EXPRESSIVE AND ENFORCEABLE INFORMATION SECURITY POLICIES

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

Stephen Nathaniel Chong

August 2008

EXPRESSIVE AND ENFORCEABLE INFORMATION SECURITY POLICIES

Stephen Nathaniel Chong, Ph.D.

Cornell University 2008

Declassification and erasure are both common, and often crucial, security requirements. Declassification occurs when the confidentiality of information is weakened; erasure occurs when the confidentiality of information is strengthened, perhaps to the point of completely removing the information from the system.

This dissertation presents and explores a framework for declassification and erasure information security policies, and it shows how these policies can help in building trustworthy systems.

First, this dissertation demonstrates that the declassification and erasure policies can be provably enforced. It presents a type system that, in conjunction with run-time mechanisms, can enforce the declassification and erasure policies on information from the start of the program until its termination, regardless of how the information propagates through the system, or where it enters and leaves. The dissertation defines a novel, precise, end-to-end semantic security condition, *noninterference according to policy*, and proves that all well-typed programs satisfy it. Thus, enforcement of declassification and erasure policies provides well-defined security guarantees.

Second, this dissertation investigates declassification and erasure in the presence of mutual distrust: the principals with a security concern in the system do not necessarily trust each other. Mutual distrust is pervasive. The dissertation defines *decentralized robustness*, a semantic security condition that ensures that each principal is convinced declassification and erasure occur only at appropri-

ate times, regardless of the actions of principals he distrusts. A type system to enforce decentralized robustness is presented.

Finally, this dissertation demonstrates the practicality of declassification and erasure policies. The enforcement mechanisms for the policies and decentralized robustness are incorporated into the Jif programming language (an extension of the Java programming language with information-flow control). The resulting language is used to implement a secure remote voting system. The use of erasure and declassification policies provides additional assurance that the voting system implementation satisfies some of its security requirements.

BIOGRAPHICAL SKETCH

Stephen Nathaniel Chong was born at a young age in Palmerston North, New Zealand. The first few years of his life involved little direct interaction with computers. That changed when his brother won an Aquarius computer as a door prize at a fair in Sydney, Australia. Stephen and his brother would painstakingly type in BASIC programs from the manual (including comments), and, due to the lack of permanent storage, beg their parents to leave the computer plugged in.

Stephen's next encounter with computers was at Monrad Intermediate School. There he was given the opportunity to administer the school computers: four Poly computers, networked together. The Poly was a home and educational computer, proudly developed in New Zealand in the early 1980s.

Computers got personal (and more mainstream) for Stephen when his mother acquired an Apple IIc in 1989, ostensibly for work. It was on this machine that Stephen learned how to program, in Applesoft BASIC. However, the pride and joy of Stephen's early adolescence was the first computer he owned: an Amiga 500, purchased in 1989 with carefully saved Bar Mitzvah money.

Since that time, Stephen has owned several personal computers that have both helped and hindered him in his pursuit of higher education. He completed a Bachelor of Arts from Victoria University of Wellington in 1996, and a Bachelor of Science (Honours) from the same institution the following year. Several years working as consultant drove Stephen back to university. Since August 2001, he has been studying for a Ph.D. at Cornell University.

To my family,

especially the newest part of it.

ACKNOWLEDGMENTS

There are many people to thank and acknowledge for this existence of this dissertation. First and foremost, my sincere thanks to my advisor, Andrew Myers, who has always provided me with superb advice and guidance. I am deeply indebted to him for his support and encouragement. I am continually impressed by his insights, enthusiasm, and ability for research. Indeed, it was his enthusiastic teaching of Advanced Programming Languages that led me to this area of research. He has been, and will continue to be, a role model for me.

I am also grateful to the other members of my committee: Fred Schneider, Dexter Kozen, and David Easley. They were an insightful and considered readership, and I have learned much from their feedback.

During my time at Cornell, I have found the Computer Science Department to be always a friendly, supportive, and creative environment to work (and play) in. This is due to the excellent students, faculty, and staff.

The various members of the APL group, and the Programming Language Discussion Group have had a huge influence on my development as a computer scientist; several have been mentors to me during my time at grad school. I am grateful to Riccardo Pucella, who has been a great mentor and friend to me, and was always ready to discuss stupid questions, on any subject. I am also grateful to Steve Zdancewic, Dan Grossman, and Andrei Sabelfeld, who have given me much good advice. I look forward to continuing to learn from all of these mentors.

I have been very fortunate to have had the chance to collaborate with many others, on papers closely (and not-so-closely) related to my dissertation; each paper has been an engaging and learning experience. My thanks to Hubie Chen,

Michael Clarkson, Jed Liu, Nate Nystrom, Kevin O'Neill, Riccardo Pucella, Xin Qi, Radu Rugina, K. Vikram, Steve Zdancewic, Lantian Zheng, and Xin Zheng.

I'm also grateful to my office mates and others in the department, who, over the years, have offered a wealth of advice and welcome distractions: Kamal Aboul-Hosn, Eric Breck, Hubie Chen, James Cheney, Siggi Cherem, Michael Clarkson, Amy Gale, Dan Grossman, Jeff Hartline, Bill Hogan, Prakash Linga, Stephanie Meik, Alex Niculescu-Mizil, Nate Nystrom, Kevin O'Neill, Kelly Patwell, Sabina Petride, Riccardo Pucella, Filip Radlinski, Ganesh Ramanarayanan, Matt Schultz, Alexa Sharp, Becky Stewart, Yanling Wang, Vicky Weissman, and Tom Wexler. And the nth years of Upson 5154 (and neighbors) provided a great support group for the final surge...

People make the place, and I am grateful to the many friends that have made my years in Ithaca memorable, enjoyable, and illuminative. Without these people, getting a Ph.D. would have harder and much less fun.

Finally, I would like to thank my family. To Mum and Dad: Thank you for always being loving and supportive, and encouraging independence in your offspring. To Lewis and Lissa: I am proud to have you as my siblings and my friends. I wish we got to spend more time together in the same place. And to Kiran: Thank you, for the last seven years and the many years to come.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Many computer systems have detailed and complicated information security requirements, perhaps derived from legislation, user requirements, or organizational policy. Broadly, information security requirements address the confidentiality, integrity, and availability of data (Bishop, 2002).

It is important that information security requirements be satisfied in system implementations. Violations of information security can have severe financial, legal, or ethical implications (e.g., Bello 2008; Krebs 2008; British Broadcasting Corporation 2007; Project on Government Oversite 2007; Wagner and Bishop 2007; Federal Trade Commission 2005a; Marlin 2005; Federal Trade Commission 2005b, 2002). Thus, to be trustworthy, a system should enforce information security correctly.

Current methodologies for building systems provide little assurance that information security requirements are satisfied because enforcement is not clearly connected to requirements. Instead, the security of information is implicit in the system implementation's use and manipulation of the information. To reason that an implementation correctly enforces a security requirement may require reasoning about many pieces of the implementation, often in different modules. Current implementation techniques do not facilitate this reasoning.

The goal of this work is to demonstrate how information security policies can help to build trustworthy systems. Expressive, formal, information security policies help bridge the gap between the information security requirements of a system and the system implementation. Information security policies allow the unambiguous specification of a system's security requirements, and can be provably enforced. Expressive and enforceable information security policies

can provide assurance that an implementation of a system satisfies the system's security requirements by providing stronger connections between a system's requirements and implementation.

This dissertation shows in two major steps how information security policies can help to build trustworthy systems. First, it presents expressive policies that can capture common information security requirements, and can be provably enforced, with well-understood security guarantees. Second, it incorporates these policies into a practical programming language, and uses the programming language to implement a large system to validate the expressiveness and utility of the policies.

1.1 End-to-end information security

Information security is an end-to-end requirement: the end-to-end behavior of a system must respect the security requirements, regardless of the implementation of the system.

Consider, for example, a medical information website that offers (among other functionality) a diagnostic application, where a user enters symptoms, and the application presents information about possible diseases consistent with the symptoms. Suppose there is a confidentiality requirement that the system never reveal a user's symptoms to anyone other than the user. It is important to ensure that the system never outputs the user's symptoms (other than to the user), as this would violate the confidentiality requirement.

It is important to protect not only sensitive information, but also data derived from sensitive information. The medical information website should be prevented not only from revealing a user's symptoms, but also from revealing the diagnoses, as the diagnoses may be used to infer the symptoms.

Conservatively, it may be necessary to treat all data derived from, or dependent on, sensitive information, as also being sensitive, since derived data may suffice to deduce some or all of the original information. End-to-end enforcement of security requires protecting *information*, not just bits. Information can be regarded as bits in context. For example, a sequence of bits may reveal very little information by itself; however, knowing that the bit sequence is a UTF-8 encoding of symptoms entered by the user provides context, and reveals the sensitive information.

End-to-end security cannot be enforced by the commonly used technique of (discretionary) access control. Access control is able to enforce appropriate security at the time of the access control check, but not before or after. Consider using access control to enforce confidentiality. Access control can restrict the release of information (i.e., can restrict who may read what information when) but cannot restrict the subsequent propagation of the information. For example, access control for a file system can prevent a program from reading a file, but cannot prevent the program from distributing the file's contents.

Cryptography can provide end-to-end information security in some settings. For example, end-to-end security can be enforced in a communication network by applying appropriate cryptographic operations at the endpoints (Saltzer et al., 1984); this can protect both the confidentiality and integrity of information sent over the network. However, for a computer system that must examine, manipulate, combine and compute information, it is infeasible to simply encrypt information as it enters the system, and decrypt as it leaves. Homomorphic encryption schemes permit only limited computation to be performed on encrypted data. Similarly, current techniques for secure multiparty computation (Yao, 1982) cannot perform arbitrary secure computation.

Information-flow control is an approach for achieving end-to-end enforcement. Information-flow techniques enforce security by restricting the flow, or propagation, of information in a system.

Conceptually, information-flow control techniques label data with security-relevant annotations (e.g., security classifications, security categories, or confidentiality levels). As data are updated and created, the security labels are also updated to reflect data dependencies. The security labels can be used, for example, to prevent confidential data from being output on public channels, or to prevent untrusted data from being used in trusted contexts. Different classes of labels can be used to reason about information flow; Denning (1976) advocates a lattice structure for the labels.

Dynamic information-flow control techniques restrict the flow of information at runtime. Static information-flow control techniques (Sabelfeld and Myers, 2003), which restrict information flow prior to execution, do not incur the performance overheads of representing security labels at runtime.

Noninterference (Goguen and Meseguer, 1982) is an end-to-end semantic security condition that requires that high security inputs do not affect low security outputs. Information-flow control can enforce noninterference. However, noninterference is too strong a requirement for many systems.

1.2 Information security changes

In many systems, the security to enforce on information changes over time. Often the change in security is key to the correct functionality of the system, and the reasons for change are as varied as the systems themselves. The following examples illustrate the variety in requirements for changing information security.

Example 1.1 (Decrease in confidentiality) *In a round of poker, players are dealt cards, and do not reveal their cards until the end of the round. Initially, and throughout the round, information about who got dealt which cards is highly confidential; at the end of the round, (some of) the cards are revealed, and the information about who has which cards is no longer confidential.*

Example 1.2 (Increase in confidentiality) *A company may desire to share customers' information (such as demographics, and email addresses) with business partners. Often a customer has the opportunity to opt out of such information sharing schemes. When a customer opts out, the company must increase the confidentiality of the customer's information, to prevent business partners from using that information.*

Example 1.3 (Increase in integrity) *To register on a website, a user supplies an email address, and perhaps other information. The email address is initially treated as low-integrity information by the website provider, as the provider does not know whether the email address is valid. The provider sends a message to the user's email address requesting the user perform an action, for example, present a nonce to the website. After the action has been performed, the email address is validated, and can now be treated as higher integrity information.*

Example 1.4 (Increase in availability) *Consider a website with users primarily from one geographic region. The website has high load between the hours of 7 a.m. to 11 p.m. in that region, and much lower load outside of those hours. The information provided by the website needs to be highly available during high-load hours. This can perhaps be obtained by provisioning more web servers during those hours.*

1.3 Declassification and erasure

Examples 1.1 and 1.2 above concern the decrease and increase, respectively, of the confidentiality of information. *Declassification* occurs when the confidentiality of information is decreased—that is, when the confidentiality of information is changed to be less restrictive. We define *erasure* (Chong and Myers, 2005) to be the opposite case, when the confidentiality of information is changed to be more restrictive.

The semantic security condition of noninterference is too strong in the presence of declassification, which intentionally makes secret information public. Noninterference cannot express erasure requirements, which make publicly observable information less observable.

Declassification and erasure are both common and crucial information security requirements of many applications. The examples below demonstrate the pervasiveness and diversity of declassification and erasure requirements.

Example 1.5 (Mobile computing) *A mobile device, such as a laptop computer, may operate in several environments of varying sensitivity and vulnerability. When a mobile device leaves a secure environment (where sensitive information is accessible) for a less secure environment, it may be necessary to ensure that the mobile device does not contain any sensitive information. There is an erasure requirement: the mobile device needs to erase sensitive information before entering a less secure environment.*

Example 1.6 (Public computers) *Some computers are intended for use by members of the public, for example, computers in public libraries or kiosks. Users of these computers may enter and access sensitive information via these computers, such as email, user names and passwords, or web sites they would prefer to remain confidential.*

Each user's session on a public computer should be independent of previous users' sessions, to ensure both that a user is not able to learn sensitive information from a

previous user's session, and that a user is not able influence a future user's session, for example, by directing a future user to a phishing web site.

Therefore, public computers have an erasure requirement: when a session has finished, use of information from the session needs to be restrictive enough that it cannot influence new sessions. Note that session information may be required to be removed completely from the system, or to remain in the system but be inaccessible by new sessions (for example, in audit logs).

Example 1.7 (Online transaction) *Consider a consumer purchasing a product from a merchant over a network. In order to complete the transaction, the consumer has to provide a credit card number to the merchant. The merchant promises not to keep any record of the credit card number after the transaction. However, once the consumer has approved the purchase, the merchant must send the credit card number to the bank, which* will *keep a record of the credit card number.*

The merchant's system has both a declassification requirement and an erasure requirement: the credit card number needs to be released by the merchant's system and sent to the bank, and needs to be removed from the system at the end of the transaction.

Example 1.8 (Medical information website) *Consider the diagnostic application on a medical information website, introduced above. A user enters information about symptoms, and the application presents diseases consistent with the symptoms. Suppose the website's privacy policy states that symptoms the user enters are confidential, and no record of them will be kept after the user has finished using the diagnostic application.*

The website has an erasure requirement: when the user has finished using the diagnostic application, the symptom data that the user has entered must be erased. Note that the information the user has entered may need to persist over several user requests, but also might need to be erased before the session has finished. Thus, the lifetime of the information does not necessarily match that of any web server resource. Another

subtlety is that diagnoses the system has produced must also be erased, as they may reveal information about the symptoms entered.

Example 1.9 (Sealed-bid auction) *In a* sealed-bid auction *(also known as a* closed auction*) participants bid for the auctioned item by submitting secret bids. (In a physical setting, the secrecy of bids might be enforced by placing each bid in an envelope and sealing the envelope.) Once all bids have been submitted, the bids are revealed, and the winner (the participant with the highest bid) is determined.*

Sealed-bid auctions have declassification requirements on the bids: each bid must be secret until all bids have been submitted, whereupon it should be declassified, either to the auctioneer or to all participants, depending on the auction scheme.

Example 1.10 (Poker) *As described in Example 1.1, players in a round of poker are not allowed to reveal their cards until the end of the round. At the end of the round, some players may be required to reveal their cards, but all players are allowed to reveal their cards. Thus, for at least some players, the revelation of their cards is optional.*

A round of poker has declassification requirements on the cards: a player's cards must be secret until the end of the round, whereupon a player may declassify their cards to all other players.

1.4 Mutual distrust

Many systems interact with multiple *principals*, entities with security concerns. Often these principals are mutually distrusting, yet they must cooperate to achieve a common goal or perform some combined computation. For example, in sealed-bid auctions, described above in Example 1.9, mutually distrusting participants cooperate to conduct the auction. In some settings, the auctioneer

may also be distrusted (Jones and Menezes, 1995) and distrusting, and yet the auction must still occur.

In settings with mutually distrusting principals, security becomes relative. Different principals may have different security requirements. Moreover, in the presence of mutual distrust, different principals may have very different notions of who are the potential attackers.

Mutually distrusting principals need tools to express and enforce their security requirements, including declassification and erasure requirements. Moreover, principals need assurance that their requirements are satisfied, regardless of the actions of principals they distrust.

1.5 Contributions and outline

This dissertation presents and explores a framework for expressive and enforceable information security policies. A policy language for declassification and erasure requirements is presented in Chapter 2, including defining semantics for the policy language. The policy semantics are used to define a precise end-to-end security condition, *noninterference according to policy*.

Chapter 3 demonstrates that the policies can be enforced in a simple imperative language, IMP_E, using a type system for information-flow control, in conjunction with run-time mechanisms. Information in IMP_E is labeled with declassification and erasure policies. Any well-typed IMP_E program satisfies noninterference according to policy.

The decentralized label model (Myers and Liskov, 2000) (DLM) allows mutually distrusting principals to specify information security requirements, and is thus suitable for reasoning about security requirements in the presence of mutual distrust. In Chapter 4 we extend the DLM with declassification and erasure

requirements. We also define *decentralized robustness* (Chong and Myers, 2006), a security condition that uses the DLM to generalize robustness (Zdancewic and Myers, 2001; Myers et al., 2004; Zdancewic, 2003). Decentralized robustness requires that any change to information security is sufficiently trusted by the principals affected by the change. We describe how decentralized robustness restricts declassification and erasure. We also modify the language IMP_E so that information is labeled with labels from the DLM, and show that this language enforces decentralized robustness.

Chapters 2–4 introduce expressive policies that can capture common information security requirements, and that can be provably enforced. In Chapter 5, we incorporate the DLM with declassification and erasure policies into the Jif programming language (Myers, 1999; Myers et al., 2001–2008), a practical programming language that extends Java with information-flow control. We also adapt and incorporate into Jif the enforcement mechanisms for noninterference according to policy and decentralized robustness. We use the resulting language, Jif_E, to implement Civitas (Clarkson et al., 2008), a secure remote voting service, and describe the benefits derived therefrom.

Related work is discussed in Chapter 6, and Chapter 7 summarizes and considers future directions for this work.

The material in Chapters 2–5 is joint work with Andrew Myers, and is adapted from Chong and Myers (2005, 2006, 2008). The remote voting system Civitas is joint work with Michael Clarkson and Andrew Myers.

CHAPTER 2

ERASURE AND DECLASSIFICATION POLICIES

Formal, checkable, security policies can provide assurance that a computer system satisfies its security requirements. Designers and developers can formally express requirements using the policies, which can then be provably enforced in the subsequent implementation. There are two main challenges: designing policy languages that are rich enough to express the security requirements, yet simple enough to enforce provably; and providing end-to-end enforcement of the policies.

This chapter presents a policy language for two kinds of security requirements: erasure and declassification. The security policies describe how the confidentiality of information changes over time. This chapter also considers what semantic security guarantees hold when the security policies are enforced. We consider the end-to-end enforcement of policies in Chapter 3.

2.1 Policies

We assume there is a lattice $(\mathcal{L}, \sqsubseteq)$ of *confidentiality levels* that restrict the use of information and give a base vocabulary for expressing erasure and declassification policies. Appropriate lattices include the two-point lattice $\{L, H\}$ where $L \sqsubseteq H$ and $H \not\sqsubseteq L$, and the lattice of principals ordered by an *acts-for* relation (Myers and Liskov, 2000). (In Chapter 4 we use the lattice of principals when extending the decentralized label model (Myers and Liskov, 2000).) We assume there is a clear notion of enforcement of confidentiality level $\ell \in \mathcal{L}$ on information.

We also assume there is a language for specifying *conditions*, which indicate when declassification and erasure occur. Many condition languages are possible; Chapter 3 uses program expressions as conditions.

$\ell \in \mathcal{L}$	Lattice element
c, d	Conditions
$p, q ::=$	Policies
$\quad \ell$	$\quad$ Lattice policy
$\quad p \searrow^{c} q$	$\quad$ Declassification policy
$\quad p \overset{c}{\nearrow} q$	$\quad$ Erasure policy

Figure 2.1: Syntax of policies

Security policies describe what confidentiality level is currently enforced on information, and how this may and must change in the future. Figure 2.1 shows the syntax of policies.

Lattice policy $\ell \in \mathcal{L}$ means that confidentiality level ℓ (or a more restrictive confidentiality level) must be enforced on information now and at all times in the future.

Declassification policy $p \searrow^{c} q$ means that policy p is currently enforced on information, and when condition c is satisfied, information may be declassified, after which policy q must be enforced (regardless of the subsequent satisfaction or non-satisfaction of c).

Erasure policy $p \overset{c}{\nearrow} q$ means that policy p is currently enforced on information, and when condition c is satisfied, information must be made more restricted, by enforcing both policies p and q on the information (regardless of the subsequent satisfaction or non-satisfaction of c).

The satisfaction of conditions controls when declassification may occur, and when erasure must occur. Condition satisfaction is specific to the condition language used. We assume the condition satisfaction depends only on the current system state s (which may include the history of the system), and write $s \vDash c$ if condition c is satisfied in state s, and $s \nvDash c$ if c is not satisfied in state s. To instantiate the policy framework, sound decision procedures for the satisfaction relation $s \vDash c$ and non-satisfaction relation $s \nvDash c$ must be specified. For expres-

sive condition languages, the checking of condition satisfaction is likely to be incomplete. The effect of incompleteness will be just to make security analysis more conservative.

For example, if we are enforcing policy $H \searrow^{c} L$ on information, then we must enforce the confidentiality level H on the information; however, when condition c is satisfied, we are permitted to change the confidentiality level enforced on the information to L. Figure 2.2(a) depicts this graphically. The lines indicate the flow of information within a system. Initially the information may only flow within parts of the system where confidentiality level H is enforced, but may flow to other parts when the condition c is satisfied. Figure 2.2(b) shows what happens if condition c is never satisfied: the information can never be declassified, and must always have the policy H enforced on it.

If we are enforcing erasure policy $L \nearrow^{c} H$ on information, then we must enforce the confidentiality level L on the information, and if and when condition c is satisfied, we must change the confidentiality level we are enforcing to be at least as restrictive as both L and H—since $L \sqsubseteq H$, it suffices to enforce the confidentiality level H. Figure 2.2(c) shows this graphically. Information is initially in parts of the system where confidentiality level L is enforced, but may flow at any time to parts where the more restrictive confidentiality level H is enforced. However, the information must have confidentiality level H enforced on the information by the time c is satisfied. Figure 2.2(d) shows that when c is never satisfied, the information never needs to be erased.

Consider enforcing policy $(H \searrow^{c} L) \nearrow^{d} H$ on information. Initially policy $H \searrow^{c} L$ is enforced on information, meaning that the confidentiality level H must be enforced, and if condition c is satisfied (before d is satisfied) then confidentiality level L can be enforced on information. However, once condition d is satisfied,

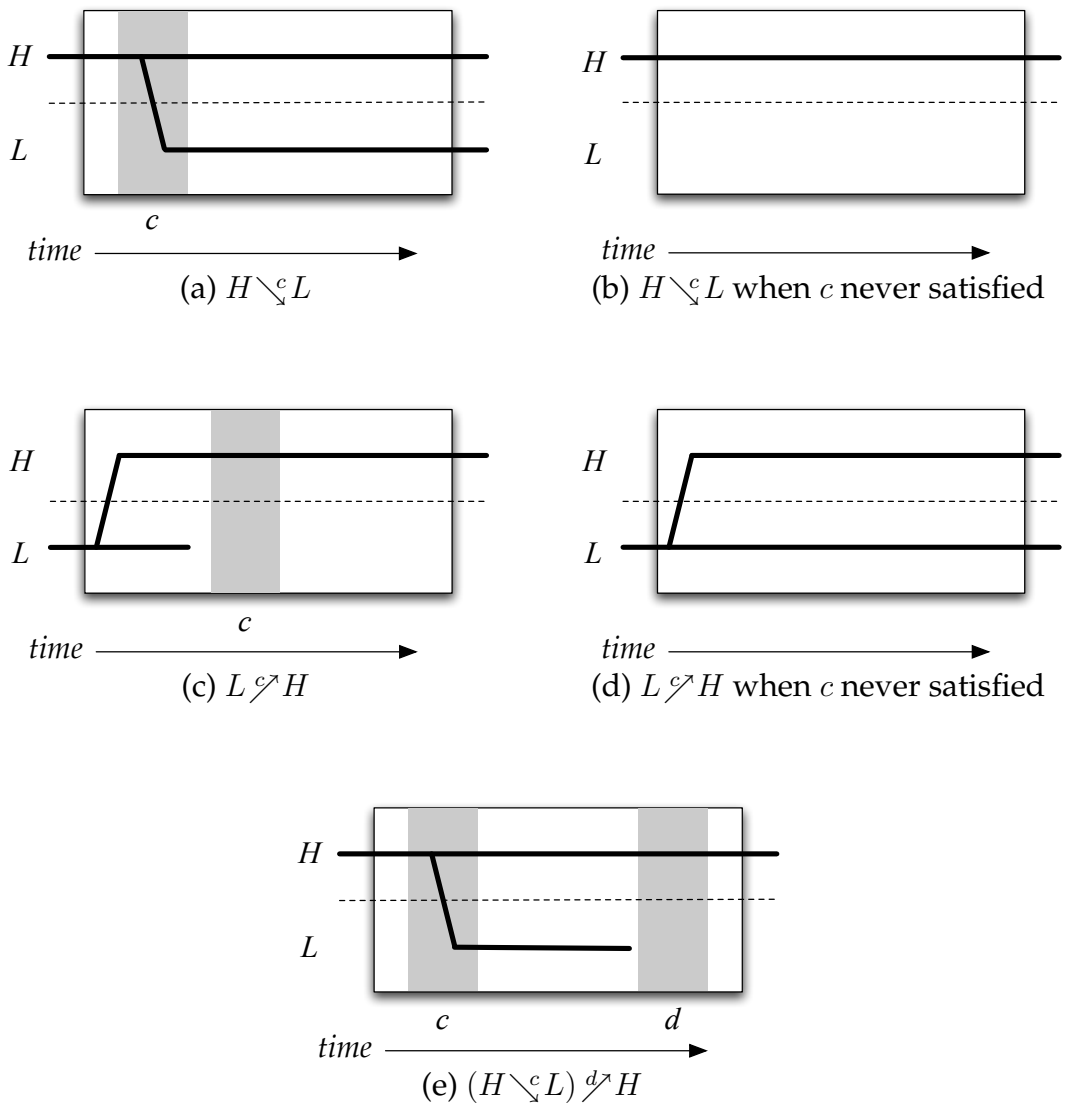

Figure 2.2: Declassification and erasure examples

$$\frac{\mathsf{reqErase}(p, s)}{\mathsf{reqErase}(p \searrow^{c} p', s)} \qquad \frac{\mathsf{reqErase}(p, s)}{\mathsf{reqErase}(p \nearrow^{c} p', s)} \qquad \frac{s \vDash c}{\mathsf{reqErase}(p \nearrow^{c} p', s)}$$

Figure 2.3: Definition of $\mathsf{reqErase}(p, s)$

we must enforce policy H on information, meaning that confidentiality level H will be enforced then and at all times in the future. Figure 2.2(e) shows this visually.

Condition satisfaction determines when policies mandate erasure. Policy p *requires information erasure* in state s (or simply, *requires erasure* in state s), denoted $\mathsf{reqErase}(p, s)$, if there is a currently enforced erasure policy whose condition is satisfied. Figure 2.3 gives inference rules defining $\mathsf{reqErase}(p, s)$. Lattice policy ℓ never requires erasure. Declassification policy $p \searrow^{c} q$ requires erasure if subpolicy p (the policy currently enforced) requires erasure. Erasure policy $p \nearrow^{c} q$ requires erasure if subpolicy p requires erasure, or c is satisfied.

To develop intuition for the erasure and declassification policies, and to show the expressiveness of the policy framework, we show how declassification and erasure security requirements presented in the examples of Section 1.3 can be represented using our policy framework.

Example 2.1 (Mobile computing) *When a mobile device moves from a secure environment to a less secure environment, there may be a requirement to erase sensitive information from the device.*

For example, suppose a laptop is used both at corporate headquarters and on client sites. At corporate headquarters, it is connected to the corporate LAN, and has access to sensitive documents; at the client site, it may be possible for client personnel to use the laptop.

When sensitive documents are downloaded onto the laptop at headquarters, a suitable security policy for the documents is $H \nearrow^{leaveHQ} \top$*, where* $leaveHQ$ *is satisfied when the*

laptop has left the secure environment of corporate headquarters, H is a confidentiality level for the sensitive documents, and $\top$ is a confidentiality level so high that the laptop is not permitted to hold any data at that level. Thus, the sensitive documents must be removed from the laptop at or before the time that the laptop is removed from corporate headquarters. Rather than leave the enforcement of this policy to the laptop user, the document management system on the laptop could automatically enforce this policy, erasing sensitive documents whenever the laptop leaves corporate headquarters, perhaps detected by disconnection from the corporate LAN. An efficient alternative to erasing the actual documents would be to encrypt them and remove the key from the laptop.

Example 2.2 (Public computers) *A user's session on a public computer should be independent of previous users' sessions, to prevent a user's sensitive information from being learned by a later user.*

One technique to enforce independence between sessions is to ensure that any information specific to a user session is erased before the start of the next user session. Erasure policies can provide a suitable expression of this security requirement. Let U be a confidentiality level corresponding to any information specific to a user's session, such as the time a session started, and which web sites were visited during the session. Let the condition $newSess$ be satisfied at some time before a new user's session begins, and let $\top$ be a confidentiality level so high that the public computer is not permitted to hold any data at that level. Applying the security policy $U \overset{newSess}{\nearrow} \top$ to all information entered or accessed in a user's session ensures that information from one user's session will be erased before another user's session begins.

However, it may be necessary to record information about each user's session for administrative purposes; for example, to gather statistics regarding how many people use the public computer and for how long. Thus, completely removing all information specific to a user's session may not be possible. We can adapt the erasure policy for

user information to permit the recording of information for administrative purposes while ensuring that each user's session is independent of other users' sessions. Let A be a confidentiality level for administrative information, and assume that $U \sqsubseteq A$ and $A \not\sqsubseteq U$. Then the erasure policy $U \overset{newSess}{\nearrow} A$ will allow information from users' sessions to be recorded for administrative purposes, but because $A \not\sqsubseteq U$, information held for administrative purposes cannot influence users' sessions.

Example 2.3 (Online transaction) *Consider a consumer who gives his credit card number to a merchant, to make some purchase. The merchant should not keep any record of the credit card number after the transaction, but the merchant must send the credit card number to the bank, which* will *keep a record of it.*

Let M be a confidentiality level corresponding to information stored by the merchant. Let B be a confidentiality level corresponding to information stored by the bank. Then a suitable policy for the credit card number is $(M \overset{pur}{\searrow} B) \overset{end}{\nearrow} B$, where pur is a condition that is satisfied once the consumer has approved the purchase, and end is a condition that is satisfied by the end of the transaction.

Note that policy $(M \overset{pur}{\searrow} B) \overset{end}{\nearrow} B$ allows the merchant to release the credit card details to the bank once the customer has approved the purchase, since (as will be made precise in Section 2.2.3) information labeled with policy $(M \overset{pur}{\searrow} B) \overset{end}{\nearrow} B$ is permitted to be relabeled with the policy B, provided the condition pur is satisfied at the time of relabeling. However, at the end of the transaction, the policy B should be enforced on the credit card number, meaning that the bank is allowed to store the number, but the merchant must have removed the number from his system.

Now suppose we extend the example so that the consumer can optionally allow the merchant to store the credit card number. This may allow the merchant to maintain a customer profile, and save the consumer from needing to re-enter the credit card number for subsequent purchases. A suitable policy for the credit card number is now

$((M \searrow^{pur} B) \overset{end}{\nearrow} B) \searrow^{pro} (M \searrow^{pur} B)$, *where* pro *is a condition that is satisfied when the consumer has given permission for the merchant to maintain a customer profile. Note that if the consumer gives permission, then the merchant may store the credit card number with a policy* $M \searrow^{pur} B$*, allowing the merchant to send the credit card number to the bank when the consumer makes a purchase; if the consumer does not give permission, then the merchant is still required to erase the credit card number by the end of the transaction.*

Example 2.4 (Medical information website) *A diagnostic application on a medical information website takes symptoms entered by the user, and produces possible diagnoses consistent with the symptoms. The website's privacy policy states that no record of the user's symptoms will be kept after the user has finished using the application.*

A suitable policy for symptoms entered by the user could be $session \overset{appEnd}{\nearrow} \top$*, where* $session$ *is a confidentiality level allowing only the session client and server to read the information,* $\top$ *is a confidentiality level so restrictive that it prevents the server from storing the information, and* $appEnd$ *is a condition that is satisfied when the user has finished using the diagnosis application. Thus, the data entered by the user will initially have the confidentiality level* $session$ *enforced on it. Once condition* $appEnd$ *is satisfied, the confidentiality level* $\top$ *must be enforced, implying that the data will be removed completely from the system. End-to-end enforcement of the policies will ensure that information derived from the user's symptoms will have the same policy,* $session \overset{appEnd}{\nearrow} \top$ *enforced on it, or something more restrictive. Thus, any diagnoses derived from the user's symptoms must also have the confidentiality level* $\top$ *enforced on them once* $appEnd$ *is satisfied.*

Example 2.5 (Poker) *A poker player's cards should not be revealed until the end of the round.*

A suitable security policy for the cards of player P_i is $P_i \searrow^{end} \bot$, where end is satisfied at the end of the round, P_i is a confidentiality level that only player P_i can read, and $\bot$ is the least secret confidentiality level; we assume all players may observe information at the confidentiality level $\bot$. This policy allows player P_i to reveal his cards to the other players at the end of the round.

The above examples demonstrate that policies can express a wide range of application-specific declassification and erasure requirements. There are, however, declassification and erasure requirements that cannot be expressed using the policies. For example, consider data that has confidentiality level H enforced on it, but may be declassified to exactly one of confidentiality levels L_1 or L_2, where $L_1 \sqsubseteq H$ and $L_2 \sqsubseteq H$ and L_1 and L_2 are incomparable. No policy can capture this requirement. For example, assuming conditions c and d describe when the data may be declassified to L_1 and L_2 respectively, policy $(H \searrow^c L_1) \searrow^d L_2$ allows the data to be declassified to first L_1 and then L_2. Due to their nested structure, policies cannot restrict the declassification of data appropriately.

Although the policy syntax could be altered to be more expressive, we refrain from doing so. The policies are sufficiently expressive to capture many useful declassification and erasure requirements.

2.2 Semantics

Erasure and declassification policies describe how the confidentiality of information changes over time. We formalize this intuition by defining a semantics for policies, and exploring properties of the semantics. We also define a relabeling judgment that soundly approximates the semantics, but can be checked syntactically.

2.2.1 Notation

We first introduce some concepts and notation for reasoning about the execution of systems.

Let S be a system. Let Σ_S denote the *feasible states* of S, that is, all states that may occur in some execution of the system S.

For any two states $s, s' \in \Sigma_S$, we write $s \rightarrow s'$ if and only if the system can atomically transition from s to s'. The relation $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$.

A trace $\tau = s_0 s_1 \ldots$ is a finite or infinite sequence of states such that $s_i \rightarrow s_{i+1}$ for all i such that $i \geq 0$ and i is less than the length of τ. We use $|\tau|$ to denote the length of τ, defining $|\tau| = \infty$ if τ is an infinite sequence. The ith element of τ is denoted $\tau[i]$, that is, $\tau[i] = s_i$. Thus, the last element of a finite sequence τ is denoted $\tau[|\tau| - 1]$.

Let $\tau = s_0 s_1 \ldots$ be a finite or infinite sequence of states. We use $\tau[..k]$ to denote the sequence of states $s_0 s_1 \ldots s_k$, where $k < |\tau|$. For finite trace τ and trace τ' where $\tau[|\tau| - 1] \rightarrow \tau'[0]$ we write $\tau\tau'$ for the trace obtained by appending τ' to τ. For notational convenience, if τ is a finite trace, we write $\mathsf{reqErase}(p, \tau)$ as an abbreviation for $\mathsf{reqErase}(p, \tau[|\tau| - 1])$.

As mentioned in Section 2.1, we write $s \vDash c$ if condition c is satisfied when the system state is s, and write $s \nvDash c$ if condition c is not satisfied when the system state is s.

We write $[s, s'] \nvDash c$, where $s \rightarrow^* s'$, to mean that condition c is not satisfied in any state from s to s' inclusive:

$$[s, s'] \nvDash c \quad \triangleq \quad \forall s''.\ (s \rightarrow^* s'' \wedge s'' \rightarrow^* s') \Rightarrow s'' \nvDash c.$$

$$
\begin{aligned}
[\![\ell]\!]_s &= \{(s', \ell') \mid s \rightarrow^* s' \text{ and } \ell \sqsubseteq \ell'\} \\
[\![p \searrow^c q]\!]_s &= [\![p]\!]_s \cup \bigcup \{[\![q]\!]_{s'} \mid s \rightarrow^* s' \text{ and } s' \vDash c\} \\
[\![p \nearrow^c q]\!]_s &= [\![p]\!]_s \cap \Big(\{(s', \ell) \in [\![p]\!]_s \mid [s, s'] \nvDash c\} \ \cup \\
&\qquad \bigcup \{[\![q]\!]_{s''} \mid s \rightarrow^* s'' \text{ and } [s, s'') \nvDash c \text{ and } s'' \vDash c\} \Big)
\end{aligned}
$$

Figure 2.4: Semantics for policies $[\![p]\!]_s$

Similarly, we use $[s, s') \nvDash c$, to mean that condition c is not satisfied in any state from s up to but not including, state s':

$$[s, s') \nvDash c \quad \triangleq \quad \forall s''.\ (s \rightarrow^* s'' \wedge s'' \rightarrow^* s' \wedge s'' \neq s') \Rightarrow s'' \nvDash c.$$

2.2.2 Policy semantics

Suppose the current state of the system is s, and we have some information on which we are enforcing the policy p. The semantics of policy p in state s, denoted $[\![p]\!]_s$, describe what confidentiality levels may be enforced on the information, as the system evolves from state s. Thus, the semantics describe how the confidentiality of the information is allowed to change during the execution of the system.

More formally, the semantics of policy p in state s is a set of pairs (s, ℓ) of system states s and confidentiality levels ℓ. If policy p is enforced on information in state s, and $(s', \ell') \in [\![p]\!]_s$, then the state s' is reachable from the state s in zero or more steps, and confidentiality level ℓ' may be enforced on the information in state s'. Figure 2.4 defines the semantics $[\![p]\!]_s$.

The semantics for confidentiality level ℓ allow any confidentiality level at least as restrictive as ℓ to be enforced at all times in the future. The semantics for

a lattice policy ℓ is thus all possible pairs (s', ℓ') where the state s' is reachable from the state s, and $\ell \sqsubseteq \ell'$.

The semantics of declassification policy $p \searrow^c_{\searrow} q$ is a superset of the semantics of policy p. If p permits enforcing confidentiality ℓ in state s', then $p \searrow^c_{\searrow} q$ also permits it, and in addition, permits policy q to be enforced on information, starting in any state s' such that $s' \vDash c$.

By contrast, the semantics of erasure policy $p \overset{c}{\nearrow} q$ in state s is a subset of the semantics of p in s. If erasure condition c has not been satisfied from state s to s', then confidentiality level ℓ may be enforced on information in state s' provided policy p permits it. However, if condition c has been satisfied, then ℓ may be enforced in state s' only if the semantics of p in s and q in s'' permit it, for some state s'' where condition c is not satisfied between s and s''. The intuition is that the information will only be present in the system if the system started enforcing both policies p and q in state s''.

Properties of $[\![p]\!]_s$

The semantics of policies have several useful and interesting properties.

First, for any given policy p and states s and s', the set of confidentiality levels $\{\ell \mid (s', \ell) \in [\![p]\!]_s\}$ is closed upward.

Property 2.6 *For all policies p, states s and s' and confidentiality levels ℓ, if $(s', \ell) \in [\![p]\!]_s$, then for all ℓ' such that $\ell \sqsubseteq \ell'$ we have $(s', \ell') \in [\![p]\!]_s$.*

Proof: Proof is by induction on the structure of p. Suppose for any subpolicy of p the result holds. Let s and s' be states and ℓ a confidentiality level such that $(s', \ell) \in [\![p]\!]_s$. Let ℓ' be a confidentiality level such that $\ell \sqsubseteq \ell'$. We need to show $(s', \ell') \in [\![p]\!]_s$.

- $p \equiv \ell''$. If $(s', \ell) \in [\![\ell'']\!]_s$, then $(s', \ell') \in [\![\ell'']\!]_s$.

- $p \equiv q \searrow^{d} q'$. If $(s', \ell) \in [\![p]\!]_s$ then either $(s', \ell) \in [\![q]\!]_s$ or $(s', \ell) \in [\![q']\!]_{s''}$ for some s''. Either way, by the inductive hypothesis, the result holds.
- $p \equiv q \not\nearrow^{d} q'$. If $(s', \ell) \in [\![p]\!]_s$ then either $(s', \ell) \in [\![q]\!]_s \cap [\![q']\!]_{s''}$, for some s'', or $(s', \ell) \in [\![q]\!]_s$. Either way, by the inductive hypothesis the result holds.

∎

For all policies p, we observe that as time goes on, as long as the information does not require erasure, the set of possible confidentiality levels that may be enforced on information decreases.

Property 2.7 *Let p be a policy and s and s' be states such that $s \rightarrow^* s'$. If for all states s'' such that $s \rightarrow^* s'' \rightarrow^* s'$ and $s' \neq s''$ we have $\neg\mathsf{reqErase}(p, s'')$, then $[\![p]\!]_{s'} \subseteq [\![p]\!]_s$.*

Proof: Proof is by induction on the structure of p. Suppose for any subpolicy of p the result holds. Let s and s' be states such that $s \rightarrow^* s'$ and for all states s'' such that $s \rightarrow^* s'' \rightarrow^* s'$ and $s' \neq s''$ we have $\neg\mathsf{reqErase}(p, s'')$. We need to show $[\![p]\!]_{s'} \subseteq [\![p]\!]_s$.

- $p \equiv \ell$. Trivial.
- $p \equiv q \searrow^{d} q'$. If $(t, \ell) \in [\![p]\!]_{s'}$ then either $(t, \ell) \in [\![q]\!]_{s'}$ or $(t, \ell) \in [\![q']\!]_{t'}$, for some t' such that $s' \rightarrow^* t'$ and $t' \vDash d$. If the former, then since for all s'', $\mathsf{reqErase}(p, s'')$ if and only if $\mathsf{reqErase}(q, s'')$, by the inductive hypothesis we have $[\![q]\!]_{s'} \subseteq [\![q]\!]_s$. If the latter, then $s \rightarrow^* t'$, so $[\![q']\!]_{s'} \subseteq [\![p]\!]_s$. Thus, $[\![p]\!]_{s'} \subseteq [\![p]\!]_s$.
- $p \equiv q \not\nearrow^{d} q'$. If $(t, \ell) \in [\![p]\!]_{s'}$ then either $(t, \ell) \in [\![q]\!]_{s'}$ or $(t, \ell) \in [\![q]\!]_{s'} \cap [\![q']\!]_{t'}$, for some t' such that $s' \rightarrow^* t'$ and $[s', t) \nvDash d$. If the former, then since for all s'', $\neg\mathsf{reqErase}(p, s'')$ implies $\neg\mathsf{reqErase}(q, s'')$, by the inductive hypothesis we have $[\![q]\!]_{s'} \subseteq [\![q]\!]_s$. If the later, then $s \rightarrow^* t'$ and $[s, t) \nvDash d$, so $[\![q]\!]_{s'} \cap [\![q']\!]_{t'} \subseteq [\![p]\!]_s$. Thus, $[\![p]\!]_{s'} \subseteq [\![p]\!]_s$.

$$\text{RL-LATTICE}\ \frac{\ell \sqsubseteq \ell'}{c_0, \ldots, c_k \vdash \ell \leq \ell'} \qquad \text{RL-TRANS}\ \frac{c_0, \ldots, c_k \vdash p \leq p' \quad c_0, \ldots, c_k \vdash p' \leq p''}{c_0, \ldots, c_k \vdash p \leq p''} \qquad \text{RL-DECL}\ \frac{c \in \{c_0, \ldots, c_k\}}{c_0, \ldots, c_k \vdash p \searrow^{c} p' \leq p'}$$

$$\text{RL-DECL-I}\ \frac{c_0, \ldots, c_k \vdash q \leq p \quad c \vdash q \leq p'}{c_0, \ldots, c_k \vdash q \leq p \searrow^{c} p'} \qquad \text{RL-DECL-E}\ \frac{}{c_0, \ldots, c_k \vdash p \searrow^{c} p' \leq p} \qquad \text{RL-DECL-DECL}\ \frac{c_0, \ldots, c_k \vdash p \leq q \quad c \vdash p' \leq q'}{c_0, \ldots, c_k \vdash p \searrow^{c} p' \leq q \searrow^{c} q'}$$

$$\text{RL-ERASE-I}\ \frac{}{c_0, \ldots, c_k \vdash p \leq p \nearrow^{c} p'} \qquad \text{RL-ERASE-E}\ \frac{c_0, \ldots, c_k \vdash p \leq q \quad \vdash p' \leq q}{c_0, \ldots, c_k \vdash p \nearrow^{c} p' \leq q} \qquad \text{RL-ERASE-ERASE}\ \frac{c_0, \ldots, c_k \vdash p \leq q \quad \vdash p' \leq q'}{c_0, \ldots, c_k \vdash p \nearrow^{c} p' \leq q \nearrow^{c} q')}$$

Figure 2.5: Inference rules for $c_0, \ldots, c_k \vdash p \leq q$

∎

2.2.3 Relabeling judgment

The policy semantics provides a formal meaning for policies, and allows us to reason about the relative restrictiveness of policies. However, when considering enforcement of policies in a language-based setting, it is useful to avoid reasoning directly about the semantics, and instead use a sound syntactic approximation to the semantics.

We define the *relabeling judgment* $c_0, \ldots, c_k \vdash p \leq q$ over policies such that if $c_0, \ldots, c_k \vdash p \leq q$ then, assuming conditions $c_0, \ldots, c_k$ are all satisfied, information labeled with policy p can safely be relabeled with policy q. That is, in any state s that satisfies all conditions $c_0, \ldots, c_k$, enforcing q on the information in that state is consistent with policy p.

Intuitively, for the relabeling judgment $c_0, \ldots, c_k \vdash p \leq q$ to be sound, policy q must be at least as restrictive as policy p. That is, anything that q permits to be done to information, p permits as well. More formally, soundness requires that if $c_0, \ldots, c_k \vdash p \leq q$ then for any state s such that $s \vDash c_i$ for all $i \in 0..k$, $[\![q]\!]_s \subseteq [\![p]\!]_s$. Theorem 2.9 below proves the soundness of the relabeling judgment.

Figure 2.5 shows inference rules for the $c_0, \ldots, c_k \vdash p \leq q$ judgment. The rule RL-LATTICE states that information may be relabeled from lattice policy ℓ to lattice policy ℓ' in any state, provided that $\ell \sqsubseteq \ell'$. The rule RL-TRANS makes the judgment transitive on policies.

The declassification rule RL-DECL permits relabeling from a declassification policy $p \searrow^c p'$ to policy p', provided that condition c is satisfied. This rule captures the intuitive meaning of declassification policies: declassification may occur when the appropriate condition is satisfied. Note that rule RL-DECL permits relabeling from $p \searrow^c p'$ to p', and p' may permit declassifications or require erasures that $p \searrow^c p'$ does not.

The declassification introduction rule RL-DECL-I describes when it is permissible to relabel information from some policy q to the policy $p \searrow^c p'$. First, it must be permitted to relabel information from q to p when $c_0, \ldots, c_k$ are satisfied; second, in any state where condition c is satisfied, it must be permitted to relabel information from q to p'.

The declassification elimination rule RL-DECL-E allows information to be relabeled from a declassification policy $p \searrow^c p'$ to the policy p. Intuitively, it is acceptable to relabel information from $p \searrow^c p'$ to p, since policy p is always more restrictive than policy $p \searrow^c p'$, which enforces everything that p does but also permits declassification to p'.

The rule RL-DECL-DECL describes when information may be relabeled from

one declassification policy $p \searrow^{c} p'$ to another, more restrictive declassification policy $q \searrow^{d} q'$. The intuition is that this is permitted if q is at least as restrictive as p when $c_0, \ldots, c_k$ are satisfied, the policy $q \searrow^{d} q'$ permits declassification only when $p \searrow^{c} p'$ does (that is, $d = c$), and, whenever declassification is permitted, q' is at least as restrictive as p'.

As can be seen by inspection of Figure 2.5, each of the relabeling rules for erasure policies corresponds to a relabeling rule for declassification. For example, erasure introduction RL-ERASE-I is analogous to RL-DECL-E: information may be relabeled from p to $p \nearrow^{c} p'$, since $p \nearrow^{c} p'$ is always more restrictive than p. An erasure policy $p \nearrow^{c} p'$ enforces everything that p does, and in addition requires the information to be erased at certain times.

Erasure elimination RL-ERASE-E is analogous to the rule for declassification introduction, allowing information to be relabeled from $p \nearrow^{c} p'$ to q provided that p can be relabeled to q when conditions $c_0, \ldots, c_k$ are satisfied, and p' can be relabeled to q at all times. Intuitively, information may be relabeled to q since information labeled q would not need to be erased when c is satisfied, as q is at least as restrictive as both p and p'.

The rule RL-ERASE-ERASE compares two erasure policies, $p \nearrow^{c} p'$ and $q \nearrow^{d} q'$, and is similar to RL-DECL-DECL. Information may be relabeled from $p \nearrow^{c} p'$ to $q \nearrow^{d} q'$ provided that q is at least as restrictive as p when $c_0, \ldots, c_k$ are satisfied, and whenever $p \nearrow^{c} p'$ requires information to be erased, so does $q \nearrow^{d} q'$ (that is, $d = c$), and at all times, q' is at least as restrictive as p'.

There is no erasure rule analogous to RL-DECL. This is because erasure policies specify information flows that must not happen, which is difficult to capture with inference rules of this style. Instead, the onus of ensuring information is erased at appropriate times falls upon the system that enforces the policies.

Properties of $c_0, \ldots, c_k \vdash p \leq q$

What properties does the relabeling judgment exhibit? It is easy to establish that, given any conditions $c_0, \ldots, c_k$, the relation over policies implied by the judgment $c_0, \ldots, c_k \vdash p \leq q$ is a pre-order: transitive and reflexive. However, it does not form a partial order, as it is not anti-symmetric. For example, for any condition c, and confidentiality $\ell \in \mathcal{L}$, we have both $\vdash \ell \leq \ell \searrow^c_\searrow \ell$ and $\vdash \ell \searrow^c_\searrow \ell \leq \ell$.

The top and bottom elements of the confidentiality lattice $\mathcal{L}$, denoted $\top_\mathcal{L}$ and $\bot_\mathcal{L}$ respectively, are greatest and least elements: for all policies p, we have $c_0, \ldots, c_k \vdash p \leq \top_\mathcal{L}$ and $c_0, \ldots, c_k \vdash \bot_\mathcal{L} \leq p$.

Property 2.8 *For any conditions $c_0, \ldots, c_k$, and all policies p and q, the relation $\leq_{c_0,\ldots,c_k}$ over policies, where $p \leq_{c_0,\ldots,c_k} q$ if $c_0, \ldots, c_k \vdash p \leq q$, is a pre-order. Moreover, for all policies p, $c_0, \ldots, c_k \vdash p \leq \top_\mathcal{L}$ and $c_0, \ldots, c_k \vdash \bot_\mathcal{L} \leq p$.*

Proof: For any conditions $c_0, \ldots, c_k$, the transitivity of $\leq_{c_0,\ldots,c_k}$ follows immediately from RL-TRANS. We prove reflexiveness by induction on policies p. Assume that for all sub-policies p' of p, we have $c_0, \ldots, c_k \vdash p' \leq p'$.

Consider the form of p.

- $p \equiv \ell$: By RL-LATTICE and reflexiveness of the lattice ordering $\sqsubseteq$, we can conclude $c_0, \ldots, c_k \vdash \ell \leq \ell$.
- $p \equiv q \searrow^d_\searrow q'$: By the inductive hypothesis, $c_0, \ldots, c_k \vdash q \leq q$ and $c_0, \ldots, c_k \vdash q' \leq q'$. It follows immediately by RL-DECL-DECL that $c_0, \ldots, c_k \vdash p \leq p$ as required.
- $p \equiv q \nearrow^d_\nearrow q'$: By the inductive hypothesis, $c_0, \ldots, c_k \vdash q \leq q$ and $c_0, \ldots, c_k \vdash q' \leq q'$. It follows immediately by RL-ERASE-ERASE that $c_0, \ldots, c_k \vdash p \leq p$ as required.

Thus, $\leq_{c_0,\ldots,c_k}$ is a pre-order.

To show that $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq p$, and $c_0, \ldots, c_k \vdash p \leq \top_{\mathcal{L}}$, we proceed by induction on the structure of policies p.

Assume that for all sub-policies p' of p we have $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq p'$, and $c_0, \ldots, c_k \vdash p' \leq \top_{\mathcal{L}}$. Consider the form of p.

- $p \equiv \ell$: By the lattice properties of $\mathcal{L}$, we have $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq \ell$, and $c_0, \ldots, c_k \vdash \ell \leq \top_{\mathcal{L}}$.
- $p \equiv q \searrow^{d} q'$: By the inductive hypothesis, we have $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq q$ and $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq q'$, so by RL-DECL-I we can conclude $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq p$. Similarly, we have $c_0, \ldots, c_k \vdash q \leq \top_{\mathcal{L}}$, so by RL-DECL-E we can conclude $c_0, \ldots, c_k \vdash p \leq \top_{\mathcal{L}}$.
- $p \equiv q \nearrow^{d} q'$: By the inductive hypothesis, we have $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq q$, so by RL-ERASE-I we can conclude $c_0, \ldots, c_k \vdash \bot_{\mathcal{L}} \leq p$. Similarly, we have $c_0, \ldots, c_k \vdash q \leq \top_{\mathcal{L}}$ and $c_0, \ldots, c_k \vdash q' \leq \top_{\mathcal{L}}$, so by RL-ERASE-E we can conclude $c_0, \ldots, c_k \vdash p \leq \top_{\mathcal{L}}$.

∎

The following theorem shows that the relabeling judgment $c_0, \ldots, c_k \vdash p \leq q$ is sound, in the sense that if information may be relabeled from p to q in some state s in which conditions $c_0, \ldots, c_k$ are all satisfied, then $[\![p]\!]_s \supseteq [\![q]\!]_s$. That is, if $c_0, \ldots, c_k \vdash p \leq q$, then q is at least as restrictive as p, in the sense that if it is secure to enforcing confidentiality level ℓ on information labeled with policy q in state s (for example, allowing some set of people to read the information), then it would also be secure to enforce ℓ on information labeled p in state s.

Theorem 2.9 *If $c_0, \ldots, c_k \vdash p \leq q$ then for all states s, such that $\forall i \in 0..k.\ s \vDash c_i$, we have $[\![p]\!]_s \supseteq [\![q]\!]_s$.*

Proof: We proceed by induction on the proof of $c_0, \ldots, c_k \vdash p \leq q$. The inductive hypothesis is that for any premise of the form $c_0, \ldots, c_k \vdash p' \leq q'$, we have $[\![p']\!]_s \supseteq [\![q']\!]_s$ for any state s such that $\forall i \in 0..k.\ s \vDash c_i$.

- **RL-LATTICE, RL-TRANS.** Trivial.
- **RL-DECL.** Here $p \equiv p' \searrow^c q$, and $s \vDash c$. We have $[\![p' \searrow^c q]\!]_s = [\![p']\!]_s \cup \bigcup\{[\![q]\!]_{s'} \mid s \rightarrow^* s' \text{ and } s' \vDash c\} \supseteq [\![q]\!]_s$, since $s \vDash c$.
- **RL-DECL-I.** Here $q \equiv q' \searrow^d q''$, and $c_0, \ldots, c_k \vdash p \leq q'$, and $d \vdash p \leq q''$. By the inductive hypothesis, we have $[\![p]\!]_s \supseteq [\![q']\!]_s$ and $[\![p]\!]_{s'} \supseteq [\![q'']\!]_{s'}$ for any state s' such that $s' \vDash d$. Thus, $[\![p]\!]_s \supseteq [\![q]\!]_s$ as required.
- **RL-DECL-E.** Here $p \equiv q \searrow^c p'$, and $[\![q \searrow^c p']\!]_s = [\![q]\!]_s \cup \bigcup\{[\![p']\!]_{s'} \mid s \rightarrow^* s' \text{ and } s' \vDash c\} \supseteq [\![q]\!]_s$.
- **RL-DECL-DECL.** Here $p \equiv p' \searrow^c p''$ and $q \equiv q' \searrow^c q''$. By the inductive hypothesis, we have $[\![p']\!]_s \supseteq [\![q']\!]_s$. Also, for any s' such that $s \rightarrow^* s'$ and $s' \vDash c$, we have $c \vdash p'' \leq q''$, so by the inductive hypothesis, $[\![p'']\!]_{s'} \supseteq [\![q'']\!]_{s'}$. So we have $[\![p' \searrow^c p'']\!]_s \supseteq [\![q' \searrow^c q'']\!]_s$.
- **RL-ERASE-E.** Here $p \equiv p' \nearrow^c p''$, and, by the inductive hypothesis, $[\![p']\!]_s \supseteq [\![q]\!]_s$ and $[\![p'']\!]_{s'} \supseteq [\![q]\!]_{s'}$ for any state s'. Suppose we have a pair $(s', \ell) \in [\![q]\!]_s$. Either (1) $(s', \ell) \in [\![p']\!]_s$ and $[s, s'] \nvDash c$, or (2) $(s', \ell) \in [\![p']\!]_s \cap [\![p'']\!]_{s''}$ for some s'' such that $s \rightarrow^* s''$ and $[s, s''] \nvDash c$. If case (1) then $(s', \ell) \in [\![p]\!]_s$. If case (2) then by assumption $\vdash p'' \leq q$, and by the inductive hypothesis, we have $[\![p'']\!]_{s''} \supseteq [\![q]\!]_{s''}$, so $(s', \ell) \in [\![p]\!]_s$. Thus $[\![p]\!]_s \supseteq [\![q]\!]_s$ as required.
- **RL-ERASE-I.** Here $q \equiv p \nearrow^d q'$. Clearly, $[\![p]\!]_s \supseteq [\![p \nearrow^d q']\!]_s = [\![q]\!]_s$.
- **RL-ERASE-ERASE.** Here $p \equiv p' \nearrow^c p''$ and $q \equiv q' \nearrow^c q''$. By the inductive hypothesis, we have $[\![p']\!]_s \supseteq [\![q']\!]_s$ and for all s', $[\![p'']\!]_{s'} \supseteq [\![q'']\!]_{s'}$. Suppose we have a pair $(s', \ell) \in [\![q']\!]_s$. Either (1) $(s', \ell) \in [\![q']\!]_s$ and $[s, s'] \nvDash c$, or (2)

$(s', \ell) \in [\![q]\!]_s \cap [\![q'']\!]_{s''}$ for some s'' such that $s \rightarrow^* s''$ and $[s, s'') \not\models c$. If case (1) then $(s', \ell) \in [\![p]\!]_s$. If case (2) then by assumption $\vdash p'' \leq q''$, and by the inductive hypothesis, we have $[\![p'']\!]_{s''} \supseteq [\![q'']\!]_{s''}$, so $(s', \ell) \in [\![p]\!]_s$. Thus $[\![p]\!]_s \supseteq [\![q]\!]_s$ as required.

∎

2.3 Security properties

Using the policy semantics, we define an end-to-end semantic security condition, *noninterference according to policy* (Chong and Myers, 2005), a generalization of noninterference (Goguen and Meseguer, 1982). We first present the observational model used in the definitions of the semantic security conditions.

2.3.1 Observational model

Semantic security conditions based on noninterference (Goguen and Meseguer, 1982) require that high security inputs do not affect low security outputs. The observational model, and the precise definitions of input, output, high and low security, lead to slightly different definitions of noninterference (O'Neill et al., 2006).

We define an observational model suitable for state-based systems. The model is general, and we instantiate it for a simple imperative language in Chapter 3.

For system S, let O be the set of *observables*, the possible observations that can be made on S. We assume that the result of an observation depends only on the current system state, and for a state $s \in \Sigma_S$ and observable $o \in O$, we write $s(o)$ for the result of making observation o when the system is in state s. Observables

$$
\begin{aligned}
\mathsf{obs}(\ell) &\triangleq \ell \\
\mathsf{obs}(p \searrow^{c}_{\searrow} q) &\triangleq \mathsf{obs}(p) \\
\mathsf{obs}(p \overset{c}{\nearrow} q) &\triangleq \mathsf{obs}(p)
\end{aligned}
$$

Figure 2.6: Observation level $\mathsf{obs}(p)$

might include the contents of a memory location or register, the temperature of the CPU, or the image currently displayed on a monitor. The intention is that the set of observables O is general enough to model all observable aspects of a system, such as user or network input and output. If the result of an observation depends on the history of execution, then without loss of generality we can assume that the system state maintains a history of the system's execution, and thus the result of any observation depends only on the current state of the system.

Attackers can observe some, but not necessarily all, of a system's observables. We assume that for every observable $o \in O$ there is security policy associated with o, denoted $\mathsf{pol}(o)$, and a given attacker's ability to make observation o depends only on the security policy $\mathsf{pol}(o)$. Intuitively, the security policy $\mathsf{pol}(o)$ is the policy the system should enforce on the information that may be revealed by making the observation. Our model assumes that each observable o has a fixed policy for the duration of the execution; it is possible to model observables with changing policies using collections of observables (cf. Hunt and Sands 2006).

We assume that the confidentiality lattice $\mathcal{L}$ can describe the observability of information, and associate a confidentiality level with each policy p that describes the *policy observability*, denoted $\mathsf{obs}(p)$. We assume that for a given attacker A, there is some lattice element ℓ_A such that the attacker can make all and only observations o such that $\mathsf{obs}(\mathsf{pol}(o)) \sqsubseteq \ell$.

Figure 2.6 presents inference rules for the function $\mathsf{obs}(\cdot)$. The observability of a lattice policy ℓ is simply ℓ, and the observability of declassification and erasure

policies is the observability of the left subpolicy. Intuitively, the left subpolicy is the policy that is currently enforced; the right subpolicy indicates the policy that may (for declassification) or must (for erasure) be enforced in the future. Thus, observability is determined by the left subpolicy.

Using the policy observability function, we define the *observational equivalence relation* $\approx_\ell$ on states, such that for any two states $s, s' \in \Sigma_S$, we have $s \approx_\ell s'$ if the two states are indistinguishable to the attacker; that is, if for all observables $o \in O$, if the attacker can observe o, then $s(o) = s'(o)$.

Definition 2.10 (Observational equivalence) *For any confidentiality level $\ell \in \mathcal{L}$, and states $s, s' \in \Sigma_S$, s and s' are* observationally equivalent at level ℓ, *denoted $s \approx_\ell s'$, if for all observables $o \in O$ such that $\mathsf{obs}(\mathsf{pol}(o)) \sqsubseteq \ell$, we have $s(o) = s'(o)$.*

We use *correspondences* (Banerjee et al., 2008) between traces to indicate which states appear equivalent to an observer that sees first one trace, then the other. A correspondence R is a relation over the natural numbers. If R is a correspondence for traces τ and τ', and $(i, j) \in R$, we will use it to mean that $\tau[i]$ and $\tau'[j]$ look the same to a given observer.

Definition 2.11 (Correspondences) *A* correspondence R *between traces τ and τ' is a subset of $\mathbb{N} \times \mathbb{N}$ such that*

1. *(Completeness) either $\{i \mid (i, j) \in R\} = \{i \in \mathbb{N} \mid i < |\tau|\}$ or $\{j \mid (i, j) \in R\} = \{j \in \mathbb{N} \mid j < |\tau'|\}$; and*
2. *(Initial states) if $|R| > 0$ then $(0, 0) \in R$; and*
3. *(Monotonicity) for all $(i, j) \in R$ and $(i', j') \in R$, if $i < i'$ then $j \leq j'$; and, symmetrically, if $j < j'$ then $i \leq i'$.*

For a correspondence R between traces τ and τ', and numbers i, j, if $(i, j) \in R$ then we say that state $\tau[i]$ corresponds to state $\tau'[j]$, and vice versa.

This definition of correspondences ensures that a correspondence covers all states in at least one of τ or τ', and if both traces are non-empty, then the initial states of the traces correspond to each other. The monotonicity requirement implies that the observer observes each trace as it executes, and time moves only forward.

Correspondences are both timing and termination insensitive, implicitly assuming that an observer cannot directly observe atomic transitions, and cannot detect if an execution has terminated. The definition can be refined to provide timing or termination sensitivity. Termination sensitivity is achieved by strengthening completeness to require that the correspondence covers all states in both τ and τ', and that no state in τ or τ' corresponds to an infinite set of states. Timing sensitivity is achieved by strengthening the definition so that every state in τ and τ' corresponds to exactly one other state. Note that a timing-sensitive correspondence for two traces is also a termination-sensitive correspondence for those traces: non-termination of a system can be seen as an extreme timing channel.

Definition 2.12 (Termination-sensitive correspondences) *A* termination-sensitive correspondence *R between traces τ and τ' is a correspondence between traces τ and τ' such that*

- *(Finiteness) for all $i < |\tau|$ and $j < |\tau'|$, $\{k \mid (i, k) \in R\}$ and $\{k \mid (k, j) \in R\}$ are finite and non-empty.*

Definition 2.13 (Timing-sensitive correspondences) *A* timing-sensitive correspondence *R between traces τ and τ' is a correspondence between traces τ and τ' such that*

- *(Uniqueness) for all* $i < |\tau|$ *and* $j < |\tau'|$, $\{k \mid (i,k) \in R\}$ *and* $\{k \mid (k,j) \in R\}$ *are of size one.*

Finally, we assume that a system has a set M of *modifiables*, elements of the system that can be directly modified by users, attackers, or other entities. The set of modifiables model the ways that entities can directly interact with the system. Examples include input channels (such as network, keyboard, and mouse) and the physical environment (for example, modifying the thermostat in the server room). The set of modifiables M may intersect with the set of observables O, depending on how the system is modeled. For example, if memory locations can be both observed and altered (a common assumption in language-based modeling), then each memory location may be present in both M and O.

We write $s[m \mapsto v]$ to indicate system state s with modifiable m set to value v. Similar to observables, we assume that each modifiable m has a security policy associated with it, denoted $\mathsf{pol}(m)$. The policy $\mathsf{pol}(m)$ is the security policy that the system should enforce on information stored in m.

2.3.2 Noninterference

Noninterference (Goguen and Meseguer, 1982) is a semantic security condition that requires that high security inputs do not affect low security outputs. We assume that the system's input is available in a single modifiable location $m \in M$, and that the output of the system is the sequence of observations during the subsequent execution of the system.

Noninterference at level ℓ requires that for any trace of the system with some input v_1, there is another trace of the system with input v_2 that looks the same to an observer at level ℓ. This means that the input does not affect the observable output.

Definition 2.14 (Noninterference) *A system S is* (termination-sensitive, timing-sensitive) noninterfering at level ℓ *for input $m \in M$ if for any two values v_1 and v_2, and any state s such that both $s_0 = s[m \mapsto v_1]$ and $s'_0 = s[m \mapsto v_2]$ are feasible, if τ is a trace of S such that $\tau[0] = s_0$, then there is a trace τ' such that $\tau'[0] = s'_0$, and there is a (termination-sensitive, timing-sensitive) correspondence R between τ and τ' such that for all $(i, j) \in R$, $\tau[i] \approx_\ell \tau'[j]$.*

Noninterference is too strong in the presence of declassification, which intentionally makes secret information public. Also noninterference cannot express erasure requirements, which make publicly observable information less observable. Both declassification and erasure change the permitted information flows; noninterference is unable to reason about these changes. Motivated by these shortcomings of noninterference, we define noninterference according to policy.

2.3.3 Noninterference according to policy

Noninterference according to policy (Chong and Myers, 2005) is a semantic security condition that generalizes noninterference. It allows precise reasoning about the observability of information that undergoes declassification and erasure.

Noninterference according to policy is defined in terms of the policy semantics. The intuition behind the policy semantics is that if information in state s has policy p enforced on it, then when the system enters state s', the information (or anything derived or influenced by it) should be observable at level ℓ only if $(s', \ell) \in [\![p]\!]_s$. Noninterference according to policy makes this intuition precise.

Definition 2.15 (Noninterference according to policy) *A system is* (termination-sensitive, timing-sensitive) noninterfering according to policy *for input $m \in M$ if for any two values v_1 and v_2, and any state s such that both $s_0 = s[m \mapsto v_1]$ and*

$s'_0 = s[m \mapsto v_2]$ are feasible, then for any trace τ such that $\tau[0] = s_0$ there is a trace τ' such that $\tau'[0] = s'_0$, and a (termination-sensitive, timing-sensitive) correspondence R for τ and τ' such that for all confidentiality levels $\ell \in \mathcal{L}$, and all $(i, j) \in R$, if $(\tau[i], \ell) \notin [\![\mathsf{pol}(m)]\!]_{s_0}$ or $(\tau'[j], \ell) \notin [\![\mathsf{pol}(m)]\!]_{s'_0}$, then $\tau[i] \approx_\ell \tau'[j]$.

Like noninterference, noninterference according to policy places restrictions on whether information input to the system is observable by an attacker during the execution of the program. However, whereas noninterference required all corresponding states to be observationally equivalent at a fixed level ℓ, noninterference according to policy is more precise, and requires corresponding states to be observationally equivalent at confidentiality levels determined by the semantics of the policy enforced on the input. Thus, noninterference according to policy reflects how the observability of input may change during the execution of the system, as declassifications and erasures occur.

Noninterference according to policy generalizes noninterference. In the absence of declassification or erasure of the input, noninterference according to policy reduces to noninterference. More specifically, if the policy enforced on input m indicates that information will never be observable at a confidentiality level ℓ, then noninterference according to policy for variable m implies noninterference at level ℓ for variable m. For example, a program that is noninterfering according to policy for input m, where $\mathsf{pol}(m) = H$, will never declassify the input to level L, and thus is noninterfering at level L for m. The following theorem states this formally.

Theorem 2.16 *If S is (termination-sensitive, timing-sensitive) noninterfering according to policy for input $m \in M$, then for any confidentiality level $\ell \in \mathcal{L}$ such that $\ell \notin \{\ell' \mid (s', \ell') \in [\![\mathsf{pol}(m)]\!]_s, s \in \Sigma_S\}$, S is (termination-sensitive, timing-sensitive) noninterfering at level ℓ for input m.*

Proof: Let $m \in M$ be fixed. Let $s \in \Sigma_S$, and let v_1 and v_2 be values such that both $s_0 = s[m \mapsto v_1]$ and $s'_0 = s[m \mapsto v_2]$ are feasible. Let τ be a trace such that $\tau[0] = s_0$.

Since S is (termination-sensitive, timing-sensitive) noninterfering according to policy, there is a trace τ' such that $\tau'[0] = s'_0$, and a (termination-sensitive, timing-sensitive) correspondence R for τ and τ' such that for all confidentiality levels $\ell' \in \mathcal{L}$, and all $(i, j) \in R$, if $(\tau[i], \ell') \notin [\![p]\!]_{s_0}$ or $(\tau'[j], \ell') \notin [\![p]\!]_{s'_0}$, then $\tau[i] \approx_{\ell'} \tau'[j]$.

Since $\ell \notin \{\ell' \mid (s', \ell') \in [\![\mathsf{pol}(m)]\!]_{s_0}\}$ and $\ell \notin \{\ell' \mid (s', \ell') \in [\![\mathsf{pol}(m)]\!]_{s'_0}\}$, we must have for all $(i, j) \in R$, $\tau[i] \approx_\ell \tau'[j]$. Thus, S is (termination-sensitive, timing-sensitive) noninterfering at level ℓ for input m. ∎

CHAPTER 3

ENFORCEMENT OF ERASURE AND DECLASSIFICATION

The declassification and erasure policies of Chapter 2 can be enforced in a simple imperative language using a security-type system and simple run-time mechanisms. Any well-typed program in this language satisfies noninterference according to policy.

3.1 The IMP$_E$ language

This section presents IMP$_E$, a simple imperative language that incorporates declassification and erasure policies. The language has run-time mechanisms for erasure and declassification, and a type system to control the flow of information. In Section 3.2, we show that these together suffice to enforce declassification and erasure policies.

3.1.1 Syntax

Figure 3.1 presents the syntax of IMP$_E$. We assume there is a countable set of variables $Vars$. Language expressions include integer literals $n \in \mathbb{Z}$, and variables $x \in Vars$. The metavariable $\oplus$ ranges over total binary operations on integers.

In this chapter, we restrict the conditions of policies in IMP$_E$ to expressions. A condition is satisfied when it evaluates to a non-zero value. For example, if policy $H \searrow^{x+3} L$ is enforced on information, that information may be declassified when expression $x + 3$ is non-zero. The lattice of confidentiality levels remains unspecified.

The commands are standard, with the exception of declassification. The *guarded declassification* command $x :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$

$e ::=$	Expressions
n	Integer literal
x	Variable
$e_0 \oplus e_1$	Binary operation
$c ::=$	Commands
skip	No-op
$x := e$	Assignment
$c_0; c_1$	Sequence
if e **then** c_0 **else** c_1	Selection
while e **do** c	Iteration
$x :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$	Guarded declassification

Figure 3.1: Syntax of IMP$_E$

evaluates expression e, and assigns the result to variable x, provided that expression e_i evaluates to a non-zero value, for all $0 \leq i \leq k$. If there is some e_i that evaluates to zero, then declassification fails. The type system will ensure that policy p_f is an upper bound on information that e may reveal, and that information labeled p_f can safely be relabeled p_t provided all conditions e_i are satisfied.

3.1.2 Operational semantics

A *memory* σ is a map from variables to integers, and is thus a function from $Vars$ to $\mathbb{Z}$. We write $\sigma(e)$ for the result of evaluating expression e using memory σ, that is, using $\sigma(x)$ as the value of each variable x that occurs in e. We write $\sigma[x \mapsto v]$ for the memory that maps variable x to integer v, and otherwise behaves exactly as σ does.

A *configuration* is a pair of a command c and memory σ, written $\langle c, \sigma \rangle$. A configuration fully describes the system state, and thus we take Σ_{IMP_E}, the set of feasible states of IMP$_E$, to be the set of all configurations. Since policy conditions

OS-SKIP

$$\frac{}{\langle \textbf{skip}; c,\ \sigma\rangle \rightarrow \langle c,\ \sigma\rangle}$$

OS-ASSIGN

$$\frac{\sigma' = \mathit{update}(\sigma, x, \sigma(e))}{\langle x := e,\ \sigma\rangle \rightarrow \langle \textbf{skip},\ \sigma'\rangle}$$

OS-SEQUENCE

$$\frac{\langle c_0,\ \sigma\rangle \rightarrow \langle c_0',\ \sigma'\rangle}{\langle c_0; c_1,\ \sigma\rangle \rightarrow \langle c_0'; c_1,\ \sigma'\rangle}$$

OS-IF

$$\frac{i = \begin{cases} 0 & \text{if } \sigma(e) \neq 0 \\ 1 & \text{if } \sigma(e) = 0 \end{cases}}{\langle \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1,\ \sigma\rangle \rightarrow \langle c_i,\ \sigma\rangle}$$

OS-WHILE

$$\frac{}{\langle \textbf{while } e \textbf{ do } c,\ \sigma\rangle \rightarrow \langle \textbf{if } e \textbf{ then } c; \textbf{while } e \textbf{ do } c \textbf{ else skip},\ \sigma\rangle}$$

OS-DECLASSIFY

$$\frac{v = \begin{cases} \sigma(e) & \text{if } \forall i \in 0..k.\ \sigma(e_i) \neq 0 \\ 0 & \text{if } \exists i \in 0..k.\ \sigma(e_i) = 0 \end{cases} \qquad \sigma' = \mathit{update}(\sigma, x, v)}{\langle x := \textbf{declassify}(e, p_f \textbf{ to } p_t \textbf{ using } e_0, \ldots, e_k),\ \sigma\rangle \rightarrow \langle \textbf{skip},\ \sigma'\rangle}$$

Figure 3.2: Operational semantics of IMP_E

$$\mathit{update}(\sigma, x, v) = \begin{cases} \mathit{erasure}(\sigma) & \text{if } \mathsf{reqErase}(\Gamma(x), \sigma) \\ \mathit{erasure}(\sigma[x \mapsto v]) & \text{otherwise} \end{cases}$$

$$\mathit{erasure}(\sigma) = \bigsqcup_{i \in \omega} \sigma_i$$

where $\sigma_0 = \sigma$, and

$$\sigma_{i+1} = \lambda x \in \mathit{Vars}.\ \begin{cases} 0 & \text{if } \mathsf{reqErase}(\Gamma(x), \sigma_i) \\ \sigma_i(x) & \text{otherwise} \end{cases}$$

and $\bigsqcup_{i \in \omega} \sigma_i$ denotes the least upper bound of the chain $\sigma_0\sigma_1\sigma_2\ldots$ under the ordering $\sqsubseteq$, where
$\sigma' \sqsubseteq \sigma'' \triangleq \forall x \in \mathit{Vars}.\ \sigma'(x) = \sigma''(x) \vee \sigma''(x) = 0$

Figure 3.3: $\mathit{update}(\sigma, x, v)$ and $\mathit{erasure}(\sigma)$

are expressions, the satisfaction of a condition depends only on the memory of the current configuration. For brevity, we thus write $\mathsf{reqErase}(p, \sigma)$ instead of $\mathsf{reqErase}(p, \langle c, \sigma \rangle)$.

We assume there is a *typing context* that indicates what policy should be enforced on information stored in each variable. A typing context Γ is a function from $Vars$ to policies, and $\Gamma(x)$ is the policy that must be enforced on information stored in variable x. The typing context does not change during execution: a variable x always has the same policy $\Gamma(x)$ enforced on it.

Figure 3.2 presents the operational semantics for IMP_E, showing how configurations are updated as commands execute. The enforcement of policies relies on two run-time mechanisms, embodied in the operational semantics. The first is run-time overwriting of variables to enforce erasure; the second is run-time checking of conditions for declassification. Except for these two mechanisms, the operational semantics of the language are standard.

Overwriting variables. IMP_E enforces erasure by setting the contents of a variable to zero whenever the policy for the variable requires information erasure. Policy p requires information erasure when $\mathsf{reqErase}(p, \sigma)$ holds, where σ is the current memory. For example, policies $L \overset{x \geq 0}{\nearrow} H$ and $(L \overset{x=3}{\nearrow} H) \overset{y}{\searrow} L$ both require information erasure if $\sigma(x) = 3$. Since conditions are expressions, a condition may become satisfied when the memory is updated. The operational semantics for commands that update memory (assignment and declassification) use the utility function $update(\sigma, x, v)$ to overwrite variables, defined in Figure 3.3. The function $update(\sigma, x, v)$ takes memory σ, variable x, and integer v, and, provided policy $\Gamma(x)$ does not require erasure, returns $erasure(\sigma[x \mapsto v])$. The utility function $erasure(\sigma)$ checks for each variable y if policy $\Gamma(y)$ requires erasure given

the memory σ; if so, it overwrites variable y with the value zero. Overwriting y changes the memory, and thus may trigger the overwriting of other variables.

The function $erasure(\sigma)$ is defined for all memories σ, and it provably overwrites variables as required: if memory $\sigma' = erasure(\sigma)$ then for all variables x, $\mathsf{reqErase}(\Gamma(x), \sigma')$ implies $\sigma'(x) = 0$.

Theorem 3.1 *For all memories σ, the memory $\sigma' = erasure(\sigma)$ is defined. Moreover, for all variables $x \in Vars$, if $\mathsf{reqErase}(\Gamma(x), \sigma')$ then $\sigma'(x) = 0$.*

Proof: The chain $\sigma_0\sigma_1\ldots$ is nondecreasing under the ordering $\sqsubseteq$, and has the least upper bound

$$\sigma' = \lambda x \in Vars. \begin{cases} 0 & \text{if } \exists i \in \omega.\ \mathsf{reqErase}(\Gamma(x), \sigma_i) \\ \sigma(x) & \text{otherwise.} \end{cases}$$

If for some $x \in Vars$ we have $\mathsf{reqErase}(\Gamma(x), \sigma')$ then there is some finite set X of variables that occur in conditions of policy $\Gamma(x)$. Because the chain is nondecreasing, and from the definition of $\sqsubseteq$, there must be some $i \in \omega$ such that for all $j \geq i$, and all $y \in X$, $\sigma_i(y) = \sigma_j(y) = \sigma'(y)$. Thus, $\mathsf{reqErase}(\Gamma(x), \sigma_i)$, and so, since σ' is the least upper bound of the chain, $\sigma'(x) = 0$. ∎

Run-time mechanism for declassification. Declassification of information can occur only when appropriate conditions are satisfied. For example, the declassification policy $H \searrow^{x>0} L$ allows information to be declassified to L when the expression $x > 0$ is non-zero, that is, when x is positive. The operational semantics for a guarded declassification command, x := **declassify**(e, p_f **to** p_t **using** $e_0, \ldots, e_k$), evaluates e and assigns the result to variable x provided the expressions $e_0, \ldots, e_k$ all evaluate to non-zero values. If one or more expressions e_i evaluate to zero, then declassification fails, and variable x is updated with the

constant value zero. (Other reasonable semantics include leaving the value of x unchanged, or stopping execution.)

The use of run-time mechanisms to aid in the enforcement of declassification and erasure policies allows simpler static enforcement mechanisms. The policies can be enforced without these run-time mechanisms, but would require either more complex static enforcement, or less expressive conditions. See Chapter 6 for more discussion on this trade-off.

3.1.3 Type system

The run-time mechanisms of IMP$_E$ ensure that declassification only occurs if appropriate conditions are satisfied, and that variables are overwritten when their policies require erasure. However, the run-time mechanisms alone are not sufficient to ensure that erasure and declassification policies are enforced. What prevents information with erasure policy $L \overset{c}{\nearrow} H$ from being stored in a variable x that has policy L enforced on it? Information in variable x has low security enforced on it, and is not necessarily overwritten when condition c is satisfied. Similarly, what prevents information with policy H from being stored in a variable with policy $H \overset{c}{\searrow} L$ enforced on it, and subsequently (and incorrectly) declassified?

The type system of IMP$_E$ restricts information flow within a program, ensuring that appropriate policies are enforced on information at all times. The type system restricts both explicit flows, where information flows from direct assignments to variables, and implicit flows (Denning and Denning, 1977), where information flows via the program's control structure. The type system does not restrict timing or termination channels.

T-SKIP

$$\frac{\Gamma \vdash pc\ \mathsf{pol}}{pc, \Gamma \vdash \mathbf{skip}\ \mathsf{com}}$$

T-ASSIGN

$$\frac{\Gamma \vdash e\ :\ p_e\ \mathsf{exp} \quad \vdash pc \leq \Gamma(x) \quad \vdash p_e \leq \Gamma(x) \quad \Gamma \vdash pc\ \mathsf{pol}}{pc, \Gamma \vdash x := e\ \mathsf{com}}$$

T-SEQUENCE

$$\frac{pc, \Gamma \vdash c_0\ \mathsf{com} \quad pc, \Gamma \vdash c_1\ \mathsf{com}}{pc, \Gamma \vdash c_0; c_1\ \mathsf{com}}$$

T-WHILE

$$\frac{\Gamma \vdash e\ :\ p_e\ \mathsf{exp} \quad pc', \Gamma \vdash c\ \mathsf{com} \quad \Gamma \vdash pc\ \mathsf{pol} \quad \vdash pc \leq pc' \quad \vdash p_e \leq pc'}{pc, \Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathsf{com}}$$

T-IF

$$\frac{\Gamma \vdash e\ :\ p_e\ \mathsf{exp} \quad pc', \Gamma \vdash c_0\ \mathsf{com} \quad pc', \Gamma \vdash c_1\ \mathsf{com} \quad \Gamma \vdash pc\ \mathsf{pol} \quad \vdash pc \leq pc' \quad \vdash p_e \leq pc'}{pc, \Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\ \mathsf{com}}$$

T-DECLASSIFY

$$\frac{\Gamma \vdash e\ :\ p_f\ \mathsf{exp} \quad \vdash pc \leq \Gamma(x) \quad \vdash p_t \leq \Gamma(x) \quad \Gamma \vdash pc\ \mathsf{pol} \quad \forall i \in 0..k.\ \Gamma \vdash e_i\ :\ \Gamma(x)\ \mathsf{exp} \quad e_0, \ldots, e_k \vdash p_f \leq p_t}{pc, \Gamma \vdash x := \mathbf{declassify}(e, p_f\ \mathbf{to}\ p_t\ \mathbf{using}\ e_0, \ldots, e_k)\ \mathsf{com}}$$

T-VAL

$$\frac{}{\Gamma \vdash n\ :\ p\ \mathsf{exp}}$$

T-VAR

$$\frac{\vdash \Gamma(x) \leq p}{\Gamma \vdash x\ :\ p\ \mathsf{exp}}$$

T-OP

$$\frac{\Gamma \vdash e_0\ :\ p_0\ \mathsf{exp} \quad \Gamma \vdash e_1\ :\ p_1\ \mathsf{exp} \quad \vdash p_0 \leq p \quad \vdash p_1 \leq p}{\Gamma \vdash e_0 \oplus e_1\ :\ p\ \mathsf{exp}}$$

T-POL

$$\frac{\forall e \in \mathsf{eraseConds}(p).\ \Gamma \vdash e\ :\ p\ \mathsf{exp}}{\Gamma \vdash p\ \mathsf{pol}}$$

$$\begin{aligned} \mathsf{eraseConds}(\ell) &\triangleq \emptyset \\ \mathsf{eraseConds}(p \searrow^{c} q) &\triangleq \mathsf{eraseConds}(p) \\ \mathsf{eraseConds}(p \nearrow^{c} q) &\triangleq \{c\} \cup \mathsf{eraseConds}(p) \end{aligned}$$

Figure 3.4: Typing rules for IMP_E

The typing judgment $pc, \Gamma \vdash c$ com means that command c is well-typed under typing context Γ and program counter policy pc. The program counter policy is used to restrict implicit flows. It is an upper bound on the policies of information that may have influenced the value of the program counter, so it is an upper bound on the information that may be gained by knowing that command c is executed. The typing judgment $\Gamma \vdash e : p$ exp means that under typing context Γ, policy p is an upper bound on the policies of information that may be gained by evaluating expression e.

Figure 3.4 presents inference rules for these typing judgments. The rules track and restrict the flow of information within a program. For example, the rule T-ASSIGN for an assignment $x := e$ ensures that information that may be revealed by evaluating expression e is allowed to flow to variable x ($p_e \leq \Gamma(x)$), and that information that may be revealed by learning the assignment is executed is also allowed to flow to variable x ($pc \leq \Gamma(x)$).

All the inference rules for the judgments $pc, \Gamma \vdash c$ com and $\Gamma \vdash e : p$ exp are standard for information-flow security type systems, with the exception of the rule for guarded declassification, T-DECLASSIFY. A guarded declassification command $x :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$ declassifies information with policy p_f to policy p_t. Rule T-DECLASSIFY requires that p_f can be relabeled p_t for any memory σ in which all conditions $e_0, \ldots, e_k$ are satisfied ($e_0, \ldots, e_k \vdash p_f \leq p_t$). The typing rule also requires that the declassified information is allowed to be stored in x ($p_t \leq \Gamma(x)$), that the information gained by knowing the declassification occurred can flow to x ($pc \leq \Gamma(x)$), and that the information gained by evaluating e is bounded above by policy p_f ($\Gamma \vdash e : p_f$ exp).

There is a flow of information from the conditions $e_0, \ldots, e_k$ to the variable x. The operational semantics for a guarded declassification will assign the result of

evaluating e into x only if all conditions $e_0, \ldots, e_k$ evaluate to non-zero values. Thus, the value of the variable x after the declassification command may reveal information about the value of the conditions. The typing rule for declassification, T-DECLASSIFY, tracks this information flow by requiring $\Gamma(x)$ to be an upper bound on the information that may be gained by knowing if condition e_i was satisfied ($\Gamma \vdash e_i : \Gamma(x)$ exp).

Well-formed contexts

A variable x is overwritten when $\Gamma(x)$, the policy enforced on x, requires erasure. Thus, if satisfaction of condition e can cause policy $\Gamma(x)$ to require erasure, there is information flow from e to x. For example, if $\Gamma(x) = L \not\nearrow^{y>0} H$, then the value of variable y affects the value of variable x, and an observer who learns that x is not erased learns information about y, to wit, that $y \leq 0$. To track and control this information flow, we restrict the typing contexts that may be used.

For all variables x, we require that policy $\Gamma(x)$ is well-typed, written $\Gamma \vdash \Gamma(x)$ pol. Any policy pc that is used as a program counter policy in the proof of a typing judgment $pc, \Gamma \vdash c$ com must also be well-typed. The inference rule for well-typed policies, T-POL, is given in Figure 3.4. It requires that if condition e may cause policy p to require erasure, then p is an upper bound on the information that may be obtained by evaluating e, which is expressed by the premise $\Gamma \vdash e : p$ exp.

The recursively defined function eraseConds(p) returns the set of expressions that may cause policy p to require erasure. That is, reqErase(p, σ) if and only if there is some condition $e \in$ eraseConds(p) such that $\sigma(e) \neq 0$. We define the *overwrite dependency* relation $\prec_\Gamma$ over variables such that $x \prec_\Gamma y$ if changing the

```
1  if ( userReqExit ) then
2    appEnd = 1; exit()
3  else
4    // get user's symptoms
5    symp := getUserSymptoms();
6    ...
7    // diagnosis
8    if (contains(symp, 'fever') &&
9        contains(symp, 'malaise') && ...) then
10     diag := 'Influenza'
11   else if ...
```

$$\Gamma(\texttt{symp}) = \mathit{session}\ {}^{\texttt{appEnd}}\!\not\nearrow \top \qquad \Gamma(\texttt{appEnd}) = \mathit{session}$$
$$\Gamma(\texttt{diag}) = \mathit{session}\ {}^{\texttt{appEnd}}\!\not\nearrow \top \qquad \Gamma(\texttt{userReqExit}) = \mathit{session}$$

Figure 3.5: Medical information website example

value of x may cause policy $\Gamma(y)$ to require erasure. More formally, $x \prec_\Gamma y$ if there is an expression e such that $e \in \mathsf{eraseConds}(\Gamma(y))$ and x appears in e.

To make it easier to track information flows that occur due to overwriting, and to simplify security proofs, we require the overwrite dependency relation to be well-founded. This prevents infinite chains of distinct variables $x_0, x_1, \ldots$, such that the overwriting of variable x_i depends on the value of variable x_{i+1}. Well-foundedness of the overwrite dependency relation also prevents recursively defined policies. For example, if $\Gamma(x) = L\ {}^{x=0}\!\not\nearrow H$ then $x \in \mathsf{eraseConds}(\Gamma(x))$ and $x \prec_\Gamma x$.

Well-formed contexts are exactly the typing contexts that have a well-founded overwrite dependency relation and contain only well-typed policies.

Definition 3.2 (Well-formed typing context) *Typing context* Γ *is* well-formed *if the overwrite dependency relation* $\prec_\Gamma$ *is well-founded and for all* $x \in \mathit{Vars}$, $\Gamma \vdash \Gamma(x)$ pol.

3.1.4 Example

Figure 3.5 shows a fragment of IMP_E code that could be used to process a client request to the medical information website described in Example 1.8, and elaborated in Example 2.4. For ease of presentation, we assume the existence of functions and strings.

The code first checks if the user has requested to exit the diagnosis application, and if so, sets variable `appEnd` and exits. Otherwise, the code gets the user's symptoms and uses them to produce a diagnosis, which would then be displayed to the user. Modulo the use of strings and functions, the code is well-typed, and the relevant parts of the typing context Γ are also shown in Figure 3.5.

The policy enforced on the user symptoms, $\Gamma(\texttt{symp})$, is $\mathit{session} \overset{\texttt{appEnd}}{\nearrow} \top$. As described in Example 2.4, $\mathit{session}$ is a confidentiality level allowing only the session client and server to read the information, and $\top$ is a confidentiality level so restrictive that it prevents the server from storing the information. There is an implicit flow of information from `symp` to `diag`, as `symp` is used in the conditional test on lines 8–9, and `diag` is assigned to in the body of the conditional. By typing rule T-IF, the program counter policy for the conditional's body must be at least as restrictive as $\Gamma(\texttt{symp})$. Similarly, by rule T-ASSIGN, $\Gamma(\texttt{diag})$ must be as restrictive as that program counter policy. These constraints are satisfied by using policy $\Gamma(\texttt{symp})$ as the program counter policy for the body of the conditional, since $\Gamma(\texttt{symp}) = \Gamma(\texttt{diag})$.

The value of variable `appEnd` can cause policy $\mathit{session} \overset{\texttt{appEnd}}{\nearrow} \top$ to require erasure. Indeed, when variable `appEnd` is set (line 2), variables `symp` and `diag` are overwritten. There is thus information flow from `appEnd` to `symp` and `diag`. The requirement for a well-formed typing context tracks this flow, and requires that

$\vdash \Gamma(\texttt{appEnd}) \leq \Gamma(\texttt{symp})$ and $\vdash \Gamma(\texttt{appEnd}) \leq \Gamma(\texttt{diag})$, which are satisfied, as

$$\Gamma(\texttt{appEnd}) = \mathit{session},$$
$$\Gamma(\texttt{symp}) = \Gamma(\texttt{diag}) = \mathit{session} \overset{\texttt{appEnd}}{\nearrow} \top,$$

and

$$\vdash \mathit{session} \leq \mathit{session} \overset{\texttt{appEnd}}{\nearrow} \top.$$

3.2 Noninterference according to policy

The central result of this chapter is that the type system and runtime mechanisms of IMP$_E$ suffice to enforce erasure and declassification policies. Thus, any well-typed IMP$_E$ program is noninterfering according to policy.

In order to state and prove this result, we first instantiate the observational model (Section 2.3.1) and refine the general definition of noninterference according to policy (Definition 2.15) for IMP$_E$ commands.

We assume that the input to the system is given in a variable, and an attacker can observe some but not necessarily all memory locations. Thus, the set of observables O and the set of modifiables M are equal to the set of variables Vars. The policy enforced on a variable x is given by the typing context Γ, so $\mathsf{pol}(x) = \Gamma(x)$. Thus, configurations $\langle c, \sigma\rangle$ and $\langle c', \sigma'\rangle$ are observationally equivalent at level ℓ if for all variables $x \in \mathit{Vars}$, if $\mathsf{obs}(\Gamma(x)) \sqsubseteq \ell$ then $\sigma(x) = \sigma'(x)$.

Definition 3.3 (Noninterference according to policy) *A command c is* noninterfering according to policy for variable x *if for all integers $v_1, v_2 \in \mathbb{Z}$, all memories σ, memories $\sigma_1 = \mathit{update}(\sigma, x, v_1)$ and $\sigma_2 = \mathit{update}(\sigma, x, v_2)$, and all traces τ_1 and τ_2 such that $\tau_i[0] = \langle c, \sigma_i\rangle$ for $i \in \{1, 2\}$, there exists a correspondence R for τ_1 and τ_2 such that*

for all $(i, j) \in R$*, for all* $\ell \in \mathcal{L}$*, if* $(\tau_1[i], \ell) \notin [\![\Gamma(x)]\!]_{\langle c, \sigma_1 \rangle}$ *and* $(\tau_2[j], \ell) \notin [\![\Gamma(x)]\!]_{\langle c, \sigma_2 \rangle}$*, then* $\tau_1[i] \approx_\ell \tau_2[j]$.

Theorem 3.4 *For all typing contexts* Γ *and commands* c*, if* Γ *is well-formed, and* $pc, \Gamma \vdash c$ com *for some policy* pc*, then for all variables* $x \in Vars$*,* c *is noninterfering according to policy for variable* x.

The proof of Theorem 3.4 uses the technique of Pottier and Simonet (2002) for showing noninterference in the ML programming language. The key concept of their technique is to define a new language that can represent two executions of a program with different inputs, and reduces the proof of noninterference to demonstrating type-soundness of the new language. We present the syntax and semantics of the language IMP_E^2, show that it is adequate to represent evaluation of two IMP_E programs, and show that type preservation in IMP_E^2 implies Theorem 3.4.

3.2.1 Syntax and semantics of IMP_E^2

The language IMP_E^2 extends IMP_E with pair constructs for commands $(\!| c_1 \mid c_2 |\!)$, and integers $(\!| v_1 \mid v_2 |\!)$. The pair constructs represent different commands and integers that may arise in two different executions of an IMP_E program. This allows a single execution of an IMP_E^2 program to represent the two different executions of an IMP_E program. A command pair cannot be nested inside another command pair, but can otherwise appear nested at arbitrary depth. Integer pairs are used to track how memories differ in different executions of a program: memories in IMP_E^2 are functions from variables to integers and integer pairs. Figure 3.6 shows the extended syntax of IMP_E^2.

For an extended command c, let the projection functions $\lfloor c \rfloor_1$ and $\lfloor c \rfloor_2$ represent the two IMP_E commands that c encodes. The projection functions satisfy

$c ::=$	Commands
$\ldots$	IMP$_E$ commands
$(\!\mid c_1 \mid c_2 \mid\!)$	Pair command

Figure 3.6: Syntax of IMP$_E^2$

$\lfloor (\!\mid c_1 \mid c_2 \mid\!) \rfloor_i = c_i$, and are homomorphisms on other commands. Similarly for integer pairs, $\lfloor (\!\mid v_1 \mid v_2 \mid\!) \rfloor_i = v_i$. The projection functions are extended to memories, so that

$$\lfloor \sigma \rfloor_i(x) = \begin{cases} v & \text{if } \sigma(x) = v \\ v_i & \text{if } \sigma(x) = (\!\mid v_1 \mid v_2 \mid\!) \end{cases}$$

The evaluation of expressions are also extended, so that binary operations $\oplus$ are homomorphic on integer pairs. Thus, the evaluation of an expression e in a memory σ may be either an integer v or an integer pair $(\!\mid v_1 \mid v_2 \mid\!)$.

We extend configurations to triples $\langle c,\, \sigma \rangle_i$ for an index $i \in \{\bullet, 1, 2\}$. The index indicates if the command c and memory σ represent a pair of configurations ($\bullet$), or the left (1) or right (2) side of a pair of configurations. A configuration $\langle c,\, \sigma \rangle_i$ is well formed if $i \in \{1, 2\}$ implies that c does not contain any command pairs, and the image of σ does not contain any integer pairs.

The operational semantics of IMP$_E^2$ are given in Figure 3.7, and extend the operational semantics of IMP$_E$. The rule OS-PAIR-LIFT allows the evaluation of either element of a command pair $(\!\mid c_1 \mid c_2 \mid\!)$. The rule OS-PAIR-SKIP removes a command pair when both elements of the pair have finished execution. The rule OS-PAIR-IF is used when the conditional of an **if** command evaluates to different values in the two executions, and as a result, a command pair is introduced, representing the different commands that each execution will evaluate. Note that this is the only way in which a command pair can be introduced into a configuration. For succinctness, this rule uses a ternary expression, $(v_i \neq 0)?c_0{:}c_1$,

OS-PAIR-LIFT

$$\frac{\begin{array}{c}\langle c_i,\, \lfloor\sigma\rfloor_i\rangle_i \to \langle c_i',\, \sigma_i'\rangle_i \\ \{i,j\} = \{1,2\} \qquad c_j' = c_j \qquad \sigma_j' = \lfloor\sigma\rfloor_j \\ \sigma' = \lambda x.\ \begin{cases} 0 & \text{if } \mathsf{reqErase}(\Gamma(x), \lfloor\sigma\rfloor_1) \text{ and} \\ & \quad \mathsf{reqErase}(\Gamma(x), \lfloor\sigma\rfloor_2) \\ (\!|\, \sigma_1'(x) \,|\, \sigma_2'(x) \,|\!) & \text{if } \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ \sigma(x) & \text{otherwise} \end{cases}\end{array}}{\langle (\!|\, c_1 \,|\, c_2 \,|\!),\, \sigma\rangle_\bullet \to \langle (\!|\, c_1' \,|\, c_2' \,|\!),\, \sigma'\rangle_\bullet}$$

OS-PAIR-SKIP

$$\frac{}{\langle (\!|\, \mathbf{skip} \,|\, \mathbf{skip} \,|\!),\, \sigma\rangle_\bullet \to \langle \mathbf{skip},\, \sigma\rangle_\bullet}$$

OS-PAIR-IF

$$\frac{\begin{array}{c}\sigma(e) = (\!|\, v_1 \,|\, v_2 \,|\!) \\ c_i' = (v_i \neq 0)\ ?\ c_0 : c_1\end{array}}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1,\, \sigma\rangle_\bullet \to \langle (\!|\, c_1' \,|\, c_2' \,|\!),\, \sigma\rangle_\bullet}$$

OS-PAIR-DECLASSIFY

$$\frac{\begin{array}{c}\sigma(e_0 \times \cdots \times e_k) = (\!|\, v_1 \,|\, v_2 \,|\!) \\ v_i' = (v_i \neq 0)\ ?\ \lfloor\sigma(e)\rfloor_i : 0 \\ \sigma' = \mathit{update}^2(\sigma, x, (\!|\, v_1' \,|\, v_2' \,|\!))\end{array}}{\langle x := \mathbf{declassify}(e, L_f\ \mathbf{to}\ L_t\ \mathbf{using}\ e_0, \ldots, e_k),\, \sigma\rangle_\bullet \to \langle \mathbf{skip},\, \sigma'\rangle_\bullet}$$

$$\mathit{update}^2(\sigma, y, w) = \lambda x.\ \begin{cases} 0 & \text{if } \forall i \in \{1,2\}.\ \mathsf{reqErase}(\Gamma(x), \sigma_i') \\ (\!|\, \sigma_1'(x) \,|\, \sigma_2'(x) \,|\!) & \text{if } \exists i \in \{1,2\}.\ \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ & \quad \text{and } \exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(x), \sigma_i') \\ \sigma(x) & \text{otherwise} \end{cases}$$

where $\sigma_i' = \mathit{update}(\lfloor\sigma\rfloor_i, y, \lfloor w\rfloor_i)$

$$\mathit{erasure}^2(\sigma) = \lambda x.\ \begin{cases} 0 & \text{if } \forall i \in \{1,2\}.\ \mathsf{reqErase}(\Gamma(x), \sigma_i') \\ (\!|\, \sigma_1'(x) \,|\, \sigma_2'(x) \,|\!) & \text{if } \exists i \in \{1,2\}.\ \lfloor\sigma\rfloor_i(x) \neq \sigma_i'(x) \\ & \quad \text{and } \exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(x), \sigma_i') \\ \sigma(x) & \text{otherwise} \end{cases}$$

where $\sigma_i' = \mathit{erasure}(\lfloor\sigma\rfloor_i)$

Figure 3.7: Operational semantics of IMP_E^2

which is equal to c_0 if the predicate $v_i \neq 0$ is true, and to c_1 otherwise. The rule OS-PAIR-DECLASSIFY is used when the evaluation of conditions for a declassification differ in the two executions. The rule uses the evaluation of the product of the conditions, $e_0 \times \cdots \times e_k$, since this product will be zero if and only if there is some e_i that evaluates to zero.

The rules for IMP$_E$, given in Figure 3.2, are adapted by indexing each configuration with i to become rules for IMP$_E^2$. We write a premise of the form $\sigma(e) \neq 0$ to mean there is an integer v (not an integer pair) such that $\sigma(e) = v$ and $v \neq 0$. The utility functions $update(\cdot, \cdot, \cdot)$ and $erasure(\cdot)$ are adapted for IMP$_E^2$; the new versions, $update^2(\cdot, \cdot, \cdot)$ and $erasure^2(\cdot)$, are presented in Figure 3.2. For the adapted IMP$_E$ rules, the version of the function to use depends upon the configuration index: the IMP$_E^2$ versions if the index is $\bullet$; the IMP$_E$ versions otherwise.

3.2.2 Adequacy of IMP$_E^2$

The language IMP$_E^2$ is adequate for reasoning about the execution of two IMP$_E$ programs. We show that execution of a IMP$_E^2$ program is sound (a step taken by a IMP$_E^2$ program corresponds to one or zero steps taken by its projections), and complete (given two IMP$_E$ executions, there is a IMP$_E^2$ execution whose projection agrees with at least one of them). We use $\rightarrow^=$ to denote the reflexive closure of the relation $\rightarrow$.

Lemma 3.5 (Soundness) *If* $\langle c, \sigma\rangle_\bullet \rightarrow \langle c', \sigma'\rangle_\bullet$, *then* $\langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i\rangle \rightarrow^= \langle \lfloor c' \rfloor_i, \lfloor \sigma' \rfloor_i\rangle$ *for* $i \in \{1, 2\}$.

Proof: By induction on the derivation $\langle c, \sigma\rangle_\bullet \rightarrow \langle c', \sigma'\rangle_\bullet$. The interesting cases are the new rules introduced for IMP$_E^2$: OS-PAIR-LIFT, OS-PAIR-SKIP, OS-PAIR-IF, and OS-PAIR-DECLASSIFY. For a reduction using OS-PAIR-LIFT, only one of

the two projections takes a step, while the other projection remains unchanged. For OS-PAIR-SKIP, both projections remain unchanged. For both OS-PAIR-IF, and OS-PAIR-DECLASSIFY, both projections take a step. ∎

Lemma 3.6 (Stuck configurations) *If* $\langle c, \sigma\rangle_\bullet$ *is stuck (i.e., cannot be reduced and* $c \neq$ **skip***), then* $\langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle$ *is stuck for some* $i \in \{1, 2\}$.

Proof: By structural induction on command c. ∎

Lemma 3.7 (Completeness) *If* $\langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle \rightarrow^* \langle c_i', \sigma_i' \rangle$ *for* $i \in \{1, 2\}$*, then there exists a* IMP_E^2 *configuration* $\langle c', \sigma' \rangle_\bullet$ *such that* $\langle c, \sigma\rangle_\bullet \rightarrow^* \langle c', \sigma'\rangle_\bullet$ *and* $\langle \lfloor c' \rfloor_i, \lfloor \sigma' \rfloor_i \rangle = \langle c_i', \sigma_i' \rangle$ *for some* $i \in \{1, 2\}$.

Proof: Let $\tau_i = \langle \lfloor c \rfloor_i, \lfloor \sigma \rfloor_i \rangle \ldots \langle c_i', \sigma_i' \rangle$. Let n_i be the length of τ_i. For a IMP_E^2 trace $\tau = \langle c, \sigma\rangle_\bullet \ldots \langle c', \sigma'\rangle_\bullet$, let $f_i(\tau)$ be n_i minus the number of reduction steps in τ that reduce the ith projection. Note that $f_i(\tau)$ is non-negative. Consider $g(\tau) = \min(f_1(\tau), f_2(\tau))$. If $g(\tau) = 0$, then τ is a trace that satisfies the conditions.

Suppose $g(\tau) > 0$. Consider the function

$$h(\tau) = (g(\tau),\ |f_1(\tau) - f_2(\tau)|,\ numPairs(\tau[|\tau| - 1]))$$

where $numPairs(\langle c, \sigma\rangle_\bullet)$ returns the number of pair commands in c. Note that all elements of the triple returned by $h(\tau)$ are non-negative. If we can extend τ by one step to a trace τ' such that $h(\tau') < h(\tau)$ under lexicographic ordering, then, by repeated applications, eventually we will produce a trace τ'' such that $g(\tau'') = 0$.

We now show how to extend τ to a trace τ' such that $h(\tau') < h(\tau)$. By assumption, $g(\tau) > 0$, so neither τ_1 or τ_2 is stuck. By Lemma 3.6, we can extend τ by one more step, producing trace τ'. By Lemma 3.5, either $f_i(\tau') = f_i(\tau) - 1$ for

T-PAIR

$$\frac{\begin{array}{cc} pc \sqsubseteq pc' & \mathit{protected}(pc', \tau) \\ \neg\mathsf{reqErase}(pc', \lfloor \tau \rfloor_1) & pc', \Gamma \vdash c_1\ \mathsf{com} \\ \neg\mathsf{reqErase}(pc', \lfloor \tau \rfloor_2) & pc', \Gamma \vdash c_2\ \mathsf{com} \end{array}}{\tau, pc, \Gamma \vdash (\!|\, c_1 \mid c_2 \,|\!)\ \mathsf{com}}$$

T-CONFIG

$$\frac{\begin{array}{c} \tau, pc, \Gamma \vdash c\ \mathsf{com} \\ \forall x \in \mathit{Vars}.\ (\sigma(x) = (\!|\, v_1 \mid v_2 \,|\!)) \Rightarrow \mathit{protected}(\Gamma(x), \tau) \end{array}}{\tau, pc, \Gamma \vdash \langle c,\ \sigma \rangle_\bullet\ \mathsf{config}}$$

Figure 3.8: Typing rules for IMP_E^2

some $i \in \{1, 2\}$, or $f_i(\tau') = f_i(\tau)$ for all $i \in \{1, 2\}$. If the former, then $h(\tau') < h(\tau)$. If the latter, then the rule OS-PAIR-SKIP was used in the reduction, and the last configuration of τ' has one fewer pair command than the last configuration of τ, so $h(\tau') < h(\tau)$. ∎

3.2.3 Type preservation of IMP_E^2

We extend the type system of IMP_E to type IMP_E^2 commands and configurations. The extended type system will allow us to show that type preservation of an IMP_E^2 program implies noninterference according to policy for an IMP_E program.

The typing judgment for commands is now of the form $\tau, pc, \Gamma \vdash c\ \mathsf{com}$, where τ is an execution trace. If $\tau, pc, \Gamma \vdash c\ \mathsf{com}$, then command c is well-typed with typing context Γ and program counter policy pc at the program point when trace τ has been produced. Typing rules for IMP_E (given in Figure 3.4) are made typing rules for IMP_E^2 by adding the additional typing parameter τ to each rule. Similarly, the judgment $\tau, pc, \Gamma \vdash \langle c,\ \sigma \rangle_\bullet\ \mathsf{config}$ means that configuration $\langle c,\ \sigma \rangle_\bullet$ is well-typed with typing context Γ and program counter policy pc at the program point when trace τ has been produced.

$$\frac{}{p \leq_{\langle c,\sigma\rangle} p} \qquad \frac{\tau = \tau'\langle c, \sigma\rangle \quad \neg\mathsf{reqErase}(p', \tau') \quad p \leq_{\tau'} p' \quad [\![p']\!]_\sigma \supseteq [\![q]\!]_\sigma}{p \leq_\tau q}$$

Figure 3.9: Inference rules for $p \leq_\tau q$

The two new typing rules, shown in Figure 3.8, make use of the predicate $protected(p, \tau)$. Informally, if for policy p the predicate $protected(p, \tau)$ is true, then the program input may have flowed through the program, and now be labeled with the policy p. Thus, this predicate depends on the execution trace τ. Note that the premises for the typing rule for pairs, T-PAIR, uses the typing judgments for IMP$_E$, i.e., without the trace τ. This is because well-formed commands do not have nested command pairs.

To formalize how program input may be relabeled with different policies, we use the *extended relabeling* relation $p \leq_\tau q$. For policies p and q and finite trace τ, if $p \leq_\tau q$, then input in $\tau[0]$, the initial configuration of trace τ, labeled with policy p can influence information labeled with policy q in final configuration of τ. Inference rules for this relation are given in Figure 3.9.

More formally, we define $protected(p, \tau)$ as

$$\begin{aligned} protected(p, \tau) \triangleq\ & (\neg\mathsf{reqErase}(p, \lfloor\tau\rfloor_1) \vee \\ & \neg\mathsf{reqErase}(p, \lfloor\tau\rfloor_2)) \wedge \\ & \forall i \in \{1, 2\}.\ \neg\mathsf{reqErase}(p, \lfloor\tau\rfloor_i) \Rightarrow \\ & \Gamma(x) \leq_{\lfloor\tau\rfloor_i} p \end{aligned}$$

where x is the variable in which program input is placed.

The extended relabeling relation has a nice property with respect to the semantics of policies. If $\tau = \langle c, \sigma\rangle \ldots \langle c', \sigma'\rangle$ and $p \leq_\tau q$ then the semantics of q in $\langle c', \sigma'\rangle$ are a subset of the semantics of p in $\langle c, \sigma\rangle$. We prove this using the following lemma.

Lemma 3.8 *If* $\langle c,\, \sigma\rangle \rightarrow \langle c',\, \sigma'\rangle$ *and* $\neg\mathsf{reqErase}(p, \sigma)$ *then* $[\![p]\!]_{\langle c',\, \sigma'\rangle} \subseteq [\![p]\!]_{\langle c,\, \sigma\rangle}$.

Proof: By induction on the structure of p. ∎

Property 3.9 *If* $\tau = \langle c,\, \sigma\rangle \ldots \langle c',\, \sigma'\rangle$ *and* $p \leq_\tau q$ *then* $[\![q]\!]_{\langle c',\, \sigma'\rangle} \subseteq [\![p]\!]_{\langle c,\, \sigma\rangle}$.

Proof: By induction on the derivation of $p \leq_\tau q$, using Lemma 3.8. ∎

A well-typed IMP_E^2 program tracks information flow from the initial input. The execution of a IMP_E^2 program preserves typing. This key theorem will allow us to prove that well-typed IMP_E^2 programs satisfy noninterference according to policy.

Theorem 3.10 (Type preservation) *Let* Γ *be a well-formed typing context, and* c_0 *a* IMP_E *command,* c, c' IMP_E^2 *commands, and* $\sigma_0, \sigma, \sigma'$ IMP_E^2 *memories such that* $\sigma_0 = \mathit{erasure}^2(\sigma_0)$ *and* $\langle c_0,\, \sigma_0\rangle_\bullet \rightarrow^* \langle c,\, \sigma\rangle_\bullet \rightarrow \langle c',\, \sigma'\rangle_\bullet$.

Let $\tau = \langle c_0,\, \sigma_0\rangle_\bullet \ldots \langle c,\, \sigma\rangle_\bullet$, *and let* $\tau' = \tau\langle c',\, \sigma'\rangle_\bullet$.

If $\tau, \bot, \Gamma \vdash \langle c,\, \sigma\rangle_\bullet$ config *then* $\tau', \bot, \Gamma \vdash \langle c',\, \sigma'\rangle_\bullet$ config.

Before we prove Theorem 3.10, we first state and prove a series of useful lemmas.

The first two lemmas relate to the program counter policy. If a program is well-typed for some program counter policy pc, then it is also well-typed for any weaker program counter policy, and also, any variable x that is updated in the next step satisfies $pc \leq \Gamma(x)$.

Lemma 3.11 *If* $pc \leq pc'$ *and* $\tau, pc', \Gamma \vdash c$ com *then* $\tau, pc, \Gamma \vdash c$ com.

Proof: By induction on $\tau, pc', \Gamma \vdash c$ com. ∎

Lemma 3.12 *Let Γ be a well-formed typing context, τ a trace, $i \in \{1, 2, \bullet\}$, c, c' commands, and $\sigma, sigma'$ memories such that $\langle c,\ \sigma\rangle_i \rightarrow \langle c',\ \sigma'\rangle_i$ and $\tau, pc, \Gamma \vdash c$ com. For all $x \in Vars$, if $\sigma(x) \neq \sigma'(x)$ then $pc \leq \Gamma(x)$.*

Moreover, if the execution step assigned some variable y, then for all $x \in Vars$, if $\sigma(x) \neq \sigma'(x)$ then $\Gamma(y) \leq \Gamma(x)$.

Proof: By induction on the derivation of $\langle c,\ \sigma\rangle_i \rightarrow \langle c',\ \sigma'\rangle_i$. The only way the memory can change is by assigning some variable y the value (or pair value) v, via the utility function $update(\sigma, y, v)$.

If $x = y$, then the appropriate typing rule (OS-ASSIGN, OS-DECLASSIFY, or OS-PAIR-DECLASSIFY) requires that $pc \leq \Gamma(y) = \Gamma(x)$. If $x \neq y$, then, considering the definition of $erasure(\sigma[y \mapsto v])$ there must be some k such that $\sigma'(x) = \sigma_k(x) \neq \sigma(x)$. By induction on k, we can show that for any variable z, if $\sigma_k(z) \neq \sigma(z)$, then $pc \leq \Gamma(y) \leq \Gamma(z)$. The base case $k = 0$ follows from the typing rules requiring $pc \leq \Gamma(y)$. The inductive case is that if $\neg$reqErase(z, σ) but reqErase(z, σ_{k+1}), then there must be some variable z' such that $\sigma(z') \neq \sigma_k(z') = \sigma'(z')$, z' appears in an expression in eraseConds$(\Gamma(z))$. By the induction hypothesis, we have $pc \leq \Gamma(y) \leq \Gamma(z')$. Since Γ is well-formed, we have $\Gamma(z') \leq \Gamma(z)$, so $pc \leq \Gamma(y) \leq \Gamma(z)$ as required. ∎

To prove type preservation in IMP^2_E, it is helpful to know that IMP$_E$ also preserves types.

Lemma 3.13 (Type preservation for IMP$_E$) *Let Γ be a well-formed typing context, and c a IMP$_E$ command, and σ a IMP$_E$ memory, and pc a policy such that $pc, \Gamma \vdash c$ com. For $i \in \{1, 2\}$, if $\langle c,\ \sigma\rangle_i \rightarrow \langle c',\ \sigma'\rangle_i$ then $pc, \Gamma \vdash c'$ com.*

Proof: By induction on $\langle c,\ \sigma\rangle_i \rightarrow \langle c',\ \sigma'\rangle_i$, using Lemma 3.11 applied to IMP$_E$ type judgments. ∎

The following series of lemmas are related to showing that nice properties hold for the utility functions $update^2(\cdot,\cdot,\cdot)$ and $erasure^2(\cdot)$. Several of them are concerned with memories σ that satisfy $\sigma = erasure^2(\sigma)$. We call such memories *consistent*, as they are consistent with erasure requirements: $\forall i \in \{1,2\}.\ \forall x \in Vars.\ \mathsf{reqErase}(\Gamma(x), \lfloor\sigma\rfloor_i) \Rightarrow \lfloor\sigma(x)\rfloor_i = 0$.

The IMP_E^2 utility functions $update^2(\cdot,\cdot,\cdot)$ and $erasure^2(\cdot)$ agree with their IMP_E versions.

Lemma 3.14 *Let σ and σ' be IMP_E^2 memories. If $\sigma' = update^2(\sigma, x, (\!| v_1 \mid v_2 |\!))$ for some variable x and values v_1 and v_2, then for all $i \in \{1,2\}$, $\lfloor\sigma'\rfloor_i = update(\lfloor\sigma\rfloor_i, x, v_i)$. Similarly, if $\sigma' = erasure^2(\sigma)$, then for all $i \in \{1,2\}$, $\lfloor\sigma'\rfloor_i = erasure(\lfloor\sigma\rfloor_i)$*

Proof: Suppose $\sigma' = update^2(\sigma, x, (\!| v_1 \mid v_2 |\!))$ for some variable x and values v_1 and v_2. Let $\sigma'_i = update(\lfloor\sigma\rfloor_i, x, v_i)$. Let y be a variable. If $\mathsf{reqErase}(\Gamma(y), \sigma'_1)$ and $\mathsf{reqErase}(\Gamma(y), \sigma'_2)$ then $\sigma'_i(y) = 0 = \lfloor\sigma'\rfloor_i(y)$ as required. If $\sigma'_1(y) = \lfloor\sigma\rfloor_1(y)$ and $\sigma'_2(y) = \lfloor\sigma\rfloor_2(y)$ then $\lfloor\sigma'(y)\rfloor_i = \lfloor\sigma(y)\rfloor_i = \sigma'_i(y)$ as required. Otherwise, $\sigma'(y) = (\!| \sigma'_1(y) \mid \sigma'_2(y) |\!)$, so $\lfloor\sigma(y)\rfloor_i = \sigma'_i(y)$ as required.

Now suppose $\sigma' = erasure^2(\sigma)$. Let $\sigma'_i = erasure(\lfloor\sigma\rfloor_i)$. Let y be a variable. If $\mathsf{reqErase}(\Gamma(y), \sigma'_1)$ and $\mathsf{reqErase}(\Gamma(y), \sigma'_2)$ then $\sigma'_i(y) = 0 = \lfloor\sigma'\rfloor_i(y)$ as required. If $\sigma'_1(y) = \lfloor\sigma\rfloor_1$ and $\sigma'_2(y) = \lfloor\sigma\rfloor_2$ then $\lfloor\sigma'(y)\rfloor_i = \lfloor\sigma(y)\rfloor_i = \sigma'_i(y)$ as required. Otherwise, $\sigma'(y) = (\!| \sigma'_1(y) \mid \sigma'_2(y) |\!)$, so $\lfloor\sigma(y)\rfloor_i = \sigma'_i(y)$ as required. ∎

The utility function $update^2(\cdot,\cdot,\cdot)$ establishes a consistent memory.

Lemma 3.15 *Let σ and σ' be IMP_E^2 memories such that $\sigma' = update^2(\sigma, x, w)$ for some variable x and value w. Then $\sigma' = erasure^2(\sigma')$*

Proof: Note that for IMP_E memories σ_0 and σ'_0 if $\sigma'_0 = update(\sigma_0, x, v)$ for some variable x and value v, then $\sigma'_0 = erasure(\sigma'_0)$. This follows easily from the definition of $update(\sigma_0, x, v)$ and the idempotency of $erasure(\cdot)$.

Let $\sigma'_i = erasure(\lfloor \sigma' \rfloor_i)$. We have

$$\begin{aligned}
\sigma'_i &= erasure(\lfloor \sigma' \rfloor_i) \\
&= erasure(\lfloor update^2(\sigma, x, w) \rfloor_i) \\
&= erasure(update(\lfloor \sigma \rfloor_i, x, \lfloor w \rfloor_i)) \\
&= update(\lfloor \sigma \rfloor_i, x, \lfloor w \rfloor_i).
\end{aligned}$$

Let y be a variable. Consider $erasure^2(\sigma')(y)$. If $\forall i \in \{1,2\}.\ \mathsf{reqErase}(\Gamma(y), \sigma'_i)$ then from the definition of $update^2(\sigma, x, w)$ we have $\sigma'(y) = 0 = erasure^2(\sigma')(y)$. Similarly, if $\exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(y), \sigma'_i)$ and $\exists i \in \{1,2\}.\ \lfloor \sigma \rfloor_i(y) \neq \sigma'_i(y)$, then from the definition of $update^2(\sigma, x, w)$ we have $\sigma'(y) = (\!|\, \sigma'_1(y) \,|\, \sigma'_2(y) \,|\!) = erasure^2(\sigma')(y)$. Finally, if we have both $\exists i \in \{1,2\}.\ \neg\mathsf{reqErase}(\Gamma(y), \sigma'_i)$ and $\forall i \in \{1,2\}.\ \lfloor \sigma \rfloor_i(y) = \sigma'_i(y)$, then from the definition of $update^2(\sigma, x, w)$ we have $\sigma'(y) = \sigma(y) = erasure^2(\sigma')(y)$. ∎

The semantics of IMP^2_E preserves consistent memories.

Lemma 3.16 *Let Γ be a well-formed typing context, and c_0 a IMP_E command, c a IMP^2_E command, and σ_0, σ IMP^2_E memories such that $\sigma_0 = erasure^2(\sigma_0)$ and $\tau = \langle c_0,\ \sigma_0 \rangle_\bullet \rightarrow^* \langle c,\ \sigma \rangle_\bullet$. Then $\sigma = erasure^2(\sigma)$.*

Proof: By induction on $\rightarrow$. When a step does not change the memory, this is trivial. For OS-ASSIGN, OS-DECLASSIFY, and OS-PAIR-DECLASSIFY, the result follows from the idempotency of $erasure^2(\cdot)$. For OS-PAIR-LIFT, it follows from the idempotency of $erasure(\cdot)$. ∎

For consistent memories σ, a variable x will map to a pair value in a σ only if the policy $\Gamma(x)$ does not require erasure in at least one of the projections. Equivalently, if $\Gamma(x)$ requires erasure in both projections, then $sigma(x)$ will not be a pair value.

Lemma 3.17 *For all IMP^2_E memories σ, and all variables $x \in Vars$, if $\sigma = erasure^2(\sigma)$ and $\sigma(x) = (\!| v_1 \mid v_2 |\!)$, then* $\neg$reqErase$(\Gamma(x), \lfloor\sigma\rfloor_i)$ *for some $i \in \{1, 2\}$.*

Proof: Immediate from the definition of $erasure^2(\sigma)$. ∎

For consistent memories σ, if there is a variable x that maps to a pair value, then there is some variable y such that y also maps to a pair value, $\Gamma(y)$ does not require erasure in either projection, and information is allowed to flow from y to x. The proof of this lemma uses the well-foundedness of the overwrite dependency relation $\prec_\Gamma$.

Lemma 3.18 *For any IMP^2_E memory σ, and variable x, if $\sigma = erasure^2(\sigma)$ and $\sigma(x) = (\!| v_1 \mid v_2 |\!)$, then there is a variable y such that $\sigma(y) = (\!| v_1' \mid v_2' |\!)$ and for all $i \in \{1, 2\}$* $\neg$reqErase$(\Gamma(y), \lfloor\tau\rfloor_i)$, *and $\Gamma(y) \leq \Gamma(x)$*

Proof: If $\neg$reqErase$(\Gamma(x), \lfloor\sigma\rfloor_i)$ for all $i \in \{1, 2\}$, then we are done. If not, then by Lemma 3.17, $\neg$reqErase$(\Gamma(x), \lfloor\sigma\rfloor_i)$ for some $i \in \{1, 2\}$. This means there is some expression $e \in$ eraseConds$(\Gamma(x))$ such that $\sigma(e)$ is a pair value, so there is some variable x_0 that appears in e such that $\sigma(x_0)$ is a pair value. Since Γ is well-formed, we have $\Gamma(x_0) \leq \Gamma(x)$. Note that $x_0 \prec_\Gamma x$. If $\neg$reqErase$(\Gamma(x_0), \lfloor\tau\rfloor_i)$ for all $i \in \{1, 2\}$, then we are done. Otherwise, we repeat the argument, forming a chain $x_0, x_1, \ldots$ such that $x_{k+1} \prec_\Gamma x_k$ and $\Gamma(x_{k+1}) \leq \Gamma(x_k)$. Since Γ is a well-formed typing context, the relation $\prec_\Gamma$ is well-founded, and thus eventually a variable x_n will be found such that $\sigma(x_n)$ is a pair value and $\neg$reqErase$(\Gamma(x_n), \lfloor\tau\rfloor_i)$ for all $i \in \{1, 2\}$. ∎

The next two lemmas concern the preservation of predicates $protected(p, \tau)$ and $\neg$reqErase$(p, \lfloor\tau\rfloor_i)$ when the trace τ is extended. The first claims that if τ is extended by one step to τ' but the memory is not changed in that step, then $protected(p, \tau)$ implies $protected(p, \tau')$. The second lemma claims that if a IMP^2_E

command c is well-typed for a trace τ, and trace τ' satisfies all $protected(\cdot,\cdot)$ and $\neg$reqErase$(\cdot,\cdot)$ predicates that τ does, then c is well-typed for τ'.

Lemma 3.19 *Let Γ be a well-formed typing context, and c_0, c IMP$_E$ commands, and σ_0, σ IMP$_E^2$ memories such that $\sigma_0 = erasure^2(\sigma_0)$ and $\langle c_0, \sigma_0\rangle_\bullet \rightarrow^* \langle c, \sigma\rangle_\bullet$. Suppose $\langle c, \sigma\rangle_\bullet \rightarrow \langle c', \sigma\rangle_\bullet$. Let $\tau = \langle c_0, \sigma_0\rangle_\bullet \ldots \langle c, \sigma\rangle_\bullet$, and let $\tau' = \tau\langle c', \sigma\rangle_\bullet$. Then for all policies p, if $protected(p, \tau)$ then $protected(p, \tau')$.*

Proof: Suppose $protected(p, \tau)$. We need to show that either $\neg$reqErase$(p, \lfloor\tau'\rfloor_1)$ or $\neg$reqErase$(p, \lfloor\tau'\rfloor_2)$ and that for $i \in \{1, 2\}$ if $\neg$reqErase$(p, \lfloor\sigma\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} p$.

First note that the final memories of τ and τ' are identical, so we have $\neg$reqErase$(p, \lfloor\tau\rfloor_i)$ if and only if $\neg$reqErase$(p, \lfloor\tau'\rfloor_i)$.

Since $protected(p, \tau)$, either $\neg$reqErase$(p, \lfloor\tau\rfloor_1)$ or $\neg$reqErase$(p, \lfloor\tau\rfloor_2)$, so either $\neg$reqErase$(p, \lfloor\tau'\rfloor_1)$ or $\neg$reqErase$(p, \lfloor\tau'\rfloor_2)$.

Suppose for some i we have $\neg$reqErase$(p, \lfloor\tau'\rfloor_i)$. Then $\neg$reqErase$(p, \lfloor\tau\rfloor_i)$, and since $protected(p, \tau)$, $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} p$. By the inference rules for extended relabeling, we can conclude $protected(p, \tau')$. ∎

Lemma 3.20 *Let τ and τ' be IMP$_E^2$ traces, Γ a well-formed context, pc a policy, and c a IMP$_E^2$ command such that $\tau, pc, \Gamma \vdash c$ com. If for all policies p we have $protected(p, \tau) \Rightarrow protected(p, \tau')$ and $\neg$reqErase$(p, \lfloor\tau\rfloor_i) \Rightarrow \neg$reqErase$(p, \lfloor\tau'\rfloor_i)$, then $\tau', pc, \Gamma \vdash c$ com.*

Proof: By induction on the derivation of $\tau, pc, \Gamma \vdash c$ com, the only interesting case being T-PAIR. ∎

Pair commands are introduced into a configuration only when an **if** command is executed, and the conditional expression evaluates to a pair value. This restricts where pair commands may appear.

Lemma 3.21 *Let Γ be a well-formed context, c a IMP$_E$ command, and σ a IMP$_E^2$ memory. For any configuration $\langle c',\ \sigma'\rangle_\bullet$ such that $\langle c,\ \sigma\rangle_\bullet \rightarrow^* \langle c',\ \sigma'\rangle_\bullet$, and any sequence $d_0; d_1$ that is a sub-command of c', the command d_1 does not contain any pair commands.*

Proof: By induction on $\langle c,\ \sigma\rangle_\bullet \rightarrow^* \langle c',\ \sigma'\rangle_\bullet$. ∎

Using these lemmas, we can now prove that IMP$_E^2$ preserves typing.

Proof of Theorem 3.10: Proof is by induction on the judgment $\langle d,\ \sigma\rangle_\bullet \rightarrow \langle d',\ \sigma'\rangle_\bullet$. Let $\tau = \langle c_0,\ \sigma_0\rangle_\bullet \ldots \langle c,\ \sigma\rangle_\bullet$, and let $\tau' = \tau\langle c',\ \sigma'\rangle_\bullet$. Note that by Lemma 3.16, we have $\sigma = \mathit{erasure}^2(\sigma)$ and $\sigma' = \mathit{erasure}^2(\sigma')$.

- OS-SKIP. Here $d = \textbf{skip}; d'$. Since the memory is unchanged, by Lemmas 3.19 and 3.20 we have $\tau', \bot, \Gamma \vdash d'$ com.
- OS-ASSIGN. Here $d = y := e$ and $d' = \textbf{skip}$. By the typing rule for **skip**, we have $\tau', \bot, \Gamma \vdash d'$ com. We need to show that $\forall z \in \mathit{Vars}.\ (\sigma'(z) = (\!|\, v_1 \,|\, v_2 \,|\!)) \Rightarrow \mathit{protected}(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z) = (\!|\, v_1 \,|\, v_2 \,|\!)$. By Lemma 3.17, we either have $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$, where x is the input variable. Assume that we have $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$.

  First, suppose $\sigma(z) = \sigma'(z)$. If $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$, then, by the inference rules for the extended relabeling relation, we can conclude that $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$, and we are done. Otherwise $\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$, so by Lemma 3.17 and Lemma 3.18, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_i)$ and $\Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c,\ \sigma\rangle_\bullet$ config, we have $\mathit{protected}(\Gamma(w), \tau)$, so $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus, by the inference rules for the extended relabeling relation, $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

Otherwise, $\sigma(z) \neq \sigma'(z)$ so either $z = y$, or z was updated by the utility function $erasure(\cdot)$.

If $z = y$, then $\sigma(e)$ is a pair value, since $\sigma'(z)$ is a pair value. Thus there must be a variable w that appears in e such that $\sigma(w)$ is a pair, and $\Gamma(w) \leq \Gamma(z)$ (by the typing rule for assignment). By Lemma 3.18, there is a variable w' such that $\sigma(w')$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w'), \lfloor\tau\rfloor_i)$ and $\Gamma(w') \leq \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c,\, \sigma\rangle_\bullet$ config, we have $protected(\Gamma(w'), \tau)$, so $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w')$, and thus, by the inference rules for the extended relabeling relation, $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

Finally, if $\sigma(z) \neq \sigma'(z)$ and $z \neq y$, then z was updated by the utility function $erasure(\cdot)$. Note that this means $\Gamma(z)$ requires erasure on at least one of the two projections. By Lemma 3.18, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$ and $\Gamma(w) \leq \Gamma(z)$. Since $\Gamma(w)$ does not require erasure in either projection, w was not updated by the utility function $erasure(\cdot)$. Thus, by the previous cases, we have $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

- OS-SEQUENCE. Here $d = d_1; d_2$ and $d' = d'_1; d_2$. By the inductive hypothesis, we have $\tau', \bot, \Gamma \vdash d'_1$ com, and that $\forall x \in Vars.\ (\sigma'(x) = (\!|\, v_1 \,|\, v_2 \,|\!)) \Rightarrow protected(\Gamma(x), \tau')$. We need to show that $\tau', \bot, \Gamma \vdash d_2$ com. We do this by an easy induction on the derivation of $\tau, \bot, \Gamma \vdash d_2$ com which relies on the fact that, by Lemma 3.21, the command d_2 cannot contain a command pair $(\!|\, d_3 \,|\, d_4 \,|\!)$. Thus we have $\tau', \bot, \Gamma \vdash \langle d'_1; d_2,\, \sigma'\rangle_\bullet$ config as required.

- OS-IF. Here $d =$ **if** e **then** d_0 **else** d_1 and $d' = d_i$ for some $i \in \{0, 1\}$. By the typing rule for **if**, and Lemma 3.11, we have $\tau, \bot, \Gamma \vdash d'$ com. Since the memory is unchanged, by Lemma 3.19 and Lemma 3.20 we have $\tau', \bot, \Gamma \vdash d'$ com.

- OS-WHILE. Here $d = \textbf{while}\ e\ \textbf{do}\ d_1$ and $d' = \textbf{if}\ e\ \textbf{then}\ (d_1; \textbf{while}\ e\ \textbf{do}\ d_1)\ \textbf{else skip}$. Since $\tau, \bot, \Gamma \vdash d$ com, we have $\Gamma \vdash e : p_e$ exp, and there exists some policy pc' such that $\bot \leq pc'$, and $p_e \leq pc'$, and $\tau, pc', \Gamma \vdash d_1$ com. From this we can derive $\tau, pc', \Gamma \vdash \textbf{while}\ e\ \textbf{do}\ d_1$ com, and thus $\tau, pc', \Gamma \vdash d'$ com, so by Lemma 3.11, we have $\tau, \bot, \Gamma \vdash d'$ com. Since the memory is unchanged, by Lemma 3.19 and Lemma 3.20 we have $\tau', \bot, \Gamma \vdash d'$ com.

- OS-DECLASSIFY. Here $d = y := \textbf{declassify}(e, p_f\ \textbf{to}\ p_t\ \textbf{using}\ e_0, \ldots, e_k)$ and $d' = \textbf{skip}$. By the typing rule for **skip**, we have $\tau', \bot, \Gamma \vdash d'$ com. We need to show that $\forall z \in \mathit{Vars}.\ (\sigma'(z) = (\!|\, v_1 \,|\, v_2 \,|\!)) \Rightarrow \mathit{protected}(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z) = (\!|\, v_1 \,|\, v_2 \,|\!)$. By Lemma 3.17, either $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_i)$ then $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(z)$. Suppose $\neg\mathsf{reqErase}(\Gamma(z), \lfloor \tau' \rfloor_i)$.

  First, suppose $\sigma(z) = \sigma'(z)$. The reasoning is exactly the same as the analogous subcase in OS-ASSIGN.

  Otherwise, $\sigma(z) \neq \sigma'(z)$ so either $z = y$, or z was updated by the utility function $\mathit{erasure}(\cdot)$.

  If $z = y$, then $\sigma(e)$ is a pair value, since $\sigma'(z)$ is a pair value. (This implies that all conditions e_i were satisfied, as otherwise, z is updated with the non-pair value 0.) The reasoning in this case is exactly the same as the analogous subcase in OS-ASSIGN.

  Finally, if $\sigma(z) \neq \sigma'(z)$ and $z \neq y$, then z was updated by the utility function $\mathit{erasure}(\cdot)$. The reasoning here is exactly the same as the analogous subcase in OS-ASSIGN.

- OS-PAIR-SKIP. Immediate by typing rule for **skip**.

- OS-PAIR-LIFT. Here $d = (\!|\, d_1 \,|\, d_2 \,|\!)$ and $d' = (\!|\, d_1' \,|\, d_2' \,|\!)$. If $\sigma' = \sigma$, then by Lemma 3.19, we have $\tau', \bot, \Gamma \vdash d'$ com. So suppose $\sigma' \neq \sigma$.

  Since $\tau, \bot, \Gamma \vdash d$ com, there is a policy pc' such that we have $\mathit{protected}(pc', \tau)$, $\neg\mathsf{reqErase}(pc', \tau)$, and $pc', \Gamma \vdash d_i$ com for $i \in \{1, 2\}$. By Lemma 3.13, we have $pc', \Gamma \vdash d_i'$ com, for $i \in \{1, 2\}$.

  Without loss of generality, assume that it is the left execution that makes progress, and thus $d_2 = d_2'$. Note that since the left projection made progress, the right projection was unchanged, so $\lfloor \sigma' \rfloor_2 = \lfloor \sigma \rfloor_2$, so we have $\neg\mathsf{reqErase}(pc', \lfloor \sigma' \rfloor_2)$.

  If $\neg\mathsf{reqErase}(pc', \lfloor \sigma' \rfloor_1)$, then we can easily show that $\mathit{protected}(pc', \tau')$, so we have $\tau', \bot, \Gamma \vdash d'$ com, as required.

  Otherwise, suppose $\mathsf{reqErase}(pc', \lfloor \sigma' \rfloor_1)$. Since $\sigma' \neq \sigma$, there must have been either an assignment or a declassification, updating some variable y. By the definition of $\mathit{update}(\cdot, \cdot, \cdot)$, this means that $\neg\mathsf{reqErase}(\Gamma(y), \lfloor \tau \rfloor_1)$, and by the well-formedness of Γ, $\neg\mathsf{reqErase}(\Gamma(y), \lfloor \tau' \rfloor_1)$ (since whether $\Gamma(y)$ requires erasure cannot depend on y). By Lemma 3.12, $pc' \leq \Gamma(y)$. Since $\neg\mathsf{reqErase}(pc', \lfloor \sigma \rfloor_1)$ but $\mathsf{reqErase}(pc', \lfloor \sigma' \rfloor_1)$, there is a variable y' such that $\lfloor \sigma' \rfloor_1(y') \neq \lfloor \sigma \rfloor_1(y')$, and there is an expression $e \in \mathsf{eraseConds}(pc')$ such that y' appears in e, and either $y = y'$ or y can affect whether $\Gamma(y')$ requires erasure, that is (y, y') is in the transitive closure of the overwrite dependency relation $\prec_\Gamma$. From the well-formedness of Γ, we have $\Gamma(y) \leq \Gamma(y')$, and since $\Gamma \vdash pc'$ pol, $\Gamma(y') \leq pc'$, and thus $\Gamma(y) \leq pc'$.

  Since $\sigma'(y)$ is a pair value, by Lemma 3.18, there is a variable w such that $\sigma'(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor \tau' \rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor \tau' \rfloor_2)$ and $\Gamma(w) \leq \Gamma(y)$. If $\Gamma(y) \leq \Gamma(w)$, then $pc' \leq \Gamma(w)$, so $\Gamma(x) \leq_{\lfloor \tau' \rfloor_i} \Gamma(w)$, so $\mathit{protected}(\Gamma(w), \tau')$. If $\Gamma(y) \not\sqsubseteq \Gamma(w)$,

then by Lemma 3.12, $\sigma(w) = \sigma'(w)$, and so $\neg$reqErase$(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg$reqErase$(\Gamma(w), \lfloor\tau\rfloor_2)$. Because $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $protected(\Gamma(w), \tau)$, and using the inference rules for the extended relabeling relation, we can show that $protected(\Gamma(w), \tau')$. Moreover, by Lemma 3.11, we have $\tau, \Gamma(w), \Gamma \vdash d_1'$ com and $\tau, \Gamma(w), \Gamma \vdash d_2'$ com. Thus, $\tau', \bot, \Gamma \vdash d'$ com.

We also need to show that for any variable $z \in Vars$, if $\sigma'(z) = (\!|\, v_1 \mid v_2 \,|\!)$ then $protected(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z)$ is a pair value. By Lemma 3.17, either $\neg$reqErase$(\Gamma(z), \lfloor\tau'\rfloor_1)$ or $\neg$reqErase$(\Gamma(z), \lfloor\tau'\rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg$reqErase$(\Gamma(z), \lfloor\tau'\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$. Suppose $\neg$reqErase$(\Gamma(z), \lfloor\tau'\rfloor_i)$.

Suppose $\sigma(z) = \sigma'(z)$. If $\neg$reqErase$(\Gamma(z), \lfloor\tau\rfloor_i)$ we are done. Otherwise reqErase$(\Gamma(z), \lfloor\tau\rfloor_i)$, so by Lemma 3.17 and Lemma 3.18, there is a variable w such that $\sigma(w)$ is a pair value and $\neg$reqErase$(\Gamma(w), \lfloor\tau\rfloor_i)$ and $\Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

Otherwise, $\sigma(z) \neq \sigma'(z)$ so $\lfloor\sigma\rfloor_1(z) \neq \lfloor\sigma'\rfloor_1(z)$, and since $\tau, pc', \Gamma \vdash d_1$ com, by Lemma 3.12 we have $pc' \leq \Gamma(z)$. Thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_1} \Gamma(z)$ as required.

- OS-PAIR-IF. Here $d =$ **if** e **then** d_0 **else** d_1 and $d' = (\!|\, d_1' \mid d_2' \,|\!)$. Since $\sigma(e) = (\!|\, v_1 \mid v_2 \,|\!)$, there is at least one variable y that appears in e such that $\sigma(y)$ is a pair value. By Lemma 3.18, there is a variable w such that $\sigma(w)$ is a pair value and $\neg$reqErase$(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg$reqErase$(\Gamma(w), \lfloor\tau\rfloor_2)$ and $\Gamma(w) \leq \Gamma(y) \leq p_e$. Therefore, by Lemma 3.11, we have $\Gamma(w), \Gamma \vdash d_1'$ com and $\Gamma(w), \Gamma \vdash d_2'$ com. Since $\tau, pc, \Gamma \vdash \langle c, \sigma\rangle_\bullet$ config, we have $protected(\Gamma(w), \tau)$, and by Lemma 3.19, $protected(\Gamma(w), \tau')$ and $\neg$reqErase$(\Gamma(w), \lfloor\sigma'\rfloor_1)$ and $\neg$reqErase$(\Gamma(w), \lfloor\sigma'\rfloor_2)$. Thus, $\tau', \bot, \Gamma \vdash d'$ com.

- OS-PAIR-DECLASSIFY. This case is similar to OS-ASSIGN. Here $d = y := \textbf{declassify}(e, p_f \textbf{ to } p_t \textbf{ using } e_0, \ldots, e_k)$ and $d' = \textbf{skip}$. By the typing rule for **skip**, we have $\tau', \bot, \Gamma \vdash d'$ com. We need to show that $\forall z \in \mathit{Vars}.\ (\sigma'(z) = (\!|\, v_1 \,|\, v_2 \,|\!)) \Rightarrow \mathit{protected}(\Gamma(z), \tau')$. Let z be a variable such that $\sigma'(z)$ is a pair value. By Lemma 3.17, either $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_1)$ or $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_2)$. We now just need to show that for $i \in \{1, 2\}$, if $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$ then $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$. Suppose $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau'\rfloor_i)$.

  Suppose $\sigma(z) = \sigma'(z)$. If $\neg\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$ we are done. Otherwise $\mathsf{reqErase}(\Gamma(z), \lfloor\tau\rfloor_i)$, so by Lemma 3.17 and Lemma 3.18, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$ and $\Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c,\, \sigma\rangle_\bullet$ config, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

  Otherwise, $\sigma(z) \neq \sigma'(z)$ so either $z = y$, or z was updated by the utility function $\mathit{erasure}(\cdot)$.

  Suppose $z = y$. Since $\sigma(e_0 \times \cdots \times e_k)$ is a pair value, there must be a variable w that appears in $e_0 \times \cdots \times e_k$ such that $\sigma(w)$ is a pair, and $\Gamma(w) \leq \Gamma(z)$ (by the typing rule for declassification). By Lemma 3.18, there is a variable w' such that $\sigma(w')$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w'), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w'), \lfloor\tau\rfloor_2)$ and $\Gamma(w') \leq \Gamma(w) \leq \Gamma(z)$. Since $\tau, pc, \Gamma \vdash \langle c,\, \sigma\rangle_\bullet$ config, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w')$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

  Finally, if $\sigma(z) \neq \sigma'(z)$ and $z \neq y$, then z was updated by the utility function $\mathit{erasure}(\cdot)$. Note that this means $\Gamma(z)$ requires erasure on at least one of the two projections. By Lemma 3.18, there is a variable w such that $\sigma(w)$ is a pair value and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_1)$ and $\neg\mathsf{reqErase}(\Gamma(w), \lfloor\tau\rfloor_2)$ and

$\Gamma(w) \leq \Gamma(z)$. Since $\Gamma(w)$ does not require erasure on either projection, w was not updated by the utility function $erasure(\cdot)$. Thus, by the previous cases, we have $\Gamma(x) \leq_{\lfloor\tau\rfloor_i} \Gamma(w)$, and thus $\Gamma(x) \leq_{\lfloor\tau'\rfloor_i} \Gamma(z)$ as required.

∎

Using the type preservation property of IMP^2_E, we are now ready to prove Theorem 3.4: well-typed IMP_E programs satisfy noninterference according to policy.

Proof of Theorem 3.4: Let $v_1, v_2 \in \mathbb{Z}$, let σ be a IMP_E memory. Let $\sigma_0 = update^2(\sigma, x, (\!|\, v_1 \,|\, v_2 \,|\!))$. By Lemma 3.16, $\sigma_0 = erasure^2(\sigma_0)$. Let τ_1 and τ_2 be traces such that $\tau_i[0] = \langle c, \lfloor\sigma_0\rfloor_i\rangle$.

By Lemma 3.7 and Lemma 3.5, there is a IMP^2_E trace τ such that $\lfloor\tau\rfloor_i$ is a prefix of τ_i for all $i \in \{1, 2\}$, and for some $i \in \{1, 2\}$, $\lfloor\tau\rfloor_i = \tau_i$.

We construct a correspondence R for τ_1 and τ_2 such that R is the smallest set such that for all $k \in 1..|\tau|$, $(f_1(\tau[..k]), f_2(\tau[..k])) \in R$, where $f_i(\tau')$ is the number of reduction steps of the IMP^2_E execution τ' that reduce the ith projection. Thus, if $(i, j) \in R$, then there is some k such that $\lfloor\tau[k]\rfloor_1 = \tau_1[i]$ and $\lfloor\tau[k]\rfloor_2 = \tau_2[j]$.

Let $\ell \in \mathcal{L}$, and let $(i, j) \in R$. There is some k such that $\lfloor\tau[k]\rfloor_1 = \tau_1[i]$ and $\lfloor\tau[k]\rfloor_2 = \tau_2[j]$. Let $\tau[k] = \langle c_k, \sigma_k\rangle_\bullet$. Note that $\tau_1[i] = \langle\lfloor c_k\rfloor_1, \lfloor\sigma_k\rfloor_1\rangle$ and $\tau_2[j] = \langle\lfloor c_k\rfloor_2, \lfloor\sigma_k\rfloor_2\rangle$.

Suppose for some variable y, $\lfloor\sigma_k\rfloor_1(y) \neq \lfloor\sigma_k\rfloor_2(y)$. Then $\sigma_k(y) = (\!|\, v_1 \,|\, v_2 \,|\!)$. By Theorem 3.10, we have $\tau[..k], \bot, \Gamma \vdash \langle c_k, \sigma_k\rangle_\bullet$ config. Therefore, we have $protected(\Gamma(y), \tau[..k])$. Thus, either $\Gamma(x) \leq_{\tau_1[..i]} \Gamma(y)$ or $\Gamma(x) \leq_{\tau_2[..j]} \Gamma(y)$. By Property 3.9 and Lemma 3.14, either $(\tau_1[i], \mathsf{obs}(\Gamma(y))) \in [\![\Gamma(x)]\!]_{update(\sigma,x,v_1)}$ or $(\tau_2[j], \mathsf{obs}(\Gamma(y))) \in [\![\Gamma(x)]\!]_{update(\sigma,x,v_2)}$. Therefore, if $(\tau_1[i], \ell) \notin [\![\Gamma(x)]\!]_{update(\sigma,x,v_1)}$ and $(\tau_2[j], \ell) \notin [\![\Gamma(x)]\!]_{update(\sigma,x,v_2)}$, then $\tau_1[i] \approx_\ell \tau_2[j]$, so c is noninterfering according to policy for variable x. ∎

We have shown that well-typed IMP_E programs enforce declassification and erasure policies, in that they satisfy noninterference according to policy. We established this result by presenting a language, IMP_E^2, that can represent in a single execution, the execution of two IMP_E programs. The type-system of IMP_E^2 is designed so that type-preservation of IMP_E^2 implies noninterference according to policy for IMP_E programs.

CHAPTER 4

DECENTRALIZED POLICIES AND ROBUSTNESS

The *decentralized label model* (DLM), introduced by Myers and Liskov (2000), allows mutually distrusting principals to express security requirements. Principals can declare, and retain ownership of, information security policies. Different principals can have different security policies for the same information.

In this chapter we extend the decentralized label model to include declassification and erasure policies, increasing the expressiveness of the security requirements the DLM can capture. We also extend the DLM to allow both confidentiality and integrity policies in a single label, and extend the structure of labels from a semi-lattice to a lattice (Chong and Myers, 2006). We then present the security property of decentralized robustness (Chong and Myers, 2006), a generalization of robustness (Zdancewic and Myers, 2001; Myers et al., 2004; Zdancewic, 2003) that accounts for mutually distrusting principals, and show how to enforce decentralized robustness in a language with erasure and declassification.

The DLM is used to specify security policies in the Jif programming language. In Chapter 5 we will extend Jif with declassification and erasure policies using the extended DLM presented here.

4.1 Decentralized Label Model

The DLM allows mutually distrusting principals to express information security policies. A *principal* is any entity with security concerns, such as a user, a process, a machine, or a collection of users. The set of principals is denoted $Principals$. A principal may delegate its authority to other principals: if principal a delegates its authority to principal b, then b is said to *act for* a, written $b \succeq a$. The *acts-for*

relation is reflexive and transitive; it is similar to the *speaks-for* relation (Lampson et al., 1991), and can be used to encode groups and roles.

A *conjunctive principal* $a \wedge b$ represents the joint authority of both a and b. The conjunctive principal $a \wedge b$ has the authority of both a and b, and can act for each of them: $a \wedge b \succeq a$ and $a \wedge b \succeq b$. Similarly, a *disjunctive principal* $a \vee b$ represents the disjoint authority of a and b. It can be regarded as a group consisting of the principals a and b. Both a and b are able to act for the disjunctive principal $a \vee b$: $a \succeq a \vee b$ and $b \succeq a \vee b$. For historical reasons, a disjunction of principals $a_0 \vee \cdots \vee a_k$ is sometimes written using commas: $a_0, \ldots, a_k$.

Principals express their security concerns with *labels*. A label is a pair of a confidentiality policy and an integrity policy. Labels are associated with information, and a system that enforces labels ensures that the policies of a label are enforced on the appropriate information. Confidentiality policies are formed from conjunctions and disjunctions of *owned reader policies*, and integrity policies are formed from conjunctions and disjunctions of *owned writer policies*. Each owned reader policy and owned writer policy has an owning principal; a policy owned by a principal a is a statement of a's beliefs or requirements about the security of information.

4.1.1 Confidentiality policies

An *owned reader policy* allows the owner of the policy to specify which principals the owner permits to read a given piece of information. As originally presented, the DLM permitted owned reader policies to be of the form $o \rightarrow r$, where the principal o is the owner of the policy, and the principal r (which may be a conjunctive or disjunctive principal) is the specified reader.

We extend the DLM to include erasure and declassification policies as reader policies. The lattice of confidentiality levels with which we instantiate the policy framework of Chapter 2 is the lattice of conjunctive and disjunctive principals, ordered by the *acts-for* relation $\succeq$. Any language for specifying conditions can be used, including the program expressions seen in Chapter 3.

Owned reader policies are thus of the form $o \rightarrow p$, where p is either a principal r, an erasure policy $p_0 \overset{c}{\nearrow} p_1$, or a declassification policy $p_0 \overset{c}{\searrow} p_1$.

An owned reader policy $o \rightarrow p$ means that o requires that the confidentiality enforced on information conforms to the policy p. Principal o permits a principal b to read information only if policy p permits b to observe the information ($b \succeq \mathsf{obs}(p)$) and that declassification (allowing more principals to read) and erasure (allowing fewer principals to read) of the information must conform to the reader policy p.

Example 4.1 *Suppose the owned reader policy* $Alice \rightarrow (Bob \overset{cond}{\searrow} Bob \vee Chuck)$ *is enforced on some information. Alice requires that the declassification policy* $Bob \overset{cond}{\searrow} Bob \vee Chuck$ *is enforced on the information. This means that Alice permits only Bob (and principals that can act for Bob) to read the information. When the condition* $cond$ *is satisfied, the information may be declassified, and after declassification any principal that can act for the disjunctive principal* $Bob \vee Chuck$ *may read the information.*

As a formal semantics for owned reader policies, we define the function $\mathsf{readerpol}(a, o \rightarrow p)$ to be the policy that principal a believes should be enforced on information according to owned reader policy $o \rightarrow p$:

$$\mathsf{readerpol}(a, o \rightarrow p) \triangleq \begin{cases} p & \text{if } o \succeq a \\ \bot & \text{otherwise} \end{cases}$$

A principal a believes that a reader policy $o \rightarrow p$ should restrict the readers of information only if the owner of the policy o can act for a. The parameterization

on principal a is important in the presence of mutual distrust, because it allows the significance of the policy to be expressed for every principal independently. If principal o owns a policy that restricts the readers of information, it does not necessarily mean that another principal a also believes those restrictions should apply. Thus, if o does not act for a, then $\mathsf{readerpol}(a, o \rightarrow p)$ is the most permissive reader policy, the bottom principal $\bot$, a principal that all other principals are able to act for; in other words, a does not credit the policy with any significance.

Example 4.2 *For the owned reader policy* $Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)$*, Alice, and any principal that Alice acts for, require the declassification policy* $Bob \searrow^{cond} Bob \vee Chuck$*to be enforced. Assuming that Alice can act for Edith, but not Frank, we have*

$$\mathsf{readerpol}(Alice, Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)) = Bob \searrow^{cond} Bob \vee Chuck$$

$$\mathsf{readerpol}(Edith, Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)) = Bob \searrow^{cond} Bob \vee Chuck$$

$$\mathsf{readerpol}(Frank, Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)) = \bot$$

We can also define a useful function of owned reader policies $\mathsf{readers}(a, \kappa)$, that for a given principal a describes the set of principals that a believes can read information labeled with owned reader policy κ. The definition of $\mathsf{readers}(a, \kappa)$ makes use of the $\mathsf{obs}(\cdot)$ function, defined in Chapter 2, that describes the observation level of policies.

$$\mathsf{readers}(a, o \rightarrow p) \triangleq \begin{cases} \{b \in Principals \mid b \succeq \mathsf{obs}(p)\} & \text{if } o \succeq a \\ \{b \in Principals \mid b \succeq \bot\} & \text{otherwise} \end{cases}$$

If o, the owner of the policy, can act for a, then a believes that a reader policy $o \rightarrow p$ restricts the readers of information to the set of principals that can act for $\mathsf{obs}(p)$. If o does not act for a, then $\mathsf{readers}(a, o \rightarrow p)$ is the set of all principals, since a does not believe the policy restricts who may read the information at all.

Example 4.3 *Considering again the owned reader policy* $Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)$*, and assuming that Alice can act for Edith, but not Frank, we have*

$$\begin{aligned}
\mathsf{readers}(Alice, Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)) &= \{b \in Principals \mid b \succeq Bob\} \\
\mathsf{readers}(Edith, Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)) &= \{b \in Principals \mid b \succeq Bob\} \\
\mathsf{readers}(Frank, Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)) &= \{b \in Principals \mid b \succeq \bot\} \\
&= Principals.
\end{aligned}$$

Conjunction and disjunction.

Greater expressiveness can be achieved by taking conjunctions and disjunctions of owned reader policies. We define *confidentiality policies* to be the smallest set containing all owned reader policies and closed under the binary operators $\sqcup$ and $\sqcap$: if κ and κ' are confidentiality policies, then both $\kappa \sqcap \kappa'$ and $\kappa \sqcup \kappa'$ are too.

The operator $\sqcup$ is conjunction for confidentiality policies: $\kappa \sqcup \kappa'$ is the policy that enforces both κ and κ'. The policy $\kappa \sqcup \kappa'$ permits a principal to read information only if both κ and κ' allow it; declassification to another confidentiality policy can only occur if both κ and κ' permit the relabeling; and erasure of the information must occur if either κ or κ' require erasure.

The operator $\sqcap$ is disjunction for confidentiality policies: $\kappa \sqcap \kappa'$ enforces only the restrictions that both κ and κ' require. Thus, the confidentiality policy $\kappa \sqcap \kappa'$ allows a principal to read information if either κ or κ' allows it; declassification to another reader policy can occur if either κ or κ' permit the relabeling; and erasure of the information must occur only if both κ and κ' require erasure. The confidentiality policy $\kappa \sqcap \kappa'$ is no more restrictive than either κ or κ'.

We extend $\mathsf{readerpol}(a, \kappa)$ and $\mathsf{readers}(a, \kappa)$ for confidentiality policies. The codomain of $\mathsf{readerpol}(\cdot, \cdot)$ is the *free lattice* (Freese et al., 1995) over reader policies:

the set of reader policies closed under a meet operator $\sqcap$ and a join operator $\sqcup$.

$$\begin{aligned}
\mathsf{readerpol}(a, \kappa \sqcup \kappa') &\triangleq \mathsf{readerpol}(a, \kappa) \sqcup \mathsf{readerpol}(a, \kappa') \\
\mathsf{readerpol}(a, \kappa \sqcap \kappa') &\triangleq \mathsf{readerpol}(a, \kappa) \sqcap \mathsf{readerpol}(a, \kappa')
\end{aligned}$$

The codomain of $\mathsf{readers}(\cdot, \cdot)$ continues to be sets of principals. Since the confidentiality policy $\kappa \sqcup \kappa'$ imposes the restrictions of both κ and κ', the set of principals that a permits to read information labeled $\kappa \sqcup \kappa'$ is the intersection of the readers of κ and κ'. Similarly, the reader set for $\kappa \sqcap \kappa'$ is the union of the reader sets for κ and κ'.

$$\begin{aligned}
\mathsf{readers}(a, \kappa \sqcup \kappa') &\triangleq \mathsf{readers}(a, \kappa) \cap \mathsf{readers}(a, \kappa') \\
\mathsf{readers}(a, \kappa \sqcap \kappa') &\triangleq \mathsf{readers}(a, \kappa) \cup \mathsf{readers}(a, \kappa')
\end{aligned}$$

Example 4.4 *Let* $\kappa = \mathit{Alice} \rightarrow (\mathit{Bob} \searrow^{\mathit{cond}} \mathit{Bob} \vee \mathit{Chuck}) \sqcup \mathit{Bob} \rightarrow \mathit{Dave}$. *Assuming that Bob does not act for Alice, we have*

$$\begin{aligned}
\mathsf{readerpol}(\mathit{Alice}, \kappa) &= \mathit{Bob} \searrow^{\mathit{cond}} \mathit{Bob} \vee \mathit{Chuck} \;\sqcup\; \bot \\
&= \mathit{Bob} \searrow^{\mathit{cond}} \mathit{Bob} \vee \mathit{Chuck} \\
\mathsf{readers}(\mathit{Alice}, \kappa) &= \{b \in \mathit{Principals} \mid b \succeq \mathit{Bob}\} \cap \mathit{Principals} \\
&= \{b \in \mathit{Principals} \mid b \succeq \mathit{Bob}\}.
\end{aligned}$$

More interestingly, if both Alice and Bob act for Edith, then

$$\begin{aligned}
\mathsf{readerpol}(\mathit{Edith}, \kappa) &= (\mathit{Bob} \searrow^{\mathit{cond}} \mathit{Bob} \vee \mathit{Chuck}) \sqcup \mathit{Dave} \\
\mathsf{readers}(\mathit{Edith}, \kappa) &= \{b \in \mathit{Principals} \mid b \succeq \mathit{Bob}\} \;\cap \\
&\qquad\qquad \{b \in \mathit{Principals} \mid b \succeq \mathit{Dave}\} \\
&= \{b \in \mathit{Principals} \mid b \succeq \mathit{Bob} \text{ and } b \succeq \mathit{Dave}\} \\
&= \{b \in \mathit{Principals} \mid b \succeq \mathit{Bob} \wedge \mathit{Dave}\}.
\end{aligned}$$

Ordering confidentiality policies.

Using the $\mathsf{readerpol}(\cdot,\cdot)$ function, we can define a relabeling judgment $c_0,\ldots,c_k \vdash \kappa \sqsubseteq^C \kappa'$ on confidentiality policies. Similar to the relabeling judgment $c_0,\ldots,c_k \vdash p \leq q$ on declassification and erasure policies, presented in Section 2.2.3, the confidentiality judgment $c_0,\ldots,c_k \vdash \kappa \sqsubseteq^C \kappa'$ describes when information labeled with confidentiality policy κ can safely be relabeled with confidentiality policy κ'. Indeed, the confidentiality judgment is defined using the policy relabeling judgment.

$$c_0,\ldots,c_k \vdash \kappa \sqsubseteq^C \kappa' \quad \triangleq$$
$$\forall a \in \mathit{Principals}.\ c_0,\ldots,c_k \vdash \mathsf{readerpol}(a,\kappa) \leq \mathsf{readerpol}(a,\kappa')$$

where we extend the definition of $c_0,\ldots,c_k \vdash p \leq q$ for the free lattice over declassification and erasure policies as follows.

$$\frac{c_0,\ldots,c_k \vdash p \leq q \quad c_0,\ldots,c_k \vdash p' \leq q}{c_0,\ldots,c_k \vdash p \sqcup p' \leq q} \qquad \frac{c_0,\ldots,c_k \vdash p \leq q}{c_0,\ldots,c_k \vdash p \sqcap p' \leq q} \qquad \frac{c_0,\ldots,c_k \vdash p' \leq q}{c_0,\ldots,c_k \vdash p \sqcap p' \leq q}$$

$$\frac{c_0,\ldots,c_k \vdash p \leq q}{c_0,\ldots,c_k \vdash p \leq q \sqcup q'} \qquad \frac{c_0,\ldots,c_k \vdash p \leq q'}{c_0,\ldots,c_k \vdash p \leq q \sqcup q'} \qquad \frac{c_0,\ldots,c_k \vdash p \leq q \quad c_0,\ldots,c_k \vdash p \leq q'}{c_0,\ldots,c_k \vdash p \leq q \sqcap q'}$$

If $c_0,\ldots,c_k \vdash \kappa \sqsubseteq^C \kappa'$ then every principal a believes that, provided conditions $c_0,\ldots,c_k$ are satisfied, κ' is at least as restrictive as κ is, and so information labeled κ can be used in at least as many places as information labeled κ'.

We define the confidentiality relabeling relation $\sqsubseteq^C$ such that $\kappa \sqsubseteq^C \kappa'$ if $\vdash \kappa \sqsubseteq^C \kappa'$. The following key property, relating the confidentiality relabeling relation to the $\mathsf{readers}(\cdot,\cdot)$ function.

Property 4.5 *For all confidentiality policies κ and κ', all states s, and all principals a, if $\kappa \sqsubseteq^C \kappa'$ then* $\mathsf{readers}(a, \kappa) \supseteq \mathsf{readers}(a, \kappa')$.

Proof: By induction on the derivation of $\vdash \kappa \sqsubseteq^C \kappa'$. The only interesting case is $\kappa = p$ and $\kappa' = q$, and we have a derivation for $\vdash p \leq q$. In that case, we proceed by induction on the derivation of $\vdash p \leq q$, where it holds trivially for RL-LATTICE, RL-ERASE-I, and RL-DECL-E, and follows simply from the inductive hypothesis for all other cases. ∎

The relation $\sqsubseteq^C$ forms a pre-order over confidentiality policies, and the equivalence classes form a lattice. The operators $\sqcup$ and $\sqcap$ are the join and meet operators of this lattice. The least restrictive confidentiality policy is the reader policy $\bot \rightarrow \bot$, where $\bot$ is a principal that all principals can act for, since it means all principals believe that information labeled $\bot \rightarrow \bot$ has the lowest possible reader policy $\bot$ enforced on it. The most restrictive expressible confidentiality policy is $\top \rightarrow \top$, where $\top$ is a principal that can act for all principals, and is the most restrictive possible reader policy. Information labeled $\top \rightarrow \top$ is allowed to be read only by principal $\top$, all principals believe that it cannot be declassified, and there is no more restrictive reader policy that it can be erased to.

Previous presentations of the DLM have considered only conjunctions of confidentiality policies, resulting in a join semi-lattice structure. This work adds disjunctions of confidentiality policies, producing a lattice structure that is exploited in Section 4.2.4.

4.1.2 Integrity policies

Integrity and confidentiality are well-known duals, and we define integrity policies dually to confidentiality policies. The set of *integrity policies* is formed by

closing *owned writer policies* under conjunction and disjunction.

Owned writer policies are of the form $o \leftarrow r$, where r is a principal (Myers and Liskov, 2000). Although it is possible to extend writer policies with declassification policies or erasure policies, we refrain from doing so, focusing our attention on how confidentiality changes over time.

An owned writer policy $o \leftarrow w$ allows the owner o to specify which principals may have influenced ("written") the value of a given piece of information. The owned writer policy $o \leftarrow w$ means that according to the owner o, a principal b could have influenced the value of information only if b can act for w. Owned writer policies describe the integrity of information in terms of its provenance.

We define the function $\mathsf{writers}(a, \iota)$ to be the set of principals that principal a believes may have influenced information according to the owned writer policy ι. Like owned reader policies, a principal a believes that the owned writer policy $o \leftarrow w$ describes the writers of information only if o can act for a; if o does not act for a, then a conservatively believes that any principal may have influenced the information.

$$\mathsf{writers}(a, o \leftarrow w) \triangleq \begin{cases} \{b \in \mathit{Principals} \mid b \succeq w\} & \text{if } o \succeq a \\ \{b \in \mathit{Principals} \mid b \succeq \bot\} & \text{otherwise} \end{cases}$$

Dual to confidentiality policies, we denote disjunction for integrity policies with the operator $\sqcup$, and conjunction with $\sqcap$. The integrity policy $\iota \sqcap \iota'$ is the conjunction of ι and ι', meaning that a principal a could have influenced information labeled $\iota \sqcap \iota'$ only if both ι and ι' agree that a could have influenced it. The writer sets for ι and ι' are thus intersected to produce the writer set for $\iota \sqcap \iota'$. The integrity policy $\iota \sqcup \iota'$ is the disjunction of ι and ι'; the writer set for $\iota \sqcup \iota'$ is

thus the union of the writer sets for ι and ι'.

$$\begin{aligned}\mathsf{writers}(a, \iota \sqcap \iota') &\triangleq \mathsf{writers}(a, \iota) \cap \mathsf{writers}(a, \iota') \\ \mathsf{writers}(a, \iota \sqcup \iota') &\triangleq \mathsf{writers}(a, \iota) \cup \mathsf{writers}(a, \iota')\end{aligned}$$

Example 4.6 *Let $\iota = Alice \leftarrow Chuck \ \sqcup \ Bob \leftarrow Chuck \vee Dave$. Assuming that Bob does not act for Alice, we have*

$$\begin{aligned}\mathsf{writers}(Alice, \iota) &= \{b \in Principals \mid b \succeq Chuck\} \cup Principals \\ &= Principals.\end{aligned}$$

If both Alice and Bob act for Edith, then

$$\begin{aligned}\mathsf{writers}(Edith, \kappa) &= \{b \in Principals \mid b \succeq Chuck\} \ \cup \\ &\qquad \{b \in Principals \mid b \succeq Chuck \vee Dave\} \\ &= \{b \in Principals \mid b \succeq Chuck \vee Dave\}.\end{aligned}$$

We define a relabeling relation $\sqsubseteq^I$ on integrity policies: for two integrity policies ι and ι', we have $\iota \sqsubseteq^I \iota'$ if and only if for all principals a, $\mathsf{writers}(a, \iota) \subseteq \mathsf{writers}(a, \iota')$. Intuitively, information with a smaller writer set has higher integrity than information with a larger writer set, since fewer principals may have influenced the value of the former; the higher the integrity of information, the fewer restrictions on where that information may be used.

Unlike the relabeling judgment on confidentiality policies $c_0, \ldots, c_k \vdash \kappa \sqsubseteq^C \kappa'$ which was defined in terms of the relabeling judgment $c_0, \ldots, c_k \vdash p \leq q$ on declassification and erasure policies, the relabeling relation for integrity policies does not need to consider condition satisfaction, as owned writer policies are restricted to being of the form $o \leftarrow w$, for a principal w.

The relation $\sqsubseteq^I$ forms a pre-order over integrity policies, and the equivalence classes form a lattice, with join and meet operators $\sqcup$ and $\sqcap$ respectively. The

most restrictive integrity policy is $\bot \leftarrow \bot$, since it means all principals believe that any principal may have influenced the information. The policy $\top \leftarrow \top$ is the least restrictive expressible integrity policy, as it means that all principals believe that only principal $\top$ (who can act for all other principals) has influenced the information.

4.1.3 Labels

A label is a pair of a confidentiality policy and an integrity policy. We write a label $\{\kappa; \iota\}$, where κ is a confidentiality policy, and ι is an integrity policy. The confidentiality projection of $\{\kappa; \iota\}$, written $C(\{\kappa; \iota\})$, is κ, and the integrity projection $I(\{\kappa; \iota\})$ is ι. We extend the $\mathsf{readerpol}(\cdot,\cdot)$, $\mathsf{readers}(\cdot,\cdot)$ and $\mathsf{writers}(\cdot,\cdot)$ functions appropriately:

$$\begin{aligned} \mathsf{readerpol}(a, \{\kappa; \iota\}) &\triangleq \mathsf{readerpol}(a, \kappa) \\ \mathsf{readers}(a, \{\kappa; \iota\}) &\triangleq \mathsf{readers}(a, \kappa) \\ \mathsf{writers}(a, \{\kappa; \iota\}) &\triangleq \mathsf{writers}(a, \iota) \end{aligned}$$

Example 4.7 *Consider the following label.*

$$\{Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck) \;;\; Alice \leftarrow Chuck \sqcup Bob \leftarrow Chuck \vee Dave\}$$

The confidentiality policy of this label is the owned reader policy $Alice \rightarrow (Bob \searrow^{cond} Bob \vee Chuck)$*, and the integrity policy is the disjunction of two owned writer policies:* $Alice \leftarrow Chuck$ *and* $Bob \leftarrow Chuck \vee Dave$*.*

Ordering labels

We define the relabeling judgment $c_0, \ldots, c_k \vdash L \sqsubseteq L'$ on labels using the judgment $c_0, \ldots, c_k \vdash \kappa \sqsubseteq^C \kappa'$ and the relation $\sqsubseteq^I$.

$$c_0, \ldots, c_k \vdash \{\kappa; \iota\} \sqsubseteq \{\kappa'; \iota'\} \triangleq c_0, \ldots, c_k \vdash \kappa \sqsubseteq^C \kappa' \text{ and } \iota \sqsubseteq^I \iota'.$$

We again define the relation $\sqsubseteq$ over labels such that $L \sqsubseteq L'$ if $\vdash L \sqsubseteq L'$. The relation $\sqsubseteq$ forms a pre-order, whose equivalence classes form a lattice. We use $\sqcup$ and $\sqcap$ for the join and meet operations over this lattice,

$$L_1 \sqcup L_2 \triangleq \{C(L_1) \sqcup C(L_2) \ ; \ I(L_1) \sqcup I(L_2)\}$$
$$L_1 \sqcap L_2 \triangleq \{C(L_1) \sqcap C(L_2) \ ; \ I(L_1) \sqcap I(L_2)\}$$

4.2 Decentralized robustness

Robustness (Zdancewic and Myers, 2001; Myers et al., 2004; Zdancewic, 2003) is a useful semantic security condition that requires that an attacker should not be able to affect the security of information flow. A system is robust if an active attacker (who can modify low-integrity aspects of the system and observe its execution), is unable to learn more than a passive attacker (who can only observe the system's execution).

Previous work, by Myers, Sabelfeld, and Zdancewic (2004), defined robustness with respect to a single attacker. However, in a decentralized setting, with multiple, mutually distrusting principals, the identity and power of an attacker depends on whose viewpoint is considered. The decentralized label model provides sufficient expressiveness for each principal to state their beliefs about the abilities of other principals to read and influence information.

In this section we extend the semantic security condition of robustness to *decentralized robustness* (Chong and Myers, 2006). In a system satisfying decentralized robustness, every principal believes that the system is robust with respect to attacks by any attacker. We consider the enforcement of decentralized robustness in Section 4.3.

4.2.1 Robustness

We define robustness for a nondeterministic state-based system. As in Chapter 2, we assume that for a system S, the set Σ_S is the set of feasible states of S, and that O is the set of observations that can be made on the system, and M is the set of modifiable elements of the system. For an observable $o \in O$ and state $s \in \Sigma_S$, we write $o(s)$ for the result of making observation o while the system is in state s. For an modifiable $m \in M$, a value v and state $s \in \Sigma_S$, we write $s[m \mapsto v]$ for state s with m set to v. The function $\mathsf{pol}(\cdot)$ is a function from observables and modifiables to security labels, and describes the security policy enforced on the information that may be revealed by making an observation. The transition relation of S is denoted $\rightarrow$, and traces of S are finite or infinite sequences of feasible states $s_0 \ldots s_k$ such that $s_i \rightarrow s_{i+1}$ for $i \in 0..(k-1)$.

For the purposes of defining robustness most generally, we do not assume that labels are from the DLM. Instead, we require that a label L is a pair of a confidentiality policy and an integrity policy, and there is an ordering $\sqsubseteq$ over labels. We write $C(L)$ for the confidentiality policy of L, and $I(L)$ for the integrity policy. For security labels L and L', if $L \sqsubseteq L'$, then L requires confidentiality lower than (or equal to) that of L', and higher (or equal) integrity.

The definition of robustness assumes that there is an *attacker*, an entity that is able to modify the behavior of the system in limited ways. An attacker is characterized by its *power*, its ability to modify and observe system behavior. The power of an attacker A is a pair of security labels: $\langle R_A, W_A \rangle$. Security label R_A is an upper bound on the security label of observations that A can make, and W_A is a lower bound on the security label of modifiable elements that A can influence.

An *attack* by attacker A is a modification to some or all modifiable elements with a security label bounded below by W_A. That is, an attack by A can modify

any modifiable $m \in M$ whose associated security label $\mathsf{pol}(m)$ indicates that A is able to influence it: $W_A \sqsubseteq \mathsf{pol}(m)$. An attack a applied to a state s is denoted $s[a]$.

After attacking a system, the attacker observes the subsequent execution of the system. The observational ability of the attacker is characterized by the security label R_A. We define two indistinguishability relations over states: weak indistinguishability and strong indistinguishability.

A *terminating trace* τ is a finite trace such that the state $\tau[|\tau| - 1]$ is a terminal state of the system. Two states s_0 and s_0' are *weakly indistinguishable* to A if for every terminating trace $\tau = s_0 s_1 \ldots s_m$, there is a terminating trace $\tau' = s_0' s_1' \ldots s_n'$ such that there is a correlation R for τ and τ' such that for all $(i, j) \in R$ and all observables o, if $\mathsf{pol}(o) \sqsubseteq R_A$ then $\tau[i](o) = \tau'[j](o)$. States s_0 and s_0' are *strongly indistinguishable* to A if s_0 and s_0' are weakly indistinguishable to A and all traces $\tau = s_0 s_1 \ldots s_m$ and $\tau' = s_0' s_1' \ldots s_n'$ are terminating.

Having defined systems, attackers, and attacks, we can now present the definition of robustness.

Definition 4.8 (Robustness) *A system has* robustness *with respect to attacks by attacker A with power $\langle R_A, W_A \rangle$ if for all states s and s', and all attacks a and a' by attacker A, if $s[a]$ and $s'[a]$ are strongly indistinguishable to A, then $s[a']$ and $s'[a']$ are weakly indistinguishable to A.*

Robustness captures the idea that the observations of an attacker should be independent of what attacks the attacker can make. In particular, an attacker should be unable to force the system to declassify information, or to influence what information is declassified by the system. (The latter is known as a *laundering attack*.) In the context of erasure, an attacker should also be unable to prevent the system from erasing information.

By requiring strong indistinguishability in the premise of the condition, the robustness condition ignores inept attacks that cause a system to diverge and thus to present the attacker with fewer observations. See Myers, Sabelfeld, and Zdancewic (2004) for more discussion of this technical issue.

The definition of robustness assumes a single distinguished attacker A. However, in applications with multiple, mutually distrusting principals, there may be many attackers, and moreover, the identity and power of an attacker may depend on whose viewpoint is considered.

Using the DLM, the security condition of robustness can be generalized to consider attacks launched by an arbitrary principal. To motivate this, we first present an example of a simple system with mutually distrusting principals. We then define *robustness against all attackers*, and derive label constraints that ensure a declassification is robust against all attackers.

4.2.2 Example

Consider a simple sealed-bid auction, shown in Figure 4.1. There are two bidders, Alice and Bob. There are ten consecutive auctions, indexed by the variable i, each auction for a different item. In each auction, both bidders submit a secret bid; after all bids for the ith auction have been submitted, the secret bids are declassified, and the winner computed. We model each bidder as a principal, and have an auctioneer principal au. We assume there are no *acts-for* relationships between these principals. Every variable in the program is annotated with a security label from the DLM (including the extensions for declassification and erasure), which is enforced on information stored in the variable. We assume there is a condition $allBids$ that is satisfied only once all bids for the current auction have been submitted.

```
1  int{⊥→⊥; Alice←au⊓Bob←au} winner[10];
2  int{⊥→⊥; Alice←au⊓Bob←au} i;
3  for (i = 1..10) {
4    int{Alice→(au↘allBids⊥); Alice←au⊓Bob←au} bidAlice
5                                  = getAliceBid(i);
6    int{Bob→(au↘allBids⊥); Alice←au⊓Bob←au} bidBob
7                                  = getBobBid(i);
8
9    // end of auction i
10   int{Alice→⊥; Alice←au⊓Bob←au} openAlice =
11             declassify(bidAlice
12                 from {Alice→(au↘allBids⊥); Alice←au⊓Bob←au}
13                 to {Alice→⊥; Alice←au⊓Bob←au}
14                 using allBids);
15
16   int{Bob→⊥; Alice←au⊓Bob←au} openBob =
17             declassify(bidBob
18                 from {Bob→(au↘allBids⊥); Alice←au⊓Bob←au}
19                 to {Bob→⊥; Alice←au⊓Bob←au}
20                 using allBids);
21
22   // compute winner
23   winner[i] = computeWinner(openAlice, openBob);
24
25   // process payment of winning bid
26   ...
27 }
```

Figure 4.1: Sealed-bid auction example.

Consider the auction program from Alice's perspective. In each auction, Alice submits a bid, stored in variable `bidAlice`, with the label

$$\{Alice \rightarrow (au \searrow^{allBids} \bot); Alice \leftarrow au \sqcap Bob \leftarrow au\}$$

enforced on it. Thus, Alice specifies that her bid should be readable only by the auctioneer, but may be declassified when the condition $allBids$ is true, and both Alice and Bob are prepared to accept the bid as high integrity, influenced only by the auctioneer (due to his ability to control when the ith auction commences). After Bob has submitted his bid, Alice's bid is declassified to $\{Alice \rightarrow \bot; Alice \leftarrow au \sqcap Bob \leftarrow au\}$, allowing the bid to be read by all principals, and to be stored in `openAlice`.

Alice may be concerned with attempts by Bob to corrupt the auction. For example, could Bob corrupt the control flow so that Alice's bid is declassified before Bob has submitted his bid, permitting Bob to always win with the minimal winning bid? Or could Bob alter the value stored in `bidAlice`, and fool the system into releasing sensitive information of Alice's, such as her credit card number, or her bid for auction $i + 1$?

Alice would like assurance that the program is robust against attacks by Bob. However, Bob also needs assurance that the program is robust against attacks by Alice. And both principals may be concerned with the auctioneer's ability to corrupt the auction. Even in this simple example there are several potential attackers, and it is necessary to reason about robustness against all possible attackers.

4.2.3 Robustness against all attackers

The power of an attacker A is defined by the pair of labels $\langle R_A, W_A \rangle$, which bounds the information that A can observe and influence. In the setting of Myers,

Sabelfeld, and Zdancewic (2004), there is no *a priori* relationship between A, R_A, and W_A, making it difficult to characterize an arbitrary attacker's power, and therefore difficult to prove robustness against all possible attackers.

However, in the DLM the power of an attacker A can be expressed in terms of the attacker's identity, because all entities are represented by principals. Moreover, we can express the power of an attacker as perceived by a particular principal: for principals a and b, the security labels $R_{a\to b}$ and $W_{a\leftarrow b}$ are bounds on the labels of information that a believes b can read and write:

Definition 4.9 *The label $R_{a\to b}$ is the least upper bound on labels of information that principal a believes principal b can read:*

$$L \sqsubseteq R_{a\to b} \textit{ if and only if } b \in \mathsf{readers}(a, L)$$

The label $W_{a\leftarrow b}$ is the greatest lower bound on labels of information that principal a believes principal b can influence:

$$W_{a\leftarrow b} \sqsubseteq L \textit{ if and only if } b \in \mathsf{writers}(a, L)$$

The labels $R_{a\to b}$ and $W_{a\leftarrow b}$ cannot be expressed as conjunctions and disjunctions of reader and writer policies. We can, however, characterize their reader and writer sets.

$$\begin{aligned}
\mathsf{readers}(a', R_{a\to b}) &\triangleq \{b' \mid b' \succeq b \text{ and } a' \succeq a\} \\
\mathsf{writers}(a', R_{a\to b}) &\triangleq \{b' \mid b' \succeq \bot\} \\
\mathsf{readers}(a', W_{a\leftarrow b}) &\triangleq \{b' \mid b' \succeq \bot\} \\
\mathsf{writers}(a', W_{a\leftarrow b}) &\triangleq \{b' \mid b' \succeq b \text{ and } a \succeq a'\}
\end{aligned}$$

We extend the labels of the DLM to include the labels $R_{a\to b}$ and $W_{a\leftarrow b}$ for all principals a and b. The definition of the label relation $\sqsubseteq$ is extended in the obvious

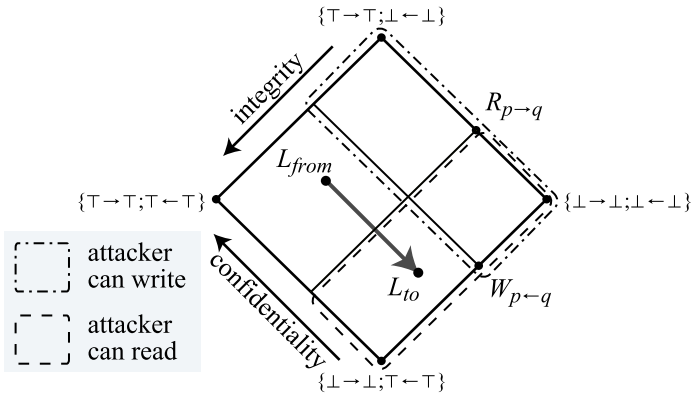

Figure 4.2: Robust declassification in a confidentiality–integrity product lattice.

way, using the definitions for $R_{a\to b}$ and $W_{a\leftarrow b}$ given above. The key property, that $\sqsubseteq$ forms a pre-order whose equivalence classes are a lattice, continues to hold.

Figure 4.2 depicts the points $R_{a\to b}$ and $W_{a\leftarrow b}$ in the product lattice of confidentiality and integrity. Their confidentiality and integrity components divide the set of all labels into four subsets characterized by the power of the attacker to either read or write information with those labels.

Having precisely described an arbitrary attacker's power, we can now define robustness against all attackers.

Definition 4.10 (Robustness against all attackers) *A system has* robustness against all attackers *if for all principals a and b, the system has robustness with respect to attacker b with power $\langle R_{a\to b}, W_{a\leftarrow b}\rangle$.*

If a system is robust against all attackers, then every principal a believes that the system is robust against attacks by any principal b.

4.2.4 Constraints for checking robustness

As will be seen in Section 4.3, the key to enforcing robustness is to ensure that that attacker A is unable to influence changes to the confidentiality of information.

For declassification, this means that if a declassification that reveals information to attacker A, then A is unable to influence either the decision to declassify or the data to be declassified. For erasure, if an erasure makes information no longer readable by attacker A, then A should be unable to influence the satisfaction of the conditions that control the erasure. These requirements have a very natural expression in the DLM. We consider first declassification, then erasure.

Declassification

Suppose L_f is the label of the information to be declassified, L_t is the label of the declassified information, and L_d is an upper bound on the labels of information contributing to the decision to declassify, which may include, for example, the information causing the program counter to reach the program point of declassification, and the information revealed by the satisfaction of conditions needed for declassification. If, from the perspective of a principal a, the declassification reveals information to a principal b, then $b \in \mathsf{readers}(a, L_t) - \mathsf{readers}(a, L_f)$; if this is the case, then we require that b cannot influence either the decision to declassify ($b \notin \mathsf{writers}(a, L_d)$), or the data to be declassified ($b \notin \mathsf{writers}(a, L_f)$).

Figure 4.2 shows part of this requirement graphically: if the declassification from L_f to L_t crosses the line defined by $R_{a \to b}$ (i.e., $b \in \mathsf{readers}(a, L_t) - \mathsf{readers}(a, L_f)$) then L_f should not be above the line defined by $W_{a \leftarrow b}$ (i.e., $b \notin \mathsf{writers}(a, L_f)$).

Since we would like this requirement to hold from every principal's perspective, and for all principals that the declassification may reveal information to, the following statement should hold at every declassification:

$$\forall a \in \mathit{Principals}.\, \forall b \in \mathsf{readers}(a, L_t).\, b \in \mathsf{readers}(a, L_f) \text{ or } (b \notin \mathsf{writers}(a, L_d) \text{ and } b \notin \mathsf{writers}(a, L_f)) \tag{4.1}$$

Unfortunately, it is difficult to prove directly that this statement is true: membership of the sets $\mathsf{writers}(a, L_d)$ and $\mathsf{writers}(a, L_f)$ depends upon the *acts-for* relation $\succeq$, and we may have only partial knowledge of the *acts-for* relation that will be in effect at run time (Chen and Chong, 2004). Demonstrating that a principal b is not a member of $\mathsf{writers}(a, L_d)$ or $\mathsf{writers}(a, L_f)$ is impossible.

However, the following two label constraints suffice to entail condition (4.1).

$$L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_d) \tag{4.2}$$

$$L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_f) \tag{4.3}$$

These label constraints can be verified syntactically, with only partial knowledge of the *acts-for* relation (Myers and Liskov, 2000). The label constraints make use of operator $\mathsf{writersToReaders}(L)$, which converts the writers of label L into readers of label $\mathsf{writersToReaders}(L)$.

$$\begin{aligned}
\mathsf{writersToReaders}(L) &\triangleq \{\mathsf{wtr}(I(L)); \top \leftarrow \top\} \\
\mathsf{wtr}(c \sqcup d) &\triangleq \mathsf{wtr}(c) \sqcap \mathsf{wtr}(d) \\
\mathsf{wtr}(c \sqcap d) &\triangleq \mathsf{wtr}(c) \sqcup \mathsf{wtr}(d) \\
\mathsf{wtr}(o \leftarrow w) &\triangleq o \rightarrow w
\end{aligned}$$

We do not define $\mathsf{writersToReaders}(\cdot)$ for the labels $R_{a\rightarrow b}$ or $W_{a\leftarrow b}$. Although suitable definitions could be given, we ensure that $R_{a\rightarrow b}$ and $W_{a\leftarrow b}$ never appear in label constraints (4.2) or (4.3).

The key property of $\mathsf{writersToReaders}(\cdot)$ is that if principal a believes b is a writer of label L, then a believes b is a reader of $\mathsf{writersToReaders}(L)$:

Property 4.11 *For all labels L, and all principals a and b, if $b \in \mathsf{writers}(a, L)$, then $b \in \mathsf{readers}(a, \mathsf{writersToReaders}(L))$.*

Proof: By induction on the structure of the integrity policy $I(L)$, exploiting the duality between confidentiality and integrity policies. ∎

The following lemma shows that if constraints (4.2) and (4.3) hold, then condition (4.1) holds; that is, every principal a believes that if the declassification reveals information to principal b, then b could not have influenced the decision to declassify or the information to be declassified.

Lemma 4.12 *If* $L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_d)$ *and* $L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_f)$ *then* $\forall a \in \mathit{Principals}.\ \forall b \in \mathsf{readers}(a, L_t).\ b \in \mathsf{readers}(a, L_f)$ *or* $(b \notin \mathsf{writers}(a, L_d)$ *and* $b \notin \mathsf{writers}(a, L_f))$.

Proof: Assume that

$$L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_d)$$

and

$$L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_f).$$

Let a be a principal, and let $b \in \mathsf{readers}(a, L_t)$. If $b \in \mathsf{readers}(a, L_f)$ then we are done. Suppose $b \notin \mathsf{readers}(a, L_f)$. By Property 4.5 we have $\mathsf{readers}(a, L_f) \supseteq \mathsf{readers}(a, L_t) \cap \mathsf{readers}(a, \mathsf{writersToReaders}(L_d))$. If $b \in \mathsf{writers}(a, L_d)$ then by Property 4.11 we have $q \in \mathsf{readers}(a, \mathsf{writersToReaders}(L_d))$. But then $b \in \mathsf{readers}(a, L_t) \cap \mathsf{readers}(a, \mathsf{writersToReaders}(L_d))$, and so we have $b \in \mathsf{readers}(a, L_f)$, a contradiction. So $b \notin \mathsf{writers}(a, L_d)$. By a similar argument, $b \notin \mathsf{writers}(a, L_f)$. ∎

Consider the declassification of Alice's bid in the auction example of Section 4.2.2. The label of Alice's bid is $\{\mathit{Alice} \rightarrow (\mathit{au} \searrow^{\mathit{allBids}} \bot);\ \mathit{Alice} \leftarrow \mathit{au} \sqcap \mathit{Bob} \leftarrow \mathit{au}\}$, and it is declassified to the label $\{\mathit{Alice} \rightarrow \bot;\ \mathit{Alice} \leftarrow \mathit{au} \sqcap \mathit{Bob} \leftarrow \mathit{au}\}$. The program counter at the declassification depends only on the variable `i`, and we assume that the condition $\mathit{allBids}$ reveals no more information that the program counter, and so the L_d label is $\{\bot \rightarrow \bot;\ \mathit{Alice} \leftarrow \mathit{au} \sqcap \mathit{Bob} \leftarrow \mathit{au}\}$. Instantiating label

constraints (4.2) and (4.3) for these labels results in the following constraint:

$$\{Alice \rightarrow (au \searrow^{allBids} \bot); Alice \leftarrow au \sqcap Bob \leftarrow au\}$$
$$\sqsubseteq \{au \rightarrow \bot \sqcup Alice \rightarrow au \sqcup Bob \rightarrow au; Alice \leftarrow au \sqcap Bob \leftarrow au\}$$

The integrity policies of both of these labels are identical, and the reader policy of the left hand side ($Alice \rightarrow (au \searrow^{allBids} \bot)$) is less than the reader policy $Alice \rightarrow au$ that is contained in a join on the right hand side, so the constraint is satisfied. This implies that every principal believes that any principal who gains the ability to read Alice's bid is unable to influence either the value declassified or the decision to declassify that value. Thus, Alice believes that if the auctioneer is trusted, the declassification will never reveal anything other than Alice's bid, and it will not occur other than at the appropriate time.

Erasure

Like declassification, erasure changes the confidentiality of information, and similar constraints can be used to check robustness.

Suppose L_f is the label of the information to be erased, L_t is the label of the information after erasure, and L_d is an upper bound on the labels of information contributing to the decision to erase, including the information revealed by the satisfaction of conditions that cause erasure. If, from the perspective of a principal a, the erasure removes information from principal b, then $b \in \mathsf{readers}(a, L_f) - \mathsf{readers}(a, L_t)$. If this is the case, we require that b cannot influence the decision to erase ($b \notin \mathsf{writers}(a, L_d)$), which means that b is unable to delay or prevent the erasure of information.

Note that there is no requirement that b cannot influence the data to be erased ($b \notin \mathsf{writers}(a, L_t)$). For declassification this requirement helped prevent laundering attacks, stopping the attacker from overwriting data to be declassified

with data of his choice. There is no equivalent attack for erasure, as it does not matter if the attacker overwrites data to be erased with his own data: the act of overwriting the data effectively erases it.

To ensure that erasure is regarded as robust from every principal's perspective, the following statement should hold at every declassification:

$$\forall a \in \mathit{Principals}.\ \forall b \in \mathsf{readers}(a, L_f).\ b \in \mathsf{readers}(a, L_t) \text{ or } b \notin \mathsf{writers}(a, L_d) \quad (4.4)$$

The following label constraint entails condition (4.4), and like constraints 4.2 and 4.3, can be verified syntactically.

$$L_t \sqsubseteq L_f \sqcup \mathsf{writersToReaders}(L_d) \quad (4.5)$$

If constraint (4.5) holds, then condition (4.4) holds; that is, every principal a believes that if the erasure removes information from principal b, then b could not have influenced the decision to erase the information.

Lemma 4.13 *If* $L_t \sqsubseteq L_f \sqcup \mathsf{writersToReaders}(L_d)$ *then*

$$\forall a \in \mathit{Principals}.\ \forall b \in \mathsf{readers}(a, L_f).\ b \in \mathsf{readers}(a, L_t) \textit{ or } b \notin \mathsf{writers}(a, L_d).$$

Proof: Assume $L_t \sqsubseteq L_f \sqcup \mathsf{writersToReaders}(L_d)$. Let a be a principal, and let $b \in \mathsf{readers}(a, L_f)$. If $b \in \mathsf{readers}(a, L_t)$ then we are done. Suppose $b \notin \mathsf{readers}(a, L_t)$. By Property 4.5 we have

$$\mathsf{readers}(a, L_t) \supseteq \mathsf{readers}(a, L_f) \cap \mathsf{readers}(a, \mathsf{writersToReaders}(L_d)).$$

If $b \in \mathsf{writers}(a, L_d)$ then by Property 4.11 we have

$$q \in \mathsf{readers}(a, \mathsf{writersToReaders}(L_d)).$$

But then $b \in \mathsf{readers}(a, L_f) \cap \mathsf{readers}(a, \mathsf{writersToReaders}(L_d))$, and so we have $b \in \mathsf{readers}(a, L_t)$, a contradiction. So $b \notin \mathsf{writers}(a, L_d)$. ∎

4.3 Enforcing robustness

In this section, we consider enforcing robustness against all attackers in the setting of a simple imperative language. Myers, Sabelfeld, and Zdancewic (2004) present a type system for a simple imperative language to enforce robustness with respect to a distinguished attacker A; their type system is parameterized on the attacker A. To enforce robustness against all attackers, we could naively require a program to be well-typed in their type system instantiated on every possible attacker A. However, this approach does not work when the set of possible attackers is unknown, or potentially infinite. Instead, we present a type system that incorporates the label constraints of Section 4.2.4, and show that this single type system enforces robustness against all attackers.

The language we consider is IMP_E, from Chapter 3, with the exception that we use labels from the DLM. Figure 4.3 presents the language grammar, including the grammar for labels. The language is similar to that of Myers, Sabelfeld, and Zdancewic (2004), but modifies the syntax for declassification. The syntax of a declassification statement in Myers, Sabelfeld and Zdancewic does not mention the label from which information is being declassified, and does not use conditions for declassification.

The operational semantics for this language are mostly the same as presented in Figure 3.2. Some changes to the semantics are required due to the use of DLM labels. We ensure that a variable that has label L enforced on it is overwritten whenever any erasure policy in L requires it. For example, if a location has the label $\{Alice \to (Bob \nearrow Chuck) \sqcap Dave \to (Alice \overset{d}{\nearrow} \top)\}$ enforced on it, then the location is overwritten whenever either c or d is satisfied. To accomplish this, we extend the definition of $\mathsf{reqErase}(\cdot, \cdot)$ to include DLM labels. The new definition is given in Figure 4.4.

$e ::=$	Expressions
$\quad n$	Integer literal
$\quad x$	Variable
$\quad e_0 \oplus e_1$	Binary operation
$c ::=$	Commands
$\quad \textbf{skip}$	No-op
$\quad x := e$	Assignment
$\quad c_0; c_1$	Sequence
$\quad \textbf{if } e \textbf{ then } c_0 \textbf{ else } c_1$	Selection
$\quad \textbf{while } e \textbf{ do } c$	Iteration
$\quad x := \textbf{declassify}(e, L_f \textbf{ to } L_t \textbf{ using } e_0, \ldots, e_k)$	Guarded declassification
$L ::=$	Labels
$\quad \{\kappa; \iota\}$	Decentralized label
$\kappa ::=$	Confidentiality policies
$\quad \kappa \sqcup \kappa'$	Join confidentiality policy
$\quad \kappa \sqcap \kappa'$	Meet confidentiality policy
$\quad a \rightarrow p$	Reader policy
a, b	Principals
$p, q ::=$	Policies
$\quad a$	Lattice policy
$\quad p \searrow^c q$	Declassification policy
$\quad p \nearrow^c q$	Erasure policy
$\iota ::=$	Integrity policies
$\quad \iota \sqcup \iota'$	Join integrity policy
$\quad \iota \sqcap \iota'$	Meet integrity policy
$\quad a \leftarrow b$	Writer policy

Figure 4.3: Syntax of IMP_E with DLM labels

$$\frac{\mathsf{reqErase}(\kappa, s)}{\mathsf{reqErase}(\{\kappa; \iota\}, s)} \qquad \frac{\mathsf{reqErase}(p, \kappa) \text{ or } \mathsf{reqErase}(p, \kappa')}{\mathsf{reqErase}(\kappa \sqcup \kappa', s)}$$

$$\frac{\mathsf{reqErase}(p, \kappa) \text{ or } \mathsf{reqErase}(p, \kappa')}{\mathsf{reqErase}(\kappa \sqcap \kappa', s)} \qquad \frac{\mathsf{reqErase}(p, s)}{\mathsf{reqErase}(o \rightarrow p, s)}$$

$$\frac{\mathsf{reqErase}(p, s)}{\mathsf{reqErase}(p \searrow^c p', s)} \qquad \frac{\mathsf{reqErase}(p, s)}{\mathsf{reqErase}(p \nearrow^c p', s)} \qquad \frac{s \vDash c}{\mathsf{reqErase}(p \nearrow^c p', s)}$$

Figure 4.4: Definition of $\mathsf{reqErase}(L, s)$

The operational semantics differ from that of Myers, Sabelfeld, and Zdancewic (2004) in two ways. First, the declassification command in this language may evaluate to 0 if the conditions for declassification are not met. Second, all updates to memory enforce erasure.

Typing contexts Γ map each variable to a label that is an upper bound (with respect to $\sqsubseteq$) on the security level of information that can be stored in the variable. The range of Γ is restricted to labels of pairs of confidentiality and integrity policies—$R_{a\to b}$ and $W_{a\leftarrow b}$ are not permitted as security levels of variables.

4.3.1 Defining robustness in IMP$_E$

In order to give a meaningful definition of robustness (and robustness against all attackers) in this language-based setting, we must first define what attacks can be made by an attacker A with power $\langle R_A, W_A\rangle$. Following Myers, Sabelfeld and Zdancewic, we define an attack by A to be a command a that will be inserted into a program. The attack a is not arbitrary code, but is restricted to a subset of the language, to model "fair" attacks. The allowed attacks are defined by the grammar shown in Figure 4.5. For each variable x occurring in an assignment $x := e$, or in an expression e of a fair attack, the attacker must be able to both influence and read the variable,

$$W_A \sqsubseteq \Gamma(x) \sqsubseteq R_A.$$

The allowed attacks do not include declassifications, because if the attacker can declassify confidential information directly, the game is already over.

Attacks may be inserted into the program at points where the attacker is able to influence the execution of code. For example, in a distributed system, the attacker may be able to insert attacks on a server that is under the attacker's

$a ::=$	Fair attacks
skip	No-op
$x := e$	Assignment
$a_0; a_1$	Sequence
if e **then** a_0 **else** a_1	Selection
while e **do** a	Iteration

Figure 4.5: Syntax of fair attacks

control. Myers, Sabelfeld and Zdancewic assume that program points at which an attacker may insert attacks are explicitly marked by *code holes* (•). There may be multiple holes in a command, represented as a vector of holes $\vec{\bullet}$; the holes in a program $c[\vec{\bullet}]$ will be replaced with a vector of attacks $\vec{a}$ to obtain a complete (hole-free) program, written $c[\vec{a}]$. The syntax of commands with holes $c[\vec{\bullet}]$ is shown in Figure 4.6. It extends the syntax of IMP_E commands from Figure 4.3.

$c ::=$	Commands
$\ldots$	IMP_E commands
$[\bullet]$	Hole

Figure 4.6: Syntax of IMP_E with holes

We can now refine Definition 4.8, the definition of robustness, for IMP_E. A configuration is a pair $\langle c, \sigma \rangle$ of command c and memory σ. A memory is a function from variables to integers.

Definition 4.14 (Robustness) *Command $c[\vec{\bullet}]$ has* robustness *with respect to attacks by attacker A with power $\langle R_A, W_A \rangle$ if for all memories σ_1 and σ_2, and all attacks $\vec{a}$ and $\vec{a'}$ by attacker A, if $\langle c[\vec{a}], \sigma_1 \rangle$ and $\langle c[\vec{a}], \sigma_2 \rangle$ are strongly indistinguishable to A, then $\langle c[\vec{a'}], \sigma_1 \rangle$ and $\langle c[\vec{a'}], \sigma_2 \rangle$ are weakly indistinguishable to A.*

This refinement of robustness assumes that the code holes where attacker A may insert code are explicitly given; however, in general, the location of code

holes depends upon which attacker we are considering. Since we are concerned with the possibility of many attackers, we need to reason about the security of code into which different attackers may insert code at different locations.

To indicate where code holes may be inserted for a given attacker A, we assume the existence of a *hole insertion relation* $\lhd_A$. Let $c_0 \lhd_A c_1[\vec{\bullet}]$ denote that the command with holes $c_1[\vec{\bullet}]$ can be obtained by inserting code holes into command c_0 at program points where attacker A is able to insert code. The actual form of the hole-insertion relation depends on the system. For example, in the context of automatic program partitioning (Zheng et al., 2003) (in which a program is automatically partitioned into code segments executed on different servers), an attacker may be able to insert code into any segment that is placed on a server controlled by the attacker.

For our purposes, we require only that the hole insertion relation $\lhd_A$ does not allow holes to be inserted into high-confidentiality contexts. That is, an attacker may not insert code at a program point whose execution depends upon information with a security label not bounded above by R_A. In the context of automatic program partitioning, program points in a high-confidentiality context correspond to code segments whose very execution would insecurely reveal sensitive information to the attacker; such code segments are never placed on a server where the attacker could insert attacks. More formally, we define the property of *safe hole insertion* as follows.

Definition 4.15 (Safe hole insertion) *A hole insertion relation $\lhd_A$ is* safe *if whenever $c_0 \lhd_A c_1[\vec{\bullet}]$, then for all holes in $c_1[\vec{\bullet}]$, if the hole is a subcommand of a command **if** e **then** c **else** c' or a subcommand of a command **while** e **do** c, then for any variable x occurring in e, we have $\Gamma(x) \sqsubseteq R_A$.*

We can now refine Definition 4.10, the definition of robustness against all attackers, for the specific language-based setting presented here.

Definition 4.16 (Robustness against all attackers) *Command c has* robustness against all attackers *if for all principals a and b, and all commands with holes $c'[\vec{\bullet}]$ such that $c \lhd_b c'[\vec{\bullet}]$, command $c'[\vec{\bullet}]$ has robustness with respect to attacker b with power $\langle R_{a\to b}, W_{a\leftarrow b}\rangle$.*

4.3.2 Enforcing robustness in IMP$_E$

Myers, Sabelfeld and Zdancewic present a type system (referred to, for brevity, as the MSZ type system) that is parameterized on a single attacker A and enforces robustness against attacks by A in a simple imperative language. We can adapt the MSZ type system in a straightforward way to account for the differences between their language and IMP$_E$.

Figure 4.7 presents the MSZ type system, adapted for our purposes. The judgment $pc, \Gamma \vdash_A c$ indicates that command c is well typed under typing context Γ and program counter label pc. The attacker A appears in the typing rules for $x :=$ **declassify**$(e, p_f$ **to** p_t **using** $e_0, \ldots, e_k)$ and command holes $[\bullet]$. All other typing rules are standard for an imperative security-typed language.

As in the type system to enforce noninterference according to policy in IMP$_E$, in Chapter 3, we also require the typing context to be well-formed.

Definition 4.17 (Well-formed typing context for attacker A) *Typing context Γ is* well-formed for attacker A *if the overwrite dependency relation $\prec_\Gamma$ is well-founded and for all $x \in Vars$, $\Gamma \vdash_A \Gamma(x)$* label.

Inference rules for the judgment $\Gamma \vdash_A L$ label are given in Figure 4.7. Well-formedness for attacker A ensures two things. First, if the value of variable x

MSZ-SKIP

$$\frac{}{pc, \Gamma \vdash_A \textbf{skip}}$$

MSZ-ASSIGN

$$\frac{\Gamma \vdash e \ : \ L \ \textsf{exp} \quad L \sqcup pc \sqsubseteq \Gamma(x)}{pc, \Gamma \vdash_A x := e}$$

MSZ-SEQUENCE

$$\frac{pc, \Gamma \vdash_A c_1 \quad pc, \Gamma \vdash_A c_2}{pc, \Gamma \vdash_A c_1; c_2}$$

MSZ-WHILE

$$\frac{\Gamma \vdash e \ : \ L \ \textsf{exp} \quad L \sqcup pc, \Gamma \vdash_A c}{pc, \Gamma \vdash_A \textbf{while } e \textbf{ do } c}$$

MSZ-IF

$$\frac{\Gamma \vdash e \ : \ L \ \textsf{exp} \quad L \sqcup pc, \Gamma \vdash_A c_1 \quad L \sqcup pc, \Gamma \vdash_A c_2}{pc, \Gamma \vdash_A \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2}$$

MSZ-PC

$$\frac{pc, \Gamma \vdash_A c \quad pc' \sqsubseteq pc}{pc', \Gamma \vdash_A c}$$

MSZ-HOLE

$$\frac{pc \sqsubseteq R_A}{pc, \Gamma \vdash_A [\bullet]}$$

MSZ-DECLASSIFY-TO-A

$$\frac{\begin{array}{c}\Gamma \vdash e \ : \ L_f \ \textsf{exp} \quad L_t \sqcup pc \sqsubseteq \Gamma(x) \quad I(L_f) \sqsubseteq I(L_t) \\ \forall i \in 0..k.\ \Gamma \vdash e_i \ : \ \Gamma(x) \ \textsf{exp} \\ L_t \sqsubseteq R_A \quad L_f \not\sqsubseteq R_A \quad W_A \not\sqsubseteq pc \qquad W_A \not\sqsubseteq L_f\end{array}}{pc, \Gamma \vdash_A x := \textbf{declassify}(e, L_f \textbf{ to } L_t \textbf{ using } e_0, \ldots, e_k)}$$

MSZ-DECLASSIFY

$$\frac{\begin{array}{c}\Gamma \vdash e \ : \ L_f \ \textsf{exp} \quad L_t \sqcup pc \sqsubseteq \Gamma(x) \quad I(L_f) \sqsubseteq I(L_t) \\ \forall i \in 0..k.\ \Gamma \vdash e_i \ : \ \Gamma(x) \ \textsf{exp} \\ L_t \not\sqsubseteq R_A \text{ or } L_f \sqsubseteq R_A\end{array}}{pc, \Gamma \vdash_A x := \textbf{declassify}(e, L_f \textbf{ to } L_t \textbf{ using } e_0, \ldots, e_k)}$$

MSZ-LABEL-FROM-A

$$\frac{\begin{array}{c}L \sqsubseteq R_A \qquad \forall e \in \textsf{eraseConds}(L).\ \Gamma \vdash e \ : \ L \ \textsf{exp} \\ \forall e \in \textsf{eraseConds}(L).\ (\textsf{erased}(L, e) \sqsubseteq R_A \text{ or} \\ (\Gamma \vdash e \ : \ L_d \ \textsf{exp} \text{ and } W_A \not\sqsubseteq L_d))\end{array}}{\Gamma \vdash_A L \ \textsf{label}}$$

MSZ-LABEL

$$\frac{L \not\sqsubseteq R_A \qquad \forall e \in \textsf{eraseConds}(L).\ \Gamma \vdash e \ : \ L \ \textsf{exp}}{\Gamma \vdash_A L \ \textsf{label}}$$

Figure 4.7: Typing rules for robustness in IMP_E

may cause variable y to be overwritten, then information is permitted to flow from x to y. Second, any erasure of information is robust; this is discussed below.

The key idea of enforcing robustness is to ensure that an attacker A is unable to influence what information is available to A. This idea is expressed in the typing rules for declassification, and in the judgment $\Gamma \vdash_A L$ label.

In the rule MSZ-DECLASSIFY-TO-A, the constraints $W_A \not\sqsubseteq pc$ and $W_A \not\sqsubseteq L'$ ensure that both the decision to declassify and the information to be declassified are high-integrity with respect to the attacker's power. We have adapted the MSZ type system by using two different typing rules for declassifications. The first, MSZ-DECLASSIFY-TO-A, is for declassifications that reveal information to the attacker A; that is, information is declassified from security level L_f (where A cannot read information) to security level L_t (where A can read information). The rule MSZ-DECLASSIFY is for declassifications that do not reveal information to attacker A, either because the attacker could already read information at level L_f, or because after declassification the attacker is still unable to read the information at its new level L_t. Only the first rule needs to enforce the robustness conditions; the original MSZ type system does not contain the second rule, requiring suitably high integrity for all declassifications, even if they do not reveal information to the attacker. We modify their declassification typing rule in anticipation of enforcing robustness against all attackers.

The judgment $\Gamma \vdash_A L$ label ensures that the decision to erase information is high-integrity with respect to the attacker's power. Like declassification, there are two rules. The rule MSZ-LABEL is for labels that the attacker cannot read, and thus erasure does not remove any information from A's observations. Rule MSZ-LABEL-FROM-A is for labels L that the attacker is able to read ($L \sqsubseteq R_A$), but which may be erased to a level that the attacker can no longer read. The function

$$
\begin{aligned}
\mathsf{eraseConds}(\ell) &\triangleq \emptyset \\
\mathsf{eraseConds}(p \searrow^{c} q) &\triangleq \mathsf{eraseConds}(p) \\
\mathsf{eraseConds}(p \not\nearrow^{c} q) &\triangleq \{c\} \cup \mathsf{eraseConds}(p) \\
\mathsf{eraseConds}(o \rightarrow p) &\triangleq \mathsf{eraseConds}(p) \\
\mathsf{eraseConds}(\kappa \sqcup \kappa') &\triangleq \mathsf{eraseConds}(\kappa) \cup \mathsf{eraseConds}(\kappa') \\
\mathsf{eraseConds}(\kappa \sqcap \kappa') &\triangleq \mathsf{eraseConds}(\kappa) \cup \mathsf{eraseConds}(\kappa') \\
\mathsf{eraseConds}(\{\kappa; \iota\}) &\triangleq \mathsf{eraseConds}(\kappa)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{erased}(\ell, c) &\triangleq \ell \\
\mathsf{erased}(p \searrow^{d} q, c) &\triangleq \mathsf{erased}(p, c) \searrow^{d} q \\
\mathsf{erased}(p \not\nearrow^{d} q, c) &\triangleq \mathsf{erased}(p, c) \not\nearrow^{d} q && \text{if } d \neq c \\
\mathsf{erased}(p \not\nearrow^{d} q, c) &\triangleq \mathsf{erased}(p, c) \sqcup \mathsf{erased}(q, c) && \text{if } d = c \\
\mathsf{erased}(o \rightarrow p, c) &\triangleq o \rightarrow \mathsf{erased}(p, c) \\
\mathsf{erased}(\kappa \sqcup \kappa', c) &\triangleq \mathsf{erased}(\kappa, c) \sqcup \mathsf{erased}(\kappa', c) \\
\mathsf{erased}(\kappa \sqcap \kappa', c) &\triangleq \mathsf{erased}(\kappa, c) \sqcap \mathsf{erased}(\kappa', c) \\
\mathsf{erased}(\{\kappa; \iota\}, c) &\triangleq \{\mathsf{erased}(\kappa, c); \iota\}
\end{aligned}
$$

Figure 4.8: Definition of $\mathsf{eraseConds}(\cdot)$ for labels and $\mathsf{erased}(\cdot, \cdot)$

$\mathsf{erased}(L, c)$ indicates the security level that must be enforced on information labeled L once condition c is satisfied. If for some condition $e \in \mathsf{eraseConds}(L)$, we have $\mathsf{erased}(L, e) \not\sqsubseteq R_A$, the attacker will not be able to read the information once e is satisfied. In that case, the attacker should not be able to influence the decision to erase the information. That is, the satisfaction of condition e should be influenced by the attacker ($\Gamma \vdash e : L_d\ \mathsf{exp}$ and $W_A \not\sqsubseteq L_d$). Definitions of $\mathsf{eraseConds}(L)$ and $\mathsf{erased}(L, e)$ are given in Figure 4.8.

The rule for command holes restricts holes from occurring in high-confidentiality contexts, which ensures that an attacker is unable to observe sensitive information through implicit flows (Denning and Denning, 1977).

Theorem 4.18 *If $pc, \Gamma \vdash_A c$ and Γ is well-formed for attacker A, then command c has robustness with respect to attacker A.*

Proof: Similar to Myers, Sabelfeld and Zdancewic's, adapted for the modified declassification typing rules, and the novel semantics of IMP$_E$ (run-time mechanisms to enforce declassification and erasure). ∎

4.3.3 Enforcing robustness against all attackers in IMP$_E$

To enforce robustness against all attackers, we derive a type system using constraints (4.2), (4.3), and (4.5), given in Section 4.2.4. This type system ensures that for all principals a and b, a well-typed program is robust against attacks by b with power $\langle R_{a\to b}, W_{a\leftarrow b}\rangle$. We prove this by showing that a well-typed program is also well-typed in the MSZ type system instantiated on the attacker b with power $\langle R_{a\to b}, W_{a\leftarrow b}\rangle$.

The new typing judgments are $\Gamma, pc \vdash c$ (command c is well-typed under program counter label pc and variable context Γ) and $\Gamma \vdash L\ \mathsf{label}$ (label L is

$$\text{TR-SKIP}\quad \frac{\Gamma \vdash pc\ \mathsf{label}}{\Gamma, pc \vdash \mathbf{skip}}$$

$$\text{TR-ASSIGN}\quad \frac{\Gamma \vdash e\ :\ L_e\ \mathsf{exp} \quad L_e \sqcup pc \sqsubseteq \Gamma(x) \quad \Gamma \vdash pc\ \mathsf{label}}{\Gamma, pc \vdash x := e}$$

$$\text{TR-SEQUENCE}\quad \frac{\Gamma, pc \vdash c_0 \quad \Gamma, pc \vdash c_1}{\Gamma, pc \vdash c_0; c_1}$$

$$\text{TR-WHILE}\quad \frac{\Gamma \vdash e\ :\ L_e\ \mathsf{exp} \quad pc', \Gamma \vdash c\ \mathsf{com} \quad \Gamma \vdash pc\ \mathsf{label} \quad L_e \sqcup pc \sqsubseteq pc'}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c}$$

$$\text{TR-IF}\quad \frac{\Gamma \vdash e\ :\ L_e\ \mathsf{exp} \quad pc', \Gamma \vdash c_0\ \mathsf{com} \quad pc', \Gamma \vdash c_1\ \mathsf{com} \quad \Gamma \vdash pc\ \mathsf{label} \quad L_e \sqcup pc \sqsubseteq pc'}{\Gamma, pc \vdash \mathbf{if}\ e\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1}$$

$$\text{TR-DECLASSIFY}\quad \frac{\begin{array}{c}\Gamma \vdash e\ :\ L_f\ \mathsf{exp} \quad L_t \sqcup pc \sqsubseteq \Gamma(x) \quad \Gamma \vdash pc\ \mathsf{label} \\ \forall i \in 0..k.\ \Gamma \vdash e_i\ :\ \Gamma(x)\ \mathsf{exp} \qquad e_0, \ldots, e_k \vdash L_f \sqsubseteq L_t \\ L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(pc) \qquad L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_f)\end{array}}{\Gamma, pc \vdash x := \mathbf{declassify}(e, L_f\ \mathbf{to}\ L_t\ \mathbf{using}\ e_0, \ldots, e_k)}$$

$$\text{TR-LABEL}\quad \frac{\begin{array}{c}\forall e \in \mathsf{eraseConds}(L).\ \Gamma \vdash e\ :\ L\ \mathsf{exp} \\ \forall e \in \mathsf{eraseConds}(L).\ \Gamma \vdash e\ :\ L_d\ \mathsf{exp}\ \text{and} \\ \mathsf{erased}(L, e) \sqsubseteq L \sqcup \mathsf{writersToReaders}(L_d)\end{array}}{\Gamma \vdash L\ \mathsf{label}}$$

Figure 4.9: Typing rules for robustness against all attackers in IMP_E

well-typed under variable context Γ). Figure 4.9 presents the inference rules for the new judgments, which are based on the rules for enforcing noninterference according to policy in IMP_E (Figure 3.4). The rules are modified to use decentralized labels instead of erasure and declassification policies. In addition, the rule for declassification, TR-DECLASSIFY, incorporates constraints (4.2) and (4.3), and the rule for labels, TR-LABEL incorporates constraint (4.5).

The judgment $\Gamma \vdash L$ label is used to ensure that typing contexts are well-formed.

Definition 4.19 (Well-formed typing context for robustness) *Typing context* Γ *is* well-formed for robustness *if the overwrite dependency relation* $\prec_\Gamma$ *is well-founded and for all* $x \in Vars$, $\Gamma \vdash \Gamma(x)$ label.

The inference rules for the new typing judgments contain no negated label ordering relations ($\not\sqsubseteq$), which is consistent with having only partial knowledge of the *acts-for* relation in effect at run time.

Unlike the rules for enforcing robustness with respect to attacker A (Figure 4.7), there is no need for a rule for command holes, as we are only concerned with complete programs; holes are introduced through hole insertion relations $\lhd_A$.

Theorem 4.20 *If* $\Gamma, pc \vdash c$ *and* Γ *is well-formed for robustness then command* c *has robustness against all attackers.*

Proof: Let $\Gamma, pc \vdash c$, and Γ be well-formed for robustness. Let a and b be principals, and let $d[\vec{\bullet}]$ be a command with holes such that $c \lhd_b d[\vec{\bullet}]$. For attacker $A = b$ with power $\langle R_{a\to b}, W_{a\leftarrow b}\rangle$, we show that $pc, \Gamma \vdash_A d[\vec{\bullet}]$, and Γ is well-formed for attacker A, Thus by Theorem 4.18, $d[\vec{\bullet}]$ has robustness with respect to attacker b with power $\langle R_{a\to b}, W_{a\leftarrow b}\rangle$.

We first show $pc, \Gamma \vdash_A c$, by induction on $\Gamma, pc \vdash c$. The only interesting case is for declassification. If the declassification doesn't reveal information to A (i.e., either $L \not\sqsubseteq R_A$ or $L' \sqsubseteq R_A$), then the declassification type-checks by MSZ-DECLASSIFY. If it does reveal information to A, then by definition of $R_{a \to b}$ and $W_{a \leftarrow b}$ and Lemma 4.12 we have $W_{a \leftarrow b} \not\sqsubseteq pc$ and $W_{a \leftarrow b} \not\sqsubseteq L'$, and so the statement type-checks by MSZ-DECLASSIFY-TO-A.

Similarly, to show that Γ is well-formed for attacker A, consider a label L such that $\Gamma \vdash L$ label. If the attacker cannot observe information labeled L ($L \not\sqsubseteq R_{a \to b}$) then $\Gamma \vdash_A L$ label by MSZ-LABEL. Otherwise, the attacker can observe information labeled L ($L \sqsubseteq R_{a \to b}$). For any condition $e \in \mathsf{eraseConds}(L)$ we have $\mathsf{erased}(L, e) \sqsubseteq L \sqcup \mathsf{writersToReaders}(L_d)$, where $\Gamma \vdash e : L_d$ exp. By definition of $R_{a \to b}$ and $W_{a \leftarrow b}$ and Lemma 4.13 we have either that $\mathsf{erased}(L, e) \sqsubseteq R_{a \to b}$ or $W_{a \leftarrow b} \not\sqsubseteq L_d$, and so the statement type-checks by MSZ-LABEL-FROM-A.

Given that $pc, \Gamma \vdash_A c$ and $c \lhd_b d[\vec{\bullet}]$, we can use the hole safety of $\lhd_b$ to show that $pc, \Gamma \vdash_A d[\vec{\bullet}]$, since code holes can only be introduced in low-confidentiality contexts, and thus any hole type-checks. ∎

CHAPTER 5

DECLASSIFICATION, ERASURE, AND ROBUSTNESS IN JIF

The Jif programming language (Myers, 1999; Myers et al., 2001–2008) extends Java (Gosling et al., 2000) with information-flow control, allowing security policy annotations on program variables and method signatures. Jif aims to provide a practical programming model for end-to-end security enforcement, and supports a large subset of Java. In this chapter, we describe how we extend Jif with declassification and erasure policies, and mechanisms to enforce these policies,. This chapter also describes the benefits obtained by using Jif_E to implement Civitas (Clarkson et al., 2008), a secure remote voting service,

5.1 Syntax and semantics

Security policies in Jif are from the decentralized label model (DLM) (Myers and Liskov, 2000). Jif_E uses security policies from the DLM extended with declassification and erasure policies, as described in Chapter 4. Jif_E extends Jif's syntax and run-time system to incorporate the guarded declassification syntax and run-time erasure mechanisms of Chapter 3.

5.1.1 Decentralized label model

In Chapter 4, declassification and erasure policies were incorporated into the DLM using as the base lattice principals ordered by the *acts-for* relation. However, the language for conditions was left unspecified.

For the condition language in Jif_E, we allow a restricted class of expressions: *access path expressions* of type `condition`, and negations of these access path expressions. The type `condition` is a new primitive type with two values: `true` and `false`. Expressions of type `condition` may be cast to `boolean`, and

vice versa. An access path expression is an expression of the form $r.f_1.\ldots.f_n$, where r is a local variable, the special variable `this`, or a class name; each f_i is a field; and all path elements other than the last are declared `final`. Immutability of path elements is needed for sound reasoning about conditions within the type system.

5.1.2 Declassification and erasure mechanisms

Jif$_E$ contains the new guarded declassification expression **declassify**(e, L_f **to** L_t **using** $e_0, \ldots, e_k$), where L_f and L_t are labels, and each expression e_i is of type `condition`. The expression is evaluated by first evaluating e to a value v, then evaluating each e_i in turn; if any e_i evaluates to `false`, then an `UnsatisfiedConditionException` is thrown; otherwise, the expression evaluates to v. If the evaluation of e or any e_i results in an exception, the declassification expression also results in the exception. As in the typing rule for declassification TR-DECLASSIFY in Figure 4.9, type checking ensures that L_f may be relabeled L_t under the assumption that all conditions e_i are satisfied.

Note that Jif already provides a mechanism for *selective declassification* (Myers and Liskov, 1997; Myers, 1999; Pottier and Conchon, 2000), whereby a declassification that weakens or removes a policy owned by principal o requires o's authority. By contrast, guarded declassification does not require the authority of any principal, since given a reader policy such as $o \rightarrow (p \searrow^c q)$, the principal o has already stated that information may be declassified when condition c is satisfied. In Jif$_E$, selective declassification and guarded declassification coexist as separate and independent mechanisms. Both mechanisms incorporate label constraints to enforce robustness against all attackers.

To enforce erasure policies, Jif$_E$ ensures that a variable or location that has label L enforced on it is overwritten whenever any erasure policy in L requires it. For example, if a location has the label $\{Alice \rightarrow (Bob \overset{\texttt{this.f}}{\nearrow} Chuck) \sqcap Dave \rightarrow (Alice \overset{\texttt{this.o.d}}{\nearrow} \top)\}$ enforced on it, then the location is overwritten whenever either `this.f` or `this.o.d` evaluates to `true`. When a location or variable is overwritten, its contents are replaced with an appropriate default value. Thus, numeric locations are overwritten with zero, and reference locations are overwritten with `null`. Section 5.3 describes the run-time mechanisms used to achieve this. This erasure mechanism is analogous to the erasure mechanism of IMP$_E$, which overwrites variables if the policy enforced on the variable requires erasure.

5.1.3 Interaction with Java and Jif features

Jif is intended for practical information-flow control. It supports a large subset of Java's language features, and provides additional features such as dynamic labels, constant arrays, and class and method polymorphism, needed for building real applications. The erasure enforcement mechanism of IMP$_E$ needs careful adaptation for these language features.

Final fields and variables. In Java, fields, local variables, and formal arguments can be marked `final`, meaning their value will not change after initialization. To respect the finality of variables and locations, Jif$_E$ requires that final variables and fields cannot be overwritten. The label L enforced on a final field or variable must not contain any erasure policies, and if L contains a dynamic label (see below), then the dynamic label must not contain any erasure policies. This ensures that label L never requires erasure.

Arrays. Jif allows different labels to be enforced on the elements of an array and the array itself. If the label enforced on the elements of an array requires erasure, the array is overwritten with appropriate default values; the length of the array is not altered. Jif supports *constant arrays*, whose elements cannot be modified after initialization. As with `final` fields, labels on elements of constant arrays must never require erasure.

Dynamic labels. Jif can represent labels at runtime and can treat labels as first-class values. The primitive type `label` is the type of run-time labels, and Jif permits run-time comparisons of dynamic labels. Jif$_E$ extends the run-time representation of labels to permit declassification and erasure policies also to be represented at runtime.

Jif extends the DLM to allow dynamic labels to appear in labels. For example, in Figure 5.1, the label that must be enforced on the variable i is the run-time value of the dynamic label `lbl`. This dynamic label may be different in different executions of the program.

Dynamic labels may be used to label fields, variables, and arrays. However, final fields and variables, and elements of constant arrays must never require erasure. We introduce a new kind of label, to reason about run-time labels that may require erasure. The primitive type `elabel` is used for dynamic labels that may require erasure. Only dynamic labels of type `elabel` may contain erasure policies; a dynamic label of type `label` cannot contain erasure policies. Thus, the labels of final fields, final variables, and elements of constant arrays, may refer to dynamic labels of type `label`, but may not refer to dynamic labels of type `elabel`. The type `label` can be cast to `elabel`, but not vice versa. The restriction that only `elabel`s may contain erasure policies also simplifies backwards compatibility of Jif$_E$ with Jif.

Polymorphism. Jif provides polymorphism for the labels of method arguments. For example, the method signature `double{`a`} sine(double{`$Alice \rightarrow Bob$`} a)` states that the label on the value returned is the same as the label of the actual argument `a`, which can be no more restrictive than $\{Alice \rightarrow Bob\}$. In Jif method bodies, the label of a formal argument is a polymorphic label, representing the label of actual argument, and bounded above by the argument label specified in the signature. However, because actual arguments may require erasure during the method body execution, we need to know what label to enforce on formal arguments in the method body. Thus, in Jif$_E$, method bodies assume that the label of a formal argument is simply the argument label bound specified in the signature. This is sound, but not as permissive as Jif, and effectively removes argument label polymorphism. However, it is not overly restrictive: we successfully implemented a remote voting system in about 13,000 lines of Jif$_E$ code, as discussed in Section 5.4.

Jif also supports polymorphic classes, permitting classes to be parameterized on labels and principals.[1] Jif$_E$ extends the class parameters to allow parameters of type `elabel`.

5.2 Tracking information flow

Jif's existing type system tracks information flow. Jif$_E$ extends Jif's run-time mechanisms, overwriting variables, and allowing declassification based on the satisfaction of condition. Thus, new information flows are introduced. Jif$_E$ extends Jif's type system to soundly track and restrict these new information flows, by adapting the typing rules of IMP$_E$.

[1] Jif as of version 3.1 does not support Java generics, another form of class parameterization for polymorphism.

5.2.1 Condition satisfaction

Condition satisfaction affects whether the expression **declassify**(e, L_f **to** L_t **using** $e_0, \ldots, e_k$) throws an `UnsatisfiedConditionException` or successfully declassifies e. Jif$_E$ requires that the label of each e_i is no more restrictive than label L_t.

Condition satisfaction may also cause variables and locations to be overwritten. Jif$_E$ tracks these information flows analogously to the IMP$_E$ label typing judgment $\Gamma \vdash L$ label. Jif$_E$ requires that for all labels L declared in a program, and for any erasure policy $p \overset{e}{\nearrow} q$ that occurs in L, the label of expression e must be no more restrictive than L. Jif$_E$ also requires that if `lbl` is a dynamic label that occurs in L, then the value `lbl` must be no more restrictive than L. So, if e is a condition that appears in `lbl`, then the label of e is no more restrictive than `lbl`, and thus no more restrictive than L.

5.2.2 Robustness

The IMP$_E$ typing rules to enforce robustness against all attackers (Figure 4.9) are also incorporated into the type system of Jif$_E$. A declassification expression **declassify**(e, L_f **to** L_t **using** $e_0, \ldots, e_k$) is restricted to ensure that $L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(L_f)$ and $L_f \sqsubseteq L_t \sqcup \mathsf{writersToReaders}(pc)$ hold, where pc is the label of the program counter at the declassification expression. Similarly, for any condition e that may case a label L to require erasure, the constraint $\mathsf{erased}(L, e) \sqsubseteq L \sqcup \mathsf{writersToReaders}(L_d)$ must hold, where L_d is an upper bound on the label of condition e.

The function $\mathsf{writersToReaders}(\cdot)$ must be extended to account for the additional labels in Jif, such as dynamic labels and polymorphic method argument labels.

```
1  final label{⊥→⊥;⊤←⊤} lbl = ...;
2  int{*lbl} i = ...;
3  if (lbl ⊑ {Alice→Bob ; Alice←Chuck}) {
4      int j = declassify(i, {⊥→⊥;Alice←Chuck});
5  }
```

Figure 5.1: Example of a dynamic label upper bound

In extending writersToReaders$(\cdot)$ to dynamic labels, we must ensure that Property 4.11 continues to hold: for all principals p and q, and labels L, if $q \in$ writers(p, L), then $q \in$ readers$(p,$ writersToReaders$(L))$. Clearly, for any dynamic label `lbl` it is safe to define writersToReaders$(\texttt{lbl})$ to be $\{\bot \rightarrow \bot; \top \leftarrow \top\}$; since readers$(p,\{\bot \rightarrow \bot;\top \leftarrow \top\})$ is the set of all principals, this would satisfy Property 4.11. While imprecise, this is the best we can do without additional information about the dynamic label.

However, in some situations, a given dynamic label has an upper bound. For example, at the declassification expression in line 4 of Figure 5.1, we know statically that $\{Alice \rightarrow Bob; Alice \leftarrow Chuck\}$ is an upper bound on the dynamic label `lbl`, because of the run-time label test on line 3.

If L is an upper bound for dynamic label `lbl`, we have $\texttt{lbl} \sqsubseteq L$, and so for any principal p, by definition of $\sqsubseteq$, we have writers$(p, \texttt{lbl}) \subseteq$ writers(p, L). Thus, if L consists only of reader and writer policies, then writersToReaders(L) is a conservative approximation for writersToReaders$(\texttt{lbl})$, and if $q \in$ writers$(p, \texttt{lbl})$, then $q \in$ readers$(p,$ writersToReaders$(L))$.

We extended this technique for approximating writersToReaders$(\cdot)$ to other labels, such as class parameters and polymorphic argument labels.

```
1 class C {
2   condition{} c = false;
3   final label{} lbl = new label {Alice→(⊥ this.c↗⊤)};
4   void m{}() {
5     String{*lbl; Alice→(⊥ this.c↗⊤)} x;
6     x = "Hello world!";
7     this.c = true;
8   }
9 }
```

Figure 5.2: Example translation source

5.3 Translation

The Jif compiler (Myers et al., 2001–2008) is a source-to-source compiler, producing Java code as output. Jif programs rely on a small trusted run-time library, implemented in Java, that provides functionality such as run-time comparisons of labels. We extend the run-time library, and modify the source-to-source translation, to provide run-time support for erasure. Figure 5.2 and Figure 5.3 show the Jif source code and Java target code, respectively, of an example translation. We refer to these figures in the description of translation below.

The key idea is that if a variable or location may need to be overwritten depending on the satisfaction of a condition c, then a listener is registered with condition c; the listener is notified whenever the value of c changes, and the listener will overwrite the variable or location if necessary.

Listeners are implementations of the interface `until.lang.ConditionListener`, which contains a single method `void conditionChange()`. Implementations of `ConditionListener.conditionChange()` simply check if erasure of a location is required, and if so, write a suitable default value into the location.

A field f of type `condition` in a class C is translated to a `boolean` field (e.g., Figure 5.2 line 2, Figure 5.3 line 2), and two methods added to the class

```
1  class C implements ConditionContainer {
2    boolean c = false;
3    jif.lang.Label lbl;
4
5    void m() {
6      final LocalPlaceHolder y = new LocalPlaceHolder() {
7          public void conditionChange() {
8            if (C.this.get_c() ||
9                   LabelUtil.requiresErasure(C.this.lbl))
10             this.erase();
11         }
12       };
13     this.register_c(y);
14     LabelUtil.register(this.lbl, y);
15     y.set(ConditionChecker.condCheckSilent("Hello world!",
16              !this.get_c() &&
17                     !LabelUtil.requiresErasure(this.lbl)));
18     this.set_c(true);
19   }
20
21   // setter and listeners for c
22   boolean set_c(boolean v) {
23     this.c = v;
24     for (ConditionListener l : this.listeners_c)
25       l.conditionChange();
26     return v;
27   }
28   final Set listeners_c = new LinkedHashSet();
29   void register_c(ConditionListener l) {
30     this.listeners_c.add(l);
31   }
32
33   // ConditionContainer methods
34   boolean accessCondition(String condName) {
35     if ("c".equals(condName)) return this.c;
36     throw new Error("Unknown condition " + condName);
37   }
38   void register(String condName, ConditionListener e) {
39     if ("c".equals(condName)) this.register_c(e);
40   }
41 }
```

Figure 5.3: Example translation target

C: a registration method `void register_`f`(until.lang.ConditionListener l)`, and a setter method `boolean set_`f`(boolean v)` (e.g., Figure 5.3 lines 31–31). The registration method adds the listener to a set of registered listeners. All assignments to the field are translated to calls to the setter method (e.g., Figure 5.2 line 7, Figure 5.3 line 18), which modifies the field, and then notifies all registered listeners.

If a local variable may need to be overwritten, then the translation moves the local variable to the heap, to allow a condition listener to access it (and overwrite it) as needed.

Assignments to fields and local variables are translated to check that the variable or field does not currently require erasure (e.g., Figure 5.2 line 6, Figure 5.3 lines 15–17). The combination of condition listeners and assignment checks ensures that whenever the label enforced on the variable or location requires erasure, the variable or location will be zero or `null` as appropriate.

Overwriting a variable or location of type `condition` may trigger the overwriting of other variables and locations. To ensure that updating a condition does not cause an infinite cascade of listener invocations, the type system of Jif$_E$ requires that for all conditions c, the value of c cannot (directly or indirectly) control whether c needs to be overwritten. This is analogous to ensuring that the overwrite dependency relation $\prec_\Gamma$ of Chapter 3 is well-founded.

5.4 Case study: Civitas

Using Jif$_E$, we implemented Civitas (Clarkson et al., 2008), a practical, secure, remote voting system. The use of declassification and erasure policies in the implementation of Civitas help ensure that the system's security requirements

are satisfied. This section discusses the experience of using Jif$_E$ to implement Civitas.

Civitas guarantees strong security properties in the presence of a strong adversary. The design of Civitas refines a cryptographic voting scheme by Juels et al. (2005). The entities involved in a Civitas election include an election supervisor, voters, and *election authorities*, which are mutually distrusting entities that collaborate to run an election. A Civitas election has several phases.

1. *Setup.* The electoral roll is established and shared keys are generated.
2. *Registration.* Voters retrieve credentials from election authorities.
3. *Voting.* Voters vote using their credentials.
4. *Tabulation.* Election authorities tabulate the election results.

More details of the design and security assurances of Civitas are available in the Civitas technical report (Clarkson et al., 2007).

Civitas is implemented in 14,000 lines of Jif$_E$ code, with about 8,000 additional lines of Java code to perform I/O and implement cryptographic operations. Declassification and erasure policies are used in four distinct places.

- *Generation of a shared key by authorities.* During setup, authorities engage in a protocol to generate a shared ElGamal key pair. Each authority generates a share of the key pair, and publishes a commitment to it. Each authority publishes its share of the public key, but only after all commitments are published.

  The label $\{A_i \rightarrow A_i \searrow^{allCommPosted} \bot; A_i \leftarrow A_i\}$ is used for authority A_i's public key share. The declassification policy requires that initially the information is readable only by election authority A_i, and may be declassified to be readable by everyone (represented by the bottom principal $\bot$) when condition $allCommPosted$ is satisfied. Condition $allCommPosted$ is a field of

type `condition`. It is easy to check that this field is only updated once A_i has successfully retrieved all key commitments. The writer policy $A_i \leftarrow A_i$ indicates that the key share was influenced only by A_i.

- *Commit-reveal protocol by authorities.* During tabulation, the authorities must jointly generate random bits, and each authority must believe that the bits are random. Each authority selects random bits, and publishes a commitment to these bits. Once all commitments are published, each authority reveals its bits, which can be combined to form a sequence of bits that all authorities agree are random.

  Similar to the key shares, the label $\{A_i \rightarrow A_i \searrow^{allBitsPosted} \bot; A_i \leftarrow A_i\}$ is used for authority A_i's random bits. Condition $allBitsPosted$ is a field of type `condition`, and it is easy to check that this field is only updated once A_i has been able to successfully retrieve all bit commitments.

- *Management of credential shares by authorities.* During registration, each authority generates a credential share for each voter. Each voter contacts each authority to retrieve his shares, combining them into a credential that can be used to vote. After delivering the share to the voter, the authority removes the share from the system. This helps ensure that the voter's anonymity is not violated should A_i be subsequently compromised.

  Authority A_i enforces the label $\{A_i \rightarrow (A_i \overset{delivered}{\nearrow} \top) \searrow^{deliveryReq} \bot; A_i \leftarrow A_i\}$ on each voter credential share. Condition $deliveryReq$ is satisfied when the voter has requested his credential share, and has authenticated himself to the authority. The satisfaction of this condition allows the declassification of the share.[2] Any copies of the information that were not declassified must

[2]Ideally, the declassification policy should allow the share to be readable only by the voter V_j it is intended for. In the protocol between authority A_i and V_j, each authenticates to the other, and they establish a shared key k; the credential share is sent to V_j encrypted with k. The reasoning supported by the DLM is not powerful enough to determine that information encrypted with k

be erased when condition $delivered$ is satisfied upon successful retrieval by the voter.

- *Management of voter credential shares by voting clients.* After voter V_j has retrieved all credential shares from the authorities, he combines them into a single credential, which he then uses to vote, publishing it together with his ballot. After combining the shares, the voter deletes them, to remove any record of which authority provided which share.

  The voter enforces the label $\{V_j \rightarrow (V_j \overset{postCombined}{\nearrow} \top) \overset{combined}{\searrow} \bot\}$ on each credential share. Upon combining the shares into a credential, condition $combined$ is satisfied, and the voter can declassify the credential to allow it to be published with his ballot. After combining shares, condition $postCombined$ is satisfied, and undeclassified copies of the shares (or of information derived from them) are erased.

Jif$_E$ allows complex declassification and erasure security requirements to be clearly and unambiguously declared on the data. In addition to stating what security must currently be enforced on the data, the policies limit how the data may be used in the future. The information flow analysis ensures that uses of the data conform to the declared security policies. This provides additional assurance that the Civitas implementation is correct. The policy annotations serve as a form of documentation, making complex information security requirements visible in the code itself.

is readable only by A_i and V_j. Extending it to reason about the subtleties of cryptography would allow a more precise declassification policy, but is largely orthogonal to this work.

CHAPTER 6

RELATED WORK

This dissertation presents expressive security policies for declassification and erasure, and demonstrates how to enforce them end-to-end, using information-flow control. In this chapter we summarize related work on information-flow control, focusing on efforts to make information security more practical, and to express and enforce declassification and erasure requirements. We also consider expressive security policies and models, beyond information flow.

6.1 Information-flow control

Information-flow control is a mechanism that can provide end-to-end enforcement of confidentiality. Seminal work by Bell and La Padula (1973) spurred the investigation of information-flow policies, and information-flow control. In Bell and La Padula's model, objects in the system are associated with security classes. Information may flow freely within a security class, but objects cannot read information from objects in more restrictive security classes, nor write information to objects in less restrictive classes ("no read up, no write down").

More generally, information-flow control techniques label data with security levels; as data are updated and created, the security labels are also updated to reflect data dependencies. The labels can be used to restrict sensitive operations and actions within the system. Denning (1976) proposes the lattice model of information flow, whereby the labels form a lattice structure. This model provides a unifying view of systems that perform information-flow control.

What does it mean to successfully enforce confidentiality end-to-end? Goguen and Meseguer (1982) answer that question by defining *noninterference*. The intuition behind noninterference is that the actions of a user with high-security

clearance should have no effect on the observations of a user with lower security clearance.

In the quarter-century since the introduction of noninterference, many similar definitions of information security have been proposed. There is little agreement on which definition of security is the correct one to use, either in general, or in specific settings. Work on semantic security conditions for information-flow control can be broadly divided into two camps: language-based security and process calculi-based security.

Language-based security is concerned with the security of implementations, and typically takes a "data-oriented" approach, seeking to prevent secret data from being leaked. Process calculi-based security is mostly concerned with the security of system specifications (as expressed in process calculi), and typically takes an "event-oriented" approach, seeking to preventing the occurrence of secret events being known by public observers.[1] Focardi and Gorrieri (2001) provide a classification of process algebra-based security conditions; and Sabelfeld and Myers (2003) survey language-based information-flow security. This dissertation takes a language-based approach to enforcing expressive security requirements; in the remainder of this chapter, we focus on language-based security.

6.1.1 Language-based information-flow control

The most common forms of information-flow control use dynamic mechanisms. However, comparatively little work considers dynamic information-flow control at the language level of abstraction. The Perl scripting language allows

[1]Some work attempts to bridge this divide: Mantel and Sabelfeld (2001) show equivalent notions of security for multi-threaded programs and event-based models of multi-threaded program, and Castellani (2007) extends their results; von Oheimb (2004) combines both views into a new security condition *noninfluence*; and Focardi et al. (2005) show formal links between language-based and process calculi-based information security.

programs to be run in *taint mode*, which attempts to ensure that "tainted" data from untrusted sources is not used in potentially unsafe system calls. Ferrari et al. (1997) consider information flow in object-oriented systems. Lam and Chiueh (2006) provide a dynamic information-flow tracking framework for C programs. McCamant and Ernst (2007) present a proof technique for bounding the amount of information leaked by a dynamic information-flow analysis on a simple imperative language. Shroff et al. (2007) dynamically track dependencies over multiple runs of a program to accurately track information flow in a lambda calculus.

There is work on dynamic control of information flow at lower levels of abstraction, including the operating system (SELinux; Efstathopoulos et al., 2005; Zeldovich et al., 2006; Krohn et al., 2007; Nightingale et al., 2008), virtual machine (Nair et al., 2007), and architecture (Fenton, 1973, 1974; Suh et al., 2004; Vachharajani et al., 2004).

One of the problems with dynamic enforcement of information-flow control is that information flow is not a predicate on a single execution trace, but rather a predicate on sets of traces (Schneider, 2000). Dynamic techniques typically do not consider executions other than the current one, and thus do not restrict information flows that arise due to the non-occurrence of events. For example, dynamic techniques typically do not soundly prevent implicit flows (Denning and Denning, 1977), where information flows via the control structure of a program.[2] Another problem is the often prohibitively high run-time overhead of label tracking.

Denning and Denning (1977) were the first to observe that compile-time, or *static*, analysis facilitates sound reasoning about information flow. They present

[2] A recent exception is the work of Shroff et al. (2007), who prove noninterference results for their dynamic information-flow control technique.

a program analysis, and show that if the analysis succeeds, then no execution of the program will produce an illegal flow of information. An additional benefit of static information-flow analyses is that they can remove the need to track labels at runtime, and reduce many of the run-time information-flow control. Feiertag (1980) and McHugh and Good (1985) developed static tools for reasoning about information flow in programs; both use theorem provers to discharge proof obligations.

Volpano et al. (1996) formulate Denning and Denning's analysis as a type system, and show soundness of the type system: well-typed programs satisfy noninterference. Subsequently, much work has focused on type systems for sound information-flow control. Many of these type systems are described in the survey of language-based information-flow security by Sabelfeld and Myers (2003).

Although IMP_E and Jif_E use run-time mechanisms to enforce erasure and declassification requirements, information-flow control in both languages is static, using a type system to track and restrict the flow of information.

6.1.2 Practical enforcement

A goal of this dissertation is to make strong information security more practical. In this vein, it furthers work on practical information-flow control. Myers developed the Jif[3] programming language (Myers, 1999; Myers et al., 2001–2008), which extends Java with information-flow control. The price of extending a complex and practical programming language such as Java is that proofs of the correctness of information flow become infeasible. Jif uses labels from the decentralized label model (Myers and Liskov, 2000; see also Chapter 4), a flexible

[3]The Jif programming language was originally called JFlow.

open-ended model for expressing security concerns, and provides mechanisms for declassification—an important requirement for real applications. In this dissertation, we extended both the decentralized label model, and the Jif programming language with support for declassification and erasure.

Simonet (2003) also extends a practical programming language, Objective Caml (Leroy et al., 2004), with information-flow control; the resulting language is called Flow Caml. In line with other ML-style language, Flow Caml provides full type inference (Pottier and Simonet, 2002). Flow Caml provides tools to visualize the information flows expressed by types, but does not provide any mechanism for declassification.

Praxis High Integrity System's language SPARK (Barnes, 2003) is based on a subset of Ada, and adds information-flow analysis. SPARK checks simple dependencies within procedures using sets of input variables as the security labels. Thus, the security lattice for a given procedure is the power set of the procedure's input variables. SPARK has been used to implement commercial systems, including air traffic control systems, avionics systems, and certificate authorities.

Deng and Smith (2004) modify the semantics of array accesses and arithmetic operations to make typing for information-flow control more permissive. They define an out-of-bound read to an array to evaluate to zero. That is, the expression $a[i]$ evaluates to zero when either i is less than zero, or greater than the length of the array a. Similarly, an out-of-bound write to an array is a no-op, and division by zero evaluates to zero. Thus, no errors are encountered during the execution of a program, so there is no information flow due to exceptions or errors. The type system can thus be simpler, and more permissive.

Hicks et al. (2007) develop Jifclipse, an extension to the Eclipse development

environment that supports writing Jif programs. Jifclipse integrates Jif's security annotations into the Eclipse development environment, allowing a programmer to navigate and view security annotations and constraints. Jifclipse can also make "quick fix" suggestions to the programmer, automating simple modifications to resolve or remove illegal information flows.

Smith and Thober (2007) provide a type-inference algorithm to facilitate the writing of security-typed code in a Java-like calculus. The programmer must provide label annotations only where data enters and leaves the system; all remaining annotations are inferred. A highly polymorphic type system permits precise expressive typings, but at the cost of whole-program analysis. Improving the inference of annotations simplifies the programmer's task. Jif currently infers type annotations on local variables, but does not infer other annotations in order to provide modular compilation.

6.2 Declassification

It has been recognized since the beginnings of work on information flow that noninterference is too rigid to describe the information security of real applications (Cohen, 1977; Goguen and Meseguer, 1982). There has been a great deal of work on mechanisms and security definitions that weaken noninterference to account for declassification. In this section, we focus on language-based security policies and semantic conditions for declassification.

Sabelfeld and Sands (2007) survey recent work on declassification, categorizing work using four dimensions: *what* information is released, *who* releases it, *where* in the system information is released, and *when* release may occur. We use these categories to discuss related work on declassification.

In the same publication, Sabelfeld and Sands also propose four "prudent principles for declassification," desirable properties for declassification policies and semantic conditions. *Semantic consistency* requires that semantically equivalent programs should satisfy the same declassification security conditions. *Conservativity* requires that the definition of security should reduce to noninterference in the absence of declassification. *Non-occlusion* requires that the presence of declassification does not hide illegal information flows. *Monotonicity* requires that adding declassification annotations cannot make a secure program insecure. Noninterference according to policy (presented in Chapter 2) satisfies semantic consistency, conservativity, and monotonicity. Because the declassification policies address *when*, but not *what*, information may be declassified, noninterference according to policy does not satisfy non-occlusion.

6.2.1 When

In many systems, it is critical to security requirements that declassification occur only at the appropriate time. Much of the work that restricts *when* declassification may occur does so by specifying conditions that must be satisfied for declassification to occur, and ensuring those conditions are satisfied. The declassification policies presented in this dissertation make these conditions explicit in the security policies.

Conditions for declassification

Giambiagi and Dam (2003) consider information flow in the secure implementation of security protocols. Security protocols are specified using *dependency rules* that declare both what information may flow, and the conditions under which the flow is allowed. They provide a semantic security condition based on their ear-

lier work on *admissability* (Giambiagi and Dam, 2000), which requires invariant behavior of a system despite perturbations to secrets input to the system.

Banerjee and Naumann (2003, 2005a) explore mediating information release with stack-based access control mechanisms. Declassification (and other sensitive operations) occur only when appropriate permissions have been enabled by calling-contexts on the stack. Thus, the condition for declassification is that access was granted by the access control system. They present a noninterference property to describe such "permission-dependent" information flows, and a security type system to enforce this property. Banerjee and Naumann (2005b) extend this work to consider history-based access control (Abadi and Fournet, 2003), where access control decisions may depend not just on the current call stack, but on the execution history. The access control decisions may form a covert channel, as they depend on possibly confidential parts of the execution history. Like the conditions for declassification and erasure seen in this work, information that may leak via this covert channel must be tracked and restricted.

Hicks et al. (2005) consider the dynamic update of information-flow policies in a program. Because updates to policies could allow new information flows, policy updates can be a form of declassification. Hicks et al. restrict updates to occur only when such updates are consistent with the currently running program, and thus may delay a policy update until soundness can be ensured. If the policy update is regarded as a declassification, then the condition for declassification is that the run-time state is consistent with the update. They define and enforce *noninterference between updates*.

Swamy et al. (2006) further develop the dynamic update work of Hicks et al. (2005) and introduce the language Rx. Instead of delaying policy updates, Rx uses transactions to ensure consistency of information flow and policy updates;

conflicting transactions will be rolled back if needed. Rx enforces noninterference between updates, and also ensures the policy update itself does not incorrectly reveal information. Rx uses role-based security (Li et al., 2002) instead of an arbitrary confidentiality lattice, or the decentralized label model. Bandhakavi et al. (2008) refine Rx, providing finer-grained specification of role-based security policies, and using established trust-management principles to restrict policy updates. More specifically, they adopt a "writers-oriented" approach to integrity instead of the "readers-oriented" approach used in Rx.

Time-complexity

A small body of work considers the time-complexity of declassification. Although connected to temporal aspects of declassification, this work is not closely related to reasoning about the conditions for declassification. Volpano and Smith (2000) define *relative secrecy* to require that a secret of length n cannot be leaked in time polynomial in n. They consider as a motivating example a password-checking program that declassifies whether a guess equaled the password, and give a type system such that no well-typed program can copy the secret in polynomial time.

6.2.2 Where

Sabelfeld and Sands (2007) identify two kinds of locality pertaining to declassification: *code locality* (*where in the code* declassification is permitted), and *level locality* (*where in the confidentiality lattice* information is permitted to flow). Restricting *where* in the code declassification is permitted can be seen as a special case of restricting *when* declassification is permitted, to wit, when execution is in the permitted code locations.

Intransitive noninterference (Rushby, 1992; Pinsky, 1995; Roscoe and Goldsmith, 1999; Mantel, 2001; Mantel and Sands, 2004; van der Meyden, 2007) is an intensional property, where in each step of computation, information only flows between security levels according to some (possibly intransitive) relation. The relation captures the intuition that some of the intended information flow is information release (i.e., the intransitive parts of the relation), and is thus a form of level locality. For example, information may be allowed to flow from L to H, from H to a $Declassifier$ level, and from $Declassifier$ to L; information would not be allowed to flow directly from H to L. Often left unstated is the code locality assumption that only trusted components will perform the computation steps that result in information release.

Mantel and Sands (2004) apply intransitive noninterference in a language-based setting, and combine code and level locality. Following Mantel's earlier work (Mantel, 2001), the permitted flows between security levels are factored into a partial order $\leq$ over security levels and a relation $\rightsquigarrow$ over security levels. The flows permitted by $\rightsquigarrow$ are the intransitive "exceptions" to the normal flows $\leq$. Declassification commands are written $[x := y]$. For each step of a program, information flow must conform to the normal flow $\leq$, except for declassification commands, which must conform to $\leq \cup \rightsquigarrow$.

Independently, Almeida Matos and Boudol (2005) combine code locality and level locality in their *nondisclosure* semantic security condition, which requires that in each step information flows according to the flow policy for that step. They use a language construct **flow** $a_1 \prec b_1, \ldots, a_n \prec b_n$ **in** c, where c is a command, $a_1, \ldots, a_n, b_1, \ldots, b_n$ are principals, and for each $i \in 1..n$, information is allowed to flow from principal a_i to b_i within the command c.

Hicks et al. (2006) propose *trusted declassification* as a mechanism to specify

which modules are trusted to perform declassification, and so provide a form of code locality. Their policy language allows principals to specify which modules to use when communicating with which principals. For example, Alice could declare that AES encryption can be used when communicating with principal Bob, but that RSA encryption should be used when communicating with Charlie. As such, trusted declassification also addresses to some extent level locality, and the *what* and *who* dimensions of declassification. Hicks et al. define a semantic security condition *noninterference modulo trusted methods* that permits only declassifications that use the trusted modules appropriately. They have implemented trusted declassification by compiling principals' policies into Jif code that grants a principal's authority to a module based on which modules the principal trusts.

6.2.3 Who

In the presence of multiple principals, perhaps mutually distrusting, it is important to reason about *who* controls or affects declassification.

The decentralized label model (DLM), introduced by Myers and Liskov (2000) and discussed extensively in Chapter 4, is a framework for specifying and reasoning about security in the presence of mutual distrust. To prevent abuse of declassification, the DLM as originally presented required that owners of policies weakened by a declassification must authorize that declassification. This requirement was termed *selective declassification* by Pottier and Conchon (2000), who presented selective declassification as a combination of information flow and access control. Pottier and Conchon express selective declassification as declassification operations "locked" at appropriate levels of authority; access control allows suitably authorized principals to unlock the declassification oper-

ations, and only unlocked declassification operations can declassify information. Selective declassification prevents inappropriate declassification by requiring a condition to be satisfied when declassification occurs: *all required owners of the data have authorized the declassification*.

Although Banerjee and Naumann (2003, 2005a,b) also use access control to mediate information release, their access control relies on the call stack and execution history, not the identity of principals.

Further extensions to the DLM have added capability mechanisms for controlling access to declassification and have integrated these mechanisms with public-key infrastructures. Chothia et al. (2003) extend the DLM by adding capability mechanisms for controlling access to declassification and integrate these mechanisms with public-key infrastructures. They introduce a programming language that uses typed cryptographic operations to restrict which principals may access information. A principal may issue a *declassification certificate* for a label he owns, enabling additional principals to access information protected by that label.

Tse and Zdancewic (2004, 2005) also consider certificates for declassifications in a type system. Using a monadic calculus, they encode both principal delegation and declassification as subtyping. Tse and Zdancewic define and enforce *conditioned noninterference*, which requires that if no one issues a declassification certificate for a given observer, then the program is noninterfering from that observer's perspective. They also define *certified noninterference*, the contrapositive of conditioned noninterference, which requires all declassifications of a program be permitted by appropriate declassification certificates.

Vaughan and Zdancewic (2007) consider a cryptographic implementation of a variant of the DLM, and incorporate it into a programming language. They show

a noninterference result for well-typed programs, under a Dolev-Yao attacker model (Dolev and Yao, 1983).

Robustness (Zdancewic and Myers, 2001; Myers et al., 2004; Zdancewic, 2003) protects declassification (and other sensitive operations; see Tse and Zdancewic 2005) from active attackers, and was described in Chapter 4. To recap, a system is robust if an active attacker (one who can observe and modify the behavior of the system) cannot learn more about the system (including secret inputs) than could a passive attacker (one who can observe but not modify the behavior of a system). Chapter 4 extends robustness to account for mutual distrust, defining and enforcing *robustness against all attackers*.

6.2.4 What

What information is permitted to be declassified is crucial to the security of systems. Much work concerned with security policies and semantic conditions for declassification has focused on expressing what information is declassified.

Sabelfeld and Myers (2004) introduce *delimited release* to reason about what information may be declassified. A collection of *escape-hatch expressions* are specified, and the only escape-hatch expressions may be declassified. Delimited release is presented as an end-to-end, extensional, security condition: if two initial states are identical on the publicly observable variables and on the evaluation of all escape-hatch expressions, then an observer will not be able to distinguish two (terminating) executions from these initial states.

Li and Zdancewic (2005a) use lambda-calculus functions to specify what information may be declassified. A declassification policy is a set of lambda-calculus functions, and the application of any of these functions to secret information may be declassified. The security condition *relaxed noninterference* requires that

information is only declassified according to the declassification policies. Li and Zdancewic (2005b) extend the policies to also consider downgrading integrity.

Askarov and Sabelfeld (2007a) explicitly model attacker knowledge, and introduce the semantic security condition *gradual release*, which requires that attacker knowledge not increase except when explicit declassifications occur. They enforce gradual release in a language that includes cryptographic operations, and model how an attacker's knowledge interacts with cryptography.

Giacobazzi and Mastroeni (2004) introduce *abstract noninterference*, which applies the machinery of abstract interpretation (Cousot and Cousot, 1977) to information flow. To model what an attacker is permitted to learn about secret information, attackers are regarded as abstract interpretations of programs. Thus, instead of observing the actual values of secret information, attackers are characterized as observing properties of the secret information, such as parity or sign of the secret information. Abstract noninterference has been further developed and elaborated (e.g., Clark et al. 2004; Giacobazzi and Mastroeni 2005; Hunt and Mastroeni 2005; Mastroeni 2005; Banerjee et al. 2007a). However, it is not apparent how abstract interpretation can be used to reason about other dimensions of declassification.

Sabelfeld and Sands (1999) use partial equivalence relations (PERs)[4] to reason about the observations an attacker can make. Although both PERs and abstract domains are used to model an attacker's observational power, it is unclear what relationship exists between the PER model and abstract noninterference (Sabelfeld and Sands, 2007).

A growing body of work considers *how much* information is declassified or leaked by a program. *Quantitative information flow* attempts to measure and restrict the amount of information released, and/or the rate at which information

[4]Partial equivalence relations are symmetric and transitive, but not necessarily reflexive.

is released. Typically, quantitative approaches measure information flow in terms of an attacker's reduction in uncertainty (e.g., Millen 1987; Gray 1991; Lowe 2002; Di Pierro et al. 2002; Clark et al. 2005; Malacaria 2007), although Clarkson et al. (2005) argue that accuracy of belief is a more appropriate measure.

6.2.5 Multiple dimensions

Researchers have recently considered multiple dimensions of declassification. Although some work has touched on multiple dimensions of declassification (e.g., Hicks et al. 2006), the recent trend is distinguished by the explicit extension or combination of previous work to cover more dimensions of declassification.

Askarov and Sabelfeld (2007b) propose *localized delimited release* to extend delimited release (Sabelfeld and Myers, 2004) with further restrictions on when information may be released. Whereas delimited release permits an observer to learn the evaluation of any escape-hatch expression e, localized delimited release permits an observer to learn e only after a **declassify**(e) expression has been executed. For example, the following program satisfies delimited release, but does not satisfy localized delimited release. (We assume the observer can view the contents of variable l but not variable h, and h **mod** 10 is the only escape-hatch expression.)

$$l := h \textbf{ mod } 10; l := \textbf{declassify}(h \textbf{ mod } 10)$$

The type system proposed by Sabelfeld and Myers (2004) to enforce delimited release is sufficiently restrictive that it also enforces localized delimited release.

Although Askarov and Sabelfeld regard localized delimited release as combining the *what* and *where* dimensions of declassification, it is more accurately characterized as combining *what* and *when*. Localized delimited release restricts

declassification based on the execution history, and not on code location. This is demonstrated by the following program.

```
1  i := 0;
2  while i < 2 do
3        if j > 0 then
4              l := h mod 10
5        else
6              skip;
7        l := declassify(h mod 10);
8        j := 1;
9        i := i + 1
```

The **while** loop executes exactly twice. The initial value of variable j determines whether the assignment on line 4 will occur on the first loop iteration. If the assignment occurs during the first iteration, then the program does not satisfy localized delimited release. However, if the assignment occurs only in the second iteration, the program does satisfy localized delimited release. Thus, for localized delimited release, it is not *where* the code is located that is significant, but *when* the code is executed.

Banerjee et al. (2007b, 2008) combine the *when*, *what*, and *where* dimensions of declassification using *flowspecs*: flexible specifications of permitted information flow. Flowspecs are a restricted form of relational Hoare triples over a logic for reasoning about information flow (Amtoft et al., 2006; Amtoft and Banerjee, 2007). Flowspec pre-conditions can specify the conditions for declassification to occur (*when*), and the expression that may be declassified (*what*). The occurrence of a flowspec in code denotes *where* in the program declassification may occur. Banerjee et al. define *conditioned gradual release* (an extension of gradual release that incorporates flowspec pre-conditions) and show that it holds when flowspecs are enforced. They suggest using program verification techniques to enforce flowspecs, and type-checking to enforce stronger information-flow controls on flowspec-free portions of code.

Broberg and Sands (2006) introduce *flow locks*, a simple mechanism for specifying conditions under which information may flow. They propose a simple calculus with explicit operations to "open" and "close" locks, and a type system to statically verify the enforcement of flow-lock policies. Broberg and Sands show that flow locks can express and enforce several declassification conditions, including noninterference until declassification (a precursor to noninterference according to policy, introduced in Chong and Myers 2004), nondisclosure (Almeida Matos and Boudol, 2005), and robust declassification (Zdancewic and Myers, 2001; Myers et al., 2004; Zdancewic, 2003). Thus, flow locks are capable of expressing and enforcing *when*, *where*, and *who* dimensions of declassification, and may be able to provide a "core calculus of dynamic information flow policies." Indeed, flow locks appear capable of enforcing erasure requirements (Sands, 2006).

Mantel and Reinhard (2007) consider the *what* and *where* dimensions of declassification. However, they present separate semantic security conditions for each dimension, as opposed to a single security condition that combines multiple dimensions. Their *where* is similar to intransitive noninterference as presented by Mantel and Sands (2004), but satisfies the prudent principles for declassification (Sabelfeld and Sands, 2007). They present two semantic *what* conditions, both of which are intensional conditions similar to delimited release. They present a type system that enforces all three of their semantic security conditions.

6.3 Information erasure

Although both erasure and declassification requirements are common in many applications, erasure is not as well-studied as declassification. Comparatively

little work has considered semantic security conditions that hold in the presence of erasure, or security policies for specifying erasure requirements.

6.3.1 Language-based erasure

Since the introduction of erasure policies (Chong and Myers, 2005), other researchers have considered language-based enforcement of information erasure, and uses of erasure policies in applications.

Concurrently with this work on erasure policy enforcement, Hunt and Sands (2008) consider the enforcement of *simple erasure policies* of the form $\ell \nearrow \ell'$, where erasure is required at the end of a lexical scope. These policies are a restricted instantiation of the policy framework used here, where only non-nested erasure policies are allowed, and the condition language is limited to specifying the end of lexical scopes. Using flow-sensitive typing contexts (Hunt and Sands, 2006), Hunt and Sands present an elegant type system to enforce erasure policies; their system requires no run-time erasure mechanism.

Comparing erasure enforcement in this work to Hunt and Sands' highlights a tension between expressiveness of erasure conditions and ease of enforcement. Simpler condition languages are easier to reason about statically, and thus easier to enforce statically. Hunt and Sands' conditions are tied to lexical scopes, and it is straightforward to reason statically about when conditions are satisfied. By contrast, the condition language used in this work is program expressions: flexible, but difficult to reason about statically. Because it is difficult or impossible to know the value of an arbitrary expression at a given program point prior to execution, it is difficult to determine statically whether a policy will require erasure at that program point, and thus difficult to enforce erasure statically. Instead, we use a simple run-time mechanism to enforce erasure, an approach similar in

spirit to hybrid type checking (Flanagan, 2006), in which proof obligations that cannot be proven at compile-time are deferred to run-time.

Hunt and Sands also consider erasure in the presence of input and output. They note that once data is output and leaves the purview of the system, the system cannot ensure erasure. They also note that a user may provide multiple inputs to a system throughout execution, and each input may require erasure. They define the semantic security condition *local erasure* to express the appropriate erasure of a single input, and lift this to a notion of *global erasure* to ensure that each input is erased appropriately. Although noninterference according to policy is defined in terms of a single input to the system provided at the start of execution, it can be easily modified to deal with input and output by adapting the approach of O'Neill et al. (2006), who consider noninterference in interactive programs. In their work, noninterference is required with respect to a user's *strategy*, a function from the trace the user has seen so far to inputs the user provides.

Of the four semantic principles for declassification proposed by Sabelfeld and Sands (2007) and described above, three seem applicable to erasure. Semantic consistency and conservativity can easily be adapted for erasure: semantic consistency requires that semantically equivalent programs should satisfy the same declassification and erasure security conditions, and conservativity requires that in the absence of declassification and erasure, the definition of security reduces to noninterference. Non-occlusion for declassification requires that the presence of declassification does not hide illegal information flows; the equivalent of non-occlusion for erasure is that the absence of erasure does not hide illegal information flows. Monotonicity does not appear to have a ready equivalent for erasure, unless there are explicit erasure annotations. The languages IMP_E and

Jif$_E$ presented in this dissertation do not have explicit erasure annotations, and neither does the language considered by Hunt and Sands (2008).

6.3.2 Uses of erasure policies

Hansen and Probst (2006) consider information-flow security in Java Card bytecode, and identify the utility of erasure policies in providing security assurance. They consider "simple erasure policies" of the form $L \overset{end}{\nearrow} H$, where end is a condition indicating the end of execution of the current program. They define a corresponding *simple erasure* security condition. Simple erasure requires that every location has one of the policies L, H, or $L \overset{end}{\nearrow} H$ enforced on it. A program satisfies simple erasure if the program satisfies noninterference, and in addition, any two executions of the program that start with equivalent values in the L locations terminate with equivalent values in the L and $L \overset{end}{\nearrow} H$ locations. Thus, values in $L \overset{end}{\nearrow} H$ locations are erased before the end of the program. Simple erasure is consistent with noninterference according to policy. Hansen and Probst conjecture, but do not demonstrate, that simple erasure is straightforward to enforce.

Hansen and Probst (2005) have also used erasure policies in secure dynamic program repartitioning. Secure program partitioning (Zdancewic et al., 2001) is a technique to split data and code across a set of mutually distrusting hosts while guaranteeing security. Hansen and Probst consider repartitioning a program when the set of hosts changes dynamically, and use erasure policies to ensure that old copies of data are removed from the system when repartitioning occurs. Hansen and Probst do not describe how to enforce the erasure policies. Søndergaard's subsequent master's thesis (Søndergaard, 2006) expands on the use of erasure policies in dynamic program repartitioning, proposing erasure

policies of the form $\ell \not\nearrow \ell'$, where conditions c are limited to Boolean formulas over two predicates that indicate whether principals of the dynamic system are active, and what components of the system principals are currently responsible for. Søndergaard discusses the trusted run-time components required to enforce these erasure policies, but does not implement them.

6.3.3 System-based and hardware-based erasure

Although not much work has considered erasure of information at the language level of abstraction, there is a body of work that considers the deletion of information at the operating system or hardware levels of abstraction.

Corner and Noble (2002) introduce *zero-interaction authentication* to automatically restrict information on a laptop computer when the user is not physically present. In their system, the user wears an authentication token that communicates with the laptop. The laptop is assumed to use a cryptographic file system, and the authentication token provides a decryption key. When the laptop detects the authentication token is not present, the laptop discards the decryption key, and encrypts the file cache. Thus, the laptop enforces an erasure policy on files, ensuring no unencrypted files are accessible without a decryption key when the authentication token is not physically present. This is similar to the mobile computing example introduced in Section 1.3 and expanded in Section 2.1.

Fenton (1974) defines a *memoryless subsystem* as a program or procedure that is "guaranteed to have kept no record of data supplied when it has completed its task." Defined this way, memoryless subsystems seem to have an erasure requirement, which can be achieved by ensuring "the machine is wiped clean at the end" of execution. Fenton points out that memoryless subsystems are inadequate to implement systems that must compute with data from mutually

distrusting principals. Fenton restates the problem as a system that must produce public output that does not depend on secret input, and defines the Data Mark machine (Fenton, 1973, 1974) to provide hardware support to solve this problem. The problem restatement has no erasure requirement, and may thus be a weakening of Fenton's initial definition.

Gutmann (1996, 2001) considers the recovery of supposedly erased data from magnetic media and semiconductor devices (DRAM, SRAM, and EEPROM). Traces of the data that was stored may be detectable even after the data has been overwritten. He describes techniques to hinder recovery of data from these media. Anderson and Kuhn (1997) consider low-cost attacks on tamper-resistant devices, and apply Gutmann's techniques to the DRAM of a security module used in a bank to check customer PIN numbers. They successfully recover approximately 90% of the bits of the secret key values.

6.4 Expressive models and policies

This dissertation develops expressive security policies, and enforces them end-to-end using information-flow control. However, the study and use of expressive security policies goes well beyond language-based information-flow control. In this section we describe other expressive security models, and security policy formalisms.

Security models Label models have been used extensively in military applications, due to the Department of Defense's "Orange Book" (Department of Defense, 1985), which mandated the use of labeled data for high-assurance systems. Feiertag et al. (1977) instantiate Denning's lattice model (Denning, 1976) as multi-level security policies for military systems. Biba (1977) shows that

information-flow control could be used to enforce integrity as well as confidentiality. Biba reverses the Bell-LaPadula information-flow control rules, requiring "no write up, no read down," demonstrating that integrity is dual to confidentiality.

Label models have also been used in commercial systems. The Clark-Wilson model (Clark and Wilson, 1987) emphasizes the need for integrity in commercial systems, and has been widely used in the banking industry. The Chinese-Wall model (Brewer and Nash, 1989) can express conflicts of interest, and restrict a user's access rights based on the access rights currently held by the user.

Foley (1991) presents a framework that can express many previous label models and requirements (including Bell-LaPadula, Biba, Clark-Wilson, Chinese Walls, and separation of duties), and induces a taxonomy on these information-flow models.

Access control The most commonly used mechanism to control information release in real systems is *discretionary access control*, also called *access control*. The key idea is that before a program performs a potentially dangerous action (such as the release of information), it performs a run-time check (at the discretion of the user and/or program) to ensure that appropriate authority exists to perform the action.

Although access control mechanisms can express declassification and erasure requirements, they are unable to enforce these information security requirements end-to-end.

Many access control mechanisms have been designed and implemented, including *capabilities* (Dennis and VanHorn, 1966; Wulf et al., 1974), *access control lists* (Lampson, 1971), and stack inspection (Wallach et al., 1997; Wallach and Felten, 1998; Fournet and Gordon, 2002). Logics for access control have been

devised as a way to explain and improve access control (see Abadi 2003 for a brief survey).

There have been many extensions and generalizations of access control mechanisms, increasing their expressiveness. Several extensions have considered temporal aspects of access control (e.g., Bertino et al., 1998; Cuppens and Gabillon, 1996), specifying the time periods that access is permitted or denied.

Jajodia et al. (2001) introduce a framework for *provisional authorization*, where access is granted only if certain conditions are satisfied, for example, an audit log entry is made. The framework is parameterized on a logic for conditions. Bettini et al. (2003) generalize this idea, and introduce *provisions* and *obligations*. Provisions are conditions that must be satisfied, or actions that must be performed, before an access decision is made; obligations are conditions or actions that must be fulfilled after the access decision. Subsequently, much recent work (Hilty et al., 2005; Barth et al., 2006; Dougherty et al., 2007; Hilty et al., 2007) has investigated the use and enforcement of obligations.

The conditions in declassification policies can be seen as provisions: condition c must be satisfied when information labeled $p \searrow^{c}_{\searrow} q$ is declassified. Similarly, erasure conditions can be viewed as obligations: there is an obligation to erase information labeled $p \overset{c}{\nearrow} q$ when condition c is satisfied. Indeed, erasure of data is a common obligation in privacy policies and digital rights management (DRM). For example, DRM may require a user who has purchased the right to watch a movie to delete the movie when it has been watched twice or one week has elapsed.

Security automata Alpern and Schneider (1987) specify safety properties[5] using a class of Büchi automata, named *security automata* by Schneider (2000). A security automaton is defined by a countable set of states, a countable input alphabet, and a transition relation. Security automata have been used to specify many useful security policies, including access control (Evans and Twyman, 1999; Erlingsson and Schneider, 1999; Abadi and Fournet, 2003), software fault isolation (Wahbe et al., 1003; Erlingsson and Schneider, 2000), Chinese Walls (Erlingsson, 2003; Fong, 2004), and information release (Swamy and Hicks, 2008). Security automata can be enforced by a variety of mechanisms, including execution monitoring (Schneider, 2000), program rewriting (Erlingsson and Schneider, 1999; Hamlen et al., 2006), and type systems (Walker, 2000; Swamy and Hicks, 2008).

Enforcement mechanisms for security automata are able to enforce (approximately) the class of safety properties (Schneider, 2000; Viswanathan, 2000; Hamlen et al., 2006). Bauer et al. (2002) introduce *edit automata*, an extension of security automata that allows the suppression and insertion of actions. Edit automata allow non-safety properties to be expressed and enforced (Ligatti et al., 2005).

Although security automata provide an expressive language for security policies, they cannot express information-flow policies. Nonetheless, a suitable adaptation of security automata could generalize the declassification and erasure policies presented here, and permit the specification of more complex declassification and erasure requirements. Because a policy may refer to distinct conditions that are both satisfied in a single state, an erasure and declassification security

[5] *Properties* are predicates over traces (Alpern and Schneider, 1985). Informally, a *safety property* (Lamport, 1985) ensures that no "bad thing" happens during any execution.

automaton may need to make multiple transitions without consuming an input.[6] Typical information-flow control mechanisms require a lattice structure over the security policies; to enforce erasure and declassification security automata, either an alternative enforcement mechanism would need to be used, or a lattice structure imposed on the automata.

[6] Automata on guarded strings (Kozen, 2003) similarly make multiple transitions on a single input.

CHAPTER 7

CONCLUSION

Trustworthy systems enforce their information security requirements. Building trustworthy systems is difficult for at least two reasons: systems have complex information security requirements; and in most implementation methodologies, enforcement of information security is only weakly tied to the security requirements. As a result, it is difficult to obtain assurance that a system's requirements are satisfied by an implementation.

The goal of this work is to make it easier to build trustworthy systems, by providing security policies that can express a system's security requirements, and provably enforcing them in the implementation. This dissertation focused on two common and important kinds of information security requirements: declassification and erasure, which both describe how the confidentiality of information changes during a system's execution.

This dissertation makes two key contributions. The first contribution is the development of an expressive security policy framework for declassification and erasure. The framework includes the definition of a precise semantic security guarantee that holds when the policies are enforced, a proof that a combination of static and dynamic mechanisms do enforce the policies in a simple imperative language IMP_E, and an extension of the decentralized label model with declassification and erasure policies. The second contribution is the incorporation of the policy framework into Jif, a practical security-typed programming language, and the use of this extended version of Jif to implement a large system, validating

the expressiveness and utility of the policies.

7.1 Declassification and erasure policies

The information security policies presented in Chapter 2 describe how permitted information flows change during the execution of a system, by expressing when information may be declassified, and when information must be erased. Declassification and erasure are both common information security requirements of many systems.

Chapter 2 also presented a formal semantics for the declassification and erasure policies, explicating their meaning in terms of the observability of information during execution of a system. The policy semantics were used to define a precise semantic security condition, noninterference according to policy. This semantic security condition allows us a formal and precise understanding of what security guarantees are obtained when a system enforces declassification and erasure policies on information.

Chapter 3 demonstrated that the policies can be provably enforced in a simple imperative language, using run-time mechanisms, and a mostly-standard type-system for information-flow control.

The decentralized label model (DLM) (Myers and Liskov, 2000) allows mutually distrusting principals to independently express their security requirements. Chapter 4 described how to incorporate declassification and erasure policies into the DLM.

The extended version of the DLM allows the specification of both confidentiality and integrity requirements. The semantic security condition of robustness (Zdancewic and Myers, 2001; Myers et al., 2004; Zdancewic, 2003) connects integrity and confidentiality by requiring changes to the confidentiality of infor-

mation to have sufficiently high integrity with respect to a distinguished attacker. Also in Chapter 4 we define *robustness against all attackers*, a generalization of robustness that accounts for mutual distrust, where any entity may be a potential attacker, and provide a type system that enforces robustness against all attackers in IMP_E.

7.2 Practical use of declassification and erasure policies

Although the declassification and erasure policies are expressive, the simple language IMP_E is not sufficiently powerful to implement real-world systems. To enable the use of declassification and erasure policies in real implementations, we incorporated the policies into the Jif programming language (Myers, 1999; Myers et al., 2001–2008), an extension of Java with information-flow control.

The language Jif_E was presented in Chapter 5. Jif_E incorporates the extended DLM into Jif, allowing declassification and erasure policies to be used in Jif code. The IMP_E type system and run-time enforcement mechanisms to enforce declassification and erasure policies required careful adaptation to interact with Java and Jif language features.

We used Jif_E to implement Civitas (Clarkson et al., 2008), a secure remote voting service. The declassification and erasure policies were useful in several places in the implementation of Civitas, and provide additional assurance that the Civitas implementation is correctly enforcing the security requirements.

7.3 Future work

There is much more work to be done to make the task of writing trustworthy programs easy enough to be incorporated into standard software methodolo-

gies, and achievable by programmers who are not security specialists. There are other aspects of information security, and software security, that are important to enforce in systems before they are trustworthy, including availability of information, reliability, and functional correctness.

Programming is ultimately a human endeavor, and tools to assist programmers have much potential to ease the production of trustworthy software. For example, tools that allow visualization and navigating of information flows within a program (both locally and globally) would help programmers develop intuition, and comprehend the impact of different policy annotations in the code. Mechanisms for easy navigation of security policies within a system would perhaps enable an even clearer connection between system security requirements and the implementation. Reducing the number of annotations, perhaps by inferring as many as possible, would reduce the burden on the programmer.

An interesting approach is to allow the programmer to decide how much security is sufficient, by allowing a tradeoff between programmer effort and security guarantees. That is, provide weak (but well-understood) security guarantees if the programmer puts little effort into information security, and stronger security guarantees for more effort. This approach would perhaps require the development of new semantic security conditions, and perhaps new program analyses for determining when the conditions hold.

Trustworthy systems should enforce fine-grained, application-specific, information security requirements. The expressive specification, and provable enforcement, of security requirements is key to building trustworthy systems, and this dissertation has focused on one aspect of this: the specification and enforcement of declassification and erasure requirements. Many significant challenges remain before the production of trustworthy systems is simple enough,

and cheap enough, to be the norm. However, recent years have seen a concentration of research effort towards this goal, and significant progress has been made. I am optimistic that the following years will bring us practical, strong, information security, to enable the building of trustworthy systems.

BIBLIOGRAPHY

Martín Abadi. Logic in access control. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science*, pages 228–233. IEEE Computer Society, June 2003.

Martín Abadi and Cédric Fournet. Access control based on execution history. In *Network and Distributed System Security Symposium*. The Internet Society, 2003. ISBN 1-891562-16-9, 1-891562-15-0.

Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 226–240, Washington, DC, USA, 2005. IEEE Computer Society.

Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct 1985.

Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

Torben Amtoft and Anindya Banerjee. Verification condition generation for conditional information flow. In *Proceedings of the Fifth ACM Workshop on Formal Methods in Security Engineering*, New York, NY, USA, November 2007. ACM Press.

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 91–102, New York, NY, USA, January 2006. ACM Press.

Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 125–136, London, UK, 1997. Springer-Verlag.

Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221. IEEE Computer Society, 2007a.

Aslan Askarov and Andrei Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 53–60, New York, NY, USA, 2007b. ACM Press. ISBN 978-1-59593-711-7.

Sruthi Bandhakavi, William Winsborough, and Marianne Winslett. A trust management approach for flexible policy management in security-typed languages. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*. IEEE Computer Society, June 2008.

Anindya Banerjee and David A. Naumann. Using access control for secure information flow in a Java-like language. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 155–169. IEEE Computer Society, June 2003.

Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005a. ISSN 0956-7968.

Anindya Banerjee and David A. Naumann. History-based access control and secure information flow. In *Proceedings of the Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards*, volume 3362 of *Lecture Notes in Computer Science*, pages 27–48. Springer-Verlag, March 2005b.

Anindya Banerjee, Roberto Giacobazzi, and Isabella Mastroeni. What you lose is what you leak: Information leakage in declassification policies. *Electronic Notes in Theoretical Computer Science*, 173:47–66, 2007a.

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Towards a logical account of declassification. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 61–66, New York, NY, USA, 2007b. ACM Press.

Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2008.

John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, April 2003. ISBN 0321136160.

Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2006.

Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Proceedings of the Foundations of Computer Security Workshop*, July 2002.

D. Elliott Bell and Leonard J. La Padula. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

Marisol Bello. Data security top tech issue for colleges. USA Today, March 20, 2008.

Elisa Bertino, Claudio Bettini, Elena Ferrari, and Pierangela Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, 1998.

Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminda Wijesekera. Provisions and obligations in policy rule management. *Journal of Network and System Management*, 11(3):351–372, 2003.

K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, April 1977.

Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002. ISBN 0-201-44099-7.

David F. C. Brewer and Michael J. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–258. IEEE Computer Society, May 1989.

British Broadcasting Corporation. Brown apologises for records loss. `http://news.bbc.co.uk/1/hi/uk_politics/7104945.stm`, November 21, 2007.

Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Symposium on Programming*, pages 180–196. Springer, 2006.

Ilaria Castellani. State-oriented noninterference for CCS. *Electronic Notes in Theoretical Computer Science*, 194(1):39–60, 2007. ISSN 1571-0661.

Hubie Chen and Stephen Chong. Owned policies for information security. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2004.

Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, New York, NY, USA, October 2004. ACM Press.

Stephen Chong and Andrew C. Myers. Language-based information erasure. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2005.

Stephen Chong and Andrew C. Myers. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2006.

Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*. IEEE Computer Society, June 2008.

Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 170–186. IEEE Computer Society, June 2003.

David Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society, 1987.

David Clark, Sebastian Hunt, and Pasquale Malacaria. Non-interference for weak observers. In *Proceedings of the Programming Language Interference and Dependence*, August 2004.

David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science*, 112: 149–166, January 2005.

Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 31–45, Washington, DC, USA, 2005. IEEE Computer Society.

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. Technical Report TR2007-2081, Cornell University, November 2007.

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2008.

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

Mark D. Corner and Brian D. Noble. Zero-interaction authentication. In *Proceedings of the The Annual International Conference on Mobile Computing and Networking*, New York, NY, USA, September 2002. ACM Press.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

Frédéric Cuppens and Alban Gabillon. Modelling a multilevel database with temporal downgrading functionalities. In *Proceedings of the Ninth Annual IFIP TC11 WG11.3 Working Conference on Database Security IX : Status and Prospects*, pages 145–164, 1996.

Zhenyue Deng and Geoffrey Smith. Lenient array operations for practical secure information flow. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 115–124. IEEE Computer Society, June 2004.

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

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

J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.

Department of Defense. Trusted computer system evaluation criteria, 1985.

Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate non-interference. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 1–15. IEEE Computer Society, June 2002.

D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Obligations and their interaction with programs. In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734, pages 375–389, Berlin, September 2007. Springer.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, New York, NY, USA, October 2005. ACM Press.

Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2000.

Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigm Workshop*, pages 87–95, September 1999.

David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 1999.

Federal Trade Commission. Eli Lilly settles FTC charges concerning security breach. Press release, available at `http://www.ftc.gov/opa/2002/01/elililly.shtm`, January 18, 2002.

Federal Trade Commission. BJ's Wholesale Club settles FTC charges. Press release, available at `http://www.ftc.gov/opa/2005/06/bjswholesale.shtm`, June 16, 2005a.

Federal Trade Commission. Internet service provider settles FTC privacy charges. Press release, available at `http://www.ftc.gov/opa/2005/03/cartmanager.shtm`, March 10, 2005b.

R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *Proceedings of the 6th ACM Symposium on Operating System Principles, ACM Operating Systems Review*, 11(5):57–66, November 1977.

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.

J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, 1973.

J. S. Fenton. Memoryless subsystems. *Computer Journal*, 17(2):143–147, May 1974.

Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–140. IEEE Computer Society, May 1997.

Cormac Flanagan. Hybrid type checking. In *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 245–256, New York, NY, USA, 2006. ACM Press.

Riccardo Focardi and Roberto Gorrieri. Classification of security properties (Part I: Information flow). In *Foundations of Security Analysis and Design*, volume 2171 of *Lecture Notes in Computer Science*, pages 331–396. Springer, 2001.

Riccardo Focardi, Sabina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *Foundations of Software Science and Computation Structure*, volume 3441 of *Lecture Notes in Computer Science*, pages 299–315, Edinburgh, UK, April 2005. Springer-Verlag.

Simon N. Foley. A taxonomy for information flow policies and models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 98–108. IEEE Computer Society, May 1991.

Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, May 2004.

Cedric Fournet and Andrew D. Gordon. Stack Inspection: Theory and Variants. In *Conference Record of the Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 2002. ACM Press.

Ralph S. Freese, Jaroslav Ježek, and James Bryant Nation. *Free Lattices*. American Mathematical Society, Providence, RI, 1995. ISBN 0821803891.

Roberto Giacobazzi and Isabella Mastroeni. Timed abstract non-interference. In *Proceedings of the International Conference on Formal Modelling and Analysis of Timed Systems*, volume 3829 of *Lecture Notes in Computer Science*, pages 289–303. Springer-Verlag, September 2005.

Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 186–197, New York, NY, USA, 2004. ACM Press.

Pablo Giambiagi and Mads Dam. Confidentiality for mobile code: The case of a simple payment protocol. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 233–244. IEEE Computer Society, 2000.

Pablo Giambiagi and Mads Dam. On the secure implementation of security protocols. In *Proceedings of the 12th European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2003.

Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, April 1982.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.

James W. Gray, III. Toward a mathematical foundation for information flow security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 21–35. IEEE Computer Society, 1991.

Peter Gutmann. Data remanence in semiconductor devices. In *The Tenth USENIX Security Symposium Proceedings*, pages 39–54. USENIX Association, 2001.

Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *The Sixth USENIX Security Symposium Proceedings*, pages 77–90. USENIX Association, 1996.

Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, January 2006.

René Rydhof Hansen and Christian W. Probst. Secure dynamic program repartitioning. In *Proceedings of the Nordic Workshop on Secure IT-Systems*, October 2005.

René Rydhof Hansen and Christian W. Probst. Non-interference and erasure policies for Java Card bytecode. In *Proceedings of the 6th International Workshop on Issues in the Theory of Security*, March 2006.

Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proceedings*

of the 2006 Workshop on Programming Languages and Analysis for Security, pages 65–74, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-374-3.

Boniface Hicks, Dave King, and Patrick McDaniel. Jifclipse: Development tools for security-typed applications. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 1–10, New York, NY, USA, 2007. ACM Press.

Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. Dynamic updating of information-flow policies. In *Proceedings of the Foundations of Computer Security Workshop*, pages 7–18, June 2005.

M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546, Berlin, September 2007. Springer.

Manuel Hilty, David Basin, and Alexander Pretschner. On obligations. In *Proceedings of the 10th European Symposium On Research In Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 98–117, Berlin, 2005. Springer.

Sebastian Hunt and Isabella Mastroeni. The PER model of abstract non-interference. In *Proceedings of the 12th International Static Analysis Symposium*, number 3672 in Lecture Notes in Computer Science, pages 171–185. Springer-Verlag, September 2005.

Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the Thirty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 79–90, New York, NY, USA, January 2006. ACM Press.

Sebastian Hunt and David Sands. Just forget it—the semantics and enforcement of information erasure. In *Proceedings of the 17th European Symposium on Programming*. Springer, 2008.

Sushil Jajodia, Michiharu Kudo, and V.S. Subrahmanian. Provisional authorizations. In Anup Gosh, editor, *E-Commerce Security and Privacy*, pages 133–159. Kluwer Academic Press, 2001.

Chris Jones and Flavio Marques Menezes. Auctions and corruption: How to compensate the auctioneer. Technical Report 291, Australian National University—Department of Economics, 1995.

Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *Proceedings of the Workshop on Privacy in the Electronic Society*, pages 61–70, November 2005.

Dexter Kozen. Automata on guarded strings and applications. *Matématica Contemporânea*, 24:117–139, 2003.

Brian Krebs. Banks: Losses from computer intrusions up in 2007. `http://blog.washingtonpost.com/securityfix/2008/02/banks_losses_from_computer_int.html`, February 26, 2008.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating System Principles*, New York, NY, USA, October 2007. ACM Press.

Lap-chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, December 2006.

Leslie Lamport. Basic concepts: Logical foundation. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 1985.

Butler Lampson. Protection. In *Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, 1971.

Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 165–182, New York, NY, USA, October 1991. ACM Press. *Operating System Review*, 253(5).

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Remy, and Jerome Vouillon. The Objective Caml – Documentation and user's manual, July 2004. Located at `http://caml.inria.fr/ocaml/htmlman/`.

Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society, May 2002.

Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Conference Record of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages*, New York, NY, USA, January 2005a. ACM Press.

Peng Li and Steve Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Proceedings of the Foundations of Computer Security Workshop*, 2005b.

Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium On*

Research In Computer Security, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373, September 2005.

Gavin Lowe. Quantifying information flow. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2002.

Pasquale Malacaria. Assessing security threats of looping constructs. In *Conference Record of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 225–235, New York, NY, USA, 2007. ACM Press.

Heiko Mantel. Information flow control and applications—bridging a gap. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 153–172, London, UK, 2001. Springer-Verlag. ISBN 3-540-41791-5.

Heiko Mantel and Alexander Reinhard. Controlling the what and where of declassification in language-based security. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 2007. ISBN 978-3-540-71314-2.

Heiko Mantel and Andrei Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, page 126, Washington, DC, USA, 2001. IEEE Computer Society.

Heiko Mantel and David Sands. Controlled Declassification based on Intransitive Noninterference. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems*, LNCS 3303, pages 129–145. Springer-Verlag, November 2004.

Steven Marlin. Citigroup's lost tapes cast spotlight on data security. http://www.informationweek.com/news/security/privacy/showArticle.jhtml?articleID=164301046, June 7, 2005.

Isabella Mastroeni. On the rôle of abstract non-interference in language-based security. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, November 2005.

Stephen McCamant and Michael D. Ernst. A simulation-based proof technique for dynamic information flow. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 41–46, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-711-7.

John McHugh and Donald I. Good. An information flow tool for Gypsy: Extended abstract. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 46–48. IEEE Computer Society, 1985.

Jonathan K. Millen. Covert channel capacity. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1987.

Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 228–241, New York, NY, USA, January 1999. ACM Press.

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, New York, NY, USA, 1997. ACM Press.

Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4): 410–442, October 2000.

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, 2001–2008.

Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2004.

Srijith K. Nair, Patrick N.D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems*. Springer, 2007.

Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, New York, NY, USA, 2008. ACM Press.

Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, June 2006.

Sylvan Pinsky. Absorbing covers and intransitive non-interference. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 102–113. IEEE Computer Society, 1995.

François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming*, pages 46–57, New York, NY, USA, 2000. ACM Press.

François Pottier and Vincent Simonet. Information flow inference for ML. In *Conference Record of the Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 319–330, New York, NY, USA, 2002. ACM Press.

Project on Government Oversite. `http://www.pogo.org/p/homeland/ha-070806-lanl.html`, August 6, 2007.

A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 1999.

John Rushby. Noninterference, transitivity and channel-control security policies. Technical report, SRI, 1992.

Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

Andrei Sabelfeld and Andrew C. Myers. A model for delimited release. In *Proceedings of the 2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming*, pages 40–58, London, UK, 1999. Springer.

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007.

J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

David Sands. Personal communication, July 2006.

Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.

SELinux. Security-enhanced Linux. Project website `http://www.nsa.gov/selinux`.

Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2007.

Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

Scott F. Smith and Mark Thober. Improving usability of information flow security in Java. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 11–20, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-711-7.

Dan Søndergaard. Secure program partitioning in dynamic networks. Master's thesis, Technical University of Denmark, 2006. IMM-M.Sc-2006-92.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

Nikhil Swamy and Michael Hicks. Verified enforcement of automaton-based information release policies. In *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security*, New York, NY, USA, June 2008. ACM Press.

Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 202–216. IEEE Computer Society, 2006. ISBN 0-7695-2615-2.

Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In *Proceedings of the 14th European Symposium on Programming*, pages 279–294. Springer, April 2005.

Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004. IEEE Computer Society.

Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture*. IEEE Computer Society, December 2004.

Ron van der Meyden. What, indeed, is intransitive noninterference? In *Proceedings of the 12th European Symposium On Research In Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 235–250. Springer, September 2007. ISBN 978-3-540-74834-2.

Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 192–206, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2848-1.

Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, University of Pennsylvania, December 2000.

Dennis Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *Conference Record of the Twenty-Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 268–276, New York, NY, USA, January 2000. ACM Press.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Proceedings of the 9th European Symposium On Research In Computer Security*, volume 3193 of *Lecture Notes in Computer Science*, pages 225–243. Springer, 2004.

David Wagner and Matthew Bishop. Voting systems top-to-bottom review. `http://www.sos.ca.gov/elections/elections_vsr.htm`, 2007.

Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 203–216, New York, NY, USA, December 1003. ACM Press.

David Walker. A type system for expressive security policies. In *Conference Record of the Twenty-Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.

Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, California, USA, May 1998. IEEE Computer Society.

Dan S. Wallach, Dirk Balfanz, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 116–128, New York, NY, USA, October 1997. ACM Press.

W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor system. *Communications of the ACM*, 17(6):337–345, June 1974.

Andrew Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164. CSREA Press, 1982.

Steve Zdancewic. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics*, Electronic Notes in Theoretical Computer Science, March 2003.

Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceed-*

ings of the 17th ACM Symposium on Operating System Principles, pages 1–14, New York, NY, USA, October 2001. ACM Press.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278. USENIX Association, 2006.

Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 236–250, Oakland, California, May 2003. IEEE Computer Society.