remote references to other classes. The translation of a field includes accessor methods that, in addition to the usual get and set operations, also perform access control checks (which are statically known, as discussed in Section 8.2). In addition, each source method is represented by one frame class per host. These frame classes correspond to the `FrameID` arguments needed by the runtime system of Figure 8.3; they encapsulate the part of the source method's activation record visible to a host.

8.6 Implementation

We have implemented the splitter and the necessary run-time support for executing partitioned programs. Jif/split was written in Java as a 7400-line extension to the existing Jif compiler. The run-time support library is a 1700-line Java program. Communica-

tion between hosts is encrypted using SSL (the Java Secure Socket Extension (JSSE) library[5], version 1.0.2). To prevent forging, tokens for entry points are hashed using the MD5 implementation from the Cryptix library, version 3.2.0.[6]

To evaluate the impact of our design, we implemented several small, distributed programs using the splitter. Because we are using a new programming methodology that enforces relatively strong security policies, direct comparison with the performance of other distributed systems was difficult; our primary concern was security, not performance. Nevertheless, the results are encouraging.

8.6.1 Benchmarks

We have implemented a number of programs in this system. The following four are split across two or more hosts:

- **List** compares two identical 100 element linked lists that must be located on different hosts because of confidentiality. A third host traverses the lists.
- **OT** is the oblivious transfer program described earlier in this chapter. One hundred transfers are performed.
- **Tax** simulates a tax preparation service. A client's trading records are stored on a stockbroker's machine. The client's bank account is stored at a bank's machine. Taxes are computed by a tax preparer on a third host. The principals have distinct confidentiality concerns, and `declassify` is used twice.
- **Work** is a compute-intensive program that uses two hosts that communicate relatively little.

Writing these programs requires adding security policies (labels) to some type declarations from the equivalent single-machine Java program. These annotations are 11–25% of the source text, which is not surprising because the programs contain complex security interactions and little real computation.

8.6.2 Experimental Setup

Each subprogram of the split program was assigned to a different physical machine. Experiments were run on a set of three 1.4 GHz Pentium 4 PCs with 1GB RAM running Windows 2000. Each machine is connected to a 100 Mbit/second Ethernet by a 3Com 3C920 controller. Round-trip ping times between the machines average about

[5] `http://java.sun.com/products/jsse/`

[6] `http://www.cryptix.org/products/cryptix31/`

Table 8.1: Benchmark measurements

Metric	List	OT	Tax	Work	OT-h	Tax-h
Lines	110	50	285	45	175	400
Elapsed time (sec)	0.51	0.33	0.58	0.49	0.28	0.27
Total messages	1608	1002	1200	600	800	800
`forward` ($\times 2$)	400	101	300	0	-	-
`getField` ($\times 2$)	2	100	0	0	-	-
`lgoto`	402	200	0	300	-	-
`rgoto`	402	400	600	300	-	-
Eliminated ($\times 2$)	402	600	400	300	-	-

310 μs. This LAN setting offers a worst-case scenario for our analysis—the overheads introduced by our security measures are relatively more costly than in an Internet setting. Even for our local network, network communication dominates performance. All benchmark programs were run using SSL, which added more overhead: the median application-to-application round-trip time was at least 640 μs for a null Java RMI [7] call over SSL.

All benchmarks were compiled with version 1.3.0 of the Sun `javac` compiler, and run with version 1.3.0 of the Java HotSpot Client VM. Compilation and dynamic-linking overhead is not included in the times reported.

8.6.3 Results

For all four benchmarks, we measured both running times and total message counts so that performance may be estimated for other network configurations. The first row of Table 8.1 gives the length of each program in lines of code. The second row gives the median elapsed wall-clock time for each program over 100 trial runs. The following rows give total message counts and a breakdown of counts by type (`forward` and `getField` calls require two messages). The last row shows the number of `forward` messages eliminated by piggybacking optimizations described below.

For performance evaluation, we used Java RMI to write reference implementations of the Tax and OT programs and then compared them with our automatically generated programs. These results are shown in the columns OT-h and Tax-h of Table 8.1. Writing the reference implementation securely and efficiently required some insight that we obtained from examining the corresponding partitioned code. For example, in the OT example running on the usual three-host configuration, the code that executes on Alice's

[7] `http://java.sun.com/products/jdk/rmi/`

machine should be placed in a critical section to prevent Bob from using a race condition to steal both hidden values. The partitioned code automatically prevents the race condition.

The hand-coded implementation of OT ran in 0.28 seconds; the automatically partitioned program ran in 0.33 seconds, a slowdown of 1.17. The hand-coded version of Tax also ran in 0.27 seconds; the partitioned program ran in 0.58 seconds, a slowdown of 2.17. The greater number of messages sent by the partitioned programs explains most of this slowdown. Other sources of added overhead turn out to be small:

- Inefficient translation of local code
- Run-time checks for incoming requests
- MD5 hashing to prevent forging and replaying of tokens

The prototype Jif/split compiler attempts only simple optimizations for the code generated for local use by a single host. The resulting Java programs are likely to have convoluted control flow that arises as an artifact of our translation algorithm—the intermediate representation of the splitter resembles low-level assembly code more than Java. This mismatch introduces overheads that the hand-coded programs do not incur. The overhead could be avoided if Jif/split generated Java bytecode output directly; however, we leave this to future work.

Run-time costs also arise from checking incoming requests and securely hashing tokens. These costs are relatively small: The cost of checking incoming messages is less than 6% of execution time for all four example programs. The cost of token hashing accounted for approximately 15% of execution time across the four benchmarks. Both of these numbers scale with the number of messages in the system. For programs with more substantial local computations, we would expect these overheads to be less significant.

For a WAN environment, the useful point of comparison between the hand-coded and partitioned programs is the total number of messages sent between hosts. Interestingly, the partitioned Tax and OT programs need fewer messages for control transfers than the hand-coded versions. The hand-coded versions of OT and Tax each require 400 RMI invocations. Because RMI calls use two messages, one for invocation and one for return, these programs send 800 messages. While the total messages needed for the Jif/split versions of OT and Tax are 1002 and 1200, respectively, only 600 of these messages in each case are related to control transfers; the rest are data forwards. The improvement over RMI is possible because the `rgoto` and `lgoto` operations provide more expressive control flow than procedure calls. In particular, an RMI call must return to the calling host, even if the caller immediately makes another remote invocation to a third host. By contrast, an `rgoto` or `lgoto` may jump directly to the third host. Thus, in a WAN environment, the partitioned programs are likely to execute more quickly

than the hand-coded program because control transfers should account for most of the execution time.

8.6.4 Optimizations

Several simple optimizations improve system performance:

- Calls to the same host do not go through the network.
- Hashes are not computed for tokens used locally to a host.
- Multiple data forwards to the same recipient are combined into a single message and also piggybacked on `lgoto` and `rgoto` calls when possible. As seen in Table 8.1, this reduces `forward` messages by more than 50% (the last row is the number of round trips eliminated).

A number of further simple optimizations are likely to be effective. For example, much of the performance difference between the reference implementation of OT and the partitioned implementation arises from the server's ability to fetch the two fields `m1` and `m2` in a single request. This optimization (combining `getField` requests) could be performed automatically by the splitter as well.

Currently, `forward` operations that are not piggybacked with control transfers require an acknowledgment to ensure that all data is forwarded before control reaches a remote host. It is possible to eliminate the race condition that necessitates this synchronous data forwarding. Because the splitter knows statically what forwards are expected at every entry point, the generated code can block until all forwarded data has been received. Data transfers that are not piggybacked can then be done in parallel with control transfers. However, this optimization has not been implemented.

8.7 Trusted Computing Base

An important question for any purported security technique is the size and complexity of the *trusted computing base* (TCB). All else being equal, a distributed execution platform suffers from a larger TCB than a corresponding single-host execution platform because it incorporates more hardware and software. On the other hand, the architecture described here may increase the participants' confidence that trustworthy hosts are being used to protect their confidentiality.

What does a principal p who participates in a collaborative program using this system have to trust? The declaration signed by p indicates to what degree p trusts the various hosts. By including a declaration of trust for a host h in the declaration, p must

trust the hardware of h itself, the h's operating system, and the splitter run-time support, which (in the prototype implementation) implicitly includes Java's.

Currently, the Jif/split compiler is also trusted. Ongoing research based on certified compilation [MWCG98] or proof-carrying code [Nec97] might be used to remove the compiler from the TCB and instead allow the bytecode itself to be verified [JVM95].

Another obvious question about the trusted computing base is to what degree the partitioning process itself must be trusted. It is clearly important that the subprograms a program is split into are generated under the same assumptions regarding the trust relationships among principals and hosts. Otherwise, the security of principal p might be violated by sending code from different partitionings to hosts trusted by p. A simple way to avoid this problem is to compute a one-way hash of all the splitter's inputs—trust declarations and program text—and to embed this hash value into all messages exchanged by subprograms. During execution, incoming messages are checked to ensure that they come from the same version of the program.

A related issue is where to partition the program. It is necessary that the host that generates the program partition that executes on host h be trusted to protect all data that h protects during execution. That is, the partitioning host could be permitted to serve in place of h during execution. A natural choice is thus h itself: each participating host can independently partition the program, generating its own subprogram to execute. That the hosts have partitioned the same program under the same assumptions can be validated using the hashing scheme described in the previous paragraph. Thus, the partitioning process itself can be decentralized yet secure.

8.8 Related Work

Besides the work on information flow already discussed in this thesis, the primary area of research related to secure program partitioning is support for transparently distributed computation.

A number of systems (such as Amoeba and Sprite [DOKT91]) automatically redistribute computation across a distributed system to improve performance, though not security. Various transparently distributed programming languages have been developed as well; a good early example is Emerald [JLHB88]. Modern distributed interface languages such as CORBA [COR91] or Java RMI do not enforce end-to-end security policies.

Jif and secure program partitioning are complementary to current initiatives for privacy protection on the Internet. For example, the recent Platform for Privacy Preferences (P3P) [p3p] provides a uniform system for specifying users' confidentiality policies. Security-typed languages such as Jif could be used for the implementation of a P3P-compliant web site, providing the enforcement mechanisms for the P3P policy.

Chapter 9

Conclusions

This chapter summarizes the contributions of this thesis and ends with some potential future directions.

9.1 Summary

This thesis has focused on the theory of various security-typed languages, in which programmers can specify security policies about the data being used in the program. Static program analysis can detect inconsistencies in the policy, revealing where insecurity might arise.

One value of security-typed programming languages is that by formalizing the information-flow problem at a particular level of abstraction it is possible to rule out a certain class of information leaks. Moreover, the abstractions used in the language definition suggest where additional security measures may be needed: for instance, the operating system access control mechanisms could perhaps be used to enforce the assumed partitioning of memory into high- and low-security portions.

Security-typed languages also can potentially enforce a richer set of security policies than traditionally provided by discretionary access control mechanisms. Because the policy descriptions are incorporated into programs, security policy creation becomes a matter of programming. Security policies that release secret data only in certain circumstances are easy to express, whereas with traditional access control mechanisms, it is difficult to conditionally release information.

This thesis extends the existing work on security-typed languages in a number of ways. It establishes a *noninterference* result for a higher-order language with structured memory. It also gives an expressive type system for noninterference in a concurrent programming language. These results highlight the importance of determinism and the closely related notion of linearity in information-flow security. Finally, it considers the

additional constraints necessary to remove the assumption that all secure computation takes place on a single, trusted computer.

This thesis also presents Jif/split, a prototype compiler for protection of confidential data in a distributed computing environment with heterogeneously trusted hosts. Security policy annotations specified in the source program allow the splitter to partition the code across the network by extracting a suitable communication protocol. The resulting distributed system satisfies the confidentiality policies of principals involved without violating their trust in available hosts. Jif/split also enforces integrity policies, which is needed because of the interaction between integrity and confidentiality in the presence of declassification. The Jif/split prototype demonstrates the feasibility of this architecture. The experience with simple example programs has shown the benefits of expressing security policies explicitly in the programming language, particularly with respect to catching subtle bugs.

The prototype implementation of Jif/split benefited from the theoretical work on security-typed languages in several ways. First, that the security types could guide an automatic partitioning process became clear only after seeing how much additional structure they add to a program. Second, the proofs of noninterference for $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ led directly to the discovery of errors and inconsistencies in the implementation of similar rules in the Jif compiler. Third, and most important, the insights about the role of ordered linear continuations used in $\lambda_{\mathrm{SEC}}^{\mathrm{CPS}}$ had direct impact on the design of the control-transfer mechanisms used in the Jif/split run time.

Building the prototype splitter also led to many observations about information security in a distributed setting. In particular, the Jif/split implementation revealed that Jif's original rule for declassification was insufficient in a distributed setting. The Jif/split prototype also validates the premise that security-typed programs can help programmers find subtle security flaws in their software

9.2 Future Work

The work in this thesis and the Jif/split prototype have yielded some insight into the difficulties of building distributed systems with strong end-to-end information-flow guarantees, but there is still much room for improvement.

This thesis has focused on one aspect of security: protecting information security. Other aspects, such as reliability and auditing of transactions, also play a role in the security of distributed computations, and they should not be neglected.

Of course security and performance are often at odds, and the same is true here. Jif/split assumes that the security of the data is more important than the performance of the system. However, encoding the security policy in the programming language makes this trade-off more explicit: if the performance of a program under a certain security

policy is unsatisfactory, it is possible to relax the policy (for instance, by declaring more trust in certain hosts, or by reducing the restrictions imposed by the label annotations). Under a relaxed policy, the compiler may be able to find a solution with acceptable performance—the relaxed security policy spells out what security has been lost for performance. The prototype allows some control over performance by allowing the user to specify relative costs of communication between hosts. The host assignment tries to find a minimum cost solution, but other constraints could be added—for example, the ability to specify a particular host for a given field.

One serious drawback of the security-typed language approach is that the security policy is decided upon either by the language designer, or the implementer of the program in question. This means that the *consumer* of the software is not able to change or specify the policy. This stands in contrast with, for example, the security automaton approach of Erlingsson and Schneider [ES99] in which security policies are applied by the user. However, there is no reason why both techniques cannot be used simultaneously—perhaps to great benefit on each side.

Realistic policies do not fall into the simple noninterference-like models of information-flow security. Programs that involve declassification are potentially dangerous, and understanding exactly what policy is enforced by a program that uses declassification is not always easy. Robust declassification and the authority model improve over previous language-based approaches, but there is still not an appropriate theoretical model for properly reasoning about declassification.

Finally, experience with larger and more realistic programs will be necessary to determine the real benefits and drawbacks to security-typed languages.

Collaborative computations carried out among users, businesses, and networked information systems continue to increase in complexity, yet there are currently no satisfactory methods for determining whether the end-to-end behavior of these computations respect the security needs of the participants. The work described in this thesis is a novel approach that is a useful step towards solving this essential security problem.

BIBLIOGRAPHY

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, January 1999.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[AG99] Martín Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[Aga00] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[AP90] M. Abadi and G. D. Plotkin. A PER model of polymorphism. In *5th Annual Symposium on Logic in Computer Science*, pages 355–365. IEEE Computer Society Press, 1990.

[App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[Bar84] H.P. Barendregt. The lambda calculus: Its syntax and semantics. In J. Barwise, D. Kaplan, H. J. Keisler, P. Suppes, and A.S. Troelstra, editors, *Studies in Logic and the Foundation of Mathematics*, volume 103. North-Holland, 1984.

[BBL84] J. Banâtre, C. Bryce, and D. Le Metáyer. Compile-time detection of information flow in sequential programs. In *Proceedings of the Europena Symposium on Research in Computer Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–73. Springer Verlag, 1984.

[BC02] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[BCY95] William R. Bevier, Richard M. Cohen, and William D. Young. Connection policies and controlled interference. In *Proc. of the 8th IEEE Computer Security Foundations Workshop*, pages 167–176, 1995.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.

[Bie99] Gavin Bierman. A classical linear lambda calculus. *Theoretical Computer Science*, 227(1–2):43–78, 1999.

[BL75] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.

[BN02] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *csfw15*, 2002.

[BORT01] Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. Linearly used continuations. In *Proceedings of the Continuations Workshop*, 2001.

[BP76] D.E. Bell and L.J. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, March 1976.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 11–19, 1988.

[CG00] Luca Cardelli and Andrew Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[CGW89] Thierry Coquand, Carl A. Gunter, and Glynn Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81(2):123–167, May 1989.

[COR91] OMG. *The Common Object Request Broker: Architecture and Specification*, December 1991. OMG TC Document Number 91.12.1, Revision 1.1.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, , and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 262–275, San Antonio, Texas, January 1999.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[DD00] Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 209–220, 2000.

[Den75] Dorothy E. Denning. *Secure Information Flow in Computer Systems*. Ph.D. dissertation, Purdue University, W. Lafayette, Indiana, USA, May 1975.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyand k-limiting. In *Proc. of the '94 SIGPLAN Conference on Programming Language Design*, pages 230–241, 1994.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[DKS99] Ivan Damgård, Joe Kilian, and Louis Salvail. On the (im)possibility of basing oblivious transfer and bit commitment on weakened security assumptions. In Jacques Stern, editor, *Advances in Cryptology – Proceedings of EUROCRYPT 99*, LNCS 1592, pages 56–73. Springer, 1999.

[DOD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, December 1985.

[DOKT91] Fred Douglis, John K. Ousterhout, M. Frans Kaashoek, and Andrew S. Tanenbaum. A comparison of two distributed systems: Amoeba and Sprite. *ACM Transactions on Computer Systems*, 4(4), Fall 1991.

[EGH94] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive points-to analysis in the presence of function pointers. In *Proc. of the '94 SIGPLAN Conference on Programming Language Design*, pages 242–256, June 1994.

[EGL83] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. In R.L. Rivest, A. Sherman, and D. Chaum, editors, *Advances in Cryptology: Proc. of CRYPTO 82*, pages 205–210. Plenum Press, 1983.

[ES99] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, September 1999.

[ET99] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, May 1999.

[FA99a] Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99—Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303, Eindhoven, The Netherlands, August 1999. Springer-Verlag.

[FA99b] Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proc. of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[FB93] J. Mylaert Filho and G. Burn. Continuation passing transformations and abstract interpretation. In *Proc. First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, 1993.

[Fei80] Richard J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International Computer Science Lab, Menlo Park, California, January 1980.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design*, pages 219–232, Vancouver, Canada, June 2000.

[FG96] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[FG97] Riccardo Focardi and Roberto Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9), September 1997.

[FG02] Cedric Fournet and Andrew Gordon. Stack inspection: Theory and variants. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–318, 2002.

[Fil92] Andrzej Filinski. Linear continuations. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, pages 27–38, 1992.

[Fis72] Michael J. Fischer. Lambda calculus schemata. *SIGPLAN Notices*, 7(1):104–109, January 1972.

[FL94] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 154–163, 1994.

[FLR77] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *Proc. 6th ACM Symp. on Operating System Principles (SOSP), ACM Operating Systems Review*, 11(5):57–66, November 1977.

[Fou98] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. Ph.D. dissertation, École Polytechnique, nov 1998.

[FSBJ97] Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, USA, May 1997.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the '93 SIGPLAN Conference on Programming Language Design*, pages 237–247, June 1993.

[Gat02] Bill Gates. Trustworthy computing. Microsoft e-mail, January 2002.

[GD72] G. S. Graham and Peter J. Denning. Protection: Principles and practice. In *Proc. of the AFIPS Spring Joint Conference*, pages 417–429, 1972.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. ISBN 0-201-63451-1.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, April 1984.

[Gra90] James W. Gray, III. Probabilistic interference. In *Proc. IEEE Symposium on Security and Privacy*, pages 170–179. IEEE Computer Society Press, May 1990.

[Gra91] James W. Gray, III. Towards a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy*, pages 21–34. IEEE Computer Society Press, 1991.

[GS92] J. W. Gray III and P. F. Syverson. A logical approach to multilevel security of probabilistic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–176. IEEE Computer Society Press, 1992.

[Hen00] Matthew Hennessy. The security picalculus and non-interference. Technical Report Report 05/2000, University of Sussex, School of Cognitive and Computing Sciences, November 2000.

[HL93] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proc. 20th ACM Symp. on Principles of Programming Languages (POPL)*, pages 206–219, January 1993.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.

[HR00] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. Technical Report report 03/2000, University of Sussex, 2000.

[HRU76] M. A. Harrison, W. L Ruzzo, and J. D. Ullman. Protection in operating systems. *Comm. of the ACM*, 19(8):461–471, August 1976.

[HVY00] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proc. of the 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000.

[HY02] Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 81–92, January 2002.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JVM95] Sun Microsystems. *The Java Virtual Machine Specification*, release 1.0 beta edition, August 1995. Available at ftp://ftp.javasoft.com/docs/-vmspec.ps.zip.

[Lam71] Butler W. Lampson. Protection. In *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in Operating Systems Review, 8(1), January 1974, pp. 18–24.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *2001 USENIX Security Symposium*, Washington, D. C., August 2001.

[LR92] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proc. of the SIGPLAN '92 Conference on Programming Language Design*, June 1992.

[LV95] Nancy Lynch and Frits Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995. Also, Technical Memo MIT/LCS/TM-486.b (with minor revisions), Laboratory for Computer Science, Massachusetts Institute of Technology.

[LWG+95] J. R. Lyle, D. R. Wallace, J. R. Graham, K. B. Gallagher, J. P. Poole, and D. W. Binkley. Unravel: A CASE tool to assist evaluation of high integrity software. IR 5691, NIST, 1995.

[Man00] Heiko Mantel. Possibilistic definitions of security: An assembly kit. In *Proc. of the 13th IEEE Computer Security Foundations Workshop*, pages 185–199, Cambridge, United Kingdom, 2000.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proc. IEEE Symposium on Security and Privacy*, pages 161–166. IEEE Computer Society Press, May 1987.

[McC88] Daryl McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society Press, May 1988.

[MCG+99] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In 2^{nd} *ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[McL88a] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, January 1988.

[McL88b] John McLean. Reasoning about security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 123–131, Oakland, CA, 1988. IEEE Computer Society Press.

[McL90] John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93. IEEE Computer Society Press, May 1994.

[MH02] Massimo Merro and Matthew Hennessy. Bisimulation congruences for safe ambients. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 71–80, January 2002.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing Series. The MIT Press, 1996.

[ML98] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, USA, May 1998.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[MNZZ01] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[Mor68] James H. Morris. *Lambda Calculus Models of Programming Languages*. Ph.D. dissertation, Massachusetts Institute of Technology, 1968.

[MPS86] David MacQueen, Gordon D. Plotkin, and Ravi Sethi. An ideal model for recursive polymorphism. *Information and Control*, 71(1/2):95–130, October/November 1986.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[MR92a] QingMing Ma and John Reynolds. Types, abstraction, and parametric polymorphism: Part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. A. Schmidt, editors, *Proceedings of the 1991 Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 1–40. Springer-Verlag, 1992.

[MR92b] M. D. McIlroy and J. A. Reeds. Multilevel security in the UNIX tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[MS01] Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, June 2001.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[MWCG98] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[Mye99] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachusetts Institute of Technology, Cambridge, MA, January 1999. Ph.D. thesis.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.

[Nie82] Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[p3p] Platform for privacy preferences (P3P). `http://www.w3.org/p3p`.

[PC00] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, September 2000.

[Pin95] Sylvan Pinsky. Absorbing covers and intransitive non-interference. In *Proc. IEEE Symposium on Security and Privacy*, 1995.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PO95] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, September 1995.

[Pot02] François Pottier. A simple view of type-secure information flow in the π-calculus. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[PP00] Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In J. Despeyroux, editor, *2nd Workshop on Logical Frameworks and Meta-languages*, Santa Barbara, California, June 2000.

[PS99] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. Technical Report MS-CIS-99-10, University of Pennsylvania, April 1999. (Summary in POPL '97).

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, January 2002.

[Rab81] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[Rei78] Richard Philip Reitman. *Information Flow in Parallel Programs: An Axiomatic Approach*. Ph.D. dissertation, Cornell University, 1978.

[Rey72] John C. Reynolds. Definitional interpreters for higherorder programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, August 1972.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Paris, France, April 1974.

[Rey78] John C. Reynolds. Syntactic control of interference. In *Proc. 5th ACM Symp. on Principles of Programming Languages (POPL)*, pages 39–46, 1978.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A Mason, editor, *Information Processing*, pages 513–523. Elsevier Science Publishers B.V., 1983.

[RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proc. of the 12th IEEE Computer Security Foundations Workshop*, 1999.

[RH99] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 93–104, San Antonio, TX, January 1999.

[Rie89] Jon G. Riecke. Should a function continue? Masters dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, 1989.

[RM96] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *Proc. 3rd International Symposium on Static Analysis*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, September 1996.

[Ros95] A. W. Roscoe. Csp and determinism in security modeling. In *Proc. IEEE Symposium on Security and Privacy*, 1995.

[RR99] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proc. of the ACM SIGPLAN 1999 Conference on Programming Language Design*, pages 77–90, May 1999.

[Rus92] John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

[Sab01] Andrei Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of the Andrei Ershov 4th International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 225–239. Springer-Verlag, July 2001.

[Sch96] B. Schneier. *Applied Cryptography*. John Wiley and Sons, New York, NY, 1996.

[Sch97] Fred B. Schneider. *On Concurrent Programming*. Springer Verlag, 1997.

[Sch99] Fred B. Schneider, editor. *Trust in Cyberspace*. National Academy Press, 1999.

[Sch01] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

[SF94] Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 1–12, 1994.

[SM02] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[SMH00] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl's 10th Anniversary.*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101, Saarbrücken, Germany, August 2000. Springer-Verlag.

[Smi01] Geoffrey Smith. A new type system for secure information flow. In *CSFW14*, pages 115–125. IEEE Computer Society Press, jun 2001.

[SNS88] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. Technical report, Project Athena, MIT, Cambridge, MA, March 1988.

[SS99] Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. In *Proc. of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 40–58. Springer-Verlag, March 1999.

[SS00] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[SS01] Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[Ste78] Guy L. Steele. Rabbit: a compiler for scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, May 1978.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[Str67] C. Strachey. Fundamental concepts in programming languages. Unpublished Lecture Notes, Summer School in Computer Programming, August 1967.

[Sut86] David Sutherland. A model of information. In *Proc. 9th National Security Conference*, pages 175–183, Gaithersburg, Md., 1986.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, January 1998.

[SV00] Sewell and Vitek. Secure composition of untrusted code: Wrappers and causality types. In *PCSFW: Proceedings of The 13th Computer Security Foundations Workshop*. IEEE Computer Society Press, 2000.

[SWM00] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proc. of the 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, 2000.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[TW99] David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 227(1-2):231–248, September 1999.

[VS97] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *10th IEEE Computer Security Foundations Workshop*, pages 156–168. IEEE Computer Society Press, June 1997.

[VS00] Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, January 2000.

[VSI96] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Progarmming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[Wad93] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag, 1993.

[WAF00] Dan S. Wallach, Andrew W. Appel, , and Edward W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[Wag00] David Wagner. *Static analysis and computer security: New techniques for software assurance*. Ph.D. dissertation, University of California at Berkeley, 2000.

[Wal00] David Walker. A type system for expressive security policies. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 254–267. ACM Press, Jan 2000.

[WF92] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Rice University, June 1992.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.

[WF98] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1998.

[WJ90] J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *Proc. IEEE Symposium on Security and Privacy*, pages 144–161, May 1990.

[WM00] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, September 2000.

[Ylo96] Tatu Ylonen. SSH – secure login connections over the Internet. In *The Sixth USENIX Security Symposium Proceedings*, pages 37–42, San Jose, California, 1996.

[Zha97] Kan Zhang. A theory for system security. In *10th IEEE Computer Security Foundations Workshop*, pages 148–155. IEEE Computer Society Press, June 1997.

[Zho01] Lidong Zhou. *Towards Fault-Tolerant and Secure On-line Services*. Ph.D. dissertation, Cornell University, May 2001.

[ZL97] Aris Zakinthinos and E. Stewart Lee. A general theory of security properties and secure composition. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1997.

[ZM00] Steve Zdancewic and Andrew C. Myers. Confidentiality and integrity with untrusted hosts. Technical Report 2000–1810, Computer Science Dept., Cornell University, 2000.

[ZM01a] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.

[ZM01b] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *Proc. of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, April 2001.

[ZM02] Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2/3), 2002.

[ZSv00] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. Technical Report 2000-1828, Department of Computer Science, Cornell University, December 2000.

[ZZNM01] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, volume 35(5) of *Operating Systems Review*, pages 1–14, Banff, Canada, October 2001.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *Transactions on Computer Systems*, 20(3):283, 2002.