TOWARDS PRACTICAL INFORMATION FLOW ANALYSIS

A Dissertation in
Computer Science and Engineering
by
Peixuan Li

© 2021 Peixuan Li

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2021
The dissertation of Peixuan Li was reviewed and approved by the following:

Danfeng Zhang  
Associate Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Trent Jaeger  
Professor of Computer Science and Engineering

Gang Tan  
Professor of Computer Science and Engineering

Dinghao Wu  
Professor of Information Sciences and Technology

Chita R. Das  
Professor of Computer Science and Engineering  
Department Head of Computer Science and Engineering
Abstract

In a world that becomes extensively connected by the internet, information is consumed and shared more than ever before. Protecting sensitive information manipulated by computing systems has been a vital task for information security, where information flow analysis has been a promising approach due to the rigorous end-to-end security guarantee that it provides.

Information flow analysis assumes that secrets are stored in variables and security levels are associated with variables to describe the intended secrecy of their values. The analysis tracks how information propagates inside a computing system and disallows any unintended usage of sensitive data. Classic Denning-style information flow analysis is well-studied with a variety of enforcement approaches backed with solid theoretical foundation.

However, classic information flow analysis is shown to be inadequate for real-world applications. First, real-world applications almost always require some dynamic policy, where the sensitivity of information can change during program execution. But security levels are assumed to be fixed in classic information flow analysis. As a result, the classic information flow approach is not applicable to applications with dynamic policies due to the lack of expressiveness to model sensitivity mutations. Second, even for a static policy, classic information flow analysis is typically flow- and path-insensitive, which raises many false alarms and thus undermines the accuracy of analysis results.
Given these limitations, this dissertation seeks to build novel and advanced information flow analyses that are more practical for real-world applications. To improve precision, we develop a flow- and path-sensitive analysis (based on a static program transformation and a dependent type system) that reduces false alarms compared with classic information flow analysis. Additionally, we develop a dependent label inference framework to free the programmers from manually providing intricate dependent labels needed in the flow- and path-insensitive analysis.

To support dynamic policy, we present a semantics framework to understand and compare existing policies. Furthermore, we present Dynamic Release, the first information flow policy that enables declassification, erasure, delegation and revocation at the same time. To make it feasible to enforce dynamic policies, we distill the conditions needed to soundly and completely decompose a dynamic policy into several code blocks with their corresponding static policies. We formalize and prove that it is possible to decompose a transient dynamic policy in a sound and complete way. However, sound and complete decomposition of a persistent policy is infeasible, as the policy by definition needs to exam the history of program execution.
# Table of Contents

List of Figures ix

List of Tables xi

Acknowledgments xii

Chapter 1
    Introduction 1

Chapter 2
    Background 7
       2.1 Information Flow Security . . . . . . . . . . . . . . . . . . . . . . . . . 7
       2.2 Information Flow Analysis on Static Policies . . . . . . . . . . . . . . . 8
       2.3 Information Flow Analysis on Dynamic Policies . . . . . . . . . . . . . . 9

Chapter 3
    A Path- and Flow-Sensitive Information Flow Analysis 10
       3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
       3.2 Sources of Imprecision . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
       3.3 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
          3.3.1 Program Transformation . . . . . . . . . . . . . . . . . . . . . . . . . 15
          3.3.2 Dependent Labels . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
       3.4 Language Syntax and Semantics . . . . . . . . . . . . . . . . . . . . . . . 16
       3.5 Program Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
          3.5.1 Bracketed Assignments and the Transformed Program . . . . . . . . 18
          3.5.2 Transformation Rules . . . . . . . . . . . . . . . . . . . . . . . . . . 18
          3.5.3 Correctness of the Transformation . . . . . . . . . . . . . . . . . . . 21
          3.5.4 Relation to Information Flow Analysis . . . . . . . . . . . . . . . . . 22
       3.6 Type System . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
          3.6.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
          3.6.2 Challenge: Statically Checking Implicit Declassification . . . . . . . 24
          3.6.3 Type Syntax and Typing Environment . . . . . . . . . . . . . . . . . . . 27
          3.6.4 Predicates and Variable Liveness . . . . . . . . . . . . . . . . . . . . 28
          3.6.5 Typing Rules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.6.6 Program Transformation and Information Flow Analysis .......... 31
3.7 Soundness ..................................................... 33
  3.7.1 Noninterference ............................................ 33
3.8 Enabling Flow-Sensitivity with Program Transformation ............ 37
  3.8.1 The Hunt and Sands System ................................ 37
  3.8.2 Program Transformation and Flow-Sensitivity ................. 38
  3.8.3 Comparison with an existing transformation .................. 42
3.9 Related Work .................................................. 43
3.10 Summary ...................................................... 45

Chapter 4
A Derivation Framework for Dependent Security Label Inference 46
4.1 Introduction .................................................... 46
4.2 Background and Overview ....................................... 49
4.3 Core Constraint Language ...................................... 53
  4.3.1 Constraint Syntax .......................................... 54
  4.3.2 Constraint Validity and Satisfiability ....................... 55
  4.3.3 Expressiveness of the Core Constraint Language .......... 57
  4.3.4 Alternative Formalizations ................................ 61
4.4 The Derivation Framework ...................................... 62
  4.4.1 Derivation Framework and Its Instances .................... 62
  4.4.2 Proof Framework ........................................... 65
4.5 Algorithms ...................................................... 71
  4.5.1 Partition Method ........................................... 71
  4.5.2 Derivation Method .......................................... 74
  4.5.3 Constraint Solving Algorithms ................................ 75
4.6 Evaluation ....................................................... 79
  4.6.1 Benchmarks ................................................ 80
  4.6.2 Performance of Inference Algorithms ....................... 81
  4.6.3 Scalability ................................................ 82
  4.6.4 Sequential vs. Combinational Partitioning .................. 83
4.7 Related Work .................................................. 84
4.8 Summary ....................................................... 86

Chapter 5
A General-Purpose Dynamic Information Flow Policy 87
5.1 Introduction .................................................... 87
5.2 Background and Overview ....................................... 90
  5.2.1 Security Levels ............................................. 90
  5.2.2 Terminology ............................................... 91
  5.2.3 Overview .................................................. 92
5.3 Dynamic Policy Specification .................................. 94
  5.3.1 Language Syntax and Security Specification ............... 94
  5.3.2 Expressiveness ............................................. 97
# List of Figures

3.1 Examples: Imprecise Information Flow Analysis Rejects Secure Programs. . 13  
3.2 Syntax of the Source Language. .......................... 17  
3.3 Semantics of the Source Language. ....................... 17  
3.4 Program Transformation. We use $A := A'$ as a shorthand for $\{A(v) := A'(v) | v \in Vars \land A(v) \neq A'(v)\}$. ....................... 19  
3.5 Merge Function. ........................................ 19  
3.6 Examples: Implicit Declassification. ..................... 25  
3.7 Examples: Implicit Declassification in Loop. ............. 26  
3.8 Syntax of Security Labels. ................................. 27  
3.9 Liveness Analysis of $\mathcal{L}_A$. ........................ 29  
3.10 Typing Rules: Expressions. ............................... 30  
3.11 Typing Rules: Commands. ................................ 30  
3.12 Soundness of original and transformed programs. .......... 35  
3.13 Erasure Semantics of Assignment. ......................... 36  
3.14 The Hunt and Sands System. .............................. 37  
3.15 Type Construction in Transformed Program. ................ 39  

4.1 Constraints and Sound Derivations. ......................... 51  
4.2 Sound and Complete Derivations. .......................... 52  
4.3 Syntax of Constraints. .................................... 54  
4.4 Correctness of a Solution $\kappa$. .......................... 55  
4.5 Program and Constraints using Boolean Expression Predicates. ........... 58  
4.6 Program and Constraints using Permission Trace. ........... 59  
4.7 Constraints and Complete Derivations. ..................... 64  
4.8 Illustrated Restrictions on Predicates. ..................... 66  
4.9 One-Shot Solver. .......................................... 76  
4.10 Early Accept Solver. ...................................... 77  
4.11 Hybrid Solver. ............................................ 78  
4.12 Performance on the MIPS benchmark. Each algorithm times out after 180s (3 mins). The gray area shows executions that time out after 180s. The X-axis represents file ID. ......................................................... 81  
4.13 Performance on the MIPS benchmark in log-log scale. The X-axis represents unique predicates involved in all constraints. ............................... 82  
4.14 Partition: Sequential vs. Combinational. .................. 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Examples of Dynamic Policies</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>Language Syntax with Security Specification</td>
<td>95</td>
</tr>
<tr>
<td>5.3</td>
<td>An Example of Encoding Paralock for $A = \langle a, {D} \rangle$</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>Interpretation of Security Labels</td>
<td>103</td>
</tr>
<tr>
<td>5.5</td>
<td>Annotated Program for Fig. 5.1-B(i)</td>
<td>127</td>
</tr>
<tr>
<td>5.6</td>
<td>Counterexamples for Crypto-Erasure and Paralocks</td>
<td>130</td>
</tr>
<tr>
<td>6.1</td>
<td>Decomposition of Dynamic Policy</td>
<td>138</td>
</tr>
<tr>
<td>6.2</td>
<td>Decomposition of A Persistent Policy</td>
<td>138</td>
</tr>
<tr>
<td>6.3</td>
<td>Decomposition of Program with False Data Dependencies</td>
<td>139</td>
</tr>
<tr>
<td>6.4</td>
<td>Language Syntax with Security Specification</td>
<td>142</td>
</tr>
<tr>
<td>6.5</td>
<td>Policy Decomposition</td>
<td>144</td>
</tr>
<tr>
<td>7.1</td>
<td>False Control-Flow Dependency</td>
<td>152</td>
</tr>
<tr>
<td>A.1</td>
<td>Extended Syntax</td>
<td>162</td>
</tr>
<tr>
<td>A.2</td>
<td>Equivalence on Memories and Commands</td>
<td>162</td>
</tr>
<tr>
<td>A.3</td>
<td>Extended Semantics: Expressions</td>
<td>163</td>
</tr>
<tr>
<td>A.4</td>
<td>Extended Semantics: Commands</td>
<td>163</td>
</tr>
<tr>
<td>A.5</td>
<td>Extended Typing Rules: Expressions</td>
<td>164</td>
</tr>
<tr>
<td>A.6</td>
<td>Extended Typing Rules: Commands</td>
<td>164</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Existing End-to-End Security Policies and Dynamic Release Written in the Formalization Framework. .................................................. 116

5.2 Evaluation Results. ................................................................. 127
Acknowledgments

This dissertation would not have been possible without the support from many people along the way. First and foremost, I am extremely grateful to my adviser, Dr. Danfeng Zhang for his invaluable advice, constant support, and willingness and patience to understand my research ideas however unsettled and obscure they were. I also would like to thank Dr. Trent Jaeger, Dr. Gang Tan and Dr. Dinghao Wu for serving as my dissertation committee members. I truly appreciate their time and feedback on improving the quality of my research and dissertation. Additionally, I would like to thank my mentors, Dr. Deepak Garg at MPI-SWS and Dr. Michael Emmi at Amazon for their guidance and efforts to make my summer internship special.

I also want to acknowledge the numerous friendships that have come from working alongside so many intelligent and kind people. I would like to especially thank my fellow and former colleagues Xulong Tang, Chen Chen, Dongrui Zeng, Lun-Pin Yuan, Yanjun Gao, Haibo Zhang and Xinxin Cai for their time to help me get through those moments of confusion, and for the brunches and coffees that made the long days/weeks better.

Lastly, I would not have made it this far without the love and support from my family. I want to thank my husband for all his love, encouragement and unwavering support over the last 12 years. I would also like to thank my father and mother for their unconditional love and firm support to let me pursue my passions.

Funding Acknowledgement  The work presented in this dissertation is supported by NSF grants CCF-1566411, CNS-1702760, CNS-1816282 and CNS-1942851. The findings and conclusions presented in this document are those of the authors and do not necessarily reflect the views of the organizations listed previously.
Dedication

To my husband, who lights up my day.
Chapter 1  |  Introduction

Information flow control has been a widely adopted approach to protect sensitive information manipulated by computing systems, due to the strong end-to-end security guarantee that it provides. In a classic information flow analysis, secrets are stored in variables and programmers provide security levels (e.g., P for public and S for secret) for all variables to describe their intended secrecy. An information flow analysis tracks how the values are processed inside a computing system and prevents any illegal information flow. Conventionally, we assume that all security levels form a lattice [1], and information can only go upwards in the lattice; otherwise, a program is insecure [2]. For example, given a two-point lattice P ⊑ S, an information flow from P to S is secure, but a flow from S to P is insecure. In this classic setting, the end-to-end security guarantee of information flow control is backed by the security property known as noninterference [3], which states that secrets cannot interfere any public outputs that are observable by an attacker.

However, classic information flow analysis still faces a few important limitations, which makes it impractical for many real-world applications. The first limitation is that reducing false alarms is a challenging task. In program analysis, false alarms are the cases where a secure program is rejected by an (imprecise) program analysis. While it is sometimes impossible to eliminate all false alarms, a high rate of false alarms greatly reduces the usefulness of a program analysis. Flow-sensitivity describes the ability to remove false alarms that are rooted from
ignoring statement order. For example, a flow-sensitive information flow analysis can correctly accept the program \((p := s; p := 0;)\) where \(s\) is secret and \(p\) is public as the final value of \(p\) does not reveal the value of \(s\). Similarly, a path-sensitive analysis removes false alarms due to ignoring the predicates at conditional branches. For example, a path-sensitive information flow analysis can correctly accept the following secure program since it can rule out the insecure sequence of statements \((y := s; p := y;)\) due to their conflicting branch predicates.

\[
\begin{align*}
\text{if } (x = 1) \text{ then } y &:= 0 \text{ else } y := s; \\
\text{if } (x = 1) \text{ then } p &:= y
\end{align*}
\]

The classic information flow analysis approach does not take into consideration for flow- or path-sensitivity. Hunt and Sands [4] propose a flow-sensitive type system which allows a variable to have multiple security levels over the course of computation. However, this floating-level system is more costly than classic fixed-level systems, and it is still path-insensitive. At the same time, dependent security labels (or dependent labels in short) are security labels that may depend on concrete program states. They are introduced in information flow analysis to improve expressiveness, and the added expressiveness leads to successful verification of real-world systems, such as a MIPS processor [5], conference management systems [6, 7], a TrustZone-like architecture [8] and Android Apps [9]. However, the connection between dependent labels and path-sensitivity was not fully explored before.

The first contribution of this dissertation is to propose a flow- and path-sensitive information flow analysis that is precise enough to accept the aforementioned secure programs. The proposed analysis consists of two major components: a general purpose program transformation that removes false dataflow dependencies that otherwise compromise the precision of a fixed-level system, as well as a fixed-label type system with dependent labels. Each component of our analysis targets one insensitive source of previous type systems. The modular design not only enables tunable precision of our analysis, but also sheds light on the design of security type systems: we show that a fixed-level system (e.g., [10]) plus the program transformation is as precise as the flow-sensitive system presented in [4].
Meanwhile, one big obstacle of information flow analysis with the promising dependent security labels is that it requires programmers to write down all (dependent) labels. However, since the dependent labels usually involve intricate security invariants, providing those labels requires a deep understanding of the program being analyzed, making it a both time-consuming and error-prone process. Moreover, when the provided labels are incorrect (i.e., program analysis fails with the provided labels), it is unclear if the program being analyzed is insecure, or the program can be verified with other correct labels.

For security labels without dependence, classic security type systems (such as Jif [11] and FlowCaml [12]) encode the restrictions on security labels into constraints in a finite semilattice. Those constraints can be solved by customized solvers (such as the Rehof-Mogensen algorithm [13] and set-constraints solvers [14, 15]). However, they cannot handle the infinite search space of dependence required for dependent labels.

The second major contribution of this dissertation is to propose the first general framework for designing security dependent label inference algorithms and checking their correctness. For generality, we propose a core constraint language that, to the best of our knowledge, can encode all static information flow analyses with dependent security labels. The framework models security label inference as an iterative process of solving derivations (typically, simpler constraints) from the original constraint set. Hence, it allows great flexibility in inference algorithm design. To facilitate the design and validation (in terms of soundness and completeness) of algorithms in the derivation framework, we distill the key properties for making an algorithm sound and/or complete. We show that various algorithms, including an extension to the inference algorithm in existing work [9], can be checked under the framework in a straightforward manner. Moreover, we also designed three novel sound and complete algorithms, namely, early-accept algorithm, early-reject algorithm, and hybrid algorithm. Based on a mix of both satisfiable and unsatisfiable constraint sets collected from the verification of an information flow policy on a MIPS processor using SecVerilog [5, 8], we found that the novel algorithms solve predicated constraints faster than an existing algorithm [9] by orders of magnitude.
The second limitation of the classic information flow analysis approach is that it assumes that secrecy of information remains fixed throughout program execution. However, the information flow policies in real applications are rarely a static notion [16, 17]. The limitation motivates the development of various kinds of *dynamic* information flow policies in the literature:

- A *declassification policy* [18–25] weakens noninterference by deliberately releasing (i.e., declassifying) sensitive information. For instance, a conference management system typically allows deliberate release of paper reviews and acceptance/rejection decisions after the notification time.

- An *erasure policy* [26–31] strengthens noninterference by requiring some public information to become more sensitive, or be erased completely when certain condition holds. For example, a payment system should not retain any record of credit card details once the transaction is complete.

- An *delegation/revocation policy* [32–35] updates dynamically the sensitivity roles in a security system to accommodate the mutable requirements of security, such as delegating/revoking the access rights of a new/leaving employee.

Moreover, there are a few case studies on the needed security properties in the light of one specific context or task [36–39], and build systems that provably enforces some variants of declassification policy (e.g., CoCon [40], CosMeDis [41]) and erasure policy (e.g., Civitas [42]). Although existing works make it possible to specify and verify some variants of dynamic policy, cherry-picking the appropriate policy is still a daunting task: different policies (even when they belong to the same kind) have very different syntax for specifying how a policy changes [43], very different nature of the security conditions (i.e., noninterference, bisimulation and epistemic [44]) and even completely inconsistent notion of security (i.e., policies might disagree on whether a program is secure or not [44]). So even for veteran researchers in information flow security, understanding the subtleties in the syntax and semantics of each...
policy is difficult, evidenced by highly-cited papers that synthesize existing knowledge on
declassification policy [43] and dynamic policy [44]. Arguably, it is currently impossible for a
system developer/user to navigate in the jungle of unconnected policies (even for the ones in
the same category) when a dynamic policy is needed [43, 44].

The third major contribution of this dissertation is to propose Dynamic Release, the first
information flow policy that enables declassification, erasure, delegation and revocation at
the same time. Moreover, Dynamic Release is built on a novel formalization framework that
is shown to subsume existing security conditions that are formalized in different ways (e.g.,
noninterference, bisimulation and epistemic [44]). More importantly, for the first time, the
formalization framework allows us to make apple-to-apple comparison among existing policies,
which are incompatible before (i.e., we cannot trivially convert one to another). We also build
a new AnnTrace benchmark for testing and understanding variants of dynamic policies in
general. The benchmark consists of examples with dynamic policies from existing papers, as
well as new subtle examples that we created in the process of understanding dynamic policies.
We implemented our policy and existing policies, and found that Dynamic Release is the only
one that is both applicable and correct on all examples.

On the other hand, existing enforcement of dynamic policies is still unsatisfactory: existing
methods are mostly policy-specific. Even for policies of the same kind, different enforcement
has completely different setups that makes it unclear whether the insights can be shared.

While it is hard to connect enforcements between different dynamic policies, we find that
a enforcement of dynamic policy almost always resembles enforcements that are designed
for static policies. We thus shift our perspective to consider enforcement of dynamic policy
in a different way: Is there a principled way to reuse enforcement on static policy for a
general-purpose dynamic policy? Based on the study of Dynamic Release, we can specify a
general-purpose dynamic policy as sensitivity mutations triggered by a set of security events or
program states. So intuitively, a dynamic policy can be interpreted as a sequence of code blocks
separated by policy mutations, where sensitivity within each code block is stable (i.e., static).
The forth major contribution of this dissertation is to draw up the conditions for a both sound and complete decomposition of a general dynamic policy. We formalize and prove that for a transient policy, it is possible to decompose it soundly and completely. Therefore, to enforce a transient policy, we can reuse well-studied enforcements of static policies in the literature. However, sound and complete decomposition of a persistent policy is infeasible, as the policy by definition needs to exam the history of program execution to determine if an information flow is legal or not.

**Outline**  The rest of this dissertation is organized in the following manner. Chapter 2 presents relevant background information. Chapter 3 integrates dependent type to build a flow- and path-sensitive information flow analysis. Chapter 4 presents an inference framework to derive dependent labels needed for the flow- and path-sensitive analysis. Chapter 5 introduces our semantic framework for general dynamic policy, and the end-to-end security property Dynamic Release. Chapter 6 discusses our strategy to decompose dynamic policies into static policies. Lastly, we provide our thoughts for interesting lines of future work in Chapter 7 and some closing remarks in Chapter 8.
Chapter 2  
Background

2.1 Information Flow Security

We first review standard information flow terminology.

Security Levels and Lattice  We assume all variables are associated with security levels. A security policy is specified as the ordering of the security levels, typically in the form of a security lattice. For data $d_1$ with security level $\ell_1$ and data $d_2$ with level $\ell_2$, the policy allows information flow from $d_1$ to $d_2$ if and only if $\ell_1 \sqsubseteq \ell_2$. In this dissertation, we use two distinguished security levels S (Secret) and P (Public) for simplicity, but keep in mind that the proposed theory is general enough to express richer security levels. The security policy on the levels P and S is defined as $P \sqsubseteq S$, while $S \not\sqsubseteq P$. That is, information flow from public data to secret variable is allowed, while the other direction is forbidden. Hereafter, we assume variable s is labeled as S, and variable p is labeled as P unless specified otherwise.

Explicit and Implicit Flows  An information flow analysis prohibits any explicit or implicit information flow that is inconsistent with the given policy. Explicit flows take place when confidential data are passed directly to public variables, such as the command $p := s$, while implicit flows arise from the control structure of the program. For example, the following
program has an implicit flow:

\[
\text{if } (s = 0) \text{ then } p := 0 \text{ else } p := 1
\]

Assume that the secret variable \( s \) is either 0 or 1. This code is insecure since it is functionally equivalent to \( p := s \). That is, the confidential data \( s \) is copied to a public variable \( p \).

An information flow analysis rules out all explicit and implicit flows; any violation of a given security policy results in an error. As in most information flow analyses, we do not consider timing, termination and other side channels in this dissertation; controlling side channel leakage (e.g., [45–47]) is largely an orthogonal issue.

### 2.2 Information Flow Analysis on Static Policies

We refer to [16] for a comprehensive survey of static information flow analysis. Here we focus on more relevant terminology and background information that are relevant to this dissertation.

A flow-sensitive information flow analysis [4, 48, 49] allows a variable to have different security labels at different program points (a.k.a., floating labels). The design of floating labels naturally preserves flow-sensitivity. Thus, false alarms can be reduced compared with flow-insensitive analysis. At the same time, however, extra efforts are needed to keep track of the security label at each program point.

Dependent labels are introduced in information flow analysis for various reasons, and in most of the cases, it eventually leads to a more accurate outcome. For example, SecVerilog is a Verilog-like language with dependent security labels for verifying timing-sensitive noninterference in hardware designs. A recent extension to SecVerilog [50] shows that dependent label can be used to recognize branch predicates and thus allows path-sensitivity.


2.3 Information Flow Analysis on Dynamic Policies

A declassification policy [18–25] deliberately releases sensitive information. For example, the seminal work Gradual Release [18] uses a special command declassify($e$) to downgrade the sensitivity of value $e$ when the command is evaluated. The Gradual Release policy is designed for declassification, however, the knowledge-based formalization it proposed is later widely-adopted to reason about the end-to-end security for various dynamic policies.

In knowledge-based formalization, information leakage is measured as attacker’s knowledge. The main insight is that the attacker’s knowledge gained by observing the outputs of a program should stay the same unless it encounters an release event. Later, Tight Gradual Release [51,52] extends Gradual Release with a tighter bound on the leakage during a release event.

While knowledge-based formalization works well for declassification policy, it becomes less applicable when upgrading policies, like a erasure policy [26–31], comes into the picture. Many works [26, 27, 33–35, 53–55] use various different forms of formalization to formalize their intended dynamic policies. For example, forgetful attacker model [32, 56] is proposed to express various kinds of attackers, so that an upgrading policy can be modeled as “forgetting” learned knowledge. RIF [53, 54] uses reclassification relation to associate label changes with program outputs. While these policies are self-contained for its own purpose, there is no consensus on how to specify or enforce a dynamic policy in general.
Chapter 3  |  
A Path- and Flow-Sensitive Information Flow Analysis

3.1 Introduction

Since Denning and Denning’s seminal paper [2], static program analysis has been widely adopted for information-flow control [16]. Among these program analyses, type systems (e.g., [10, 57, 58]) have enjoyed a great popularity due to their strong end-to-end security guarantee, and their inherently compositional nature to combine secure components forming a larger secure system as long as the type signatures agree..

Many security type systems (e.g., [10,57,58]) assume fixed levels. That is, the security level for each variable remain unchanged throughout program execution. Though this fixed-level assumption simplifies the design of those type systems, one consequence is that they tend to be over-conservative (i.e., reject secure programs). For example, given that $s$ has a level $S$ (i.e., $s$ holds a secret value) and $p$ has a level $P$, a fixed-level type system rejects secure programs, such as $(p := s; p := 0;)$, even though the publicly observable final value of $p$ is always zero.

Previous work (e.g., [4]) observes that such inaccuracy roots from the flow-insensitive nature (i.e., the order of program execution is ignored) of fixed-level systems. From this perspective,
the previous example is mistakenly considered insecure because the (impossible) execution order \( (p := 0; p := s;) \) is insecure.

Hunt and Sands [4] propose a classic flow-sensitive type system which allows a variable to have multiple security levels over the course of computation. For example, this floating-level type system correctly accepts the program \( (p := s; p := 0;) \) by assigning \( p \) with levels \( S \) and \( P \) after the first and second assignments respectively. However, this floating-level system is still path-insensitive, meaning that the predicates at conditional branches are ignored in the analysis. For example, it incorrectly rejects the following secure program since the (impossible) branch combination \( (y := s; p := y;) \) is insecure.

\[
\begin{align*}
\text{if } (x = 1) \text{ then } y := 0 & \text{ else } y := s; \\
\text{if } (x = 1) \text{ then } p := y
\end{align*}
\]

In this chapter, we develop a flow- and path-sensitive information flow analysis that is precise enough to accept the aforementioned secure programs. The novel analysis is built on two key observations. First, flow-sensitivity can be gained via a general-purpose program transformation that eliminates false dataflow dependencies that confuse a flow-insensitive type system. Consider the example \( (p := s; p := 0;) \) again. The transformation removes the false dataflow dependency between \( s \) and \( p \) by introducing an extra copy of the variable \( p \) and keeps track of the final copy of each variable at the same time. So, the example is transformed to \( (p_1 := s; p_2 := 0;) \), where \( p_2 \) is marked as the final copy. Then, a fixed-level system can easily type-check this program by assigning levels \( S \) and \( P \) to \( p_1 \) and \( p_2 \) respectively.

Second, path-sensitivity can be gained via consolidating dependent type theory (e.g., [59–61]) into security labels. That is, a security label is, in general, a function from program states to security levels. Consider the second example above with branches. We can assign \( y \) a dependent security label: \( (x = 1?P : S) \), meaning that the level of \( y \) is \( P \) when \( x = 1 \), and \( S \) otherwise. Hence, the information flow from \( y \) to \( p \) can be judged as secure since it only occurs when \( x = 1 \) (hence, \( y \) has level \( p \)).

Based on the key observations, we propose a flow- and path-sensitive information flow
analysis that consists of two major components: a general purpose program transformation that removes false dataflow dependencies that otherwise compromise the precision of a fixed-level system, as well as a fixed-label type system with dependent labels. Each component of our analysis targets one insensitive source of previous type systems. The modular design not only enables tunable precision of our analysis, but also sheds light on the design of security type systems: we show that a fixed-level system (e.g., [10]) plus the program transformation is as precise as\(^1\) the classic flow-sensitive system in [4]; furthermore, a fixed-label dependent type system can soundly control information flow in the presence of mutable variables without resorting to run-time mechanisms (e.g., [5, 8]).

In this chapter, our key contributions are:

1) We formalize a novel flow- and path-sensitive information flow analysis for a simple WHILE language. The analysis consists of a novel program transformation, which eliminates imprecision due to flow-insensitivity (Section 3.5), and a purely static type system using dependent security labels (Section 3.6).

2) We formally prove the soundness of our analysis (Section 3.7): the source program satisfies termination-insensitive noninterference whenever the transformed program type-checks. Novel proof techniques are required due to the extra variables introduced (for added precision) in the transformed program.

3) We show that our analysis is strictly more precise than a classic flow-sensitive type system [4] (Section 3.8). One interesting consequence is that the program transformation automatically makes a sound flow-insensitive type system (e.g., [10]) as precise as the classic flow-sensitive system [4].

\(^1\)We note that in the information flow literature, different terms (such as “precision” and “permissiveness”) have been used to compare the amount of false positives of various mechanisms [62]. We say a static analysis A is as precise as a static analysis B if A accepts every secure program that is accepted by B. Moreover, we say A is (strictly) more precise than B if A is as precise as B, and A accepts at least one secure program that is rejected by B.
1 \ x := s; \\
2 \ [ x := 0 ]; \\
3 \ p := x;

(a) Flow-Insensitive Analysis Rejects Secure Program.

1 \ x := s; \\
2 \ x_1 := 0; \\
3 \ p := x_1;

(b) Flow-Insensitive Analysis Accepts Equivalent Program.

1 \ x := 0; \ y := 0; \\
2 \ \text{if} \ (p_1 < 0) \ \text{then} \ y := s; \\
3 \ \text{if} \ (p_1 > 0) \ \text{then} \ x := y; \\
4 \ p_2 := x;

(c) Path-Insensitive Analysis Rejects Secure Program.

Figure 3.1. Examples: Imprecise Information Flow Analysis Rejects Secure Programs.

4) We show that our dependent type system soundly controls information flow in the presence of mutable variables without resorting to dynamic mechanisms, such as the dynamic erasure mechanism in previous work [5,8].

3.2 Sources of Imprecision

Most information flow analyses provide soundness (i.e., if the analysis determines that a program is secure, then the program provably prevents disclosure of sensitive data). However, since the problem of checking information flow security is in general undecidable [16], one key challenge of designing an information flow analysis is to maintain soundness, while improving precision (i.e., reject fewer secure programs).

In this section, we introduce the major sources of imprecision in existing type systems. In the next section (Section 3.3), we illustrate how does our novel information flow analysis alleviate those sources of imprecision.

Flow-Insensitivity The first source of imprecision is flow-insensitivity, meaning that the order of execution is not taken into account in a program analysis [63]. In the context of information flow analysis, the intuition is that an analysis is flow-insensitive if a program is analyzed as secure only when every subprogram is analyzed as secure [4].

Many security type systems, including [10, 57, 58], are flow-insensitive. Consider the program in Figure 3.1(a) (for now, ignore the brackets). This program is secure since the public
variable $p$ has a final value zero regardless of the secret variable $s$. However, it is considered insecure by a flow-insensitive analysis because of the insecure subprogram $(x := s; p := x; )$. Under the hood, the imprecision arises since the analysis requires fixed levels: the security level of a variable must remain the same throughout the program execution. But in this example, there is no fixed-level for the variable $x$: when the level is $S$, $p := x$ is insecure; when the level is $P$, $x := s$ is insecure.

**Path-Insensitivity** The second source of imprecision is path-insensitivity, meaning that the predicates at conditional branches are ignored in a program analysis [63]. In the context of information flow analysis, the intuition is that an analysis is path-insensitive if a program is analyzed as secure only when every sequential program generated from one combination of branch outcomes is analyzed as secure.

For instance, the flow-sensitive type system in [4] is path-insensitive; consequently, it rejects the secure program shown in Figure 3.1(c) (due to Le Guernic and Jensen [64]). This example is secure since the value of the secret variable $s$ never flows to the public variable $p_2$, since the assignments $y := s$ and $x := y$ never execute together in the same program execution. However, the type system in [4] rejects this program because it lacks the knowledge that the two if-statements cannot take the “then” branch in the same execution. Hence, it has to conservatively analyze the security of an impossible program execution: $x := 0; y := 0; y := s; x := y; p := x$, which is insecure due to an explicit flow from $s$ to $p$.

Under the hood, we observe that the imprecision arises from the fact that a path-insensitive analysis (e.g., [4]) requires that the security levels of a variable on two paths to be “merged” (as the least upper bound) after a branch. Consider the first branch in Figure 3.1(c). The “then” branch requires $y$ to be $S$ due to the flow from $s$ to $y$. So after that if-statement, the label of $y$ must be $S$ (i.e., which path is taken is unknown to the rest of the program). Similarly, $x$ has label $S$ after the second if-statement. Hence, $p_2 := x$ is rejected due to an explicit flow from $S$ to $P$. 

14
3.3 Overview

In order to alleviate analysis imprecision due to flow- and path-insensitivity, our novel information flow analysis has two major components: a program transformation that enables flow-sensitivity and a type system with dependent security labels, which enables path-sensitivity.

3.3.1 Program Transformation

Consider the example in Figure 3.1(a) (for now, ignore the brackets). A fixed-level type system rejects this program since the levels of $x$ at line 1 and 3 are inconsistent. We observe that there are indeed two copies of $x$ in this program but only the final one (defined at line 2) is released. So without modifying a type system, we can explicitly transform the source program to a semantically equivalent one that explicitly marks different copies.

The source language of our program analysis (Section 3.4) provides a tunable knob for improved precision: a bracketed assignment in the form of $[x := e]$. Such an assignment is semantically identical to $x := e$ but allows a programmer to request improved precision (the source language allows such flexibility since reduced precision might be preferred for reasons such as more efficient analysis on the program). In particular, for a bracketed assignment $[x := e]$, the program transformation (Section 3.5) generates a fresh copy for $x$ and uses that copy in the rest of program until another new copy is generated. For example, given the bracketed assignment at line 2 of Figure 3.1(a), the transformed program is shown in Figure 3.1(b), where the second definition of $x$ and its use at line 3 are replaced with $x_1$. The benefit is that the false dataflow dependency from $s$ to $p$ in the source program is eliminated. Hence, the transformed program can be accepted by a fixed-level type system, by assigning $x$ and $x_1$ to levels $S$ and $P$ respectively. In general, we prove that (when all assignments are bracketed) the transformation enables a fixed-level system to be at least as precise as a classic flow-sensitive type system (Section 3.8).
3.3.2 Dependent Labels

Consider the example in Figure 3.1(c). A path-insensitive type system rejects this program since such a type system ignores the path conditions under which assignments occur. Consequently, the security level of \( y \) is conservatively estimated as \( S \) after line 2, though when \( p_1 \leq 0 \), variable \( y \) only carries public information.

In our system, path-sensitivity is gained via dependent security labels (i.e., security labels that depend on program states). Compared with a security level drawn directly from a lattice, a dependent security label precisely tracks all possible security levels from different branches; hence, path-sensitivity is gained. Since dependent security labels are orthogonal to bracketed assignments, extra precision can be gained in our system even in the absence of bracketed assignments. For example, while the program in Figure 3.1(c) cannot be accepted using any simple security level for \( y \), we can assign to \( y \) a dependent label \((p_1 < 0?S : P)\), which specifies an invariant that the level of \( y \) is \( S \) when \( p_1 < 0 \) (i.e., the “then” branch is taken at line 2); the level is \( P \) otherwise. Such an invariant can be maintained by the type system described in Section 3.6. For instance, to ensure that the explicit flow from \( y \) to \( x \) at line 3 is secure, the type system generates a proof obligation \((p_1 > 0 \Rightarrow (p_1 < 0?S : P) \sqsubseteq P)\), meaning that the information flow from \( y \) to \( x \) must be permissible under the path condition \( p_1 < 0 \). This proof obligation can easily be discharged by an external solver. The soundness of our type system (Section 3.7) guarantees that all security violations are detected at compile time.

3.4 Language Syntax and Semantics

In this chapter, we consider a simple imperative WHILE language whose syntax and operational semantics are shown in Figures 3.2 and 3.3 respectively. The syntax and semantics are mostly standard: expressions \( e \) consist of variables \( x \), integers \( n \), and composed expressions \( e \text{ op } e \), where \( \text{op} \) is a binary arithmetic operation. Commands \( c \) consist of standard imperative
Figure 3.2. Syntax of the Source Language.

<table>
<thead>
<tr>
<th>Vars</th>
<th>$x, y, z \in \text{Vars}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>$e ::= x \mid n \mid e \text{ op } e$</td>
</tr>
<tr>
<td>Cmds</td>
<td>$c ::= \text{skip} \mid c_1; c_2 \mid x := e \mid {x := e} \mid$</td>
</tr>
<tr>
<td></td>
<td>if ($e$) then $c_1$ else $c_2$ \mid \text{while} ($e$) $c$</td>
</tr>
</tbody>
</table>

Figure 3.3. Semantics of the Source Language.

- **S-Skip**
  $$\langle \text{skip}; c, m \rangle \rightarrow \langle c, m \rangle$$

- **S-Assign**
  $$\langle e, m \rangle \downarrow n \quad \langle x := e, m \rangle \rightarrow \langle \text{skip}, m\{x \mapsto n\} \rangle$$

- **S-Assign-Bracket**
  $$\langle [x := e], m \rangle \rightarrow \langle \text{skip}, m\{x \mapsto n\} \rangle$$

- **S-Seq**
  $$\langle c_1; c_2, m \rangle \rightarrow \langle c_1' ; c_2' , m' \rangle$$

- **S-While**
  $$\langle \text{while} (e) \ c, m \rangle \rightarrow \langle \text{if} (e) \ \text{then} \ (c; \ \text{while} (e) \ c) \ \text{else} \ \text{skip}, m \rangle$$

- **S-If1**
  $$\langle e, m \rangle \downarrow n \quad n \neq 0 \quad \langle \text{if} (e) \ \text{then} \ c_1 \ \text{else} \ c_2, m \rangle \rightarrow \langle c_1, m \rangle$$

- **S-If2**
  $$\langle e, m \rangle \downarrow n \quad n = 0 \quad \langle \text{if} (e) \ \text{then} \ c_1 \ \text{else} \ c_2, m \rangle \rightarrow \langle c_2, m \rangle$$

Instructions, including skip, sequential composition $c_1; c_2$, assignments, conditional if branch and while loop. The semantics of expressions are given in the form of $\langle e, m \rangle \downarrow n$ (big-step semantics), where memory $m$ maps variables to their values. The small-step semantics of commands has the form of $\langle c, m \rangle \rightarrow \langle c', m' \rangle$, where $\langle c, m \rangle$ is a configuration. We use $m\{x \mapsto n\}$ to denote the memory that is identical to $m$ except that variable $x$ is updated to the new value $n$.

The only interesting case is the bracketed assignment $[x := e]$, which is semantically
equivalent to normal assignment $x := e$ in the source language. These commands are tunable knobs for improved precision in our information flow analysis, as we show shortly.

3.5 Program Transformation

To alleviate the imprecision due to flow-insensitivity, one component of our analysis is a novel program transformation that introduces extra variable copies to the source program, so that false dataflow dependencies that otherwise may confuse flow-insensitive analyses are removed.

3.5.1 Bracketed Assignments and the Transformed Program

We propose a general and flexible design for the program transformation. In particular, the program transformation is triggered only for assignments that are marked with brackets. Such a design enables a tunable control of analysis precision for programmers or high-level program analysis built on our meta source language: when there is no bracketed assignment, the transformed program is simply identical to the source program; when all assignments have brackets, the transformation generates a fresh copy of $x$ for each bracketed assignment $[x := e]$.

Due to the nature of the transformation, the transformed program follows the same syntax and semantics as the source language, except that all bracketed assignments are removed.

To avoid confusion, we use underlined notations for the transformed program: $e$ for expressions, $c$ for commands and $m$ for memories, when both the original and the transformed programs are in the context; otherwise, we simply use $e$, $c$ and $m$ for the transformed programs as well.

3.5.2 Transformation Rules

The program transformation maintains one active copy for each variable in the source code. One invariant maintained by the transformation is that for each program point, there is exactly
one active copy for each source-program variable. Intuitively, that unique active copy holds the most recent value of the corresponding source-program variable.

**Definition 1 (Active Set)** An active set \( A : \text{Vars} \mapsto \text{Vars} \), is an injective function that maps a source variable to a unique variable in the transformed program.

For simplicity, we assume that the variables in the transformed program follow the naming convention of \( x_i \) where \( x \in \text{Vars} \) and \( i \) is an index. Hence, for any variable \( v \) in the range of \( A \), we simply use \( \lfloor v \rfloor \) to denote its corresponding source variable (i.e., a variable without the
index). Hence, $v = A\lfloor v \rfloor$ always holds by definition. Moreover, since we frequently refer to the range of $A$, we abuse the notation of $A$ to denote active copies that $A$ may map to (i.e., the range of $A$). That is, we simply write $v \in A$ instead of $v \in \text{Ran}(A)$. Moreover, we use $A\{x \mapsto x_i\}$ to denote an active set that is identical to $A$ except that $x$ is mapped to $x_i$.

The transformation rules are summarized in Figure 3.4. For an expression $e$, the transformation has the form of $\langle e, A \rangle \Rightarrow e'$, where $e'$ is the transformed expression. The transformation of an expression simply replaces the source variables with their active copies in $A$.

For a command $c$, the transformation has the form of $\langle c, A \rangle \Rightarrow \langle c', A' \rangle$, where $c'$ is the source command and $c'$ is the transformed one. Since assignments may update the active set, $A'$ represents the active set after $c$.

Rule (TRSF-Assign) applies to a normal assignment. It transforms the assignment to one with the same assignee and update $A$ accordingly. Rule (TRSF-Assign-Create) applies to a bracketed assignment $[x := e]$. It renames the assignee to a fresh variable. For example, line 1 of the transformed program in Figure 3.1(b) is exactly the same as the original program in 3.1(a); but the assignee of line 2 is renamed to $x_1$. Rule (TRSF-If) uses a special $\Phi$ function, defined in Figure 3.5, to merge the active sets generated from the branches. In particular, $\Phi(A_1, A_2) \Rightarrow A_3$ generates an active set $A_3$ that maps $x$ to a fresh variable iff $A_1(x) \neq A_2(x)$. Transformation for the while loop is a little tricky since we need to compute an active set that is active both before and after each iteration. Rule (TRSF-While) shows one feasible approach: the rule transforms the loop in a way that $A_1$ is a fixed-point: the active set is always $A_1$ before and after an iteration by the transformation.

We note that given an identity function as the initial active set $A$, a program without any bracketed assignment is transformed to itself with a final active set $A$. At the other extreme, the transformation generates one fresh active copy for each assignment when all assignments are bracketed.
3.5.3 Correctness of the Transformation

One important property of the proposed transformation is its correctness: a transformed program is semantically equivalent to the source program. To formalize this property, we need to build an equivalence relation on the memory for the source program \((m : \text{Vars} \rightarrow \mathbb{N})\) and the memory for the transformed program \((\underline{m} : \text{Vars} \rightarrow \mathbb{N})\). We note that the projection of \(m\) on an active set \(\mathcal{A}\) defined as follows shares the same domain and range as \(m\). Hence, it naturally specifies an equivalence relation on \(m\) and \(\underline{m}\) w.r.t. \(\mathcal{A}\): \(m\) can be directly compared with \(m^\mathcal{A}\).

**Definition 2 (Memory Projection on Active Set)** We use \(m^\mathcal{A}\) to denote the projection of \(m\) on the active set \(\mathcal{A}\), defined as follows:

\[
\forall x \in \text{Vars}. \quad m^\mathcal{A}(x) = m(\mathcal{A}(x))
\]

We formalize the correctness of our transformation as the following theorem. As stated in the theorem, the correctness is not restricted to any particular initial active set \(\mathcal{A}\).

**Theorem 1 (Correctness of Transformation)** Any transformed program is semantically equivalent to its source:

\[
\forall c, \underline{c}, m, m', m', \mathcal{A}, \mathcal{A}'.
\langle c, \mathcal{A} \rangle \Rightarrow \langle c, \mathcal{A}' \rangle \land \langle c, m \rangle \rightarrow^* \langle \text{skip}, m' \rangle \\
\land \langle c, m \rangle \rightarrow^* \langle \text{skip}, m' \rangle \land m = m^\mathcal{A} \\
\Rightarrow m' = (m')^{\mathcal{A}'}.
\]

**Proof sketch.** By induction on the transformation rules. The full proof is available in Appendix A.1.
### 3.5.4 Relation to Information Flow Analysis

Up to this point, it might be unclear why introducing extra variables can improve the precision of information flow analysis. We first note that transformed programs enable more precise reasoning for dataflows. Consider the program in Figure 3.1(a) and Figure 3.1(b). In the transformed program, it is clear that the value stored in $x$ never flows to variable $p$; but such information is not obvious in the source program. Moreover, Theorem 1 naturally enables a more precise analysis of the transformed program, since it implies that if any property holds on the final active set $A'$ for the transformed program, then the property holds on the entire final memory for the original program. That is, in terms of information flow security, the original program leaks no information if the transformed program leaks no information in the subset $A'$ of the final memory. Consider the example in Figure 3.1(b) again. Theorem 1 allows a program analysis to accept the (secure) program even though the variable $x$, which is not in $A'$, may leak the secret value.

In Section 3.8, we show that, in general, the program transformation automatically makes a flow-insensitive type system (e.g., the Volpano, Smith and Irvine’s system [10] and the system in Section 3.6) at least as precise as a classic flow-sensitive type system [4].

### Relation to Single Static Assignment (SSA)

SSA [65] is used in the compilation chain to improve and simplify dataflow analysis. Viewed in this way, it is not surprising that our program transformation shares some similarity with the standard SSA-transformation. However, our transformation is different from the latter in major ways:

- Most importantly, our transformation does not involve the distinguishing $\phi$-functions of SSA. First of all, removing $\phi$-functions simplifies the soundness proof, since the resulting target language syntax and semantics are completely standard. Moreover, it greatly
simplifies information flow analysis on the transformed programs. Intuitively, the reason is that in the standard SSA form, the $\phi$-function is added after a branch (i.e., in the form of $(\text{if } e \text{ then } c_1 \text{ else } c_2); x := \phi(x_1, x_2)$). However, without a nontrivial program analysis for the $\phi$-function, the path conditions under which $x := x_1$ and $x := x_2$ occur (needed for path-sensitivity) is lost in the transformed program. On the other hand, extra assignments are inserted under the corresponding branches in our transformation. The consequence is that the path information is immediately available for the analysis on the transformed program. We defer a more detailed discussion on this topic to Section 3.6.6, after introducing our type system.

- As discussed in Section 3.5.4, the final active set $A'$ generated from the transformation is crucial for enabling a more precise program analysis on the transformed program (intuitively, an information flow analysis may safely ignore variables not in $A'$); however, such information is lost in the standard SSA form.

- Our general transformation offers a full spectrum of analysis precision: from adding no active copy to adding one copy for each assignment, but the standard SSA transformation only performs the latter.

### 3.6 Type System

The second component of the analysis is a sound type system with expressive dependent labels. The type system analyzes a transformed program along with the final active set; the type system ensures that the final values of the public variables in the final active set are not influenced by the initial values of secret variables.
3.6.1 Overview

We first introduce the nonstandard features in the type system: dependent security labels and program predicates.

Return to the example in Figure 3.1(c). We observe that this program is secure because: 1) $y$ holds a secret value only when $p_1 < 0$, and 2) the information flow from $y$ to $x$ at line 3 only occurs when $p_1 > 0$. Accordingly, to gain path-sensitivity, two pieces of information are needed in the type system: 1) expressive security labels that may depend on program states, and 2) an estimation of program states that may reach a program point.

We note that such information can be gained by introducing dependent security labels and program predicates to the type system. For the example in Figure 3.1(c), the relation between the level of $y$ and the value of $x$ can be described as a concise dependent label ($p_1 < 0?S:P$), meaning that the security level of $x$ is $S$ when $p_1 < 0$; the level is $P$ otherwise. Moreover, for precision, explicit and implicit flows should only be checked under program states that may reach the program point. In general, a predicate overestimates such states. For the example in Figure 3.1(c), checking that the explicit flow from $y$ to $x$ is secure under any program state is too conservative, since it only occurs when $p_1 > 0$. With a program predicate that $p_1 > 0$ for the assignment $x := y$, the label of $y$ can be precisely estimated as $P$. Note that our analysis agrees with the definition of path-sensitivity: it understands that the two assignments $y := s$ and $x := y$; never execute together in one execution. The example in Figure 3.1(c) is accepted by our type system.

3.6.2 Challenge: Statically Checking Implicit Declassification

Though designing a dependent security type system may seem simple at the first glance, handling mutable variables can be challenging. The implicit declassification problem, as defined in [5], occurs whenever the level of a variable changes to a less restrictive one, but its value remains the same. Consider the insecure program in Figure 3.6(a), which is identical
to the secure program in Figure 3.1(c) except for line 4. This program is obviously insecure since the sequence \( y := s; p_1 := 1; x := y; p_2 := x; \) may be executed together. Compared with Figure 3.1(c), the root cause of this program being insecure is that at line 4 (when \( p_1 \) is updated), \( y \)'s new level \( P \) (according to the label \( p_1 < 0?S : P \)) is no longer consistent with the value it holds.

The type systems in [5, 8] resort to a run-time mechanism to tackle the implicit declassification problem. However, that also means that the type system might change the semantics of the program being analyzed. In this dissertation, we aim for a purely static solution.

**Program Transformation and Implicit Declassification** Although the program transformation in Section 3.5 is mainly designed for flow-sensitivity, we observe that it also helps to detect implicit declassification. Consider the example in Figure 3.6(a) again, where the assignment at line 4 has brackets. The corresponding transformed program (Figure 3.6(b)) does not have an implicit declassification problem since updating \( p_3 \) at line 4 does not change \( y \)'s level, which depends on the value of \( p_1 \), rather than \( p_3 \). Moreover, the insecure program cannot be type-checked since both “then” branches might be executed together.

While adding extra variable copies helps in the previous example, it unfortunately does not eliminate the issue. The intuition is that even for a fully-bracketed program, variables modified in a loop might still be mutable (since the local variables defined in the loop might change in each iteration). Consider the program in 3.7(a). This program is insecure since it copies \( s \) to
$y$ in the first iteration, and copies $y$ to $p$ in the next iteration. When fully-bracketed, the loop body becomes

$$\text{if } (x_2 \% 2 = 0) \text{ then } y_1 := s; y_3 := y_1 \text{ else } ...;$$

$$x_3 := x_2 + 1; x_2 := x_3; y_2 := y_3;$$

where the labels of $y_1$ and $y_3$ depend on $x_2$. In this program, implicit declassification happens when $x_2$ is updated.

One naive solution is to disallow mutable variables in a program. However, dependence on mutable variables does not necessarily break security. Consider the program in Figure 3.7(b), which is identical to the previous example except that $y$ is updated at line 8. In this program, $y$’s level depends on the mutable variable $x$, but it is secure since the value of $s$ never flows to the next iteration.

**Our Solution**  Our insight is that changing $y$’s level at line 7 in Figure 3.7(b) is secure since the value of $y$ is not used in the future (in terms of dataflow analysis, $y$ is dead after line 6). This observation motivates us to incorporate a customized *liveness analysis* (Section 3.6.4) into the type system: an update to a variable $x$ is allowed if no labels of the *live variables* at that program point depend on $x$. 

Figure 3.7. Examples: Implicit Declassification in Loop.
3.6.3 Type Syntax and Typing Environment

In our type system, types are extended with security labels, whose syntax is shown in Figure 3.8. The simplest form of label $\tau$ is a concrete security level $\ell$ drawn from a security lattice $L$. Dependent labels, specifying levels that depend on run-time values, have the form of $(e?\tau_1 : \tau_2)$, where $e$ is an expression. Semantically, if $e$ evaluates to a non-zero value, the dependent label evaluates to $\tau_1$, otherwise, $\tau_2$. A security label can also be the least upper bound, or the greatest lower bound of two labels.

We use $\Gamma$ to denote a typing environment, a function from program variables to security labels. The integration of dependent labels puts constraints on the typing environment $\Gamma$ to ensure soundness. In particular, we say $\Gamma$ is well-formed, denoted as $\vdash \Gamma$, if: 1) no variable depends on a more restrictive variable, preventing leakage from labels; 2) there is no chain of dependency. These restrictions are formalized as follows, where $FV(\tau)$ denotes the free variables in $\tau$:

**Definition 3 (Well-Formedness)** A typing environment $\Gamma$ is well-formed, written $\vdash \Gamma$, if and only if:

$$\forall x \in \text{Vars.} \ (\forall x' \in FV(\Gamma(x))). \ (\Gamma(x') \sqsubseteq \Gamma(x))$$

$$\wedge (\forall x' \in FV(\Gamma(x))). \ (FV(\Gamma(x')) = \emptyset)$$

We note that the definition rules out self-dependence, since if $x \in FV(\Gamma(x))$, we have $FV(\Gamma(x)) = \emptyset$. Contradiction.
3.6.4 Predicates and Variable Liveness

Our type system is parameterized on two static program analyses: a predicate generator and a customized liveness analysis. Instead of embedding these analyses into our type system, we follow the modular design introduced in [5] to decouple program analyses from the type system. Consequently, the soundness of the type system is only based on the correctness of those analyses, regardless of the efficiency or the precision of those analyses.

**Predicate Generator** We assume a predicate generator that generates a (conservative) program predicate for each assignment \( \eta \) in the transformed program, denoted as \( P(\eta) \). A predicate generator is correct as long as each predicate is always true when the corresponding assignment is executed.

A variety of techniques, regarding the trade-offs between precision and complexity, can be used to generate predicates that describe the run-time state. For example, weakest preconditions [66] or the linear propagation [5] could be used. Our observation is that for path-sensitivity, only shallow knowledge containing branch conditions is good enough for our type system.

**Liveness Analysis** Traditionally, a variable is defined as alive if its value will be read in the future. But in our type system, if a variable \( x \) is alive, then any free variable in the label of \( x \) should also be considered as alive, because the concrete level of \( x \) depends on those variables. Moreover, we assume at the end of a program, only the variables in the final active set are alive, due to Theorem 1.

The liveness analysis is defined in Figure 3.9, where \( s \) denotes a program command, and \( \text{final} \) refers to the last command of the program being analyzed. Here, \( \text{final} \) is the initial state for the backward dataflow analysis. \( \text{succ}[s] \) returns the successors (as a set) of the command \( s \). In the GEN set of an assignment \( x := e \), both \( \text{FV}(e) \), and \( \bigcup_{v \in \text{FV}(e)} \text{FV}(\Gamma(v)) \), the free variables inside their labels, are included. Since we are analyzing the transformed program, the state of the final active set is crucial for precision. Therefore, the analysis also enforces that, at the end...
LIVE\textsubscript{out}[\text{final}] = \mathcal{A}

LIVE\textsubscript{in}[s] = GEN[s] \cup (LIVE\textsubscript{out}[s] - KILL[s])

LIVE\textsubscript{out}[s] = \bigcup_{p \in \text{succ}[s]} \text{LIVE}\textsubscript{in}[p]

GEN[x := \eta \ e] = \text{FV}(e) \cup (\bigcup_{v \in \text{FV}(e)} \text{FV}(\Gamma(v)))

KILL[x := \eta \ e] = \{x\}

Figure 3.9. Liveness Analysis of \mathcal{L}_\mathcal{A}.

of the program, all active copies in \mathcal{A} are alive. Other rules are standard for liveness analysis.

**Interface to the Type System**  We assume each assignment in the transformed language is associated with a unique identifier \eta. We use •\eta and \eta• to denote the precise program points right before and after the assignment respectively. For example, \mathcal{P}(•\eta) represents the predicates right before statement \eta, and \mathcal{L}_\mathcal{A}(\eta•) denotes the alive set right after statement \eta with initialization of \mathcal{A} as the final live set.

### 3.6.5 Typing Rules

The type system is formalized in Figure 3.10 and Figure 3.11. Typing rules for expressions have the form of \( \Gamma \vdash e : \tau \), where \( e \) is the expression being checked and \( \tau \) is the label of \( e \). The typing judgment of commands has the form of \( \Gamma, pc \vdash c \). Here, \( pc \) is the usual program-counter label [16], used to control implicit flows.

Most rules are standard, thanks to the modular design of our type system. The only interesting one is rule (T-ASSIGN). For an assignment \( x := \eta \ e \), this rule checks that both the explicit and implicit flows are allowed in the security lattice: \( \tau \sqcup pc \sqsubseteq \Gamma(x) \). Note that since \( \tau \) might be a dependent label that involves free program variables, the \( \sqsubseteq \) relation is technically the lifted version of the relation on the security lattice. Hence, the constraint \( \tau \sqcup pc \sqsubseteq \Gamma(x) \) requires the label of \( x \) to be at least as restrictive as the label of current context \( pc \) and the label.
Γ ⊢ n : ⊥ \quad \text{T-Const} \quad \frac{\Gamma (x) = \tau}{\Gamma ⊢ x : \tau} \quad \text{T-VAR} \quad \frac{\Gamma ⊢ e : \tau_1}{\Gamma ⊢ e' : \tau_2} \quad \text{T-OP}

\begin{align*}
\frac{\Gamma \vdash e : \tau}{\Gamma, pc \vdash \text{skip}} & \quad \text{T-Skip} & \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1 ; c_2} & \quad \text{T-SEQ} \\
\frac{\Gamma ⊢ e : \tau \quad \Gamma, \tau \cup pc ⊢ c_1 \quad \Gamma, \tau \cup pc ⊢ c_2}{\Gamma, pc ⊢ \text{if} (e) \text{ then } c_1 \text{ else } c_2} & \quad \text{T-If} \\
\frac{\Gamma \vdash e : \tau}{\Gamma, pc \vdash \text{P}(\eta \bullet) \Rightarrow \tau \cup pc \subseteq \Gamma (x) \quad \forall v \in L_A(\eta \bullet).x \notin \text{FV}(\Gamma (v))}{\Gamma, pc \vdash x := \eta e} & \quad \text{T-Assign} \\
\frac{\Gamma \vdash e : \tau \quad \Gamma, \tau \cup pc ⊢ c}{\Gamma, pc \vdash \text{while} (e) \ c} & \quad \text{T-While}
\end{align*}

\text{Figure 3.10. Typing Rules: Expressions.}

\text{Figure 3.11. Typing Rules: Commands.}

\(e\) under any program execution. For precision, the type system validates the partial ordering under the predicate \(\text{P}(\bullet \eta)\), the predicate that must hold for any execution that reaches the assignment.

Moreover, the assignment rule checks that for any variable in the liveness set after the assignment, its security label must not depend on \(x\); otherwise, its label might be inconsistent with its value. As discussed in Section 3.6.2, this check is required to rule out insecure implicit declassification.

At the top level, the type system collects proof obligations in the form of \(\models \text{P} \Rightarrow \tau_1 \sqsubseteq \tau_2\), where \(\tau_1\) and \(\tau_2\) are security labels, and \(\text{P}\) is a predicate. Such proof obligations can easily be discharged by theorem solvers, such as Z3 [67].

As an example, consider again the interesting examples in Figure 3.7. In both programs, we can assign \(y\) to the dependent label \((x \% 2 = 0? S : P)\), and assign \(x\) to the label \(P\). From the liveness analysis, we know that the live sets right after line 7 are \(\{x, y, s\}\) and \(\{x, p, s\}\).
for Figure 3.7(a) and Figure 3.7(b) respectively. Hence, the type system correctly rejects the
insecure program in Figure 3.7(a) since the check at line 7, \( \forall v \in \mathcal{L}_A(\eta\bullet). x \not\in \text{FV}(\Gamma(v)) \), fails.
On the other hand, the check at line 7 succeeds for the program in Figure 3.7(b). For line 4 in
Figure 3.7(b), the assignment rule generates one proof obligation

\[
\models (x \% 2 = 0) \Rightarrow P \sqcup S \sqsubseteq (x \% 2 = 0?S : P)
\]

which is clearly true for any value of \( x \). In fact, the secure program in Figure 3.7(b) is correctly
accepted by the type system in Figure 3.10 and Figure 3.11.

### 3.6.6 Program Transformation and Information Flow Analysis

We now discuss the benefits of the program transformation in Section 3.5 for information flow
analysis in details.

**Simplifying Information Flow Analysis**

As discussed in Section 3.5.4, our transformation does not involve the distinguishing \( \phi \)-functions
of SSA. Doing so simplifies information flow analysis on the transformed programs. We
illustrate this using the following example, where \( y \) is expected to have the label \( (x = 1?P : S) \)
afterwards.

\[
\text{if } (x = 1) \text{ then } y := 0 \text{ else } y := s
\]

Our transformation yields the following program, which can be verified with labels \( y_1 : P, y_2 : S, y_3 : (x = 1?P : S) \).

\[
\text{if } (x = 1) \text{ then } (y_1 := 0; y_3 := y_1) \\
\text{else } (y_2 := s; y_3 := y_2);
\]

In comparison, the standard SSA form is:

\[
(\text{if } (x = 1) \text{ then } y_1 := 0 \text{ else } y_2 := s;)y_3 := \phi(y_1, y_2);
\]

To verify this program, a type system would need at least a nontrivial typing rule for \( \phi \), which
somehow “remembers” that \( y_3 := y_2 \) occurs only when \( x = 1 \). Even with such knowledge, the type of \( y_2 \) cannot simply be \( S \), since otherwise, assigning \( y_2 \) to \( y_3 \) at \( \phi \) is insecure. In fact, the labels required for verification are \( y_1, y_2, y_3 : (x = 1?S : P) \).

Similar complexity is also involved for the \( \phi \)-functions inserted for loops: to precisely reason about information flow, the semantics and typing rules of \( \phi \) also need to track the number of iterations.

**Improving Analysis Precision**

Precision-Wise, bracketed assignments improve analysis precision in two ways. First, as discussed in Section 3.5.4, they improve flow-sensitivity by introducing new variable definitions. Second, they also improve path-sensitivity by enabling more accurate program predicates.

Consider the following example.

\[
\begin{align*}
x &:= -1; \\
\text{if } (x > 0) \text{ then } y := S; \text{ else } y := 1; \\
[y := -x]; \\
\text{if } (x > 0) \text{ then } p := y;
\end{align*}
\]

This program is secure since \( p \) becomes 1 regardless of the value of \( s \). However, without the bracket shown, the type system rejects it since no such label \( \tau_y \) satisfies the constraints that \((x > 0) \Rightarrow (S \subseteq \tau_y)\) (arising from the first if) and \((x > 0) \Rightarrow (\tau_y \subseteq P)\) (arising from the second if).

However, with the bracket, the last two lines become

\[
\begin{align*}
x_1 &:= -x; \\
\text{if } (x_1 > 0) \text{ then } p := y;
\end{align*}
\]

This program can be type-checked with \( y \)'s label as \((x > 0?S : P)\) and a precise enough predicate generator, which generates \( x_1 = -x \) after the assignment \( x_1 := -x \), because constraints \((x > 0) \Rightarrow (S \subseteq \tau_y)\) and \((x_1 > 0 \land x_1 = -x) \Rightarrow (\tau_y \subseteq P)\) can be solved with \( y \)'s label mentioned above.
3.7 Soundness

Central to our analysis is rigorous enforcement of a strong information security property. We formalize this property in this section and sketch a soundness proof. The complete proof is available in Appendix A.2.

3.7.1 Noninterference

Our formal definition of information flow security is based on noninterference [3]. Informally, a program satisfies noninterference if an attacker cannot observe any difference between two program executions that only differ in their confidential inputs. This intuition can be naturally expressed by semantics models of program executions.

Since a security label may contain program variables, its concrete level cannot be determined statically in general. But it can always be evaluated under a concrete memory:

**Definition 4** For a security label $\tau$, we evaluate its concrete level under memory $m$ as follows:

$$V(\tau, m) = \ell, \text{ where } \langle \tau, m \rangle \Downarrow \ell$$

Moreover, to simplify notation, we use $T_\Gamma(x, m)$ to denote the concrete level of $x$ under $m$ and $\Gamma$ (i.e., $T_\Gamma(x, m) = V(\Gamma(x), m)$).

To formally define noninterference in the presence of dependent labels, we first introduce an equivalence relation on memories. Intuitively, two memories are $(\Gamma, \ell)$-equivalent if all variables with a level below level $\ell$ agree on both their concrete levels and values.

**Definition 5** $(\Gamma, \ell)$-Equivalence: Given any concrete level $\ell$ and $\Gamma$, we say two memories $m_1$ and $m_2$ are equivalent up to $\ell$ under $\Gamma$ (denoted by $m_1 \approx_\Gamma^\ell m_2$) iff

$$\forall x \in \text{Vars}.$$
\[(\mathcal{T}_\Gamma(x, m_1) \subseteq \ell \iff \mathcal{T}_\Gamma(x, m_2) \subseteq \ell) \land \mathcal{T}_\Gamma(x, m_1) \subseteq \ell \implies m_1(x) = m_2(x)\]

It is straightforward to check that \(\approx^\ell \) is an equivalence relation on memories. Note that we require type of \(x\) be bounded by \(\ell\) in \(m_2\) whenever \(\mathcal{T}_\Gamma(x, m_1) \subseteq \ell\). The reason is to avoid label channels, where confidential data is leaked via the security level of a variable \([5, 48]\).

Given initial labels \(\Gamma\) on variables and final labels \(\Gamma'\) on variables, we can formalize noninterference as follows:

**Definition 6 (Noninterference)** We say a program \(c\) satisfies noninterference w.r.t. \(\Gamma, \Gamma'\) if equivalent initial memories produce equivalent final memories:

\[
\forall m_1, m_2, \ell, \quad m_1 \approx^\ell \Gamma m_2 \land \langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m'_1 \rangle \land \langle c, m_2 \rangle \rightarrow^* \langle \text{skip}, m'_2 \rangle \implies m'_1 \approx^\ell \Gamma' m'_2
\]

The main theorem is the soundness of our analysis: informally, if the transformed program type-checks, then the original program satisfies noninterference. Since the type system applies to the transformed program, we first need to connect the types in the original and the transformed programs. To do that, we define the projection of types for the transformed program in a way similar to Definition 2:

**Definition 7 (Projection of Types)** Given an active set \(A\) and \(\Gamma\), types of variables in the transformed program, we use \(\Gamma^A\) to denote a mapping from \(\text{Vars}\) to \(\tau\) as follows:

\[
\forall v \in \text{Vars}. \quad \Gamma^A(v) = \Gamma(A(v))
\]
Formally, the soundness theorem states that if a program $c$ under active set $A$ (e.g., an identity function) is transformed to $c$ and final active set $A'$, and $c$ is well-typed under the type system (parameterized on $A'$), then $c$ satisfies noninterference w.r.t. $\Gamma^A$ and $\Gamma^{A'}$:

**Theorem 2 (Soundness)**

$$
\forall c, m_1, m_2, m'_1, m'_2, \ell, \Gamma, A, A'.
$$

$$
\langle c, A \rangle \Rightarrow \langle c, A' \rangle \land \vdash \Gamma \land \Gamma \vdash c \land m_1 \approx^{\ell} m_2 \land
$$

$$
\langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m'_1 \rangle \land \langle c, m_2 \rangle \rightarrow^* \langle \text{skip}, m'_2 \rangle
$$

$$
\implies m'_1 \approx^{\ell}_{\Gamma^{A'}} m'_2
$$

To approach a formal proof, we notice that by the correctness of the program transformation (Theorem 1), it is sufficient to show that the transformed program leaks no information on the subset $A'$. Such connection is illustrated in Figure 3.12. We formalize the soundness for the transformed program w.r.t. initial and final active sets as follows:

**Theorem 3 (Soundness of Transformed Program)**

$$
\forall c, m_1, m_2, m_3, m_4, \ell, \Gamma, A, A'.
$$

$$
\langle c, A \rangle \Rightarrow \langle c, A' \rangle \land \vdash \Gamma \land \Gamma \vdash c \land m_1^A \approx^{\ell} m_2^A
$$

$$
\land \langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m_3 \rangle \land \langle c, m_2 \rangle \rightarrow^* \langle \text{skip}, m_4 \rangle
$$

35
\begin{align*}
\text{erase}(m, x, \eta)(x') &= \begin{cases} 
0, & x \in \text{FV}(x') \land x' \not\in \mathbb{L}_{A'}(\eta) \\
(\overline{m}(x'))', & \text{otherwise}
\end{cases} \\
\langle x, m \rangle \Downarrow n \quad m' &= m\{x \mapsto n\} \\
\langle x := \eta e, m \rangle &\rightarrow \text{ER}(A') \langle \text{skip, erase}(m', x) \rangle
\end{align*}

Figure 3.13. Erasure Semantics of Assignment.

Proof sketch. One challenge in the formal proof is that the equivalence relation $\approx^e_{\Gamma}$ only holds on the active copies and it may break temporarily during the program execution. Consider the example in Figure 3.7(b). During the first iteration of the loop body, $y$ holds a secret value but its level is $P$ right after line 8. Hence, the relation $\approx^e_{\Gamma}$ may break at that point in the small-step evaluation starting from two memories that only differ in secrets. To tolerate such temporary violation of the $\approx^e_{\Gamma}$ relation, we prove the soundness with a new semantics which enforces that the relation $\approx^e_{\Gamma}$ holds for all variables, and the final values of variables in $A'$ agree with those in the standard semantics. The new semantics, called the erasure semantics is shown in Figure 3.13. The semantics is parameterized on the final active set $A'$. The only difference from the standard one is for assignments: the new assignment rule (ST-ERASE) sets variables that are not alive and whose types depend on $x$ to be zero. It is easy to check that the erasure semantics agrees on the final value of the variables in $A'$. Also, it removes the temporary violation of the equivalence relation by forcing value of $y$ to be zero after line 7 of Figure 3.7(b). The complete proof is available in Appendix A.2.
Recall that the dependent security type system (without program transformation) is flow-insensitive; yet, our program analysis is flow-sensitive with the novel program transformation in Section 3.5. In this section, we show that this is not a coincidence: the program transformation automatically makes a flow-insensitive type system (e.g., the Volpano, Smith and Irvine’s system [10] and the system in Section 3.6) flow-sensitive.

3.8.1 The Hunt and Sands System

In [4], Hunt and Sands define a classic flow-sensitive type system where the security level of a program variable may “float” in the program. In particular, Hunt and Sands (HS) judgments have the form of \( pc \vdash_{\text{HS}} \Gamma \{ c \} \Gamma' \), where \( \Gamma \) and \( \Gamma' \) are intuitively the typing environments before and after executing \( c \) respectively.

Consider the program in Figure 3.1(a). While a flow-insensitive type system rejects it, the
HS system accepts it with the following typing environments:

\[ \Gamma \{ x := s ; \} \Gamma \{ x := 0 ; \} \Gamma' \{ p := x ; \} \Gamma' \]

where \( \Gamma = \{ x \mapsto S, p \mapsto P \} \) and \( \Gamma' = \{ x \mapsto P, p \mapsto P \} \).

The HS typing rules for commands are summarized in Figure 3.14. We use \( \vdash_{\text{HS}} \) to distinguish those judgments from the ones in our system. The interesting rules are rule (HS-If) and rule (HS-While): the former computes the type for each variable as the least upper bound of its labels in the two branches; the latter computes the least fixed-point of a monotone function (the while loop) on a finite lattice.

### 3.8.2 Program Transformation and Flow-Sensitivity

We show that the program transformation in Section 3.5 along with a flow-insensitive type system subsumes the HS system: for any program \( c \) that can be type-checked in the HS system, the transformed program of \( \llbracket c \rrbracket \) (i.e., a fully-bracketed program) can be type-checked in a flow-insensitive type system. This result has at least two interesting consequences:

1. The program transformation removes the source of “flow-insensitivity”; a flow-insensitivity type system can be automatically upgraded to a flow-sensitive one.

2. The flow- and path-sensitive system in this chapter strictly subsumes the HS system: any secure program accepted by the latter is accepted by the former, but not vice versa (e.g., the program in Figure 3.1(c)).

To construct types in the transformed program, we first introduce a few notations. Given a typing environment \( \Gamma : \text{Vars} \rightarrow \tau \) for the original program and an active set \( \mathcal{A} \), we can straightforwardly construct a (minimal) typing environment, written \( \Gamma_{\mathcal{A}} \), whose projection on \( \mathcal{A} \) is \( \Gamma \):

\[ \forall \underline{u} \in \mathcal{A}. \; \Gamma_{\mathcal{A}}(\underline{u}) \triangleq \Gamma(\lfloor \underline{u} \rfloor) \]
C-Skip \[ (pc, \Gamma, A)\{\text{skip} \Rightarrow \text{skip}\}(\Gamma, A) \hookrightarrow \Gamma_A \]

C-Seq \[ (pc, \Gamma, A)[\{c_1\} \Rightarrow \{c_1\}] (\Gamma'', A') \hookrightarrow \Gamma_1 \quad (pc, \Gamma, A)[\{c_2\} \Rightarrow \{c_2\}] (\Gamma', A') \hookrightarrow \Gamma_2 \]

C-Assgn \[ (pc, \Gamma, A)[\{x := e\} \Rightarrow \{x := e\}](\Gamma(x \mapsto \tau), A(x \mapsto x_i)) \]

C-If \[ (pc, \Gamma, A)[\{\text{if } (e) \text{ then } c_1 \text{ else } c_2\} \Rightarrow \{\text{if } (e) \text{ then } (c_1 \land A_i) \text{ else } (c_2 \land A_2)\}](\Gamma', A_i) \hookrightarrow \Gamma_i \cup \Gamma_{\text{if } (e)} \]

C-While \[ (pc, \Gamma, A)[\{\text{while } (e) \text{ c}\} \Rightarrow \{\text{while } (e) \land A_i\}]\{\{c\} \Rightarrow \{c\}\}(\Gamma', A_i) \hookrightarrow \Gamma_0 \cup \Gamma_{\text{while } (e) \text{ c}} \]

**Figure 3.15.** Type Construction in Transformed Program.

Moreover, given a sequence of tying environments for the transformed program, say \(\Gamma_1, \Gamma_2, \ldots\), we define a merge function, denoted as \(\cup\), that returns the union of \(\Gamma_1, \Gamma_2, \ldots\) so that conflicts in the environments are resolved in the order of \(\Gamma_1, \Gamma_2, \ldots\). For example, \(\cup(\{x_1 \mapsto S, y_2 \mapsto P\}, \{x_1 \mapsto P, y_2 \mapsto P\}) = \{x_1 \mapsto S, y_2 \mapsto P\}\).

For a fully bracketed program \([c]\), we can inductively define the construction of \(\Gamma\) as inference rules in the form of

\[ (pc, \Gamma, A)[\{c\} \Rightarrow \{c\}](\Gamma', A') \hookrightarrow \Gamma \]

where \(pc, \Gamma, c, \Gamma'\) are consistent with the HS typing rules in the form of \(pc \vdash_{\text{HS}} \Gamma\{c\}\Gamma'\); \(A, [c], A', \{c\}\) are consistent with the program transformation rules in the form of \(\langle [c], A \rangle \Rightarrow \{c, A'\}\). \(\Gamma\) is the constructed typing environment that, as we show shortly in Theorem 4, satisfies \(\Gamma, pc \vdash \{c\}\). The construction algorithm is formalized in Figure 3.15.

Most parts of the rules are straightforward; they are simply constructed to be consistent with the HS typing rules and the transformation rules in Figure 3.4. The following lemma makes such connections explicit.
Lemma 1

\[
\forall pc, \Gamma, \Gamma', A, A', c, \varsigma. pc \vdash_{HS} \Gamma\{c\}\Gamma' \land \langle [c], A \rangle \Rightarrow \langle \varsigma, A' \rangle
\]

\[
\Rightarrow \exists \Gamma. (pc, \Gamma, A)\{[c] \Rightarrow \varsigma\}(\Gamma', A') \Rightarrow \Gamma
\]

Proof. By induction on the structure of \(c\). \(\square\)

To construct types for the transformed program: for \texttt{skip}, we use \(\Gamma_A\) (the typing environment before this command); for assignment, since \(x_i\) must be fresh, we can simply augment \(\Gamma_A\) with \(\{x_i \mapsto \tau\}\). Other rules simply merge constructed types from subexpressions in a conflict-solving manner, using \(\cup\). An eagle-eyed reader may find the construction is intuitively correct if there is no conflict at all in the merge operations.

We show that there is no conflict during construction by two observations. First, if a variable has the same active copy before and after transforming a fully-bracketed command \([c]\), then its type must remain the same (before and after \(c\)) in the HS system. This property is formalized as follows:

Lemma 2

\[
pc \vdash_{HS} \Gamma\{c\}\Gamma' \land \langle [c], A \rangle \Rightarrow \langle \varsigma, A' \rangle \Rightarrow
\]

\[
\forall v \in \text{Vars}. (A(v) = A'(v)) \Rightarrow (\Gamma(v) = \Gamma'(v))
\]

Proof sketch. By induction on the structure of \(c\). The most interesting cases are for branch and loop.

- if \((e)\) then \(c_1\) else \(c_2\): by the HS typing rule, \(pc \vdash_{HS} \Gamma\{c_1\}\Gamma_1 \land pc \vdash_{HS} \Gamma\{c_2\}\Gamma_2 \land \Gamma' = \Gamma_1 \cup \Gamma_2\). By the transformation rules, \(\langle [c_i], A_i \rangle \Rightarrow \langle \varsigma_i, A_i \rangle, i \in \{1, 2\}\). Suppose \(A(v) \neq A_1(v), A_1(v)\) must be a fresh variable generated in \(\varsigma_1\), and hence, cannot be in \(A_2\). By the definition of \(\Phi\), \(A_3(v)\) must be fresh. This contradicts the assumption
$A(v) = A'(v)$. Hence, $\Gamma(v) = \Gamma_1(v)$ by the induction hypothesis. Similarly, we can infer that $\Gamma(v) = \Gamma_2(v)$. So $\Gamma'(v) = \Gamma_1(v) \sqcup \Gamma_2(v) = \Gamma(v)$.

- while $(e)$: By rule (TRS-WHILE), we have $\langle [e], A \rangle \Rightarrow \langle c_1, A_1 \rangle$, where $A'$ is $A_1$ in this case. Hence, by the assumption, we have $A(v) = A_1(v)$. By rule (HS-WHILE), there is a sequence of environments $\Gamma'_i, \Gamma''_i$ such that $pc \sqcup \tau_i \vdash \Gamma'_i \{e\} \Gamma''_i$. By the induction hypothesis, $\Gamma''_i(v) = \Gamma'_i(v)$. Since $\Gamma'_0 = \Gamma$ and $\Gamma'_{i+1} = \Gamma \sqcup \Gamma''_i$ in rule (HS-WHILE), we can further infer that $\Gamma'_{i+1}(v) = \Gamma''_i(v)$. Hence, we have $\Gamma'(v) = \Gamma_n(v) = \Gamma_0(v) = \Gamma(v)$.

Second, the constructed environment is minimal, meaning that it just specifies types for the variables in $A$ and the freshly generated variables in $c$ (denoted as $\text{FVars}(c)$).

For technical reasons, we formalize this property along with the main correctness theorem of the construction, stating that the transformed program $c$ type-checks under the constructed environment $\Gamma$. Note that given any $A$, a fully bracketed command $[c]$ always transforms to some $c$ and $A'$. Hence, by Lemma 1, the following theorem is sufficient to show that our program analysis is at least as precise as the HS system:

**Theorem 4**

$$\forall c, \xi, pc, A, A', \Gamma, \Gamma', \xi. \ (pc, \Gamma, A) \{[\xi] \Rightarrow c\} \ (\Gamma', A') \Rightarrow \Gamma$$

$$\Rightarrow \text{Dom}(\Gamma) \subseteq A \cup \text{FVars}(c) \land \Gamma, pc \vdash c$$

**Proof.** Complete proof is available in Appendix A.3. 

An interesting corollary of Theorem 4 is that the transformed program can be type-checked under the classic fixed-level system in [10] as well.

**Corollary 1** Theorem 4 also applies to the type system in Figure 3.10 and Figure 3.11 with the restriction that all labels are security levels (i.e., non-dependent labels), which is identical to the system in [10].

41
**Proof.** We note that the construction in Figure 3.15 only uses the non-dependent part of our type system. Given all labels are security levels, it is straightforward to check that our type system degenerates to the system in [10].

Theorem 4 has a strong prerequisite that all assignments in the original program are bracketed. We note that the result remains true when such prerequisite is relaxed. Intuitively, a bracket is unnecessary when the old and new definitions have the same security label. Otherwise, a bracket is needed for flow-sensitivity. For example, to gain flow-sensitivity, only the second assignment in Figure 3.1(a) needs a bracket. The strong prerequisite is used in Theorem 4 to make the result general (i.e., type-agnostic).

According to Corollary 1, the result that any secure program accepted by the HS system is accepted by our analysis is true even if all dependent security labels degenerate to simple security levels. On the other hand, introducing dependent security labels makes our analysis strictly more precise than the HS system. For example, the program in Figure 3.1(c) cannot be verified without dependent security labels, but it can be type-checked with a label $y : (p_1 < 0?S : P)$.

### 3.8.3 Comparison with an existing transformation

Hunt and Sands show that if a program can be type-checked in the HS system, then there is an equivalent program which can be type-checked by a fixed level system [4]. However, their construction of the equivalent program is type guided, meaning that the program transformation assumes that security labels have already been obtained in the HS system, while our program transformation (Figure 3.4) is general and syntax-directed. An interesting application of our transformation is to test the typeability of the HS system without obtaining the types needed in the HS system in the first place.

It is noteworthy that our transformation is arguably simpler than the HS transformation since our rule for loop has no fixed-point construction while the latter has one. The reason is that
compared with the HS transformation, the goal of our transformation is easier to achieve: our transformation improves analysis precision, while the HS transformation infers the type for each variable in a program. For example, consider a loop with only one assignment $x := x + 1$, and $x$ is initially $P$. In the HS system, the transformed program is $x_p := x_p + 1$, where $x_p$ is the public version of variable $x$. On the other hand, our transformation generates $x_1 := x_2; x_2 := x_1 + 1$.

From the perspective of inferring labels, introducing $x_1$ and $x_2$ might seem unnecessary since they must have the same label according to the type system. However, doing so might improve analysis precision (e.g., the type system can specify the dependencies on $x_1$ and $x_2$ separately with two copies of $x$).

### 3.9 Related Work

We refer to [16] for a comprehensive survey of static information flow analysis. Here, we focus on the most relevant ones.

**Dependent Labels and Information Flow Security** Dependent types have been widely studied and have been applied to practical programming languages (e.g., [57, 59, 61, 68–70]). New challenges emerge for information flow analysis, such as precise, sound handling of information channels arising from label changes.

For security type systems, the most related works are SecVerilog [5, 8], Lourenço and Caires [6] and Murray et al. [70]. SecVerilog is a Verilog-like language with dependent security labels for verifying timing-sensitive noninterference in hardware designs. The type systems in [5, 8] are not purely static: they remove implicit declassification by a run-time enforcement that modifies program semantics. A recent extension to SecVerilog [50] alleviates such limitation by hardware-specific static reasoning. However, those type systems do not handle loops (absent in hardware description languages), which gives rise to new challenges for soundness. Moreover, they are not flow-sensitive. The recent work [6] also allows the
security type to depend on runtime values. However, the system is flow-insensitive, and it does not have a modular design that allows tunable precision. Moreover, the language has limited expressiveness: it has no support for recursion, and it disallows dependence on mutable variables. Exploring dependent labels to their full extent exposes new challenges that we tackle in this work, such as implicit declassification. Murray et al. [70] present a flow-sensitive dependent security type system for shared-memory programs. The type system enforces a stronger security property: timing-sensitive non-interference for concurrent programs. However, even when the extra complexity due to concurrency and timing sensitivity are factored out, extra precision in their system is achieved via a floating type system that tracks the typing environments and program states throughout the program. In comparison, our analysis achieves flow-sensitivity via a separate program transformation, which results in an arguably simpler type system. Moreover, for dependency on mutable variables, their system only allows a variable’s security level to upgrade to a higher one, while our system allows a downgrade to a lower level when doing so is secure.

Some prior type systems for information flow also support limited forms of dependent labels [57, 60, 71–74]. The dependence on run-time program state, though, is absent in most of these, and most of them are flow- and path-insensitive.

**Flow-Sensitive Information Flow Analysis** Flow-sensitive information flow control [4, 48, 49] allows security labels to change over the course of computation. Those systems rely on a floating type system or a run-time monitor to track the security labels at each program point. On the other hand, the program transformation in our analysis eliminates imprecision due to flow-insensitivity. Moreover, the bracketed assignments in a source program provide tunable control for needed analysis precision. These features offer better flexibility and make it possible to turn a flow-insensitive analysis to be flow-sensitive.
**Semantic-Based Information Flow Analysis**  Another direction of information flow security is to verify the semantic definition of noninterference based on program logics. The first work that used a Hoare-style semantics to reason about information flow is by Andrews and Reitman [75]. Independence analysis based on customized logics [76–78] was proposed to check whether two variables are independent or not. Self-Composition [79, 80] composes a program with a copy of itself, where all variables are renamed. The insight is that noninterference of a program $P$ can be reduced to a safety property for the self-composition form of $P$.

Relational Hoare Logic [81] was first introduced for a core imperative program to reason about the relation of two program executions. It was later extended to verify security proofs of cryptographic constructions [82] and differential privacy of randomized algorithms [83, 84]. In the context of information flow security, Relational Hoare Type Theory [85] extends Hoare Type Theory and has been used to reason about advanced information flow policies.

Though some semantic-based information flow analyses are flow- and path-sensitive, most mechanisms incur heavy annotation burden and steep learning curve on programmers. We believe our approach shows that it is not necessary to resort to those heavyweight methods to achieve both flow- and path-sensitivity.

### 3.10 Summary

This chapter presents a sound yet flow- and path-sensitive information flow analysis. The proposed analysis consists of a novel program transformation as well as a dependent security type system that rigorously controls information flow. We show that our analysis is both flow- and path-sensitive. Compared with existing work, we show that our analysis is strictly more precise than a classic flow-sensitive type system, and it tackles the tricky implicit declassification issue completely at the compile time. Moreover, the novel design of our analysis allows a user to control the analysis precision as desired. We believe our analysis offers a lightweight approach to static information flow analysis along with improved precision.
Chapter 4  
A Derivation Framework for Dependent Security Label Inference

4.1 Introduction

Information flow control is a promising way of protecting the confidentiality of information that is manipulated by computer systems. Compared with conventional mechanisms such as access control, information flow control provides fine-grained reasoning about information flows, as well as a strong end-to-end security guarantee: secret inputs cannot be inferred by an attacker through the observations of public outputs.

Compared to dynamic enforcements (e.g., [64,86–89]), a static enforcement verifies information flow policies at compile time so that all vulnerabilities are detected before program execution. Hence, there is no computation or storage overhead at runtime. However, it is also well-known that classic static information flow analysis sometimes lacks enough expressiveness for real-world applications. This becomes a key barrier to wide adoption of those static methods.

To improve the expressiveness of static information flow analysis, dependent types in various forms have been introduced. For instance, Jif [11] and its extensions [90,91] introduce
**dynamic security labels**, security labels that can be manipulated at runtime. More recent works [5–9, 70, 92] explore the theories and techniques to apply dependent type theory [93] to information flow control. In a nutshell, dependent security labels (or dependent labels in short) are security types that may depend on concrete program states. The added expressiveness leads to successful verification of real-world systems, such as a MIPS processor [5], conference management systems [6, 7], a TrustZone-like architecture [8] and Android Apps [9].

However, one big obstacle of verification via those promising dependent security labels is that most work (except [9] and [7] that support a restricted form of dependent labels) requires programmers to write down all (dependent) labels. However, since the dependent labels usually involve intricate security invariants, providing those labels requires a deep understanding of the program being verified, making it a both time-consuming and error-prone process. Moreover, when the provided labels are incorrect (i.e., program analysis fails with the labels), it is unclear if the program being verified is insecure, or the program can be verified with other correct labels.

For security labels without dependence, classic security type systems (such as Jif [11] and FlowCaml [12]) encode the restrictions on security labels into constraints in a finite semi-lattice. Those constraints can be solved by customized solvers (such as the Rehof-Mogensen algorithm [13] and set-constraints solvers [14, 15]). However, they cannot handle the infinite search space of dependence required for dependent labels.

In this chapter, we introduce the *first general framework* for designing security label inference algorithms and checking their correctness. For generality, we propose a core constraint language that, to the best of our knowledge, can encode all static information flow analyses that allow dependent security labels, except that the current encoding for dynamic labels may not be practically efficient (we defer a more detailed discussion to Section 4.3.3). In particular, the framework models security restrictions on a program as **predicated constraints**, in the form of $P \rightarrow \tau_1 \sqsubseteq \tau_2$, meaning that security label $\tau_2$ must be more restrictive than $\tau_1$ whenever predicate $P$ holds. A key feature of the language, which also makes the inference challenging,
is that a solution may have dependence. For example, a constraint \((b = 1 \rightarrow \alpha \subseteq P) \land (b \neq 1 \rightarrow S \subseteq \alpha)\) has a dependent solution that \(\alpha\) is \(P\) (public) whenever \(b = 1\); \(\alpha\) is \(S\) (secret) whenever \(b \neq 1\). But a solver that does not allow dependent solution (e.g., an SMT solver with theory on semi-lattice) will simply reject the constraint, since neither \(P\) nor \(S\) (without dependence) is a solution for \(\alpha\).

The framework models security label inference as an iterative process of solving derivations (typically, simpler constraints) from the original constraint set. Hence, it allows great flexibility in inference algorithm design. For example, one potential algorithm may simply work on the variant of constraints where all predicates are removed; another potential algorithm may reject a set of constraints whenever a subset of the constraints is found to be unsatisfiable. An algorithm (such as the one in [9]) may also directly work on an equivalent derivation of the original constraints. More promising are the novel iterative algorithms we develop in this chapter, which allow early termination without hurting soundness and completeness.

To facilitate the design and validation (in terms of soundness and completeness) of algorithms in the derivation framework, we distill the key properties for making an algorithm sound and/or complete. We show that various algorithms, including an extension to the inference algorithm in existing work [9], can be checked under the framework in a straightforward matter. Moreover, we also designed three novel sound and complete algorithms, namely, early-accept algorithm, early-reject algorithm, and hybrid algorithm. Based on a mix of both satisfiable and unsatisfiable constraint sets collected from the verification of an information flow policy on a MIPS processor using SecVerilog [5, 8], we found that the novel algorithms solve predicated constraints faster than an existing algorithm [9] by orders of magnitude.

In summary, we make the following contributions in this chapter:

- We define a core constraint language that, to the best of our knowledge, can encode label inference for all static information flow using dependent labels, though at the moment not all encodings are efficient in practice (Section 4.3).
• We propose the *first general framework* for the design and validation of label inference algorithms for the core constraint language (Section 4.4). It formalizes the key properties for an inference algorithm to be sound and/or complete.

• Under the framework, we propose three novel inference algorithms that allow early termination in a sound and complete manner (Section 4.5).

• We implement and evaluate the novel algorithms on a corpus of predicated constraints collected from secure and insecure variants of a MIPS processor (Section 4.6). Evaluation result suggests that the novel algorithms scale well, and they outperform existing algorithms.

### 4.2 Background and Overview

**Information Flow Analysis and Security Labels**

Conventionally, we assume *security levels* (e.g., P for public and S for secret) are associated with information to describe the intended secrecy of the contents. As standard, we assume the security levels form a lattice $\mathcal{L}$ whose partial ordering $\sqsubseteq$ specifies an information flow policy: information flow from level $\ell_1$ to $\ell_2$ is allowed if and only if $\ell_1 \sqsubseteq \ell_2$. However, note that our mechanism applies to any general security lattice.

Most information-flow type systems (e.g., [4, 10, 57, 58]) associate one security level from the security lattice to each program variable. To ensure information flow security (typically, some variant of the noninterference property [3]), a type system checks security restrictions on information flows in a program. For example, for an assignment $s := p$, the restriction is that the security level of $s$ is more restrictive than that of $p$ (i.e., $P \sqsubseteq S$). When the security levels of some variables are missing, some systems (e.g., Jif [11] and FlowCaml [12]) try to infer security levels for them whenever possible. Since the security levels form a lattice, a typical
inference algorithm encodes inference as solving constraints in a semi-lattice, where sound and complete algorithms exist (e.g., [13]).

However, security levels from a security lattice cannot express several demands in real-world applications. First, some applications require value-dependent security policies. For example, a function “getPwd” that takes a user name \( x \) should return a password with \( x \)’s security level attached. Second, analysis precision can be improved with value-dependent labels. For example, consider the program on the left. This is a secure program since the secret value of \( s \) never affects the public variable \( p_2 \) (the two assignments under “if” statements are never executed together). However, no static security level for \( y \) will work, even in a flow-sensitive system, since at the end of first “if” the level of \( y \) can be neither \( P \) (the flow from \( s \) to \( y \) is insecure) nor \( S \) (the flow from \( y \) to \( p_2 \) is insecure).

One promising approach for more expressive information flow security is to apply dependent type theory [93] to information flow control [5–9, 70, 92]. Such systems allow the security level associated with a variable to depend on the concrete program state. The extra expressiveness is shown to be useful in the verification of real-world applications, such as a MIPS processor [5], conference management systems [6, 7], a TrustZone-like architecture [8] and Android Apps [9]. For example, an analysis that allows dependent security labels (e.g., [92]) can verify the secure code above if the following dependent label is given manually to \( y \): \( S \) when \( p_1 < 0 \); \( P \) when \( p_1 > 0 \).

While dependent security labels are very promising in closing the gap between static information flow analysis and real-world applications, one limitation of existing works on dependent security label is that they either support label inference but only in a restricted form [7, 9], or assume that all (complex) security labels are annotated manually by a programmer. In this chapter, we tackle the problem of automatically inferring dependent security labels for general static information flow analysis with an arbitrary security lattice.
**Original Constraints:**

\[
\begin{align*}
\text{true} & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \\
d > 0 & \rightarrow S \sqsubseteq \alpha_y \\
\neg (d > 0) & \rightarrow P \sqsubseteq \alpha_y \\
d < 0 & \rightarrow \alpha_y \sqsubseteq \alpha_x 
\end{align*}
\]

**Derivation 1:** (a sound derivation)

\[
\begin{align*}
\text{true} & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \\
\land S & \sqsubseteq \alpha_y \land P \sqsubseteq \alpha_y \land \alpha_y \sqsubseteq \alpha_x 
\end{align*}
\]

**Derivation 2:** (a weaker yet sound derivation)

\[
\begin{align*}
d > 0 & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \land S \sqsubseteq \alpha_y \\
d \leq 0 & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \land P \sqsubseteq \alpha_y \land \alpha_y \sqsubseteq \alpha_x 
\end{align*}
\]

**Derivation 3:** (a sound and complete derivation)

\[
\begin{align*}
d > 0 & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \land S \sqsubseteq \alpha_y \\
d = 0 & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \land P \sqsubseteq \alpha_y \\
d < 0 & \rightarrow P \sqsubseteq \alpha_x \land \alpha_z \sqsubseteq \alpha_x \land \alpha_x \sqsubseteq P \land P \sqsubseteq \alpha_y \land \alpha_y \sqsubseteq \alpha_x 
\end{align*}
\]

**Figure 4.1.** Constraints and Sound Derivations.

**Overview**

To make label inference general, we model information flow restrictions on a program as **predicated constraints**, in the form of \( P \rightarrow \tau_1 \sqsubseteq \tau_2 \), meaning that the security label \( \tau_2 \) must be more restrictive than \( \tau_1 \) when condition \( P \) holds. We call the fragment without \( P \) (i.e., \( \tau_1 \sqsubseteq \tau_2 \)) a **label constraint**. For example, (4.2.1)-(4.2.4) in Figure 4.1 are four sets of predicated constraints (we defer a more detailed discussion on how the original constraints are generated to Section 4.3). Intuitively, each constraint can be interpreted as: whenever some abstract event \( P \) (e.g., a program execution that satisfies \( P \)) happens, the constraint \( \tau_1 \sqsubseteq \tau_2 \) is required for security. For example, the second constraint in (4.2.1) requires that \( \alpha_y \) (a security label to be inferred) is confidential whenever the condition \( d > 0 \) holds. In general, the predicate \( P \) can be parameterized over a theory on program states, linear inequalities over integers, boolean constraints, or finite sets.

What makes it challenging to infer dependent labels is that a label may depend on an
arbitrary condition, for which we propose a derivation framework (Section 4.4). The key idea is to model inference as an iterative process of transforming the original constraints into a (often) more manageable format. To see why doing so is beneficial, let us consider two simple cases.

- **Partition**: When a set of constraints have no overlapping predicates, e.g., (4.2.3), we can easily compute its solution by solving each label constraint without predicate (e.g., via the Rehof-Mogensen algorithm [13]) and merge all local solutions to a global solution.

- **Counterexample**: If under any predicate, the label constraints are unsatisfiable (i.e., there is no local solution), then there is no global solution.

In general, the original constraints, such as (4.2.1), can be more challenging: they neither form a partition, nor have an obvious counterexample. Our insight is that an inference algorithm can proceed by transforming the original constraint set into its derivations, as illustrated in Figure 4.2, where each circle represents one such derivation. Of particular interest for constraint solving are three kinds of derivations and constraint solving strategies:

- **Sound Derivation and Early-Accept** A derivation is sound if its solution is also a solution of the original constraints. For example, all three derivations: (4.2.2), (4.2.3) and (4.2.4) are sound derivations of (4.2.1). An early-accept algorithm starts with a sound yet simple derivation (e.g., removing all predicates as shown in Derivation (4.2.2))
and proceeds to the next derivation only if the current derivation has no solution. To make such an algorithm sound and complete, an algorithm will intuitively work on a weaker derivation in each iteration until reaching a sound and complete derivation (e.g., Derivation (4.2.4)), as illustrated by the green arrow in Figure 4.2.

- **Complete Derivation and Early-Reject** A derivation is *complete* if any solution of the original constraints is a solution of it (i.e., a complete derivation is unsatisfiable implies that the original constraints are unsatisfiable). An *early-reject algorithm* starts with a complete yet simple derivation (e.g., solving constraints under the same predicate) and moves to the next derivation only if the current derivation has a solution. A sound and complete algorithm can also be designed, as a dual to the early-accept strategy, as illustrated by the red arrow in Figure 4.2.

- **Equivalent Derivation and One-shot** For both early-accept and early-reject algorithms, their final derivation are both sound and complete. Another possible solving strategy may directly work on an equivalent derivation without looking for the opportunities of early termination. We call such a special case the *one-shot approach*. In fact, one existing algorithm for inferring dependent labels [9] is a one-shot algorithm.

An eagle-eyed reader may find that under the derivation model, the main challenge of designing an inference algorithm is to construct derivations and validate their soundness and/or completeness. The derivation framework offers a couple of core properties of derivations to check against for such purposes (Section 4.4.2), so that the soundness and/or completeness of derivations can be validated in a simple way.

### 4.3 Core Constraint Language

To enable label inference for various information flow analyses with dependent labels, we formalize a core constraint language with *predicated constraints*.
4.3.1 Constraint Syntax

The syntax of predicated constraints is given in Figure 4.3. In this constraint language, a security label is either a known security level $\ell$ from a lattice $L$, or an unknown constraint variable $\alpha$ to be solved. A label constraint $\tau_1 \sqsubseteq \tau_2$ denotes the security restriction that label $\tau_2$ must be at least as restrictive as $\tau_1$ (i.e., the lattice allows $\tau_1 \sqsubseteq \tau_2$). Concatenation of two label constraints $C_1 \land C_2$ denotes the restriction on both constraints.

A predicated constraint set $C$ is a concatenation of predicated constraints, each in the form of $P \rightarrow C$, where $P$ is a predicate on program state, and $C$ is a label constraint. In general, any practical theory of predicates, such as program logics in decidable theory, program permissions [9], are allowed in the core language. To be concrete, we use predicates of boolean program expressions $b$ to express program state without losing generality: in the inference algorithm we only assume two abstract operations on the predicate logic: $\forall(P)$ (resp. $\exists(P)$) is true if when all free variables are universally (resp. existentially) quantified, $P$ is true. In the concrete syntax of $P$, we use op for arithmetic operations (e.g., $+$, $-$), cop for binary tests on arithmetic expressions (e.g., $<$, $\geq$), and bop for boolean operations (e.g., $\land$, $\lor$).

For convenience, we also treat a predicated constraint set $C$ as a set of constraints, written as $\{P_1 \rightarrow C_1, \ldots, P_n \rightarrow C_n\}$. Moreover, we use $\mathbb{P}_C$ to denote the set of predicates in $C$ (i.e., $\{P_1, \ldots, P_n\}$), and use $\mathbb{C}_C$ to denote the set of label constraints in $C$ (i.e., $\{C_1, \ldots, C_n\}$).

Consider the sets of predicated constraints in Figure 4.1, where the predicates over-
approximate possible program states at certain program points. The second predicated constraint in (4.2.1) reads as: whenever the program state satisfies \( d > 0 \), the restriction that \( S \sqsubseteq \alpha_y \) must hold. We will provide more examples that connect predicated constraints to information flow analysis in Section 4.3.3.

### 4.3.2 Constraint Validity and Satisfiability

When a predicated constraint set \( C \) involves no variable, its \textit{validity} is defined in a trivial way: \( C \) is valid iff for any \( P \to C \) in \( C \), \( C \) is valid (i.e., all label constraints obey the partial ordering in lattice \( \mathcal{L} \)). Note that when \( P \) is \texttt{false}, a constraint is vacuously true; hence, such constraints are excluded from further analysis.

When \( C \) involves constraint variables, intuitively, \( C \) is \textit{satisfiable} if and only if there is a \textit{solution} \( \kappa \) which maps constraint variables to security levels so that constraints after substitution are valid. Since constraints are dependent, it is not surprising that a constraint solution \( \kappa \) also involves predicates, in the form of \( \{ P_1 \to s_1, \ldots, P_n \to s_n \} \) where \( s_i \) is a \textit{label solution} that maps each constraint variable to a concrete security level.\(^1\) For example, a (predicated) solution

\(^1\)Another (perhaps more intuitive) definition of solution could be a mapping from each variable to predicated security levels in the form of \( \{ P_1 \to \ell_1, \ldots, P_n \to \ell_n \} \). It is easy to check that these two forms are interchangeable.
\( (d > 0 \rightarrow \{\alpha_x \mapsto P\}, d \leq 0 \rightarrow \{\alpha_x \mapsto S\}) \) means that the solution of \( \alpha_x \) is \( P \) iff \( d > 0 \) holds.

To simplify technical development in the constraint language theory, we further restrict that all predicates in a solution form a partition, meaning that no two predicates intersect and the union of all predicates is the same as \( \text{true} \) in the predicate logic:

**Definition 8 (Predicate Partition)** We say a predicate set \( \{P_1, \ldots, P_n\} \) is a partition iff it satisfies both:

1. \( \wedge_{1 \leq i < j \leq n} \neg (P_i \land P_j) \)
2. \( \vee_{1 \leq i \leq n} P_i \)

Although such a requirement seems to be restrictive at first glance, it actually allows all legal solutions: a “solution” where some predicates overlap either has a conflict (e.g., \( (d > 0 \rightarrow \{\alpha_x \mapsto P\}, d > 1 \rightarrow \{\alpha_x \mapsto S\}) \)) or can be normalized into a solution by merging the shared solution for the intersection. For condition 2), intuitively, a solution should provide an answer for any possible predicate in the constraints.

Next, we formally define the correctness of a predicated solution \( \kappa \) for predicated constraints \( C \), written \( \kappa \models C \). The rules are given in Figure 4.4. Rule (V-PCON) requires that for any label solution \( s_j \) such that the corresponding predicate \( P_j \) “overlaps” with \( P_i \) in \( C \) (i.e., \( \circ \exists (P_i \land P_j) \)), \( s_j \) is a correct solution for the corresponding labeled constraint \( C_i \), written as \( [C_i]_{s_j} \). Checking a label solution for label constraints \( (\llbracket C \rrbracket, s) \) is more straightforward: we simply substitute constraint variables with their corresponding security levels and check the validity of the result.

Here, we use \( \langle s, \tau \rangle \downarrow \ell \) to denote the concrete level of \( \tau \) under label solution \( s \). For example,

\( (d > 0 \rightarrow \{\alpha_x \mapsto P, \alpha_y \mapsto S, \alpha_z \mapsto P\}, d \leq 0 \rightarrow \{\alpha_x \mapsto P, \alpha_y \mapsto P, \alpha_z \mapsto P\}) \) is a correct solution for the original constraints set (4.2.1). It is easy to check that the predicates form a partition. Moreover, \( d > 0 \) intersects with the predicates of lines 1, 2 in the original constraints; it is easy to check that all label constraints are valid under the label solution in this case.

We use the form of \( \{P_1 \rightarrow s_1, \ldots, P_n \rightarrow s_n\} \) to simplify some definitions (e.g., the correctness of a solution).
Figure 4.5. Program and Constraints using Boolean Expression Predicates.

Furthermore, $d \leq 0$ intersects with the predicates of lines 1, 3, 4 in the original constraints; it is
easy to check that all label constraints are valid under the label solution in this case as well.

Based on Figure 4.4, we define constraint satisfiability in the standard way:

**Definition 9 (Predicated Constraint Satisfiability)** We say a constraint set $C$ is satisfiable,
denoted as $\models C$, if and only if exists a correct solution $\kappa$:

$$\models C \iff \exists \kappa. \kappa \models C$$

### 4.3.3 Expressiveness of the Core Constraint Language

Despite its simplicity, to our best knowledge, the core constraint language can formalize all static
information flow analyses with dependent labels. Next, we informally show how to encode four
very different kinds of static information flow analyses with dependent labels [5–9,11,70,90–92]
in the core language. Since the security labels in [6] are defined as a function from predicates
to security levels, a trivial encoding exists. Here, we outline the encoding for other analyses.
**Ternary Labels** Following the syntax of ternary expression in C, some prior works [5, 8, 92] employ dependent label in form of \( b ? \tau_1 : \tau_2 \), meaning that the concrete security level is \( \tau_1 \) when \( b \) evaluates to \( \text{true} \); otherwise the level is \( \tau_2 \). Consider the program in Figure 4.5(a), where the security labels of variables \( a, b, c, d, k \) are known, while the labels of \( x, y \) (\( \alpha_x \) and \( \alpha_y \)) are to be inferred. The analysis in [92] generates the raw constraints (in their syntax) shown in Figure 4.5(c). For instance, the third constraint \( d > 0 \Rightarrow S \sqsubseteq \alpha_y \) is generated from line 4 in the source code: the level of \( y \) must be at least as restrictive as that on \( b \) whenever the “then” branch is taken. The corresponding constraints in the core language are shown in Figure 4.5(b), with the following key steps for transformation:

- **Lifting dependence:** the main mismatch between the raw constraints and our core constraints is that the former allows nested predicates, such as \( ((d > 0)?S : P) \sqsubseteq \alpha_y \). We can lift all nested predicates into the predicates in the core language. For example, \( ((d > 0)?S : P) \sqsubseteq \alpha_y \) can be lifted to two constraints: \( (d > 0) \rightarrow S \sqsubseteq \alpha_y \) and \( \neg(d > 0) \rightarrow P \sqsubseteq \alpha_y \).

- **Removing join and meet:** when a join operation \( \sqcup \) shows up on the left-hand-side of a label constraint, we simply decompose it into two constraints. For example, \( L \sqcup \alpha_z \sqsubseteq \alpha_x \) decomposes into \( L \sqsubseteq \alpha_x \wedge \alpha_z \sqsubseteq \alpha_x \). The dual can be done for the meet operation on the right-hand-side.

- **Default predicate:** without any predicate, a raw constraint means that the label constraint holds under all conditions. We make this explicit by adding a predicate \( \text{true} \) for such constraints.

**Predicate as Labels** When a security lattice only has two labels \( P \) and \( S \), some prior works use predicates to specify dependent labels [7, 70]. A predicate \( P \) in those analyses can be interpreted as \( P \) when \( P \) holds; otherwise, the level is \( S \). For example, \( \{d \leq 0\} \) represents a dependent label that when \( d \leq 0 \), the level is \( P \); otherwise, the level is \( S \). The raw constraints...
Figure 4.6. Program and Constraints using Permission Trace.

from [70] are shown in Figure 4.5(d), where constraints are in form of \( P_1 \leq_p P_2 \), where the \( P \) serves as the condition under which the restriction \( P_1 \leq P_2 \) holds. Their analysis uses \( \cup \) operation when multiple variables are used in an expression, such as line 2 in the example. To encode such constraints, the key steps are:

- **Union decomposition**: unions semantically means “join” on security labels. Hence, we can decompose them in a way similar to encoding join. For example, \( \{\text{true}\} \cup \alpha_z \leq_p \alpha_x \) decomposes into \( \{\text{true}\} \leq_p \alpha_x \wedge \alpha_z \leq_p \{\text{true}\} \alpha_x \).

- **Lifting predicates**: similar to lifting predicates for ternary labels, both \( P_1 \) and \( P_2 \) can be lifted into \( P \). For example, \( \{d \leq 0\} \leq_p \{\text{true}\} \alpha_y \) is encoded to \( \{\text{true}\} \leq_p \{d \leq 0\} \alpha_y \) and \( \{\text{false}\} \leq_p \{\neg(d \leq 0)\} \alpha_y \).

- **Replacing predicates with levels**: after the previous two steps, other than constraint variables, the remaining label constraints only involve \( \{\text{true}\} \) or \( \{\text{false}\} \), where we replace the former with \( P \) and the latter with \( S \). For instance, \( \{\text{true}\} \leq_p \{d \leq 0\} \alpha_y \) is encoded as \( d \leq 0 \rightarrow P \sqsubseteq \alpha_y \).

The encoded constraints in the core language are shown in Figure 4.5(b). This is identical to the encoding of the raw constraints in Figure 4.5(c), suggesting a strong connection between the ternary labels and predicates as labels. Though the analysis in [7] differs in certain ways, its constraints can be encoded in a similar way.
**Permission Predicates**  Security level in [9] depends on permissions granted to a program. Their analysis generates constraints in form of \((P, \tau_1 \leq \tau_2)\), corresponding to \(P \rightarrow \tau_1 \sqsubseteq \tau_2\) in the core constraint language except that \(P\) is a set of granted permissions (written as \(\oplus p\)) or permissions that are not granted (written as \(\ominus p\)). Consider the program in Figure 4.6(a), adapted from [9]. The security label of \(p, q\) are known as \(l_p, l_q\) and the label of \(r\) is unknown and annotated as \(\alpha\). The encoding (shown in Figure 4.6(c)) is straightforward, where permissions are encoded as boolean expressions in the core language.

**Dynamic Labels**  Dynamic labels that can be found in Jif [11] and its extensions [90, 91] has a restricted form of dependence: it depends on a label-typed variable. For example, a constraint \(\alpha_x \sqsubseteq l\) where \(l\) is a label-typed program variable specifies a constraint that the level of \(x\) must be at most as restrictive as the run-time value of \(l\) when the corresponding program statement is executed. For any finite lattice, such constraints can be encoded by enumerating all possible values of \(l\): \(\alpha_x \sqsubseteq l\) can be encoded as \(l = P \rightarrow \alpha_x \sqsubseteq P\) and \(l = S \rightarrow \alpha_x \sqsubseteq S\) for a two-level lattice \(\{P, S\}\). We note that such a naive encoding might be inefficient for a complex lattice, since the encoding requires one extra constraint for each level in the lattice. One more efficient alternative is to enrich the label constraints to include extra assumptions on label-typed variables and utilize more sophisticated label-constraint solvers (e.g. the solver in Jif and the SHErrLoc solver [94]) for the enriched label constraints with assumptions. However, since the efficiency of encoding is largely an orthogonal issue, we leave that as future work.

**Converting Solution Back**  In most cases of information flow analysis, what matters is the existence of a solution (meaning that the program being verified is secure), rather than what a solution it is. However, we note that if needed, a solution in the core language can be decoded as well, mostly following the same idea for encoding. Taking the analysis in [92] as an example. For any constraint variable \(\alpha\), a solution \(\kappa = \{P_1 \rightarrow s_1, \ldots, P_n \rightarrow s_n\}\) can be decoded as a ternary label \(P_1?s_1(\alpha) : \ldots P_{n-1}?s_{n-1}(\alpha) : s_n(\alpha)\) (recall that \(\mathbb{P}_\kappa\) is a partition).
4.3.4 Alternative Formalizations

Various forms of constraints have been used for program verification tasks. However, to the best of our knowledge, no existing form of constraints can directly handle dependent security labels with an arbitrary security lattice. Next, we discuss the major hurdles of handling dependent security labels via existing forms of constraints.

**SMT Constraint**  It is possible to encode dependent label subtyping as an SMT constraint with alternating quantifiers; however, such constraints are in general undecidable and handled quite poorly in practice. This stands in contrast to label subtyping without dependency, which can be encoded using efficiently-solvable, quantifier-free SMT constraints. For example, a (predicated) constraint \((b = 1 \rightarrow \alpha \sqsubseteq P) \land (b \neq 1 \rightarrow S \sqsubseteq \alpha)\) has a dependent solution that \(\alpha\) is \(P\) whenever \(b = 1\); \(\alpha\) is \(S\) whenever \(b \neq 1\). But an SMT solver with theory on semi-lattice will simply reject the constraint, since neither \(P\) nor \(S\) (without dependence) is a solution for \(\alpha\).

**Constraint Horn Clauses (CHCs)**  CHCs is an intermediate constraint format commonly used in functional verification. We note that in the simplest case of two-level security lattice, CHCs is a promising alternative formalism of dependent security labels. For instance, dependent labels can be encoded by predicates, as shown in Figure 4.5(d), which is adapted from prior work [70]. Recall that the satisfaction of a predicate encodes \(P\) in such encoding. Hence, with form of \(P_1 \leq_P P_2\), the constraints can be converted to \(P \land P_2 \Rightarrow P_1\) in CHCs. Lifty [7] employs a similar idea by encoding two security levels \(P\) and \(S\) into tagged types, with predicates over stores and users. Lifty employs an inference engine that transforms constraints into Horn Clauses. Although CHCs is a neat formalism for a two-level lattice, extending it to encode constraints with a general security lattice is still an open and challenging problem.
4.4 The Derivation Framework

The derivation framework enables the design and validation of label inference algorithms that transform original constraints into a simplified form where a solution is more feasible to obtain. In this section, we first describe three approaches under this model and then develop a proof framework that validates the soundness and completeness of algorithms under the derivation framework.

4.4.1 Derivation Framework and Its Instances

We first explore the large space of inference algorithms under our derivation framework by giving derivation examples and show why they are useful. We defer the discussion of how to check their soundness and completeness to Section 4.4.2.

We first define *derivations* as constraint set transformation:

**Definition 10 (Derivation)** A derivation abstracts a transformation from one constraint set $C_1$ to another constraint set $C_2$, denoted as $C_1 \Rightarrow C_2$.

**Sound Derivations and Early-Accept Approach** For a derivation $C_1 \Rightarrow C_2$, we say it is *sound* if any solution of $C_2$ is a solution of $C_1$ (intuitively, the derived constraints are stronger). Sound derivations are useful since if a constraint solving algorithm derives constraints in a sound way, the algorithm may terminate early, as long as there is a solution on the current derivation. We refer to such an approach as the *early-accept* approach.

The early-accept approach is an iterative approach where a sound derivation is employed at each iteration. To make the algorithm complete eventually, at each iteration, an early-accept algorithm weakens the constraints in current iteration to produce weaker yet sound derivation for the next iteration. If the final iteration is a sound and complete derivation, the algorithm becomes both sound and complete.
A set of sound derivations are shown in Figure 4.1. At iteration 1, the derived constraint set simply removes all predicates and check if the constraints can be solved without any dependence. This derivation is unsatisfiable, thus, a weaker derivation is employed in iteration 2. At iteration 2, constraints are solved under predicate \( d > 0 \) and its negation. This sound derivation has a solution (the solution shown in Section 4.3.2 already). Hence, the algorithm stops with the solution. If the derivation in iteration 2 were unsatisfiable, the algorithm continues to derivation 3, which is both sound and complete.

When a set of constraints can be solved with a small number of dependences, the early-accept approach might find a solution early. However, in the worst case (e.g., when constraints are unsatisfiable), a sound and complete algorithm may either find a solution or find there is no solution in the last derivation, which wastes computation resources.

**Complete Derivation and Early-Reject Approach** For a derivation \( C_1 \Rightarrow C_2 \), we say the derivation is complete if when there is no solution of \( C_2 \), then \( C_1 \) must be unsatisfiable (intuitively, the derived constraints are weaker). Complete derivations are useful since if a constraint solving algorithm derives constraints in a complete way, the algorithm may terminate early, as long as there is no solution on the current derivation. We refer to such an approach as the early-reject approach.

The early-reject approach is dual to the early-accept approach. Starting from a complete derivation, an early-reject algorithm works on complete derivations in each iteration. If the final iteration has a sound and complete derivation, then the algorithm is both sound and complete. For example, an unsatisfiable constraint set and its complete derivations are shown in Figure 4.7.

Figure 4.7, derivations 1 and 2 are satisfiable. Hence, an early-reject algorithm continues to the final derivation, where no solution exists due to the second constraint \( a = 0 \Rightarrow S \sqsubseteq \alpha \land \alpha \sqsubseteq S \land \alpha \sqsubseteq P \). Hence, the algorithm rejects the original constraints.

Compared with early-accept, early-reject has the potential to reject a constraint set early.
Unsatisfiable Constraint Set

\( a \leq 0 \rightarrow S \subseteq \alpha \land \alpha \subseteq S \)  
\( a \geq 0 \rightarrow \alpha \subseteq P \)  

**Derivation 1:** (A complete derivation)

\( a \leq 0 \rightarrow S \subseteq \alpha \)  

**Final Derivation:** (A sound and complete derivation)

\( a < 0 \rightarrow S \subseteq \alpha \land \alpha \subseteq S \land \alpha \subseteq P \)  

**Figure 4.7.** Constraints and Complete Derivations.

However, in the worst case (e.g., when constraints are satisfiable), a sound and complete algorithm may waste considerable computation resources in the complete but not sound derivations.

**One-Shot Approach** To achieve soundness and completeness, one direction is to start from a sound (resp. complete) derivation, and eventually reach a sound and complete derivation, as sketched above. Another direction is to transform the original constraints directly into an equivalent constraint set. We call such an approach the *one-shot* approach.

The *one-shot* approach uses one equivalent derivation in constraint solving. Consider the constraints in 4.5(b). The algorithm by [9] (an instance of one-shot approach) enumerates all combinations of predicates and their negations, generating an equivalent derivation:

\[
\{ 
  p \land q \rightarrow l_p \subseteq \alpha \land l_q \subseteq \alpha; 
  p \land \neg q \rightarrow l_p \subseteq \alpha; 
  \neg p \land q \rightarrow l_q \subseteq \alpha; 
  \neg p \land \neg q \rightarrow P \subseteq \alpha; 
\}
\]

These constraints can be solved with a solution that combines local solutions under each predicate.

Both the advantage and the disadvantage of this approach are conspicuous: the transformations are simple and intuitive to implement; however, the number of transformed constraints grows exponentially: given \( n \) different predicates, the transformation results in \( O(2^n) \) con-
straints under different predicates; this is confirmed in our evaluation (Section 4.6).

### 4.4.2 Proof Framework

The derivation framework allows a large space for constraint solving algorithms. One crucial question for those potential algorithms is that whether a derivation is *sound* (i.e., any solution of a derivation is a solution of the previous derivation) and/or *complete* (i.e., no solution of a derivation implies no solution of the previous derivation). We identify a few key properties to make it easy to check if a derivation is sound and/or complete.

**Soundness** A derivation is *sound* if the derived constraints are stronger:

**Definition 11 (Sound Derivation)** We say a derivation is sound, if any solution of the derived constraint set is also a solution of the original constraint set:

\[
\forall C_1 \Rightarrow C_2, \kappa. \kappa \models C_2 \Rightarrow \kappa \models C_1
\]

To make a derivation sound, we first note that the derived set should at least “cover” the same or more predicates in the original set, as illustrated to the left of Figure 4.8. The reason is that a solution on the stronger set should consider all cases constrained in the original set. For example, consider the (unsound) derivation, \((4.4.1) \Rightarrow (4.4.2)\) in Figure 4.7, where the case \(a > 0\) is not covered in (4.4.2). Then, a solution of (4.4.2) is not necessarily a solution of (4.4.1), since the former is “less constrained” under the uncovered case \(a > 0\). This requirement is formalized as the *Coverage* property on the predicate set:

**Property 1 (Coverage)** We say a predicate set \(P_2\) covers a predicate set \(P_1\), denoted as \(P_2 \succ P_1\), if they satisfy:

\[
\forall \forall (\bigvee P_1 \Rightarrow \bigvee P_2)
\]

Moreover, for soundness, each label constraint in the original set should be “projected”
(i.e., propagated) to the derived set. That means for each predicated constraint $P_i \rightarrow C_i$ in the derived set, $C_i$ should be stronger than any $C_j$ in the original set if $P_j \rightarrow C_j$ is in the original set and $P_i, P_j$ may occur at the same time (i.e., $\circ \exists (P_i \land P_j)$). Consider the unsound derivation (4.4.1) $\Rightarrow$ (4.4.3) and the sound derivation (4.4.1) $\Rightarrow$ (4.4.4). We notice that the constraint $\alpha \sqsubseteq P$ under $a \geq 0$ is not included under $a = 0$ in the (unsound) constraints (4.4.3). To check that the derivation (4.4.1) $\Rightarrow$ (4.4.4) is sound, we note that $a = 0$ has intersection with two predicates $a \leq 0$ and $a \geq 0$ in (4.4.1). Moreover, the label constraints under $a = 0$ involve all label constraints in (4.4.1) (i.e., the derived label constraints are stronger). Hence, the derivation (4.4.1) $\Rightarrow$ (4.4.4) is sound. We formalize this observation as the Projection property:

**Property 2 (Projection)** We say a constraint set $C$ is projected to a constraint $P \rightarrow C$, denoted as $C \rightsquigarrow_{\uparrow} P \rightarrow C$, if they satisfy:

$$\forall s. \llbracket C \rrbracket_s \Rightarrow \bigwedge_{P_i \rightarrow C_i \in C \land \exists (P \land P_i)} \llbracket C_i \rrbracket_s$$

Note that when $P$ has no intersection with any predicate in $\mathbb{P}_C$, the requirement is $\forall s. \llbracket C \rrbracket_s \Rightarrow \llbracket \emptyset \rrbracket_s$. Since any solution works on $\emptyset$, any $C$ suffice in this case. Intuitively, this does not break soundness since the original constraints put no restriction under $P$ anyway. We can lift the projection definition to constraint sets:

**Property 3 (Full Projection)** We say constraint set $C_1$ is fully projected to $C_2$, denoted as

![Figure 4.8. Illustrated Restrictions on Predicates.](image-url)
$C_1 \rightsquigarrow \uparrow C_2$, if every constraint in $C_1$ is projected to constraints in $C_2$:

$$\forall P_2 \rightarrow C_2 \in C_2, C_1 \rightsquigarrow \uparrow P_2 \rightarrow C_2$$

Altogether, a derived constraint is sound if (1) the new predicates cover the original predicates, and (2) the original constraint set is fully projected to the derived constraint set:

**Theorem 5 (Soundness)** If the original constraint set is covered by and fully projected to the derived constraint set, then the derivation is sound:

$$\forall C_1 \Rightarrow C_2, \kappa. C_1 \rightsquigarrow \uparrow C_2 \land \mathbb{P}_{C_2} \triangleright \mathbb{P}_{C_1} \land \kappa \models C_2 \Rightarrow \kappa \models C_1$$

**Proof.** By the definition of $\kappa \models C_1$, we need to show that $\forall P_i \rightarrow C_i \in C_1, P_k \rightarrow s_k \in \kappa. \circ \exists(P_i \land P_k) \Rightarrow [C_i]_{s_k}$. That is, we need to show $[C_i]_{s_k}$ under the condition that $\circ \exists(P_i \land P_k)$. By Property 1, we know that $\circ \forall(\lor \mathbb{P}_{C_1} \Rightarrow \lor \mathbb{P}_{C_2})$. Hence, $\circ \forall(P_i \land P_k \Rightarrow P_i \Rightarrow \lor \mathbb{P}_{C_1} \Rightarrow \lor \mathbb{P}_{C_2})$. Therefore, there there must be some $P_j \in \mathbb{P}_{C_2}$ such that $\circ \exists(P_k \land P_j \land P_i)$. That is, we have $\circ \exists(P_k \land P_j)$ and $\circ \exists(P_j \land P_i)$. From the former, we have $[C_j]_{s_k}$ due to validation rule V-PCon and the fact $\kappa \models C_2$; from the latter and Property 2 and 3, we have $[C_j]_{s_k} \Rightarrow [C_i]_{s_k}$. Therefore, we proved that $[C_i]_{s_k}$.

**Completeness** A derivation is complete if its derived constraint set is weaker:

**Definition 12 (Complete Derivation)** We say a derivation is complete, if any solution of the original set is also a solution of the derived set:

$$\forall C_1 \Rightarrow C_2, \kappa. \kappa \models C_1 \Rightarrow \kappa \models C_2$$

To make a derivation complete, there is one restriction on the predicate sets of $C_2$ (i.e., there is no dual to the coverage property). The reason is that, a local view is sufficient in rejecting a
constraint set. For example,

\[ a = 0 \rightarrow S \subseteq \alpha \land \alpha \subseteq S \land \alpha \subseteq P \]

which is unsatisfiable, allows us to reject (4.4.1) since when \( a = 0 \), all label constraints on the right are required in (4.4.1), but they are not satisfiable.

To make a derivation complete, each label constraint with precondition \( P \) should be *weaker* than the intersection of all \( C_i \) where \( P \) “refines” the corresponding \( P_i \) (i.e., \( P_i \) “includes” \( P \), or \( \circ \forall (P \Rightarrow P_i) \)), as illustrated in the middle of Figure 4.8. For example, (4.4.2) is a complete derivation of (4.4.1) according to this observation: first, \( a \leq 0 \) refines \( a \leq 0 \) but not \( a \geq 0 \) in (4.4.1); second, \( S \subseteq \alpha \land \alpha \subseteq S \Rightarrow S \subseteq \alpha \) (i.e., \( S \subseteq \alpha \) is weaker than the corresponding label constraint of \( a \leq 0 \) in (4.4.1)). However, a similar derivation \( \{a \leq 0 \Rightarrow \alpha \subseteq P\} \) is not a complete derivation from (4.4.1), since its label constraint \( \alpha \subseteq P \) is not weaker than the original one under the same precondition. This observation is formalized as the *Inferred* property:

**Property 4 (Inferred Constraint)** We say a constraint \( P \Rightarrow C \) is inferred from a constraint set \( C \), denoted as \( C \Rightarrow^o P \Rightarrow C \), if they satisfy:

\[
\forall s. \left[ \bigwedge_{P_i \Rightarrow C_i \in C \land \circ \forall (P \Rightarrow P_i)} C_i \right]_s \Rightarrow \left[ C \right]_s
\]

This property is a dual of the projection property 2. When \( P \) does not refine any predicate in \( \mathbb{P}_C \), the requirement is \( \forall s. \left[ C \right]_s \Rightarrow \left[ C \right]_s \). Since any solution works on \( \emptyset \), this property requires \( \forall s. \left[ C \right]_s \), which essentially makes it impossible to reject the original constraints due to \( P \Rightarrow C \).

We can lift the projection definition to constraint sets:

**Property 5 (Fully-Inferred)** We say constraint set \( C_2 \) is fully-inferred from constraint set \( C_1 \), denoted as \( C_1 \Rightarrow^o C_2 \) if all constraints are inferred constraints:

\[
\forall P_i \rightarrow C_i \in C_2. C_1 \Rightarrow^o P_i \rightarrow C_i
\]
A derivation is complete if it is fully-inferred:

**Theorem 6 (Completeness)** If the derived constraint set is fully-inferred from the original constraint set, then the derivation is complete:

\[
\forall C_1 \Rightarrow C_2, \kappa. C_1 \Rightarrow^\circ C_2 \land \kappa \models C_1 \Rightarrow \kappa \models C_2
\]

**Proof.** By the definition of \( \kappa \models C_2 \), we need to show that \( \forall P_j \rightarrow C_j \in C_2, P_k \rightarrow s_k \in \kappa. \circ \exists (P_j \land P_k) \Rightarrow [C_j]_{s_k} \). Consider the set \( Q = \{ P_i \mid P_i \in P_{C_1} \land \circ \forall (P_j \Rightarrow P_i) \} \). When \( Q = \emptyset \), we have \( C_j = \emptyset \) by Property 4. Hence, \( [C_j]_{s_k} \) is vacuously true.

Otherwise, for any \( P_i \in Q \), we have \( \circ \forall (P_j \Rightarrow P_i) \). Given \( \circ \forall (P_j \Rightarrow P_i) \) and \( \circ \exists (P_j \land P_k) \), it must be true that \( \circ \exists (P_j \land P_i \land P_k) \), which implies \( \circ \exists (P_i \land P_k) \). Since \( \kappa \models C_1 \), we have \([C_i]_{s_k}\) by validation rule V-PCon. Since this is true for any \( C_i \in Q \), we have \( \bigwedge_{C_i \in Q} [C_i]_{s_k} \). From Properties 4 and 5, we know that \( \forall s. \bigwedge_{C_i \in Q} [C_i]_{s} \Rightarrow [C_j]_{s} \). Hence, \( \bigwedge_{C_i \in Q} [C_i]_{s_k} \Rightarrow [C_j]_{s_k} \), and therefore, \([C_j]_{s_k}\). \( \square \)

**Equivalence** A derivation is equivalent if it is both sound and complete. We can check so by all of the properties defined above. However, in practice, it is more desirable to develop algorithms that start from sound (or complete) only derivations, and evolve to a sound and complete derivation eventually, with respect to the original constraint set (as illustrated in Figure 4.2).

To make it happen, we first define the weakest derivation that complies Property 3 and the strongest derivation that complies Property 5:

**Definition 13 (Weakest Sound Derivation)** A derivation is the weakest sound derivation, denoted as \( C_1 \Rightarrow^\dagger C_2 \), if

\[
P_{C_2} > P_{C_1} \text{ and } \forall P_2 \rightarrow C_2 \in C_2. C_2 = \bigwedge_{P_i \rightarrow C_i \in C_1 \land \circ \exists (P_i \land P_2)} C_i
\]
By construction, the soundness of this derivation is trivial. Similarly, we define the strongest complete derivation, whose completeness directly follows its construction:

**Definition 14 (Strongest Complete Derivation)** A derivation is the strongest complete derivation, denoted as $C_1 \Rightarrow^\circ C_2$, if

$$\forall P_2 \rightarrow C_2 \in C_2. C_2 = \bigwedge_{P_i \rightarrow C_i \in C_1 \land \forall \forall (P_2 \Rightarrow P_i)} C_i$$

What makes weakest sound derivation and strongest complete derivation interesting is that they are interchangeable when for any $P_1 \in P_{c_1}$ and $P_2 \in P_{c_2}$, $P_2$ refines $P_1$ whenever they intersect, as illustrated to the right of Figure 4.8. This is called the Refinement property:

**Property 6 (Refinement)** We say a predicate set $P_2$ is a refinement of a predicate set $P_1$, denoted as $P_1 \ll P_2$, if we have

1. $\forall P_2 \in P_2, \circ \exists (P_2)$
2. $\forall P_1, P_2 \in P_2. \circ \exists (P_1 \land P_2) \implies \circ \forall (P_2 \Rightarrow P_1)$

**Theorem 7** A refined weakest sound derivation is a strongest complete derivation:

$$\forall C_1 \Rightarrow^\dagger C_2, \kappa. P_{C_1} \ll P_{C_2} \Rightarrow C_1 \Rightarrow^\circ C_2$$

**Proof.** Consider any $P_2 \in P_{c_2}$. Consider any $P_1 \in P_{c_1}$ such that $\circ \forall (P_2 \Rightarrow P_1)$. Since $\circ \exists (P_2)$, we know that $\circ \exists (P_2 \land (P_2 \Rightarrow P_1)) = \circ \exists (P_2 \land P_1)$. Hence, the refinement property implies that $\circ \exists (P_1 \land P_2) \iff \circ \forall (P_2 \Rightarrow P_1)$. Hence, definitions 13 and 14 coincide. □

This theorem provides a constructional strategy for evolving sound derivations to a sound and complete derivation. For example, given original predicates $P_c = \{P_1, P_2\}$, the following sequence of covers of $P_c$ eventually satisfies the refinement property: $\{\text{true}\}, \{P_1, \neg P_1\},$
\{P_1 \land P_2, P_1 \land \neg P_2, \neg P_1 \land P_2, \neg P_1 \land \neg P_2\}. We discuss further on such an inference algorithm in Section 4.5.2.

On the other hand, a complete derivation becomes sound and complete if it satisfies refinement and covers the original predicate set:

**Theorem 8** A refined strongest complete-derivate is a weakest sound-derivation, if the derived constraint set covers the original one

\[ \forall C_1 \Rightarrow^\kappa C_2, \kappa. \mathbb{P}_{c_1} \ll \mathbb{P}_{c_2} \wedge \mathbb{P}_{c_2} \gg \mathbb{P}_{c_1} \Rightarrow C_1 \Rightarrow^\dagger C_2 \]

**Proof.** The proof is similar to that of Theorem 7. Note that the coverage requirement in weakest sound derivation is in the assumption. \qed

### 4.5 Algorithms

There are many approaches to utilize the derivation framework. In this section, we discuss how to instantiate the major components of the framework, as well as provide four concrete constraint solving algorithms.

#### 4.5.1 Partition Method

The general framework allows great flexibility in the predicate set \(\mathbb{P}\) of a derivation. Hereafter, we explore one approach, where the predicate set \(\mathbb{P}\) forms a partition (Definition 8) for two reasons: (1) a partition satisfies the coverage requirement for sound derivation as well as sound and complete derivations, and (2) a partition allows a solver to solve label constraints under each predicate in isolation and combine sub-solutions to one solution. We call the procedure of generating predicate set \(\mathbb{P}\) of a derivation as a partition *partition method*.

Intuitively, a partition method controls the pace of the solving process (as illustrated by the lines in Figure 4.2): in the one-shot approach, partition method directly generates a refinement
of the original constraint set; in early-accept or early-reject approach, partition algorithm is responsible to provide a different set of predicates at each iteration and also guarantee that the final predicate set is a refinement of the original constraint set.

Since the one-shot approach is straightforward, we next focus on two concrete partition algorithms, following two different intuitions:

- **Sequential Partitioning** is an avid algorithm devoted to reaching the final refinement with the least efforts wasted on the path;

- **Combinational Partitioning** is more adventurous, trying out all kindred cases along the way.

Both algorithms start from singleton partition \{\text{true}\} and eventually reach the same refinement of the original constraint set at the final iteration. We know when one predicate \(P_1\) is given, the space can be partitioned into two parts, \(P_1\) and \(\neg P_1\). When two predicates \(P_1, P_2\) are given, the space can be at most partitioned into four parts: \(P_1 \land P_2, P_1 \land \neg P_2, \neg P_1 \land P_2\) and \(\neg P_1 \land \neg P_2\). We call this relationship Partition Space:

**Definition 15 (Partition Space)** Given predicates \(P_1, P_2, ..., P_n\), a partition space of these predicates, denoted as \(\square(P_1, P_2, ..., P_n)\) is a set of predicates after partitioned :

\[
\square(P_1, P_2, ..., P_n) = \{ p_1 \land p_2 \land ... \land p_n \mid p_1 \in \{P_1, \neg P_1\}, p_2 \in \{P_2, \neg P_2\}, ..., p_n \in \{P_n, \neg P_n\}\}
\]

Given predicates \(P_1, ..., P_n\) in the original set, the sequential partitioning algorithm constructs predicate sets \{\text{true}\}, \{P_1, \neg P_1\}, \{P_1 \land P_2, P_1 \land \neg P_2, \neg P_1 \land P_2, \neg P_1 \land \neg P_2\}, ..., while the combinational partitioning algorithm constructs sets \{\text{true}\}, \{P_1, \neg P_1\}, \{P_2, \neg P_2\}, ..., as follows.

**Sequential Partitioning** Sequential partitioning refines the partition sequentially one predicate at a time. For a predicate set \(\{P_1, P_2, ..., P_n\}\), the sequential partitioning algorithm
produces:

**Iteration 0**: Starting : \{true\}

**Iteration 1**: Partition by \(P_1\): \[\Box (P_1)\]

**Iteration 2**: Partition by \(P_1, P_2\): \[\Box (P_1, P_2)\]

... 

**Iteration n**: Partition by \(P_1, P_2, \ldots, P_n\): \[\Box (P_1, P_2, \ldots P_n)\]

Consider the predicates used in the sound derivations in Figure 4.1. It follows a sequential partitioning: derivation (4.2.2) uses \{true\}, derivation (4.2.3) uses \{\(d > 0, d \leq 0\)\}, and derivation (4.2.4) uses \{\(d > 0 \land d < 0, d > 0 \land d \geq 0, d \leq 0 \land d < 0, d \leq 0 \land d \geq 0\)\}, which is equivalent to \{\(d > 0, d = 0, d = 0\)\}. The last predicate set is a refinement of the original predicates.

While sequential partitioning is intuitive and simple to implement, we observe that it is not stable at performance: the execution time of constraint solving may differ dramatically depending on the order of the input predicates. Consider a special case that only the sound derivation partitioned by \(P_1\) can be solved. In the best case, sequential partitioning picks \(P_1\) in the first iteration, resulting a partition \[\Box (P_1)\] of size 2; however, in the worse case, \(P_1\) gets picked at the last iteration. In this case, the partition algorithm will go generate a last-level partition \[\Box (P_1, P_2, \ldots P_n)\], with \(2^n\) predicated constraints.

**Combinational Partitioning**  Combinational partitioning is designed to be more stable for the performance. It tries out all partitions at the same level before going to the next level. For a predicates set: \\{\(P_1 \rightarrow C_1, P_2 \rightarrow C_2, \ldots, P_n \rightarrow C_n\)\}, the partition procedure is as follows:

**Iteration 0**: Starting : \{true\}

**Comb 1 - Iteration 1**: Partition by \(P_1\): \[\Box (P_1)\]
Comb 1 - Iteration 2: Partition by $P_2$: $\square (P_2)$

...

Comb 1 - Iteration n: Partition by $P_n$: $\square (P_n)$

Comb 2 - Iteration 1: Partition by $P_1, P_2$: $\square (P_1, P_2)$

Comb 2 - Iteration 2: Partition by $P_1, P_3$: $\square (P_1, P_3)$

...

Comb 2 - Iteration $C_n^2$: Partition by $P_{n-1}, P_n$: $\square (P_{n-1}, P_n)$

...

Comb n - Iteration 1: Partition by $P_1, P_2, ... , P_n$: $\square (P_1, P_2, ... P_n)$

For the special case that only the sound derivation partitioned by $P_1$ can be solved, in the worse case, the combinational partition picks $P_1$ at Comb 1 - Iteration n with partition $\square (P_1)$ of size 2. Though the sequential partitioning also finds the result at the n-th iteration, the size of the partition set grows exponentially at each iteration. For combinational partitioning, partition in Comb 1 are all of size 2; thus, the efforts to reach the n-th iteration is just linear, rather than exponential.

4.5.2 Derivation Method

A derivation method, the core component of a solving algorithm, constructs derivations to be checked under the current iteration. In particular, a derivation method takes in two parameters: the original predicated constraint set $\{P_1 \rightarrow C_1, \ldots, P_n \rightarrow C_n\}$, as well as a predicate set $\mathcal{P}$ (constructed by a partition method from Section 4.5.1), under which a set of label constraints are to be generated.
For each predicate $P \in \mathcal{P}$, a sound derivation method (\textsc{SoundDerive}) computes predicated constraints under $P$ following Definition 13:

$$\text{SoundDerive} \left( \left\{ P_1 \rightarrow C_1, \ldots, P_n \rightarrow C_n \right\}, \mathcal{P} \right) \triangleq \bigcup_{P \in \mathcal{P}} \left\{ P \rightarrow C_i \mid \exists (P \land P_i) \right\}$$

Similarly, a complete derivation method (\textsc{CompleteDerive}) computes predicated constraints under $P$ following Definition 14:

$$\text{CompleteDerive} \left( \left\{ P_1 \rightarrow C_1, \ldots, P_n \rightarrow C_n \right\}, \mathcal{P} \right) \triangleq \bigcup_{P \in \mathcal{P}} \left\{ P \rightarrow C_i \mid \forall (P \Rightarrow P_i) \right\}$$

Since we simply construct the sound/complete sets as the weakest sound derivation/strongest complete derivation (Definitions 13 and 14), their correctness is easy to validate. Since the partition method ensures coverage and refinement in the last step, both derivations are eventually sound and complete (Theorems 7 and 8).

### 4.5.3 Constraint Solving Algorithms

Based on derivation and partition methods, we present four concrete algorithms.

- **One-Shot Algorithm**: this algorithm is equivalent to the solver in [9]. It directly works on a refinement of the original constraint set.

- **Iterative Early-Accept Algorithm**: this algorithm iteratively uses weakest sound derivations. Hence, if an iteration is satisfiable, the algorithm accepts the original constraint set; otherwise, the algorithm continues until the final equivalent derivation.

- **Iterative Early-Reject Algorithm**: this algorithm iteratively uses strongest complete derivations. Hence, if an iteration is unsatisfiable, the algorithm rejects the original constraint set; otherwise, the algorithm continues until the final equivalent derivation.
Since we simply construct the sound/complete sets as the weakest sound derivation/strongest complete derivation (Definitions 6 and 7), their correctness is easy to validate. Since the partition method ensures coverage and refinement in the last step, both derivations are eventually sound and complete (Theorems 3 and 4).

5.3 Constraint Solving Algorithms

Based on derivation and partition methods, we present four concrete algorithms.

- **One-Shot Algorithm**: this algorithm is equivalent to the solver in Chen et al. [2018]. It directly works on a refinement of the original constraint set.
- **Iterative Early-Accept Algorithm**: this algorithm iteratively uses weakest sound derivations. Hence, if an iteration is satisfiable, the algorithm accepts the original constraint set; otherwise, the algorithm continues until the final equivalent derivation.
- **Iterative Early-Reject Algorithm**: this algorithm iteratively uses strongest complete derivations. Hence, if an iteration is unsatisfiable, the algorithm rejects the original constraint set; otherwise, the algorithm continues until the final equivalent derivation.
- **Hybrid Algorithm**: The hybrid algorithm takes advantages of both sound and complete derivations and seeks for early termination whenever possible.

Next, we discuss each algorithm in details.

**One-Shot Algorithm**

One-shot algorithm directly works on a refinement of the original constraint set, where its sound derivation is equivalent to its complete derivation. The pseudo code is shown as Algorithm 1 in Figure 4.9. It first initializes the partition algorithm and obtains the final refinement partition (Section 4.5.1). Sound derivation method is then employed at line 5 to generate label constraints for each predicate in the partition set.

At line 7, a label constraint solver, rmSolver (an implementation of the Rehof-Mogensen algorithm [13]) is used to solve label constraints under each predicate. rmSolver solves label constraints without predicates. Since the derivation at line 5 is indeed equivalent, if any constraint is unsatisfiable, the original set is rejected at line 11. Only when all label constraints

---

**Algorithm 1 One-Shot Solver**

```plaintext
1: function SOLVE-ONE-SHOT(cons-set)
2:    solution ← ∅
3:    partition.INIT(cons-set)
4:    partt ← partition.FINAL-PARTT()
5:    derive ← SOUNDDERIVE(cons-set, partt)
6:    for i ← 0, length(partt) do
7:        if rmSolver.SOLVE(derive[i]) then
8:            solution[partt[i]] ← rmSolver.SOLUTION()
9:        else
10:            unsolve ← partt[i]
11:            return false
12:        end if
13:    end for
14:    return true
15: end function
```

**Figure 4.9. One-Shot Solver.**

- **Hybrid Algorithm** The hybrid algorithm takes advantages of both sound and complete derivations and seeks for early termination whenever possible.

---

76
Iterative Early-Accept Algorithm. Early-accept algorithm makes use of sound derivations so that if any sound derivation is satisfiable, the algorithm accepts the constraint set with the solution found; otherwise, the algorithm continues until the final equivalent derivation. The pseudo code is shown in Figure 4.10.

In Figure 4.10, the partition algorithm is initialized and the current partition is obtained at line 5. The sound-derivation of each predicate is then generated and solved by the label constraint solver. The difference from the One-Shot approach is when the derived constraint is unsatisfiable: as sound, but not equivalent, derivation, it does not lead to a rejection. Instead, the predicate is recorded in new-partt, at line 12, to be further partitioned in the next iteration.
For each predicate in the current partition, it first solves the sound derivation at line 10. If the sound derivation is unsatisfiable, it solves the complete derivation at line 13 to see whether an early rejection is feasible. If not, the partition will be recorded in \( \text{partt} \rightarrow \emptyset \) to be further partitioned.

When \( \text{partt} \rightarrow \emptyset \) is empty at line 20, early-accept is triggered. A correct implementation should never reach line 24, since the final iteration is indeed an equivalence derivation.

**Algorithm 3** Hybrid Solver

```plaintext
1: function \textsc{Solve-Early-Accept}(cons-set)
2:    new-partt, solution \leftarrow \emptyset
3:    \text{partition.init}(cons-set)
4:    while partition.stop() == false do
5:        partt \leftarrow partition.next-partt(new-partt)
6:        new-partt \leftarrow \emptyset
7:        derive-sound \leftarrow \text{SoundDerive}(cons-set, partt)
8:        derive-complete \leftarrow \text{CompleteDerive}(cons-set, partt)
9:        for \( i \leftarrow 0, \text{length}(partt) \) do
10:           if \( \text{rmSolver.solve}(\text{derive-sound}[i]) \) then
11:              solution[partt][i] \leftarrow \text{rmSolver.solve}()
12:           else
13:              if \( \text{rmSolver.solve}(\text{derive-complete}[i]) \) then
14:                 unsolve \leftarrow partt[i]
15:                 return false
16:           end if
17:           new-partt.append(partt[i])
18:        end for
19:        if new-partt == \emptyset then
20:           return true
21:        end if
22:    end while
23:    return false
24: end function
```

**Figure 4.11.** Hybrid Solver.

If a derived constraint is solved, then the solution is strong enough that it does not need to be further partitioned. If the new-partt is empty at line 15, meaning that all the constraints in the current derivation are solved, then an early-accept solution is found. The algorithm thus accepts the constraint set at line 16. When the partition algorithm reaches the final partition and there still exists unsatisfiable constraints, the algorithm rejects the constraint set as unsatisfiable at line 20.

**Iterative Early-Reject Algorithm** Early-reject algorithm makes use of complete derivations and intends to early terminate with rejection. We omit the pseudo code since it is very similar
to Figure 4.10. The main difference is that it employs a complete derivation rather a sound derivation. When any label constraint is unsatisfiable, the algorithm rejects the constraint set immediately. It only accepts the input constraint set when where all the derived constraints are solved, including those from the final refinement partition.

**Hybrid Algorithm**  The hybrid algorithm takes advantages of both sound and complete derivations and seeks for early termination whenever possible. The pseudo code is shown in Figure 4.11. For each predicate in the current partition, it firsts solve the sound-derivation at line 10. If the sound derivation is unsatisfiable, it solves the complete derivation at line 13 to see whether an early rejection is feasible. If not, the partition will be recorded in new-partt to be further partitioned. When new-partt is empty at line 20, early-accept is trigged. A correct implementation should never reach line 24, since the final iteration is indeed an equivalence derivation.

### 4.6 Evaluation

We have implemented all four algorithms: One-Shot, Early-Accept, Early-Reject and Hybrid algorithms, as described in Section 4.5. We use Z3 [67], version 4.5.1, to solve arithmetic and logical constraints (arising from predicates) and implement the Rehof-Mogensen algorithm [13] for security-level constraints. The implementation is publicly available at https://github.com/psuplus/DerivationSolver. All experiments are run on a desktop with 16 GB of RAM and an Intel i7 processor at 2.2 GHz. During the evaluation, we were mostly interested in answering the following questions:

1. How efficient are those algorithms in solving predicated constraints?
2. How scalable are those algorithms in solving predicated constraints?
3. Between sequential and combinational partitioning, which one scales better?
4.6.1 Benchmarks

To evaluate constraint solving algorithms on constraints from realistic applications with information flow control, we obtain the source code of a formally verified MIPS processor with information flow control [5]. This processor is based on a classic 5-stage in-order pipeline with separate instruction and data caches. The processor also includes typical pipelining techniques, such as data hazard detection, stalling and data bypassing.

The processor consists of 1719 lines of Verilog code, separated into 18 files. In the original version, all security labels are explicitly marked. We extend the SecVerilog compiler [95] to generate fresh label variables (to be inferred) for unannotated variables and created 50 variants mutating from the original files as follows (all removed labels are randomly picked):

- partially remove manually annotated labels, or

- split multiple modules in one file into multiple files and remove labels, making label inference more local (i.e., removing cross-module constraints), or

- remove labels and inject errors by modifying some annotated labels, or removing checks needed for security (e.g., checking \( b \) is true before copying a value with label \( b?P : S \) to \( P \)). In total, 14 errors were injected to the 50 variants.

As a result, there are 2455 variables requiring security labels among all 50 files, where 1509 variables are unlabeled (i.e. to be inferred). Among the 50 files, 41 of them can only be verified with dependent labels.

The modified SecVerilog compiler generates predicated constraints in the syntax shown in Figure 4.5(c), which we then encode into our core language as sketched in Section 4.3.3. Note that since the variants of original files contain insecure code, the generated constraints tests have a mix of satisfiable and unsatisfiable constraints.

The SecVerilog compiler generates rich sets of predicates: the predicates include both path conditions and approximation of program states (details can be found in [5]).
the effect of predicates on constraint solving, we also generated predicated constraints that only contain path conditions. Our benchmark contains 100 constraint files in total: 50 set of constraints with all predicates, as well as 50 set of constraints with path conditions only.

### 4.6.2 Performance of Inference Algorithms

We first compare the constraint solving time of three algorithms (one-shot, early-accept and hybrid algorithms) on the MIPS benchmark under *combinational partition*. The comparison under sequential partitioning shows similar results, so we omit that in this section. The execution time (in log scale) for each constraint file is shown in Figure 4.12. In the evaluation, each algorithm times out after 3 minutes. Each timed-out case is plotted in the gray area, where the number on the right counts the total number of such cases for each algorithm.

We note that in most cases, the hybrid algorithm consistently performs better than the other two algorithms: the improvement is typically by orders of magnitude compared with the one-shot algorithm, and it only times-out for 3 tests, compared with 30 tests for the one-shot algorithm. The reason is that early-termination works well in practice: for most of the constraint files, a sound and complete answer can be returned without going all the way to the exponential case as the one-shot algorithm does. The result confirms the importance of early termination.

**Figure 4.12.** Performance on the MIPS benchmark. Each algorithm times out after 180s (3 mins). The gray area shows executions that time out after 180s. The X-axis represents file ID.
We do notice that in some hard cases (e.g., the 3 cases that the hybrid algorithm times out), all algorithms time out. For such cases, we plan to improve the hybrid algorithm as future work.

Compared with the early-accept algorithm, the hybrid algorithm has a comparable performance when both algorithms terminate. This is expected since for satisfiable constraints, the early-reject component of the hybrid algorithm has no effect. However, the hybrid algorithm only times-out for 3 tests, compared with 19 tests for the early-accept algorithm. When we inspect those 16 cases that these two algorithms differ, all of them are unsatisfiable constraints that require an exponential search in the early-accept algorithm. The result confirms the intuition that for a mix of satisfiable and unsatisfiable constraints, the hybrid algorithm is the most efficient.

4.6.3 Scalability

Figure 4.13 shows the execution time in terms of the number of unique predicates in the original constraints in a log-log scale (we use the number of predicates instead of the number of raw constraints since empirically, the former dominates the execution time). The hybrid method scales the best: its execution time grows slowest as the number of predicates increases (the higher plots for the hybrid algorithm suggests roughly a square-time complexity in practice). A
possible reason is that most constraints can be either accepted or rejected with a small number of dependences. The early-accept algorithm has mixed cases: for the satisfiable ones, it scales well as the hybrid algorithm; but for the unsatisfiable ones, it not very scalable. The execution time of one-shot algorithm grows consistently in an exponential manner: it time-out for most constraints with over 20 predicates.

We note that early-accept method may grow faster than the one-shot algorithm in some cases. We believe this demonstrates the trade-off between solving a large number of small-size label constraints (the one-shot algorithm) and a small number of big-size label constraints (the early-accept algorithm). For easy cases, this trade-off inclines towards the iterative approach, but for the hard cases, as the iteration goes deeper, the size of the partition is no longer small and the accumulated efforts to generate new partitions, produce derivations and validate the derived set do not pay off. Hence, in a worse case scenario (e.g., for unsatisfiable constraints), the early-accept algorithm may scale much worse than the one-shot algorithm.

4.6.4 Sequential vs. Combinational Partitioning

Figure 4.14 shows the performance of the hybrid algorithm under different partition algorithms. Almost all test cases yield a better solution on the combinational partitioning. The cases, where
two partitioning has similar performance, are mostly simple cases where a solution can be found within $10^{-2}$ seconds. The cases, where their performance diverges, are mostly hard cases and still combinational partitioning has a better result. This confirms the intuition that the combinational partitioning is a more stable algorithm that handles the worse case at the earliest combination level possible to avoid going to the exponential cases.

4.7 Related Work

**Security Label Inference**  Typical label inference algorithms, such as those employed in Jif [11] and FlowCaml [12], encode the restrictions on known and unknown security levels into constraints in a finite semi-lattice. Those constraints can be solved by customized solvers (such as the Rehof-Mogensen algorithm [13] and set-constraints solvers [14, 15]). However, constraints in finite semi-lattices cannot encode predicates on program states, which are essential for inferring dependent security labels.\(^2\)

When dependent labels come to picture, various ad-hoc inference algorithms have been proposed. Lifty [7] encodes information flow constraints into logical constraints (as illustrated in Section 4.3.3), and proposes an inference algorithm based on the inference engine of liquid types [96, 97]. While this is a neat solution for a two-level security lattice, the encoding does not apply to applications that require multiple security labels. Recent work by [9] considers security labels that depend on permissions granted to a program. Their inference algorithm is an instance of the one-shot approach: the algorithm constructs a complete partition of all (exponential) combinations of existence/absence of permissions. However, as we show in the evaluation, exploring all combinations does not scale with the increasing number of predicates in original constraints.

[98] propose a system for inferring declassification policies. But their work is largely

---

\(^2\)Conditional type [15] integrates limited control flow information (as types) into constraints. But that work does not handle general program predicates in constraints.
orthogonal to ours: their work infers declassification policies, while our work infers dependent labels *given a security policy*.

To the best of our knowledge, no existing work offers a framework for designing and checking the soundness and completeness of dependent-label inference algorithms. Although proving a variant of one-shot algorithms as did in [9] might be feasible, proving the soundness and completeness for advanced algorithms such as the hybrid algorithm can be extremely difficult without our derivation framework.

**Information Flow Analysis with Dependent Security Labels**  To improve the precision of static information flow analysis, dependent security labels have been introduced in various forms [5–9, 11, 70, 90–92]. Except for [7, 9], none of those works has dependent label inference; dependent security labels are annotated manually. Some prior type systems for information flow support limited forms of dependent labels [57, 71–74]. However, they only allow dependence on special “policy” or “label” variables, rather than run-time program states. Rich dependent type systems like Fable [60, 99], F* [100] can encode rich information flow policies, but policy inference in those systems is arguably more challenging than within our constraint language, and hence, less likely to be fully automated.

**Dependent Types**  Dependent types [93] have been widely studied and applied to practical programming languages, but most of them (e.g., [59, 61, 68, 69, 96, 97]) verifies functional correctness rather than information flow security. Though it might be possible to encode certain restricted policies into those systems (such as the encoding in [7]), at least there is no direct encoding for our *full constraint language* with a mix of predicates and security label constraints.
4.8 Summary

We present the first framework for designing and checking label inference algorithms for information flow analysis with dependent security labels. The framework models an inference algorithm as an iterative process where each step works on a sound and/or complete derivation of the original constraint set. Based on the framework, we developed novel inference algorithms that are both sound and complete. Evaluation result suggests that the novel algorithms improve performance by orders of magnitude and offers better scalability compared with existing work.
Chapter 5 | A General-Purpose Dynamic Information Flow Policy

5.1 Introduction

While noninterference [3] has become a cliché for end-to-end data confidentiality and integrity in information flow security, this well-accepted concept only describes the ideal security expectations in a static setting, i.e., when data sensitivity does not change throughout program execution. However, real-world applications almost always involve some dynamic security requirements, which motivates the development of various kinds of dynamic information flow policies:

- A declassification policy [18–25] weakens noninterference by deliberately releasing (i.e., declassifying) sensitive information. For instance, a conference management system typically allows deliberate release of paper reviews and acceptance/rejection decisions after the notification time.

- An erasure policy [26–31] strengthens noninterference by requiring some public information to become more sensitive, or be erased completely when certain condition holds. For example, a payment system should not retain any record of credit card details once
the transaction is complete.

- An delegation/revocation policy [32–35] updates dynamically the sensitivity roles in a security system to accommodate the mutable requirements of security, such as delegating/revoking the access rights of a new/leaving employee.

Moreover, there are a few case studies on the needed security properties in the light of one specific context or task [36–39], and build systems that provably enforces some variants of declassification policy (e.g., CoCon [40], CosMeDis [41]) and erasure policy (e.g., Civitas [42]).

Although the advances make it possible to specify and verify some variants of dynamic policy, cherry-picking the appropriate policy is still a daunting task: different policies (even when they belong to the same kind) have very different syntax for specifying how a policy changes [43], very different nature of the security conditions (i.e., noninterference, bisimulation and epistemic [44]) and even completely inconsistent notion of security (i.e., policies might disagree on whether a program is secure or not [44]). So even for veteran researchers in information flow security, understanding the subtleties in the syntax and semantics of each policy is difficult, evidenced by highly-cited papers that synthesize existing knowledge on declassification policy [43] and dynamic policy [44]. Arguably, it is currently impossible for a system developer/user to navigate in the jungle of unconnected policies (even for the ones in the same category) when a dynamic policy is needed [43,44].

In this chapter, we take a top-down approach and propose Dynamic Release, the first information flow policy that enables declassification, erasure, delegation and revocation at the same time. One important insight that we developed during the process is that erasure and revocation both strengthen an information flow policy, despite their very different syntax in existing work. However, an erasure policy by definition disallows the same information leaked in the past (i.e., before erasure) to be released in the future, while most revocation policies allow so. This motivates the introduction of two kinds of policies, which we call persistent and transient policies. The distinction can be interpreted as a type of information flow which is
permitted by some definitions but not by others, called facets [44].

Moreover, Dynamic Release is built on a novel formalization framework that is shown to subsume existing security conditions that are formalized in different ways (e.g., noninterference, bisimulation and epistemic [44]). More importantly, for the first time, the formalization framework allows us to make apple-to-apple comparison among existing policies, which are incompatible before (i.e., one cannot trivially convert one to another). Besides the distinction between persistent and transient policies mentioned earlier, we also notice that it is more challenging to define a transient policy (e.g., erasure), as it requires a definition of the precise knowledge gained from observing one output event, rather than the more standard cumulative knowledge that we see in existing persistent policies.

Finally, we built a new AnnTrace benchmark for testing and understanding variants of dynamic policies in general. The benchmark consists of examples with dynamic policies from existing papers, as well as new subtle examples that we created in the process of understanding dynamic policies. We implemented our policy and existing policies, and found that Dynamic

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\hline
2 submit := bid; & 2 copy := credit_card; & 2 //bk → Alice & \\
3 output(submit, S); & 3 output(copy, M); & 3 notes := half(book); & \\
4 // bid : P & 4 // credit_card : ⊤ & 4 output(notes, Alice); & \\
5 output(submit, P); & 5 copy := 0; & 5 //bk ⊏ Alice & \\
& 6 output(copy, M); & 6 output(notes, Alice); & \\
\hline
\end{tabular}
\caption{Examples of Dynamic Policies.}
\end{table}
Release is the only one that is both applicable and correct on all examples.

To summarize, in this chapter, we make the following contributions:

1. We present a language abstraction with concise yet expressive security specification (Section 5.3) that allows us to specify various existing dynamic policies, including declassification, erasure, delegation and revocation.

2. We present a new policy Dynamic Release (Section 5.4). The new definition resolves a few subtle pitfalls that we found in existing definitions, and its security condition handles transient and persistent policies in a uniform way.

3. We generalize the novel formalization framework behind Dynamic Release and show that it, for the first time, allows us to compare and contrast various dynamic policies at the semantic level (Section 5.5). The comparison leads to new insights that were not obvious in the past, such as whether an existing policy is transient or persistent.

4. We build a new benchmark for testing and understanding dynamic policies, and implemented our policy and existing ones (Section 5.6). Evaluation on the benchmark suggests that Dynamic Release is the only one that is both applicable and correct on all examples.

5.2 Background and Overview

5.2.1 Security Levels

As standard in information flow security, we assume the existence of a set of security levels \( \mathbb{L} \), describing the intended confidentiality of information\(^1\). For generality, we do not assume that all levels form a Denning-style lattice. For instance, delegation and revocation typically use principals/roles (such as Alice, Bob) where the acts-for relation on principals can change at run time. For simplicity, we use the notation \( \ell \in \mathcal{L} \) if all levels form a lattice \( \mathcal{L} \), rather than

\(^1\)Since integrity is the dual of confidentiality, we will assume confidentiality hereafter.
Moreover, we use $P$ (public), $S$ (secret) to represent levels in a standard two-point lattice where $P \subseteq S$ but $S \nsubseteq P$.

### 5.2.2 Terminology

Some terms in dynamic policy are overloaded and used inconsistently in the literature. For instance, declassification is sometimes confused with dynamic policy [44]. To avoid confusion, we first define the basic terminology that we use in this chapter.

**Definition 16 (Dynamic (Information Flow) Policy)** An information flow policy is dynamic if it allows the sensitivity of information to change during one execution of a program.

As standard, we say that a change of sensitivity is downgrading (resp. upgrading) if it makes information less sensitive (resp. more sensitive).

Next, we use the examples in Figure 5.1 to introduce the major kinds of dynamic policies in the literature. For readability, we use informal security specification in comments for most examples in this chapter; a formal specification language is given in Section 5.3.

**Declassification** Given a Denning-style lattice $L$, declassification occurs when a piece of information has its sensitivity level $\ell_1$ downgraded to a lower sensitivity level $\ell_2$ (i.e., $\ell_2 \subseteq \ell_1$). Consider Figure 5.1-A which models an online bidding system. When bidders submit their bids to the system during the bidding phase, each bid is classified that no other bidders are allowed to learn the information. When the bidding ends, the bids are public to all bidders. In the secure program (i), the bid is only revealed to a public channel with level $P$ (Line 5) when bidding ends. However, the insecure program (ii) leaks the bid during the bidding phase (Line 3).

**Erasure** Given a Denning-style lattice $L$, information erasure occurs when a piece of information has its sensitivity level $\ell_1$ upgraded to a more restrictive sensitivity level, or an incomparable level $\ell_2$ (i.e., $\ell_2 \nsubseteq \ell_1$). Moreover, when information is erased to level $\top$, the
sensitive information must be removed from the system as if it was never inputted into the system. Figure 5.1-B is from a payment system. The user of the system gives her credit card information to the merchandiser (at level $M$) as payment for her purchase. When the transaction is done, the merchandiser is not allowed to retain/use the credit card information for any other purpose (i.e., its level changes to $\top$). The secure program (i) only uses the credit card information during the transaction (Line 3), and any related information is erased after the transaction (Line 5). The insecure program (ii), however, fails to protect the credit card information after the transaction (Line 6).

**Delegation and Revocation** Delegation and revocation are typically used together, in a principal/role-based system [101–103]. In this model, information is associated with principals/roles, and a dynamic policy is specified as changes (i.e., add or remove) to the “acts-for” relationship on principals/roles. Figure 5.1-C is from a book renting system, where its customers are allowed to read books during the renting period. In this example, Alice acts-for $bk$ ($bk \to Alice$) before line 3. Hence, she is allowed to take notes from the book. When the renting is over, the book is no longer accessible to Alice ($bk \not\to Alice$), but the notes remain accessible to Alice. The secure program (i) allows the customer to get their notes (Line 6) learned during the renting period. The insecure program (ii) fails to protect the book (Line 6) after the renting is over.

### 5.2.3 Overview

We use Figure 5.1 to highlight two major obstacles of understanding/applying various kinds of dynamic policies.

First, we note that a delegation/revocation policy (Example C) and an erasure policy (Example B) use different formats to model sensitivity change. A delegation/revocation policy attaches fixed security levels to data throughout program execution; policy change is modeled as changing the acts-for relation on roles. On the other hand, an erasure policy uses a fixed lattice
throughout program execution; policy change is modeled as \textit{mutable} security levels on data. These two examples are similar from policy change perspective, as they are both upgrading policies. But due to the different specification formats, their relation becomes obscure.

Second, we note that Example B.ii and C.i are semantically very similar: both examples first read data when the policy allows so, and then try to access the data again when the policy on data forbids so. However, B.ii is considered \textit{insecure} according to an erasure policy, while C.i is considered \textit{secure} according to a revocation policy. Even when we only consider policies of the same kind (e.g., delegation/revocation), such inconsistency in the security notion also exists, which is called \textit{facets of dynamic policies} [44].

Broberg et al. [44] have identified a few facets, but identifying other differences among existing policies is extremely difficult, as they are formalized in different nature (e.g., noninterference, bisimulation and epistemic). We can peek at the semantics-level differences based on a few examples, but an apple-to-apple comparison is still impossible at this point.

In this chapter, we take a top-down approach that rethinks dynamic policy from scratch. Instead of developing four kinds of policies seen in prior work, we observe that there are only \textit{two essential building blocks} of a dynamic policy: upgrading and downgrading. With an expressive specification language syntax (Section 5.3), we show that in terms of upgrading and downgrading sensitivity, declassification (resp. erasure) is the same as delegation (resp. revocation). In terms of the formal security condition of dynamic policy, we adopt the epistemic model [18] and develop a formalization framework that can be informally understood as the following security statement:

A program $c$ is secure iff for any event $t$ produced by $c$, the “knowledge” gained about secret by learning $t$ is bounded by what’s allowed by the policy at $t$.

We note that a key challenge of a proper security definition for the statement above is to properly define the “knowledge” of learning a \textit{single} event $t$. During the process of developing the formal definition, we discovered a new facet of upgrading policies; the difference is that
whether an upgrading policy automatically allows information leakage (after upgrading) when it has happened in the past. Consequently, we precisely define the “knowledge” of learning a single event and make semantics-level choices (called transient and persistent respectively) of the new facet explicit in Dynamic Release (Section 5.4).

To compare and contrast various dynamic policies (including Dynamic Release), we cast existing policies into the formalization framework behind Dynamic Release (Section 5.5). We find that the semantics of erasure and revocation are drastically different: erasure policy is transient by definition, and most revocation policies are persistent. The semantics-level difference sheds light on why Example B.ii and C.i have inconsistent security under erasure and revocation policies, even though they are similar programs.

5.3 Dynamic Policy Specification

We first present the syntax of an imperative language with its security specification. Based on that, we show that the policy specification is powerful enough to describe declassification, erasure, delegation and revocation policies. Finally, we define a few notations to be used in this chapter.

5.3.1 Language Syntax and Security Specification

In this chapter, we use a simple imperative language with expressive security specification, as shown in Figure 5.2. The language provides standard features such as variables, assignments, sequential composition, branches and loops. Other features are introduced for security:

- We explicitly model information release by a release command \texttt{output}(b, e); it reveals the value of expression \textit{e} to an information channel with security label \textit{b}.\footnote{In the literature, it is also common to model information release as updates to a memory portion visible to an attacker. This can be modeled explicitly as requiring an assignment \textit{x := e} where \textit{x} has label \textit{b} to emit a release command \texttt{output}(b, v).}
• We introduce distinguished security events $S$. An event $s \in S$ is similar to a Boolean; we distinguish $s$ and $x$ in the language syntax to ensure that security events can only be set and unset using distinguished commands $\text{EventOn}(s)$ and $\text{EventOff}(s)$, which set $s$ to true and false respectively. We assume that all security events are initialized with false.

**Sensitivity Levels**

For generality, we assume a predefined set $\mathbb{L}$ of all security levels, and use level set $L \subseteq \mathbb{L}$ to specify data sensitivity. Intuitively, a level set $L$ consists of a set of levels where the associated information can flow to. Hence, $L_1$ is less restrictive as $L_2$, written as $L_1 \sqsubseteq L_2$ iff $L_2 \subseteq L_1$, and $L_1 \sqsubseteq L_2$ iff $L_2 \subseteq L_1$.

Although the use of level set is somewhat non-standard, we note that it provides better generality compared with existing specifications, such as a level from a Denning-style lattice [1] or a role in a role-based model [101–103].

• Denning-style lattice: let $\mathcal{L}$ be a security lattice. We can define $\mathbb{L}$ and the level set that
represents $\ell \in \mathcal{L}$ as follows:

$$\mathbb{L} = \{\ell | \ell \in \mathcal{L}\}; \quad L_\ell \triangleq \{\ell' | \ell \subseteq \ell'\}$$  \hfill (5.3.1)

Consider a two-point lattice $\{P, S\}$ with $P \sqsubseteq S$. It can be written as the follows in our syntax:

$$\mathbb{L} \triangleq \{P, S\}; \quad L_S \triangleq \{S\}; \quad L_P \triangleq \{P, S\};$$

- Role-based model: let $\mathbb{P}$ be a set of principals/roles and actsfor be an acts-for relation on roles. We can define $\mathbb{L}$ and the level set that represents $P \in \mathbb{P}$ as follows:

$$\mathbb{L} = \mathcal{P}(\mathbb{P}); \quad L_P \triangleq \{P' | P' \text{ actsfor } P\}$$  \hfill (5.3.2)

Consider a model with two roles Alice and Bob with Alice actsfor Bob but not the other way around. It can be written as the follows in our syntax:

$$\mathbb{L} \triangleq \{\text{Alice, Bob}\}; \quad L_{\text{Alice}} \triangleq \{\text{Alice}\};$$

$$L_{\text{Bob}} \triangleq \{\text{Alice, Bob}\};$$

**Sensitivity Mutation**

The core of specifying a dynamic policy is to define how data sensitivity changes at run time. This is specified by a security label $b$.

A label can simply be a level set $L$, which represents immutable sensitivity throughout program execution. In general, a label has the form of $\text{cnd?}b_1 \circ b_2$ where:

- A trigger condition $\text{cnd}$ specifies when the sensitivity changes. There are two basic kinds of trigger conditions: a security event $s$ and a (Boolean) program expression $e$. A more complicated condition can be constructed with logical operations on $s$ and $e$. We assume
that a type system checks that whenever \( cnd \) is an expression \( e \), \( e \) is of the Boolean type.

- The mutation direction \( \circ \) specifies how the information flow restriction changes. There are two one-time mutation directions: \( cnd?b_1 \to b_2 \) (resp. \( cnd?b_1 \leftarrow b_2 \)) allows a one-time sensitivity change from \( b_1 \) to \( b_2 \) (resp. \( b_2 \) to \( b_1 \)) the first time that \( cnd \) evaluates to \( \text{false} \) (resp. \( \text{true} \)). On the other hand, a two-way mutation \( cnd?b_1 \leftrightarrow b_2 \) allows arbitrary number of changes between \( b_1 \) and \( b_2 \) whenever the value of \( cnd \) flips.

**Policy Specification**

The information flow policy on a program is specified as a function from variables \( \text{Vars} \) to security labels \( \mathbb{B} \) and a policy type \( \diamond \). The policy type can either be transient, or persistent (formalized in Section 5.4).

### 5.3.2 Expressiveness

Despite the simplicity of our language syntax and security specification, we first show that all kinds of dynamic policies in Figure 5.1 can be concisely expressed. Then, we discuss how the specification covers the well-known what, who, where and when dimensions [43, 104] of dynamic policies.\(^3\) Finally, we show that the specification language is powerful enough to encode Flow Locks [105] and its successor Paralocks [106], a well-known meta policy language for building expressive information flow policies.

**Examples**

We first encode the examples in Figure 5.1.

**Declassification and Erasure** Both policies specify sensitivity changes as mutating security level of information from some level \( \ell_1 \) to \( \ell_2 \), where both \( \ell_1 \) and \( \ell_2 \) are drawn from a Denning-

---

\(^3\)The original definitions focus on declassification policy, but the dimensions are applicable for dynamic policies as well.
style lattice $\mathcal{L}$. Such a change can be specified as $L_{\ell_1} \rightarrow L_{\ell_2}$, where $L_{\ell_1}$ and $L_{\ell_2}$ are the level sets representing $\ell_1$ and $\ell_2$, as defined in Equation (5.3.1).

For example, the informal policy on credit_card in Figure 5.1-B can be precisely specified as $\text{erase}?\{\} \leftarrow \{\text{M}\} [\text{Tran}]$ (we will discuss why erasure is a transient policy in Section 5.4) with the security command $\text{EventOn(erase)}$ being inserted to Line 4 to trigger the mutation.

**Delegation and revocation**  Both policies specify sensitivity changes as modifying the acts-for relationship on principals, such as Alice and Bob. Such a change can be specified as the old and new sets of roles who acts-for the owner, say $P$, of information. That is, a change from from $\text{actsfor}_1$ to $\text{actsfor}_2$ can be specified as $L_1 \rightarrow L_2$, where $L_i \triangleq \{P' \in \mathbb{P} \mid P' \text{ actsfor}_i P\}$.

For example, the policy on book in Figure 5.1-C can be specified as $\text{revoke}?\{\} \leftarrow \{\text{Alice}\} [\text{Per}]$ (we will discuss why revocation is a persistent policy in Section 5.4) with a security command $\text{EventOn(revoke)}$ being inserted to Line 5 to trigger the mutation.  

**Dimensions of dynamic policy [43, 104]**

**What**  The what dimension regulates what information’s sensitivity is changed. Since the policy specification is defined at variable level, our language does not fully support partial release, which only releases a part of a secret (e.g., the parity of a secret) to a public domain. However, we note that the language still has some support of partial release. Consider the example in Figure 5.1-C.i. The policy allows partial value $\text{half(book)}$ to be accessible by Alice after Line 5, while the whole value of book is not. As shown in Section 5.3.2, the partial release of $\text{half(book)}$ in this example can be precisely expressed in our language. We leave the full support of partial release as future work.

Moreover, we emphasize that the policy specification regulates the sensitivity on the original

---

4We note that our encoding requires all changes to the acts-for relation to be anticipated, whereas a general delegation/revocation policy might also offer the flexibility of changing the acts-for relation dynamically.
value of the variable. For example, consider $\Gamma(h) = S, \Gamma(x) = s?S \rightarrow P$ for program:

$$x := h; \text{EventOn}(s); \text{output}(P, x);$$

The policy on $x$ states that its original value, rather than its value right before output (i.e., the value of $h$), is declassified to $P$. Hence, the program is insecure. Therefore, the specification language rules out laundering attacks [25, 43], which launders secrets not intended for declassification.

**Where** The *where* dimension regulates *level locality* (where information may flow to) and *code locality* (where physically in the code that information’s sensitivity changes). It is obvious that a label $cnd?b_1 \circ b_2$ declare where information may flow to after a policy change, and the security event $s$ with the security commands $\text{EventOn}(s)$ and $\text{EventOff}(s)$ specify the code locations where sensitivity changes.

**When** The *when* dimension is a temporal dimension, pertaining to when information’s sensitivity changes. This is specified by the trigger condition $cnd$. For example, a policy $(\text{paid}?P \leftarrow S)$ allows associated information (e.g., software key) to be released when payment has been received. This is an instance of “Relative” specification defined in [43].

**Who** The *who* dimension specifies a *principal/role*, who controls the change of sensitivity; one example is the Decentralized Label Model (DLM) [107], which explicitly defines ownership in security labels. While our specification language does not explicitly define ownership, we show next that it is expressive enough to encode Flow Locks [105] and Paralocks [106], which in turn are expressive enough to encode DLM [106]. Hence, the specification language also covers the who dimension to some extent.
Figure 5.3. An Example of Encoding Paralock for $A = \langle a, \{D\} \rangle$.

**Encoding Flow Locks [105]**

Both Flow Locks [105] and its successor Paralocks [106] introduce locks, denoted as $\sigma$, to construct dynamic policies. Let Locks be a set of locks, and $\mathbb{P}$ be a set of principals. A “flow lock” policy is specified with the following components:

- Flow locks in the form of $\Sigma \Rightarrow P$ where $\Sigma \subseteq \text{Locks}$ is the lock set for principal $P \in \mathbb{P}$.
- Distinguished commands $\text{open}(\sigma)$, $\text{close}(\sigma)$ that open and close the lock $\sigma \in \text{Locks}$.

To simplify notation, we use $\Gamma(x, P) = \Sigma$ to denote the fact that $\{\Sigma \Rightarrow P\}$ is part of the “flow locks” of $x$. Paralocks security is formalized as an extension of Gradual Release [18]. In particular, paralock security is defined based on sub-security condition for each hypothetical attacker $A = (P_A, \Sigma_A)$ where $P_A \in \mathbb{P}$ and $\Sigma_A \subseteq \text{Locks}$:

- A variable is considered “public” for attacker $A$ when $\Gamma(x, P_A) = \Sigma_x \subseteq \Sigma_A$; otherwise, it is considered “secret” for attacker $A$.
- A “release event”, in gradual release sense, is defined as a period of program execution when the set of opened locks $\Sigma_{\text{open}}$ satisfies $\Sigma_{\text{open}} \not\subseteq \Sigma_A$, as any such lock state might allow some “secret” to $A$ to be released [106].

Consider the example in Figure 5.3 and a hypothetical attacker $A = \langle a, \{D\} \rangle$. Since $\Gamma(x, a) = \{D, N\} \not\subseteq \{D\}$, variable $x$ is considered “secret” for $A$. Moreover, the first
assignment \( y := x \) is not under a release event since the set of opened locks, \( \{ D \} \), satisfies \( \{ D \} \subseteq \{ D \} = \Sigma_A \). On the other hand, the last assignment \( z := y \) is under a release event since \( \Sigma_{\text{open}} = \{ D, N \} \not\subseteq \{ D \} = \Sigma_A \). Therefore, we need to ensure that the first assignment, which is not under a releasing event, does not reveal the value of \( x \).

For each concrete attacker \( A = (P_A, \Sigma_A) \), we can encode Paralocks security as follows:

- We define a security event \( s_\sigma \) for each lock \( \sigma \in \Sigma \) and the lock command \( \text{open}(\sigma) \) (resp. \( \text{close}(\sigma) \)) is converted to \( \text{EventOn}(s_\sigma) \) (resp. \( \text{EventOff}(s_\sigma) \)).

- Let \( \Gamma'(x) = \{ P_A \} \) when \( \Gamma(x, P_A) = \Sigma_x \subseteq \Sigma_A \); otherwise, \( \Gamma'(x) = \{} \) (i.e., secret for \( P_A \)).

- We define the dynamic policy of \( x \) as \( \text{cnd}?\{ P_A \} \rightleftarrows \Gamma'(x) \) where \( \text{cnd} \triangleq \bigvee_{\sigma \in \Sigma_A} s_\sigma \).

Note that by definition, information \( x \) is under a release event (i.e., has level set \( \{ P_A \} \)) whenever at least one lock not in \( \Sigma_A \) is currently open, which implies that \( \Sigma_{\text{open}} \not\subseteq \Sigma_A \), as defined in Paralock.

Consider the transformed code in Figure 5.3 for an attacker \( A = \langle a, \{ D \} \rangle \). We note that under the encoding, the first assignment \( y := x \) is not under declassification of \( x \) since \( s_N \) is closed and the effective level set of \( x \) is \( \{} \), which prohibits attacker \( A \) from learning its value. This is consistent with Paralock semantics, where the first assignment is not under a release event. On the other hand, the second assignment \( z := y \) is under a release event since the effective level set of both \( x \) and \( y \) are \( \{ a \} \) (note that we have \( s_N \) as the security event is turned on), which allows attacker \( A \) to learn their values. This is also consistent with Paralock semantics, where the second assignment is under a release event.

Hence, we can encode Paralocks by explicitly checking that for every hypothetical attacker, the corresponding transformed program is secure.
5.3.3 Interpretation of Security Specification

Intuitively, the security specification in Figure 5.2 specifies at each program execution point, what is the sensitivity of the associated information. We formalize this as an interpretation function of the label, denoted as $[b]_\tau$, which takes in a label $b$ and a trace $\tau$, and returns a level set $L$ as information flow restrictions at the end of $\tau$.

**Execution trace**  As standard, we model program state, called memory $m$, as a mapping from program variables and security events to their values. The small-step semantics of the source language is mostly standard (hence omitted), with exception of the output and security event commands:

- $\langle e, m \rangle \Downarrow v \quad \text{S-OUTPUT}$
- $\langle \text{output}(b, e), m \rangle \xrightarrow{(b,v)} \langle \text{skip}, m \rangle$
- $\langle \text{EventOn}(s), m \rangle \rightarrow \langle \text{skip}, m\{s \rightarrow \text{true}\} \rangle \quad \text{S-SET}$
- $\langle \text{EventOff}(s), m \rangle \rightarrow \langle \text{skip}, m\{s \rightarrow \text{false}\} \rangle \quad \text{S-UNSET}$

The semantics records all output events, in the form of $\langle b, v \rangle$, during program execution, as these are the only information release events during program execution. Moreover, the distinguished security events $s$ are treated as boolean variables, which can only be set/unset by the security event commands.

Based on the small-step semantics, executing a program $c$ under initial memory $m$ produces an execution trace $\tau$ with potentially empty output events:

$$\langle c, m \rangle \xrightarrow{b_1,v_1} \langle c_1, m_1 \rangle \cdots \xrightarrow{b_n,v_n} \langle c_n, m_n \rangle.$$

We use $\tau[i]$ to denote the configuration (i.e., a pair of program and memory) after the $i$-th
\[ [L]_{\tau} = L \]

\[ [\text{cnd}?b_1 \rightarrow b_2]_{\tau} = \begin{cases} [b_1]_{\tau}, & \text{first(cnd, } \tau, \text{false)} = -1 \\ [b_2]_{\tau[i+1]}, & i = \text{first(cnd, } \tau, \text{false)} \geq 0 \end{cases} \]

\[ [\text{cnd}?b_1 \leftarrow b_2]_{\tau} = \begin{cases} [b_2]_{\tau}, & \text{first(cnd, } \tau, \text{true)} = -1 \\ [b_1]_{\tau[i+1]}, & i = \text{first(cnd, } \tau, \text{true)} \geq 0 \end{cases} \]

\[ [\text{cnd}?b_1 \Rightarrow b_2]_{\tau} = \begin{cases} [b_1]_{\tau[i+1]}, & i = \text{last(cnd, } \tau, \text{false)} \neq ||\tau|| \\ [b_2]_{\tau[i+1]}, & i = \text{last(cnd, } \tau, \text{true)} \neq ||\tau|| \end{cases} \]

where \( \text{first(cnd, } \tau, \text{bl)} \) returns the first index of \( \tau \) such that \( \text{cnd} \) evaluates to \( \text{bl} \), or \(-1\) if such an index does not exist; \( \text{last(cnd, } \tau, \text{bl)} \) returns the last index of \( \tau \) such that \( \text{cnd} \) evaluates to \( \text{bl} \), or \(-1\) if such an index does not exist.

**Figure 5.4. Interpretation of Security Labels**

evaluation step in the \( \tau \), and \( ||\tau|| \) to denote the number of evaluation steps in the trace. For example, \( \tau[0] \) is always the initial state of the execution, \( \tau[||\tau||] \) is the ending state of a terminating trace \( \tau \). We use \( \tau[i] \) (resp. \( \tau[i:] \)) to denote a prefix (resp. postfix) subtrace of \( \tau \) from the initial state up to (starting from) the \( i \)-th evaluation step. We use \( \tau[i:j] \) to denote the subtrace of \( \tau \) between \( i \)-th and \( j \)-th (inclusive) evaluation steps. Finally, we write \( \tau_1 \preceq \tau_2 \) when \( \tau_1 \) is a prefix of \( \tau_2 \).

**Interpretation of labels** We formalize the label semantics \([b]_{\tau}\) in Figure 5.4. \([b]_{\tau}\) returns a level set \( L \) that precisely specifies where the information with policy \( b \) can flow to at the end of trace \( \tau \). For a (static) level set \( L \), its interpretation is simply \( L \) regardless of \( \tau \).

For more complicated labels, the semantics also considers the temporal aspect of label changes. For example, a one-time mutation label \( \text{cnd}?b_1 \rightarrow b_2 \) allows a one-time sensitivity change from \( b_1 \) to \( b_2 \) when the first time that \( \text{cnd} \) evaluates to \( \text{false} \). Hence, let \( i \) be the first index of \( \tau \) such that \( \text{cnd} \) evaluates to \( \text{false} \). Then, \([\text{cnd}?b_1 \rightarrow b_2]_{\tau}\) reduces to \([b_1]_{\tau}\) when no such \( i \) exists (i.e., \( \text{cnd} \) always evaluates to \( \text{true} \) in \( \tau \)), and it reduces to \([b_2]_{\tau[i+1]}\) otherwise. Note that in the latter case, it reduces to \([b_2]_{\tau[i]}\) rather than \([b_2]_{\tau}\) to properly handle nested conditions: any nested condition in \( b_2 \) can only be evaluated after \( \text{cnd} \) becomes \( \text{false} \). The dual with \( \leftarrow \) is defined in a similar way. Note that \( \text{cnd}?b_1 \rightarrow b_2 \) and \( \neg \text{cnd}?b_2 \leftarrow b_1 \) are semantically the same;
we introduce both for convenience.

Finally, the bi-directional label (with ⇆) is interpreted purely based on the last configuration of $\tau$: let $i$ be the last index in $\tau$ such that $\text{cnd}$ evaluates to false. Then, $i \neq ||\tau||$ implies that $\text{cnd}$ evaluates to true at the end of $\tau$; hence, the label reduces to $b_1$. Note that $b_1$ is evaluated under $\tau^{[i+1:]}$ in this case to properly handle (potentially) nested conditions in $b_1$: any nested condition in $b_1$ can only be evaluated after $\text{cnd}$ becomes true.

Moreover, we can derive a dynamic specification for each execution point $i$, written as $\gamma_i$, such that

$$\forall x. \gamma_i(x) = \left[\Gamma(x)\right]_{\tau|_i}$$

Additionally, we overload $\gamma_i$ to track the dynamic interpretation of a label $b$ for each execution point $i$:

$$\forall b. \gamma_i(b) = \left[b\right]_{\tau|_i}$$

To simplify notation, we write

$$\langle c_0, m_0 \rangle \rightarrow \tilde{t}$$

if the execution $\langle c_0, m_0 \rangle$ terminates with an extended output sequence $\tilde{t}$, which consists of extended output events $t \triangleq \langle b, v, \gamma \rangle$, where $b, v$ are the output events on $\tau$, and $\gamma$ is the dynamic specification at the corresponding execution point. We use $t.b$, $t.v$ and $t.\gamma$ to refer to each component in the extended output event. We use the same index notation as in trace, where $\tilde{t}[i]$ returns the $i$-th output event, and $\tilde{t}[:i]$ returns the prefix output sequence up to (included) the $i$-th output. $\tilde{t}[:0]$ returns an empty sequence.

---

5In this chapter, we only consider output sequences $\tilde{t}$ produced by $\langle c_0, m_0 \rangle \rightarrow \tilde{t}$. Hence, only the terminating executions are considered, making our knowledge and security definitions in Section 5.4 termination-insensitive. Termination sensitivity is an orthogonal issue to the scope of this dissertation: dynamic policy.
5.4 Dynamic Release

In this section, we define Dynamic Release, an end-to-end information flow policy that allows information flow restrictions to downgrade and upgrade in arbitrary ways.

5.4.1 Semantics Notations

Memory Closure For various reasons, we need to define a set of initial memories that are indistinguishable from some memory $m$. Given a set of variables $X$, we define the memory closure of $m$ to be a set of memories who agree on the value of each variable $x \in X$:

**Definition 17 (Memory Closure)** *Given a memory $m$ and a set of variables $X$, the memory closure of $m$ on $X$ is:*

$$ [m]_X \triangleq \{ m' \mid \forall x \in X. m(x) = m'(x) \} $$

For simplicity, we use the following short-hands:

$$ [m]_{L,\gamma} \triangleq [m]_{\{ x \mid \gamma(x) \leq L \}} $$

$$ [m]_{\neq b} \triangleq [m]_{\{ x \mid \Gamma(x) \neq b \}} $$

where $[m]_{L,\gamma}$ is the memory closure on all variables whose sensitivity level is less or equally restrictive than level $L$ according to $\gamma$, and $[m]_{\neq b}$ is the memory closure on variables whose security policy is not $b$: a set of memories whose values only differ on variables with policy $b$.

Trace filter For various reasons, we need a filter on output traces to focus on relevant subtraces (e.g., to filter out outputs that are not visible to an attacker). Each trace filter can be defined as a Boolean function on $\langle b, v, \gamma \rangle$. With a filter function $f$ (that returns `false` for irrelevant outputs), we define the projection of outputs as follows:
Definition 18 (Projection of Trace)

\[
[\vec{t}]_f \triangleq \{ \langle b, v, \gamma \rangle \in \vec{t} \mid f(b, v, \gamma) \}
\]

We use the following short-hand for a commonly used filter, \( L \)-projection filter, where the resulting trace consists of outputs to channels that are observable to an attacker at level \( L \) :

\[
[\vec{t}]_L \triangleq [\vec{t}]_{\lambda b, v, \gamma. \gamma(b) \subseteq L}
\]

5.4.2 Key Factors of Formalizing a Dynamic Policy

Before formalizing Dynamic Release, we first introduce knowledge-based security (i.e., epistemic security) [18], which is widely used in the context of dynamic policy. Our formalization is built on the following informal security statement, which is motivated by [32]:

A program \( c \) is secure iff for any event \( t \) produced by \( c \), the “knowledge” gained about secret by observing \( t \) is bounded by what’s allowed by the policy at \( t \).

We first introduce a few building blocks to formalize “knowledge” and “allowance” (i.e., the allowed leakage).

Indistinguishability

A key component of information flow security is to define trace indistinguishability: whether two program execution traces are distinguishable to an attacker or not. Given an attacker at level set \( L \), each release event \( \langle b, v, \gamma \rangle \) is observable iff \( \gamma(b) \subseteq L \) by the attack model. Hence, as standard, we define an indistinguishability relation, written as \( \sim_L \), on traces as

\[
\sim_L \triangleq \{(\vec{t}_1, \vec{t}_2) \mid [\vec{t}_1]_L \leq [\vec{t}_2]_L \}
\]
Note that an attacker cannot rule out any execution whose prefix matches $t_1$. Hence, the prefix relation is used instead of identity.

**Knowledge gained from observation**

Following the original definition of knowledge in [18], we define the knowledge gained by an attacker at level set $L$ via observing a trace $\vec{t}$ produced by a program $c$ as:

$$k_1(c, \vec{t}, L) \triangleq \{ m \mid \langle c, m \rangle \rightarrow \vec{t} \land \vec{t} \sim_L \vec{t} \}$$

(5.4.1)

Intuitively, it states that if one initial memory $m$ produces a trace that is indistinguishable from $\vec{t}$, then the attacker cannot rule out $m$ as one possible initial memory. Note that by definition, the smaller the knowledge set is, the more information (knowledge) is revealed to the attacker.

Recall that by definition, $\langle c, m \rangle \rightarrow \vec{t}$ only considers terminating program executions. Hence, the knowledge definition above is the termination-insensitive version of knowledge defined in [18]. As a consequence, the security semantics that we define in this chapter is also termination-insensitive.

**Policy Allowance**

To formalize security, we also need to define for each output event $t$ on a trace, what is the allowed leakage to an attacker at a level set $L$. As knowledge, policy allowance, written as $A(m, \vec{t}, b, L)$, is defined as a set of memories that (1) only differs from the actual initial memory $m$ in variables whose label is $b$, and (2) should remain indistinguishable to $m$ for an attacker at $L$ who observes an output sequence $\vec{t}$ according to the dynamic policy.

Consider a dynamic label $b \in B$, memory $m$ and output sequence $\vec{t}$ of interest, as well as an

\footnote{We slightly modified the original definition to exclude “initial knowledge”, the attacker’s knowledge before executing the program.}
attacker at level \( L \), one possible policy allowance can be:

\[
\mathcal{A}(m, \vec{t}, b, L) \equiv [m] \neq b
\]

Intuitively, it specifies the initial knowledge of an attacker at level set \( L \): the attacker cannot distinguish any value difference among variables with the dynamic label \( b \). Thus, any variable with the label \( b \) is initially indistinguishable to the attacker. Eventually, Dynamic Release checks that for each label \( b \in B \), gained knowledge is bounded by the allowance with respect to \( b \). Hence, the security of each variable is checked.

### 5.4.3 Challenges of Formalizing a General Dynamic Policy

We next show that it is a challenging task to formalize the security of a general-purpose dynamic policy that allows downgrading and upgrading to occur in arbitrary ways.

**Challenge 1: Permitting both increasing and decreasing knowledge**

Allowing both downgrading and upgrading in arbitrary ways means that our general policy must permit reasoning about both increasing knowledge (as in declassification) and decreasing knowledge (as in erasure). While Equation 5.4.1 and its variants are widely used to formalize declassification policy [18, 106], they cannot reason about increasing knowledge. The reason is that, it is easy to check that for any \( c, \vec{t} \preceq \vec{t}', L \), the knowledge set decreases:

\[
\vec{t} \preceq \vec{t}' \Rightarrow k_1(c, \vec{t}, L) \supseteq k_1(c, \vec{t}', L)
\]

As other variants, the knowledge set \( k_1 \) is monotonically decreasing (hence, the knowledge that it represents is increasing by definition) as more events on the same execution are revealed to an attacker [18, 32, 56].

However, we need to reason about decreasing knowledge for an erasure policy. Consider
the example in Figure 5.1-B, where the value of credit card is revealed by the first output at Line 3. Given any program execution \(\langle c, m \rangle \rightarrow \vec{t}\), we have \(k_1(c, \vec{t}[i], M) = \{m\}\) for all \(i \geq 1\).

However, as the sensitivity of \textit{credit card} upgrades from \(M\) to \(\top\) when \(i = 2\) (i.e., the second output), the secure program (i) can be incorrectly rejected: \(k_1(c, \vec{t}[2], M) = \{m\}\) means that the value of \textit{credit card} is known to the attacker, which violates the erasure policy at that point.

\textbf{Observation 1.} Equation 5.4.1 is not suitable for an upgrading policy, since it fails to reason about decreasing knowledge. The issue is that knowledge gained from \(\vec{t}\) is defined as the full knowledge gained from observing \textit{all} outputs on \(\vec{t}\). Return to the secure program in Figure 5.1-B.i. We note that the first and second outputs together reveal the value of \textit{credit card}, but the second event alone reveals no information, as it always outputs 0. Hence, we need to precisely define the \textit{exact knowledge} gained from learning \textit{each} output to permit both increasing and decreasing knowledge.

\textbf{Challenge 2: Indistinguishability \(\sim_L\) is inadequate for a general dynamic policy}

As shown earlier, indistinguishability \(\sim_L\) is an important component of a knowledge definition; intuitively, by observing an execution \(\langle c, m \rangle \rightarrow \vec{t}\), an attacker at level set \(L\) can rule out any initial memory \(m'\) where \(m \not\sim_L m'\) (i.e., \(m' \notin k_1(c, \vec{t}, L)\)). However, the naive definition of \(\sim_L\) might be inadequate for declassified outputs. Consider the following secure program, where \(x\) is first downgraded to \(P\) and then upgraded to \(S\).

```java
1 // x : P
2 if (x>0) output(P,1);
3 output(P,1)
4 // x : S
5 output(P,2)
```

Note that the program is secure since the only output when \(x\) is secret reveals a constant value. Assume that the initial value of \(x\) is either 0 or 1. Then, there are two possible executions.
of the program with $\gamma_1(x) = P$ and $\gamma_2(x) = S$:

$$\langle c, m_1 \rangle \leftrightarrow \langle P, 1, \gamma_1 \rangle \cdot \langle P, 2, \gamma_2 \rangle$$

$$\langle c, m_2 \rangle \leftrightarrow \langle P, 1, \gamma_1 \rangle \cdot \langle P, 1, \gamma_1 \rangle \cdot \langle P, 2, \gamma_2 \rangle$$

The issue is in the first execution. By observing the first output, an attacker at $P$ cannot tell if the execution starts from $m_1$ or $m_2$, as both of them first output 1. However, the attacker can rule out $m_2$ by observing the second output with the value of 2. Note that the change of knowledge (from $\{m_1, m_2\}$ to $\{m_1\}$) violates the dynamic policy governing the second output: the policy on $x$ is $S$, which prohibits the learning of the initial value of $x$.

**Observation 2.** The inadequacy of relation $\sim_L$ roots from the fact that, due to downgrading, the public outputs of different executions might have various lengths. Therefore, outputs at the same index but produced by different executions might be incomparable. To resolve the issue, we observe that any information release (of $x$) when $x$ is $P$ is ineffective, in the sense that the restriction on $x$ is not in effect. In the example above, the outputs with value 1 are all ineffective, as $x$ is public when the outputs at lines 2 and 3 are produced. This observation motivates the secret projection filter, which finds out the effective outputs for a given secret.

**Definition 19 (Secret Projection of Trace)** Given a policy $b$ and an attacker at level $L$, a secret projection of trace is a subtrace where information with policy $b$ cannot flow to $L$ and the output channel is visible to $L$:  

$$[\tilde{t}]_{b,L} \overset{\Delta}{=} [\tilde{t}]_{\lambda b', n, \gamma, \gamma(b) \not\subseteq L \land \gamma(b') \subseteq L}$$

Return to the example above, the effective subtraces starting from $m_1$ and $m_2$ are both $\langle P, 2, \gamma_2(x) = S \rangle$, which remains indistinguishable to an attacker at level $P$. 

110
Challenge 3: Effectiveness is also inadequate

With Observation 2, it might be attempting to define indistinguishability based on $\tilde{t}_{b,L}$, rather than $t_{L}$. However, doing so is problematic as shown by the following program.

```plaintext
1 // x : S
2 if (x>0) output (P, 1);
3 // x : P
4 if (x<=0) output (P, 1);
```

With two initial memories $m_1(x) = 0, m_2(x) = 1$, we have

$$\langle c, m_1 \rangle \rightarrow \langle P, 1, \gamma_1(x) = P \rangle$$

$$\langle c, m_2 \rangle \rightarrow \langle P, 1, \gamma_2(x) = S \rangle$$

Note that only the value of $x$ is revealed on the public channel. Hence, the program is secure as it always outputs 1. However, the effective subtrace starting from $m_1$ is $\emptyset$ and that starting from $m_2$ is $\langle P, 1, \gamma_2(x) = S \rangle$, suggesting that the program is insecure: the value of $x$ is revealed by the first output from $m_2$, while the policy at that point (S) disallows so.

Observation 3. We note that both indistiguability and effectiveness are important building blocks of a general-purpose dynamic policy. However, the challenge is how to combine them in a meaningful way. We will build our security definition on both concepts and justify why the new definition is meaningful in Section 5.4.4.

Challenge 4: Transient vs. Persistent Policy

So far, the policy allowance $A(m, \tilde{t}, b, L)$ ignores what information has been leaked in the past. However, in the persistent case such as Figure 5.1-C, the learned information (note) remains accessible even after the policy on book upgrades. In general, we define transient and persistent policy as:
Definition 20 (Transient and Persistent Policy) A dynamic security policy is persistent if it always allows to reveal information that has been revealed in the past. Otherwise, the policy is transient.

Observation 4. Both transient and persistent policy have real-world application scenarios. Hence, a general-purpose dynamic policy should support both kinds of policies, in a unified way.

5.4.4 Dynamic Release

We have introduced all ingredients to formalize Dynamic Release, a novel end-to-end, general-purpose dynamic policy.

To tackle the challenges above, we first formalize the attacker’s knowledge gained by observing the last event $t'$ on a trace $\vec{t} \cdot t'$. Note that simply computing the knowledge difference between observing $\vec{t} \cdot t'$ and observing $\vec{t}$ does not work. Consider the example in Figure 5.1-B.ii. Given any program execution $\langle c, m \rangle \rightarrow \vec{t}$, we have $k_1(c, \vec{t}[i], M) = \{m\}$ for all $i \geq 1$. Hence, the difference between the knowledge gained with or without the output at Line 6 is $\emptyset$, suggesting that no knowledge is gained by observing the output at Line 6 alone, which is incorrect as it reveals the credit card number.

Instead, we take inspiration from probabilities to formalize the attacker’s knowledge gained by observing a single event on a trace. Consider a program $c$ that produces the following sequences of numbers give the corresponding inputs:

input 1: $s_1 = (1 \cdot 1 \cdot 3)$

input 2: $s_2 = (2 \cdot 2 \cdot 3)$

input 3: $s_3 = (1 \cdot 1 \cdot 3)$

input 4: $s_4 = (2 \cdot 2 \cdot 2)$
Consider the following question: what is the probability that the program generates a sequence where the last number is identical to the last number of \( s_1 \)? Obviously, besides \( s_1 \), we also need to consider sequences \( s_2 \) and \( s_3 \) since albeit a different sequence, \( s_2 \) is consistent with \( s_1 \) in the sense that the last output is 3, and \( s_3 \) is indistinguishable (i.e., identical) to \( s_1 \). More precisely, we can compute the probability as follows:

\[
\sum_{s \in \text{consist}(s_1)} P(s)
\]

where the consistent set \( \text{consist}(s_1) \) is the set of sequences that produce the same last number as \( s_1 \), i.e., \( \{(1 \cdot 1 \cdot 3), (2 \cdot 2 \cdot 3)\} \). Assuming a uniform distribution on program inputs, we have that the probability is \( P(1 \cdot 1 \cdot 3) + P(2 \cdot 2 \cdot 3) = (0.25 + 0.25) + 0.25 = 0.75 \). Note that the indistinguishable sequences \( s_1 \) and \( s_3 \) are implicitly accounted for in \( P(1 \cdot 1 \cdot 3) \).

To compute the knowledge associated with the last event on a trace \( \vec{t} \), we first use effectiveness to identify consistent traces whose last event on the effective subset is the same:

**Definition 21 (Consistency Relation)** Two output sequences \( \vec{t}_1 \) and \( \vec{t}_2 \) are consistent w.r.t. a policy \( b \) and an attack level \( L \), written as \( \vec{t}_1 \equiv_{b,L} \vec{t}_2 \) if

\[
n = \| [\vec{t}_1]_{b,L} \| = \| [\vec{t}_2]_{b,L} \| \land [\vec{t}_1]_{b,L}^{[n]} = [\vec{t}_2]_{b,L}^{[n]}
\]

Note that despite the extra complicity due to trace projection, the consistency relation is similar to the consistent set \( \text{consist}(s_1) \) in the probability computation example. Next, we define the precise knowledge gained from the last event of \( \vec{t} \) based on both the consistency relation and knowledge. Note that since knowledge is a set of memories, rather than a number, the summation in the probability case is replaced by a set union. Similar to the probability of observing each sequence, the knowledge \( k_1 \) also implicitly accounts for all indistinguishable traces (Equation 5.4.1).

**Definition 22 (Attacker’s Knowledge Gained from the Last Event)** For an attacker at level
set $L$, the attacker’s knowledge w.r.t. information with policy $b$, after observing the last event of an output sequence $\vec{t}$ of program $c$, is the set of all initial memories that produce an output sequence that is indistinguishable to some consistent counterpart of $\vec{t}$:

$$k_2(c, \vec{t}, L, b) = \bigcup_{\exists m', j. \langle c, m' \rangle \rightarrow \vec{t} \land \vec{t}^{[j]} \equiv b, L} k_1(c, \vec{t}^{[j]}, L)$$

To see how Definition 22 tackles Challenges 2 and 3, we revisit the code example under each challenge.

- **Challenge 2**: Recall that with $m_1(x) = 0$, $m_2(x) = 1$, $\gamma_1(x) = P$ and $\gamma_2(x) = S$, there are two execution traces

  $\langle c, m_1 \rangle \leftrightarrow \langle P, 1, \gamma_1 \rangle \cdot \langle P, 2, \gamma_2 \rangle$

  $\langle c, m_2 \rangle \leftrightarrow \langle P, 1, \gamma_1 \rangle \cdot \langle P, 1, \gamma_1 \rangle \cdot \langle P, 2, \gamma_2 \rangle$

  It is easy to check that the two output sequences are consistent w.r.t. the label of $x$ and an attacker at $P$ according to Definition 21 since all outputs are ineffective. Hence, in both traces, the knowledge gained from the last output is $\{m_0, m_1\}$, due to the big union in $k_2$. Hence, we correctly conclude that no information is leaked by the last output in both traces.

- **Challenge 3**: Recall that with $m_1(x) = 0$, $m_2(x) = 1$, there are two execution traces

  $\langle c, m_1 \rangle \leftrightarrow \langle P, 1, \gamma_1(x) = P \rangle$

  $\langle c, m_2 \rangle \leftrightarrow \langle P, 1, \gamma_2(x) = S \rangle$

  While the two traces are not consistent with each other, we know that $k_1(c, \langle P, 1, \gamma_2(x) = S \rangle, P) = \{\langle P, 1, \gamma_1(x) = P \rangle, \langle P, 1, \gamma_2(x) = S \rangle\}$ since the two traces satisfy $\sim_p$. Hence, the knowledge gained from the last event is $\{m_0, m_1\}$, and we correctly conclude that no
information is leaked by the last output.

To tackle Challenge 4, we observe that a persistent policy allows information leaked in the past to be released again, while a transient policy disallows so. This is made precise by the following refinement of policy allowance:

\[
\mathcal{A}(m, \vec{t}, b, L) \triangleq \begin{cases} 
[m]_{\neq b}, & b \text{ is transient} \\
[m]_{\neq b} \cap k_1(c, \vec{t}[:\|t\| - 1], L), & b \text{ is persistent}
\end{cases}
\] (5.4.2)

where \(k_1(c, \vec{t}[:\|t\| - 1], L)\) is the knowledge from every output event in \(\vec{t}\) except the last one. Note that since the knowledge here represents the cumulative knowledge gained from observing all events, we use the standard knowledge \(k_1\) instead of the knowledge gained from the last event \(k_2\) here.

Putting everything together, we have Dynamic Release security, where for any output of the program, the attacker’s knowledge gained from observing the output is always bounded by the policy allowance at that output point.

**Definition 23 (Dynamic Release)**

\[
\forall m, L \subseteq \mathcal{L}, b \in \mathcal{B}, \vec{t}, \langle c, m \rangle \leftrightarrow \vec{t} \implies \forall 1 \leq i \leq \|\vec{t}\|. \quad k_2(c, \vec{t}[i], L, b) \supseteq \begin{cases} 
[m]_{\neq b}, & \text{transient} \\
[m]_{\neq b} \cap k_1(c, \vec{t}[i-1], L), & \text{persistent}
\end{cases}
\]

### 5.5 Semantics Framework For Dynamic Policy

While various forms of formal policy semantics exist in the literature, different policies have very different nature of the security conditions (i.e., noninterference, bisimulation and epistemic [44]). In this section, we generalize the formalization of Dynamic Release (Definition 23)
<table>
<thead>
<tr>
<th>Event Type</th>
<th>Formula</th>
<th>(A(m, t, b, L)), (i = |t|)</th>
<th>(\equiv (t_1, t_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradual Release</td>
<td>([\vec{t}_1]_L \ll [\vec{t}_2]_L)</td>
<td>([m]_{L, \delta \gamma} \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{GR}}))</td>
<td>(=)</td>
</tr>
<tr>
<td>Tight Gradual Release</td>
<td>([\vec{t}_1]_L \ll [\vec{t}_2]_L)</td>
<td>([m]_{L, \delta \gamma} \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{GR}}))</td>
<td>(=)</td>
</tr>
<tr>
<td>NI According to Policy</td>
<td>(\exists R. \forall (i, j) \in R. \ [\vec{t}<em>1]</em>{L, \delta \gamma} \cong [\vec{t}<em>2]</em>{L, \delta \gamma})</td>
<td>([m]_{\delta \gamma})</td>
<td>(=)</td>
</tr>
<tr>
<td>Cryptographic Erasure</td>
<td>([\vec{t}_1]_L = [\vec{t}<em>2]</em>{\delta \gamma})</td>
<td>([m]_{L, \delta \gamma})</td>
<td>(=)</td>
</tr>
<tr>
<td>Forgetful Attacker</td>
<td>(\exists \vec{t}_2 \subset \vec{t}_1, \text{atk}(\vec{t}_1) = \text{atk}(\vec{t}_2))</td>
<td>([m]_{L, \delta \gamma} \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{FA}}))</td>
<td>(=)</td>
</tr>
<tr>
<td>Paralock</td>
<td>([\vec{t}_1]_A \ll [\vec{t}_2]_A)</td>
<td>([m]_A \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{FA}}), \vec{t}_i^{i+1} \subseteq \Sigma_A)</td>
<td>(=)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([m]_A \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{FA}}), \vec{t}_i^{i+1} \subseteq \Sigma_A)</td>
<td>(=)</td>
</tr>
<tr>
<td>Dynamic Release</td>
<td>([\vec{t}_1]_L \ll [\vec{t}_2]_L)</td>
<td>([m]<em>{\delta \gamma}, [m]</em>{\delta \gamma} \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{FA}}), b)</td>
<td>(=)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>([m]_{\delta \gamma} \cap K(c, \vec{t}<em>i^{i+1}, \sim</em>{\text{FA}}), b)</td>
<td>(=)</td>
</tr>
</tbody>
</table>

**Table 5.1.** Existing End-to-End Security Policies and Dynamic Release Written in the Formalization Framework.

by abstracting away its key building blocks. Then we convert various existing dynamic policies into the formalization framework and provide the first apple-to-apple comparison between those policies.

### 5.5.1 Formalization Framework for Dynamic Policies

We first abstract way a few building blocks of Definition 23. To define them more concretely, we consider an output sequence \(\vec{t}\) produced by \(\langle c, m \rangle\), i.e., \(\langle c, m \rangle \rightarrow \vec{t}\), as the context.

As already discussed in Section 5.4, the building blocks are:

- **Output Indistinguishability**, written as \(\sim\): two output sequences \(\vec{t}_1\) and \(\vec{t}_2\) satisfies \(\vec{t}_1 \sim \vec{t}_2\) when they are considered indistinguishable to the attacker.

- **Policy Allowance**, written as \(A\): a set of initial memory that should be indistinguishable to \(m\) according to the dynamic policy.

- **Consistency Relation**, written as \(\equiv\): when trying to precisely define the knowledge gained from each output event, two sequences are considered “consistent”, even if they are not identical (Definition 21).

With the abstracted parameters, we first generalize the knowledge definition of \(k_1\) (Equation 5.4.1) on an arbitrary relation \(\sim\) on output sequences:

\[
\sim (t_1, t_2)
\]
Definition 24 (Generalized Knowledge)

\[
\mathcal{K}(c, \vec{t}, \sim) \triangleq \{ m \mid \langle c, m \rangle \Rightarrow \vec{v} \land \vec{t} \sim \vec{v} \} \quad (5.5.1)
\]

Therefore, with abstract \( \sim \), \( A \) and \( \equiv \), we can generalize Definition 23 as the following framework:

Definition 25 (Formalization Framework) Given trace indistinguishability relation \( \sim \), consistency relation \( \equiv \) and policy allowance \( A \), a command \( c \) satisfies a dynamic policy iff the knowledge gained from observing any output does not exceed its corresponding policy allowance:

\[
\forall m, L \subseteq \mathbb{L}, b \in \mathbb{B}, \langle c, m \rangle \Rightarrow \vec{t} \implies \forall 1 \leq i \leq \| \vec{t} \|. \bigcup_{m', j. \langle c, m' \rangle \Rightarrow \vec{v} \mid \vec{t}^{[j]} \equiv \vec{v}^{[i]}} \mathcal{K}(c, \vec{v}^{[j]}, \sim) \supseteq A(m, \vec{v}^{[i]}, b, L)
\]

Let \( \sim_{\text{dr}} \triangleq \{ (\vec{t}_1, \vec{t}_2) \mid [\vec{t}_1]_L \leq [\vec{t}_2]_L \} \), \( A_{\text{dr}} \) be as defined in Equation (5.4.2), and \( \equiv_{\text{dr}} \) be as defined in Definition 22, it is easy to check that Definition 25 is instantiated to Definition 23.

Moreover, when \( \equiv \) is instantiated with an equality relation \( = \), a case that we have seen in all existing dynamic policies, the general framework can be simplified to the following form:

\[
\forall c, m, L \subseteq \mathbb{L}, b \in \mathbb{B}, \langle c, m \rangle \Rightarrow \vec{t} \implies \forall 1 \leq i \leq \| \vec{t} \|. \mathcal{K}(c, \vec{v}^{[i]}, \sim) \supseteq A(m, \vec{v}^{[i]}, b, L)
\]

We use this simpler form for existing dynamic policies where consistency is simply defined as equivalence.
5.5.2 Existing works in the formalization framework

We incorporate existing definitions into the formalization framework; the results are summarized in Table 5.1. We first highlight a few insights from Table 5.1. Then, for each work (except for Paralock due to space constraint), we sketch how to convert it into the formalization framework. The conversion of Paralock and the correctness proofs of all conversions are available in the Appendix B.

Insights from Table 5.1

To the best of our knowledge, this is the first work that enables apple-to-apple comparison between various dynamic policies. We highlight a few insights.

First, an erasure policy (e.g., According to Policy and Cryptographic Erasure) defines indistinguishability $\sim$ in a substantially more complicated way compared with others. The complexity suggests that formalizing an erasure policy is more involved compared with other dynamic policies.

Second, besides Dynamic Release, Gradual Release, Paralock and Forgetful Attacker also have $K(c, \vec{t}^{[i-1]}, \sim)$ as part of policy allowance. Recall that $K(c, \vec{t}^{[i-1]}, \sim)$ represents the past knowledge excluding the last output on $\vec{t}$. Hence, these policies are persistent policies. On the other hand, all other dynamic policies are transient policies.

Third, since an erasure policy by definition is transient, persistent policies such as Gradual Release and Paralock cannot be used as an erasure policy, such as the example in Figure 5.1-B.

Gradual Release

Gradual Release assumes a mapping $\Gamma$ from variables to levels in a Denning-style lattice. A release event is generated by a special command $x := \text{declassify}(e)$. Informally, a program is secure when illegal flow w.r.t. $\Gamma$ only occurs along with release events. Hence, at policy
syntax level, we encode a release event as

\[
\text{EventOn}(r); x := e; \text{output}(\Gamma(x), e); \text{EventOff}(r);
\]

where \( r \) is a distinguished event for release, and we set \( \forall x. \Gamma'(x) = r?\mathbb{L} \Rightarrow \Gamma(x) \) to state that any leakage of any variable is allowed when this is a release event. But otherwise, the information flow restriction of \( \Gamma \) is obeyed.

At semantics level, Gradual Release is formalized on the insight that “knowledge must remain constant between releases”:

**Definition 26 (Gradual Release [18])** A program \( c \) satisfies gradual release w.r.t. \( \Gamma \) if\(^7\)

\[
\forall c, m, L, i, \vec{t}. \langle c, m \rangle \leftrightarrow \vec{t} \implies \\
\forall i \text{ not release event. } k(c, m, \vec{t}[i], L, \Gamma) = k(c, m, \vec{t}[i-1], L, \Gamma)
\]

where \( k(c, m, \vec{t}, L, \Gamma) \triangleq \{ m' | m' \in \llbracket m \rrbracket_{L, \Gamma} \land \langle c, m \rangle \leftrightarrow \vec{t} \land \vec{t} \ll \vec{t}' \} \) (5.5.2)

While the original definition does not immediately fit our framework, we prove that they are equivalent by:

\[
\sim_{\text{GR}} \triangleq \{ (\vec{t}_1, \vec{t}_2) | [\vec{t}_1]_L \ll [\vec{t}_2]_L \} \equiv_{\text{GR}} = \\
\mathcal{A}_{\text{GR}} \triangleq \llbracket m \rrbracket_L, \vec{t}[\|\vec{t}\| - 1], \sim_{\text{GR}}
\]

Note that in our encoding, a release event sets security event \( r \) which sets all dynamic labels in the form of \( r?\mathbb{L} \Rightarrow \Gamma(x) \) to the least restrictive level set \( \mathbb{L} \). Hence, when there is a release event, the allowance check \( \mathcal{K}(\ldots) \supseteq \mathcal{A} \) trivially true, resembling Definition 26.

\(^7\)Note that \( \langle c, m \rangle \rightarrow \vec{t} \) only considers terminating program executions by definition. So we used the termination-insensitive version of Gradual Release.
Lemma 3  With \(\sim_\text{GR} \equiv \equiv_\text{GR} \) and \( A \triangleq A_\text{GR} \), Definition 25 is equivalent to Definition 26.

Tight Gradual Release

Tight Gradual Release [51, 52] is an extension of Gradual Release. Similar to Gradual Release, it assumes a base policy \( \Gamma \) and uses \( x := \text{declassify}(e) \) to declassify the value of \( e \). However, the encoding of declassification command is different for two reasons. First, we can only encode a subset of Tight Gradual Release where declassification command contains \( \text{declassify}(x) \), since our language does not fully support partial release (Section 5.3.2). Second, declassification in Tight Gradual Release is both precise (i.e., only variable \( x \) in \( \text{declassify}(x) \) is downgraded) and permanent (i.e., the sensitivity of \( x \) cannot upgrade after \( x \) is declassified). Hence, we encode \( x' := \text{declassify}(x) \) as

\[
\text{EventOn}(r_x); x' := x; \text{output}(\Gamma(x'), x);
\]

where \( r_x \) is a distinguished security event for releasing just \( x \), and we set \( \Gamma'(x) = r_x?L \leftarrow \Gamma(x) \) to state that \( x \) is declassified once \( r_x \) is set.

Tight Gradual Release uses the same knowledge definition from Gradual Release, except that its execution traces also dynamically track the set of declassified variables \( X \):

\[
\langle c, m, \emptyset \rangle \rightarrow^* \langle c', m', X \rangle
\]

Definition 27 (Tight Gradual Release) A program \( c \) satisfies tight gradual release if for any trace \( \vec{t} \), initial memory \( m \) and attacker at level \( L \), we have

\[
\forall i. 1 \leq i \leq |\vec{t}|. (\llbracket m \rrbracket_{L,L} \cap \llbracket m \rrbracket_{X_i}) \subseteq k(c, m, \vec{t}[i], L, \Gamma)
\]

where \( X_i \) is the set of declassified variables associated with the \( i \)-th output.
Due to the encoding of declassification commands, we know that for each output at index $i$ in $\vec{t}$ we have:

$$[m]_{L,\vec{t}[i],\gamma} = ([m]_{L,r} \cap [m]_{X,i})$$

Hence, we can rephrase Tight Gradual Release as follows:

$$\sim_{\text{TGR}} \triangleq \{(\vec{t}_1, \vec{t}_2) \mid \lfloor \vec{t}_1 \rfloor_L \preceq \lfloor \vec{t}_2 \rfloor_L\}$$

$$\equiv_{\text{TGR}} \triangleq \mathcal{A}_{\text{TGR}} \triangleq [m]_{L,\vec{t}[i],\gamma}$$

**Lemma 4** With $\sim \triangleq \sim_{\text{TGR}}$, $\equiv \equiv_{\text{TGR}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\text{TGR}}$, Definition 25 is equivalent to Definition 27.

**Observation:** Tight Gradual Release is more precise than Gradual Release since the policy precisely downgrades the sensitivity of $x$ but not any other variables, while as Gradual Release downgrades all variables under a release event.

Compared to Dynamic Release, the most important difference is that the consistency relation $\equiv$ is defined in completely different ways. As discussed in Section 5.4.3, it is important to define it properly for general dynamic policies. The other major difference is that the security semantics of Tight Gradual Release cannot model erasure policies. Consider the example in Figure 5.1-B.i with $m_1(\text{credit\_card}) = 0$, $m_2(\text{credit\_card}) = 1$ and attacker level $M$. Given a program execution $\langle c, m_1 \rangle \rightarrow \vec{t}$, we have $\mathcal{K}(c, \vec{t}[i], \sim_{\text{TGR}}) = \{m_1\}$ for all $i \geq 1$. However, $\text{credit\_card}$ is upgraded from $M$ to $\top$ when $i = 2$ (i.e., the second output), the secure program (i) is incorrectly rejected by Tight Gradual Release since $\mathcal{K}(c, \vec{t}[2], \sim_{\text{TGR}}) = \{m_1\} \not\supseteq \{m_1, m_2\} = [m_1]_{M,\vec{t}[2],\gamma}$.

**NI According to Policy**

Chong and Myers propose noninterference according to policy [26, 27] to integrate erasure and declassification policies. We use the formalization in the more recent paper [27] as the security definition.
This work uses compound labels, a similar security specification as ours: a label is either a simple level $\ell$ drawn from a Denning-style lattice, or in the form of $q_1 \rightarrow q_2$, where $q_1$ and $q_2$ are themselves compound labels. Hence, converting the specification to ours is straightforward.

Noninterference according to policy is defined for each variable in a two-run style. In particular, it requires that for any two program executions where the initial memories differ only in the value of the variable of interest, their traces are indistinguishable regarding a correspondence $R$:

**Definition 28 (Noninterference According To Policy [27])** A program $c$ is noninterference according to policy if for any variable $x$ (with policy $b$) we have:

$$\forall m_1, m_2, \ell, \vec{t}_1, \vec{t}_2. \forall y \neq x. m_1(y) = m_2(y) \land \langle c, m_1 \rangle \rightarrow \vec{t}_1 \land \langle c, m_2 \rangle \rightarrow \vec{t}_2 \Rightarrow \exists R. \left( \forall (i,j) \in R, \ell, \ell \notin [b]_{\tau_1[i]} \land \ell \notin [b]_{\tau_2[j]} \Rightarrow \tau_1[i] \approx_\ell \tau_2[j] \right)$$

where a correspondence $R$ between traces $\tau_1$ and $\tau_2$ is a subset of $\mathbb{N} \times \mathbb{N}$ such that:

1. (Completeness) either \{ $i$ | $(i,j) \in R$ \} = \{ $i \in \mathbb{N}$ | $i < |\tau_1|$ \} or \{ $j$ | $(i,j) \in R$ \} = \{ $j \in \mathbb{N}$ | $j < |\tau_2|$ \}, and

2. (Initial configurations) 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 $i \leq i'$.

To transform Definition 28 to our framework, we make a few important observations:

- The definition relates two memories that differ in exactly one variable (i.e., $\forall y \neq x. m_1(y) = m_2(y)$), which is different from the usual low-equivalence requirement in

---

8The original definition uses a specialized label semantics, denoted as $[b]_{(c,m)}$, and requires $\langle (c_i, m_i), \ell \rangle \notin \[b]_{(c,m)}$, which means that if by the time $\langle c, m \rangle$ reaches state $\langle c_i, m_i \rangle$, confidentiality level $\ell'$ may not observe the information. It is easy to convert that to $\ell \notin [b]_{\tau_1[i]}$ in our notation.
other definitions. However, it is easy to prove that (shown shortly) it is equivalent to a per-policy definition \([m] \neq b\) in our framework, that considers memories that differ only for variables with a particular policy \(b\).

- The component of \(\ell \not\in [q]_{t_1}^{i[i]} \land \ell \not\in [q]_{t_2}^{j[j]}\) filters out non-interesting outputs, which functions the same as the filtering function \([\ell]_{b,L}\).

- We define \(\equiv\) on two output sequence as below:

\[
\vec{t}_1 \equiv \vec{t}_2 \iff \neg (||\vec{t}_1|| = ||\vec{t}_2|| \land \exists i, \vec{t}_1^{i[i]} \neq \vec{t}_2^{i[i]})
\]

Based on the observations, we convert Definition 28 into our framework as follows:

\[
\sim_{AP} \triangleq \{(\vec{t}_1, \vec{t}_2) \mid \exists R. \forall (i, j) \in R. [\vec{t}_1^{i[i]}]_{b,L} \equiv [\vec{t}_2^{j[j]}]_{b,L}\}
\]

\[\equiv_{AP} \triangleq A_{AP} \triangleq [m] \neq b\]

**Lemma 5** With \(\sim_{AP}, A \triangleq A_{AP}, \text{ and outside equivalence } \equiv_{AP} \equiv_{AP}'\), Definition 25 is equivalent to Definition 28.

**Observation:** Compared with Gradual Release and Tight Gradual Release, the most interesting component of According to Policy is in its unique indistinguishability definition, which uses the correspondent relationship \(R\). Intuitively, According to Policy relaxes the indistinguishability definition in the way that two executions are indistinguishable as long as a correspondence \(R\) exists to allow decreasing knowledge. However, as shown later in the evaluation, the relaxation with \(R\) could be too loose: it falsely accepts some insecure programs.
Cryptographic Erasure

Cryptographic erasure [31] also uses compound labels to specify erasure policy and knowledge is defined as:

\[
k_{CE}(c, L, \vec{t}) = \{ m \mid \langle c, m \rangle \xrightarrow{\vec{f}_1} * \langle c_1, m_1 \rangle \xrightarrow{\vec{f}_2} * \langle c', m' \rangle \\
\land |\vec{f}_2|_L = |\vec{t}|_L \}\]

Unlike other policies, the definition specifies knowledge based on the subtrace relation, rather than the standard prefix relation. The reason is that it has a different attack model: it assumes an attacker who might not be able to observe program execution from the beginning.

**Definition 29 (Cryptographic Erasure Security [31])** A program \( c \) is secure if any execution starting with memory \( m \), the following holds:

\[
\forall c_0, m_0, c_i, m_i, c_n, m_n, \vec{t}_1, \vec{t}_2, L, i, n. \\
\langle c_0, m_0 \rangle \xrightarrow{\vec{f}_1} * \langle c_i, m_i \rangle \xrightarrow{\vec{f}_2} * \langle c_n, m_n \rangle \\
\Rightarrow k_{CE}(c, L, \vec{t}_2) \supseteq \bigcap_{t \in \vec{t}_2} [m]_{L,t,\gamma}
\]

To model subtraces, we adjust the \( \forall 1 \leq i \leq \|\vec{t}\| \) quantifier in the framework with \( \forall 1 \leq i < j \leq \|\vec{t}\| \), and write \( \vec{t}[i : j] \) for the subtrace between \( i \) and \( j \). Then, converting Definition 29 into our framework is relatively straightforward:

\[
\sim_{CE} \triangleq \{(\vec{f}_1, \vec{f}_2) \mid |\vec{f}_1|_L \text{ subtrace of } |\vec{f}_2|_L \}
\]

\[
\equiv_{CE} \triangleq \bigcap_{t \in \vec{t}} [m]_{L,t,\gamma}
\]

**Lemma 6** With \( \sim \sim_{CE}, \equiv \equiv_{CE} \) and \( A \triangleq A_{CE} \), Definition 25 with adjusted attack model is equivalent to Definition 29.
Observation: Compare with other dynamic policies, the most interesting part of cryptographic erasure is that its indistinguishability and policy allowance are both defined on subtraces; moreover, the latter uses the weakest policy on the subtrace. Intuitively, we can interpret Cryptographic Erasure security as: the subtrace-based knowledge gained from observing a subtrace should be bounded by the smallest allowance (i.e., the weakest policy) on the trace.

**Forgetful Attacker**

Forgetful Attacker \([32, 56]\) is an expressive policy where an attacker can “forget” some learned knowledge. To do so, an attacker is formalized as an automaton \(\text{Atk} \langle Q_A, q_{init}, \delta_A \rangle\), where \(Q_A\) is a set of attacker’s states, \(q_{init} \in Q_A\) is the initial state, and \(\delta_A\) is the transition function. The attacker observes a set of events produced by a program execution, and updates its state accordingly:

\[
\text{Atk}(\epsilon) = q_{init} \\
\text{Atk}(\vec{t} \downarrow i) = \delta(\text{Atk}_A(\vec{t} \downarrow i-1), t^{[i]})
\]

Given a program \(c\), an automaton \(\text{Atk}\) and attacker’s level \(L\), knowledge is defined as the set of initial memory that could have resulted in the same state in the automaton:

\[
k_{\text{FA}}(c, L, \text{Atk}, \vec{t}) = \{ m | \langle c, m \rangle \xrightarrow{\vec{t}_{1 \downarrow i}} * \langle c', m' \rangle \xrightarrow{\vec{t}_{2 \downarrow i}} * m'' \\
\wedge \text{Atk}([\vec{t}_{1 \downarrow i}]_L) = \text{Atk}([\vec{t}_{2 \downarrow i}]_L) \}
\]

**Definition 30 (Security for Forgetful Attacker \([32]\))** A program \(c\) is secure against an attacker \(\text{Atk} \langle Q_A, q_{init}, \delta_A \rangle\) with level \(L\) if:

\[
\forall c, c', m, m', \vec{t}, t', L. \langle c, m_1 \rangle \rightarrow \vec{t} \cdot t' \Rightarrow \\
k_{\text{FA}}(c, L, \text{Atk}, \vec{t} \cdot t') \supset k_{\text{FA}}(c, L, \text{Atk}, \vec{t}) \cap [m]_{L, \gamma'}
\]
The conversion of Definition 30 to our framework is straightforward:

\[
\sim_{\text{FA}} \triangleq \{(\vec{t}_1, \vec{t}_2) \mid \exists \vec{t}' \preceq \vec{t}_2. \text{Atk}(\vec{t}_1) = \text{Atk}(\vec{t}')\}
\]

\[
\equiv_{\text{FA}} \triangleq A_{\text{FA}} \triangleq K(c, \vec{t}[\|\vec{t}\| - 1], \sim_{\text{FA}}) \cap [m]_{L, \vec{t}[\|\vec{t}\|, \gamma}
\]

**Lemma 7** With \(\sim \triangleq \sim_{\text{FA}}, A \triangleq A_{\text{FA}},\) and outside equivalence \(\equiv \triangleq \equiv_{\text{FA}},\) Definition 25 is equivalent to Definition 30.

**Observation:** We note that Forgetful Attacker was originally formalized in a similar format as our framework, making the conversion straightforward. However, there are various differences compared with Dynamic Release. Most importantly, Forgetful Attacker security is parameterized by an automaton \(\text{Atk};\) in other words, a program might be both “secure” and “insecure” depending on the given automaton. Consider the program in Figure 5.1-B(i). The program satisfies Forgetful Attacker security with any automation that forgets about the credit card information. Nevertheless, characterizing such “willfully stupid” attackers is an open question [32]. Second, the definition of the consistency relation \(\equiv\) is completely different. As discussed in Section 5.4.3, it is important to define it properly to allow information flow restrictions to downgrade and upgrade in arbitrary ways.

### 5.6 Evaluation

In this section, we introduce \textit{AnnTrace} benchmark and implement the dynamic policies as the form shown in Table 5.1. The benchmark and implementations are available on github\(^9\).

#### 5.6.1 \textit{AnnTrace} Benchmark

To facilitate testing and understanding of dynamic policies, we created the \textit{AnnTrace} benchmark. It consists of a set of programs annotated with \textit{trace-level} security specifications. Among 58

\(^9\)https://github.com/psuplus/AnnTrace
Table 5.2 Evaluation Results.

<table>
<thead>
<tr>
<th>Examples in Fig 5.1</th>
<th>Existing(35)</th>
<th>New (23)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A(i)</td>
<td>A(ii)</td>
</tr>
<tr>
<td>Gradual Release</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tight Gradual Release</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>According to Policy p</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cryptographic Erasure</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Forgetful Attacker-Single</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic Release</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

✓ means the policy checks the program as intended (same as ground truth); × means the policy fails to check the program as intended. - means the program is not in the scope of the policy (not applicable).

Programs in the benchmark, 35 of them are collected from existing works [18,25,27,31,32,108]. References to the original examples are annotated in the benchmark programs. The benchmark also includes 23 programs that we created, such as the programs in Figure 5.1, and the counterexamples in Figure 5.6.

The benchmark is written in Python. Fig. 5.5 shows an example of annotated program for

```python
lat=Lattice()
lat.add_sub(Label("M"), lat.top)
Program(
    secure=True,
    source_code=""
        // credit_card: M
        copy := credit_Card
        output(copy, M);
        // credit_card: Top
        copy := 0;
        output(copy, M)"
    ,
    persistent=False,
    traces=[
        Trace(init_memory=dict(cc=0), outputs=[
            Out('M', 0, {'cc': 'M'}),
            Out('M', 0, {'cc': 'Top'})
        ],
        Trace(init_memory=dict(cc=1), outputs=[
            Out('M', 1, {'cc': 'M'}),
            Out('M', 0, {'cc': 'Top'})
        ],
        Trace(init_memory=dict(cc=2), outputs=[
            Out('M', 2, {'cc': 'M'}),
            Out('M', 0, {'cc': 'Top'})
        ])
    ,
    lattice=lat )
```

**Figure 5.5.** Annotated Program for Fig. 5.1-B(i)
the source code in Fig. 5.1-B(i). As shown in the example, each program consists of:

- **secure**, a boolean value indicating whether this program is a secure program; the ground truth of our evaluation.

- **source code**, written in the syntax shown in Fig 5.2;

- **persistent**, a boolean value indicating whether the intended policy in this program is persistent (or transient);

- **lattice**, \( L \), the security lattice used by the program\(^\text{10}\);

- **traces**, executions of the program. Each trace \( \tau \) has:
  
  - **initial memory**, \( m \), mapping from variables to integers
  
  - **outputs**, \( \vec{t} \), a list of output events, each \( t \) in type \( \text{Out} \):
    
    * **output level**, \( \ell \), a level from the lattice \( L \)
    
    * **output value**, \( v \), an integer value
    
    * **policy state**, \( \gamma \), mapping from variables to levels

Given a program in existing work, we (1) use the claimed security of code as the ground truth, (2) convert the program into our specification language and to a security lattice, (3) mark persistent (or transient) according to if the corresponding paper presents a persistent (or transient) policy, and (4) manually write down a finite number of traces that are sufficient for checking the dynamic policy involved in the example.

### 5.6.2 Implementation

We implemented all dynamic policies in Table 5.1 in Python, according to the formalization presented in the table. With exception of Forgetful Attacker and Paralocks, all implemented\(^\text{10}\)We use lattice instead of level set for conciseness in the implementation.
policies can directly work on the trace annotation provided by the AnnTrace benchmark. Forgetful Attack policy requires an automaton as input. So we use a single memory automaton that only remembers the last output and forgets all previous outputs. Paralocks security requires “locks” in a test program but most tests do not have locks. So we are unable to directly evaluate it on the AnnTrace benchmark.\(^\text{11}\)

Existing policies are not generally applicable to all tests. Recall that each test has a persistent/transient field. Moreover, for each test, we automatically generate the following two features from the traces field:

A. there is no policy upgrading in the trace;

B. there is no policy downgrading in the trace;

These tags are used to determine if a concrete policy is applicable to the test. For example, Cryptographic Erasure is a transient policy that only allows upgrading. Hence, it is applicable to the tests with tag\(\text{transient}\) and B.

### 5.6.3 Results

The evaluation results are summarized in Table 5.2. For the examples shown in Figure 5.1 (classical examples for declassification, erasure and delegation/revocation), we note that Dynamic Release is the only one that is both applicable and correct in all cases.

Among the 35 programs collected from prior papers and the 23 new programs, Dynamic Release is still both applicable and correct to all programs. In contrast, the existing works fall short in one way or another: with limited applicability or incorrect judgement on secure/insecure programs. Interestingly, According to Policy, Cryptographic Erasure and Gradual Release all make wrong judgment on some corner cases. Here, we discuss a few representative ones.

\(^{11}\)Although we are unable to evaluation Paralocks directly, we believe its results should resemble those of Gradual Release, as its security condition is a generalisation of the gradual release definition [106].
Figure 5.6. Counterexamples for Crypto-Erasure and Paralocks.

For According to Policy, the problematic part is the $R$ relation. The policy states that as long as a qualified $R$ can be found to satisfy the equation, a program is secure. We found that the restriction on $R$ is too weak in many cases: a qualified $R$ exists for a few insecure programs.

For Crypto-Erasure policy, the failed examples is shown in Figure 5.6-(A). It is an insecure program as the attacker learns that $x = 0$ if two outputs are observed. However, Crypto-Erasure accepts this program as secure for the reason that their policy ignores the location of an output. In this example, for the output 0, the security definition of Crypto-Erasure assumes that two executions are indistinguishable to the attacker if there exists a 0 output anywhere in the execution. Therefore, an execution with a single 0 output appears indistinguishably to the execution with two 0 outputs (both exists a 0 output). Thus, the policy fails to reject this program.

For Gradual Release, it fails on the following secure program, where $h, h1 : S$ and $l, l2 : P$.

\[
\text{if} \ (h) \ \text{then} \ l2 := \text{declassify}(h1); \\
l := 0;
\]

This example might seem insecure on the surface, as the branch condition $h$ was not part of the \text{declassify} expression. But in the formal semantics (Section 5.5.2), a release event declassifies all information in the program (i.e., Gradual Release does not provide a precise bound on the released information as pointed out in [18, 51]). The program is secure since $h1$
is assigned to $l_2$ when both $h$ and $h_1$ are declassified by the release event.

To check if a similar issue also exists in Paralocks, whose security condition is a generalization of the Gradual Release, we created a Paralock version of the same code, as shown in Figure 5.6-(B). Thanks to the cleaner syntax of Paralocks, it is more obvious that the program is secure: $h$ and $h_1$ have the same lock set \{D\}. Lock $D$ is opened before the if statement, allowing value of both $h$ and $h_1$ to flow to $l, l_2$. So the assignment in the if branch is secure. After that, only a constant 0 is assigned to $l$ when the lock $D$ is closed. However, the Paralock implementation rejects this program as insecure.

To understand why, Paralocks requires the knowledge of an attacker remains the same if the current lock state is a subset of the lock set that the attacker have. We are interested in attacker $A_1 = (a, \emptyset)$, who has an empty lock set. When lock $D$ is open, since \{D\} $\not\subseteq$ \emptyset, there is no restriction for the assignment $l_2 := h_1$. However, for the assignment $l := 0$, the current lock set is \emptyset, which is a subset of $A$’s lock set (\emptyset). That is, for all the executions, the attacker $A$’s knowledge should not change by observing the output event from assignment $l := 0$. However, this does not hold for the execution starting with $h = 0$. The initial knowledge of attacker $A$ knows nothing about $h$ or $h_1$ since they are protected by lock $D$. With $h = 0$, the assignment in the branch is not executed. The attacker only observes the output from $l := 0$. By observing that output, the attacker immediately learns that $h = 0$. Therefore, Paralock rejected this program as insecure.

5.7 Related Work

The most related works are those present high-level discussions on what/how end-to-end secure confidentiality should look like for some dynamic security policy. The major ones are already discussed and compared in the chapter.

To precisely describe a dynamic policy, RIF \[53,54\] uses reclassification relation to associate label changes with program outputs. While this approach is highly expressive, writing down the
correct relation with regards to numerous possible outputs is arguably a time-consuming and error-prone task. Similarly, flow-based declassification [55] uses a graph to pin down the exact paths leading to a declassification. However, the policy specification is tied up to the literal implementation of a program, which might limit its use in practice.

Bastys et al. [109] present six informal design principles for security definitions and enforcements. They summarize and categorize existing works to build a road map for the state-of-art. Then, from the top-down view, they provide guidance on how to approach a new enforcement or definition. In contrast, the framework and the benchmark proposed in this chapter are post-checks after one definition is formalized.

Recent work [110] presents a unified framework for expressing and understanding for downgrading policies. Similar to Section 5.4, the goal of the framework is to make obvious the meaning of existing work. Based on that, they move further to sketch safety semantics for enforcement mechanism. However, they do not provide a define a formalization framework that allows us to compare various policies at their semantics level.

Many existing work [111–113] reuses or extends the representative policies we discussed in this chapter. They adopt the major definition for their specialized interest, which are irrelevant to our interest. Hunt and Sands [28] present an interesting insight on erasure, but their label and final security definition are attached to scopes, which is not directly comparable with the end-to-end definitions discussed in this work. Contextual noninterference [114] and facets [115] use dynamic labels to keep track of information flows in different branches. The purpose of those labels is to boost flow- or path-sensitivity, not intended for dynamic policies.

5.8 Summary

We present the first formalization framework that allows apple-to-apple compassion between various dynamic policies. The comparison sheds light on new insights on existing definitions, such as the distinguishing between transient and persistent policies, as well as motivates
Dynamic Release, a new general dynamic policy proposed in this work. Moreover, we built a new benchmark for testing and understanding dynamic policies in general.
Chapter 6
Decomposition of Dynamic Policy to Static Policy

6.1 Introduction

To support flexible security requirements in real-world applications, dynamic policies in various forms have been introduced in information flow analysis. Existing dynamic policies are very different in their design, formalization and even the end-to-end security goals. Hence, not surprisingly, enforcement of various dynamic policies also diverges significantly. For example, prior works leverage security type systems [18, 31, 53, 56, 112], dynamic monitors [32, 113], secure-multi-execution [111] and special-purpose operations embedded in the source language [27, 116].

Moreover, existing enforcements are mostly built from scratch with the primary goal of soundness, i.e., rejecting all programs that might violate information flow policy. For most of them, we can find their roots back to well-studied enforcements of static information flow policy. However, in terms of enforcement, the fundamental gap between static and dynamic policies is still largely unexplored in the community.

In this chapter, we take a different approach of enforcing dynamic policy; we tackle the
following question:

*Is there a principled way to reuse enforcements of static policy for a general-purpose dynamic policy?*

Based on the findings in Chapter 5, we can specify a general-purpose dynamic policy as sensitivity mutations triggered by a set of security events or program states. So intuitively, a dynamic policy can be interpreted as a sequence of code blocks separated by policy mutations, where sensitivity within each code block is stable (i.e., static).

Based on the insight, we propose to decompose a dynamic policy into several static policies on individual code blocks, so that the end-to-end dynamic policy is enforced *if and only if* all corresponding static policies are enforced on their relevant code blocks. More precisely, let $\Gamma$ be a dynamic policy on program $c_1; c_2; \cdots; c_n$. Our goal is to break $\Gamma$ into a sequence of static policies $\gamma_1, \gamma_2, \cdots, \gamma_n$ such that $\Gamma$ is enforced on the whole program if and only if for all $1 \leq i \leq n$, the static policy $\gamma_i$ is enforced on $c_i$. Note that the overall goal contains three important sub-goals:

- **Generality**: the decomposition is not limited to any concrete variant of dynamic policy.
- **Soundness**: dynamic policy is enforced on the whole program whenever all static policies are enforced on their corresponding code blocks.
- **Completeness**: dynamic policy is enforced on the whole program only if all static policies are enforced on their corresponding code blocks.

For generality, we formalize the syntax and semantics of a language with general-purpose dynamic policy that allows sensitivity to change in arbitrary ways. Following Chapter 5, the policy also supports both transient and persistent end-to-end security semantics. For soundness, our observation is that the decomposed static policies need to soundly track data dependency between code blocks. For example, consider a program $c_1; c_2$ where $y$ depends on $x$ in the code block $c_1$. Then, the static policy of $y$ in $c_2$ should be at least as restrictive as the policy of $x$ in
to ensure end-to-end security. For completeness, our observation is that intransitive dataflow prevents a decomposition to be complete. For example, consider a program $c_1; c_2$ where $y$ depends on $x$ in the code block $c_1$ and $z$ depends on $y$ in the code block $c_2$. Completeness of decomposition is violated when the final value of $z$ does not depend on $x$, in which case we say that dataflow in the program is intransitive.

Based on the observations, we formalize the conditions under which a decomposition of transient policy is both sound and complete. Intuitively, the conditions ensure that data dependency is soundly tracked in the decomposition, and the program is free of intransitive dataflow. On the other hand, we also show a negative result regarding decomposition for persistent policy: sound and complete decomposition of a persistent policy is infeasible since its end-to-end security semantics requires extra runtime information from the past to decide whether a flow of information is legal.

In this chapter, we make the following contributions:

• We present a language with concise yet expressive security specification and end-to-end security goal (Section 6.3) to allow us to specify general-purpose dynamic policy.

• We formalize and prove the correctness of a sound and complete decomposition (Section 6.4) for transient policy. Therefore, we can reuse existing enforcements of static policies to enforce transient dynamic policy.

### 6.2 Background and Overview

#### 6.2.1 Background

**Security Levels** Following the definitions in Chapter 5, we assume the existence of a set of predefined security levels $\mathbb{L}$, and we use level sets $L \subseteq \mathbb{L}$ to describe the intended confidentiality of information. Intuitively, any level $l \in \mathbb{L}$ is allowed to view information with level set $L$ if and only if $l \in L$. Consequently, information can flow from $L_1$ to $L_2$ if and only if $L_2 \subseteq L_1$.
Dynamic Policy  An information flow policy is dynamic if it allows the sensitivity of information to change during one execution of program. For example in Figure 6.1(a), variable $x$ is associated with a dynamic policy $P \rightarrow^s S$. As we will show more details for the policy specification in Section 6.3, the policy $P \rightarrow^s S$ states that variable $x$ is of sensitivity $P$ initially; when the security event $s$ is turned on, its sensitivity changes to $S$. In general, there is no restriction on the direction of mutation: a downgrading policy makes information less sensitive, while an upgrading policy makes information more sensitive. When an upgrading mutation is involved, two opposite assumptions are found in the existing works on how to handle a previously learned information after upgrading: transient or persistent.

To understand the difference between transient and persistent policies, consider the example shown in Figure 6.2, where $x$ is initially $P$ and a partial value $x\%2$ is given to public variable $y$. Then, the sensitivity of $x$ is upgraded to $S$. Now controversy arises for the sensitivity of $x\%2$ due to the fact that $x\%2$ is already learned by a public variable $y$ before the upgrading.

Transient Policy  A transient policy does not consider the learning history. What is learned by the attacker before makes no difference to the current sensitive of a secret. In this example, a transient policy considers the learned value $x\%2$ to have sensitivity $S$, the same as the upgraded $x$. Thus, for a transient policy, this program is insecure; it is not allowed to access the learned valued $x\%2$ at Line 5 after upgrading of $x$.

Transient policies are commonly found in data retain policies, for example, credit card information during online transactions is usually associated with a transient policy, where the credit card number is first given to a merchandiser at checkout. However, when the payment is complete, an upgrading mutation requires that the merchandiser should not retain any credit.
// x : P → S[Tran];
// y, z : P
y := x % 2;
Event(s);

(a) Original Program (insecure).

Figure 6.1. Decomposition of Dynamic Policy.

// x : P → S[Per];
// y : P;
y := x % 2;
Event(s);
y := x % 2;

(b) Code Block A.

(c) Code Block B.

(a) Original Program (secure).

Figure 6.2. Decomposition of A Persistent Policy.

card information, including the information obtained before the payment is complete.

**Persistent policy** On the other hand, a persistent policy assumes that an attacker remembers the value of \( x \% 2 \) that has already leaked to a public variable. Hence, such a policy considers the access to the same value \( x \% 2 \) safe even after \( x \) is upgraded. In other words, a persistent policy allows the attacker to own any information forever after it is exposed to them. For example, in a book renting system, customers can read books during the renting period, and are allowed to take notes from the books they rent. When the renting is over, the book is no longer accessible to the customer. However, the notes owned by the customer should remain accessible to the customer.

### 6.2.2 Overview

While dynamic policies are intricate, as shown in Chapter 5, we observe that in between any two adjacent sensitivity mutations, the dynamic policy is expected to have a static (i.e., fixed) sensitivity during that period of time. The observation motivates us to study if it is possible to decompose a dynamic policy into a set of static policies, each governing a period of time
Figure 6.3. Decomposition of Program with False Data Dependencies.

when no mutation occurs. If so, we can enforce an intricate dynamic policy via enforcing a set of static policies on code fragments, which is well-studied in the literature. Moreover, if the decomposition is both sound and complete, enforcing the decomposed static policies is identical to enforcing the original dynamic policy.

- Soundness: a policy decomposition is sound if the original dynamic policy is enforced whenever all of the decomposed static policies are enforced.

- Completeness: a policy decomposition is complete if the original dynamic policy is enforced implies that all of the decomposed static policies are enforced.

Therefore, a policy decomposition is both sound and complete when the following condition holds: the original dynamic policy is enforced if and only if all of the decomposed static policies are enforced. In this chapter, we show that for a persistent policy, policy decomposition can be both sound and complete. However, policy decomposition cannot be both sound and complete for a persistent policy.

Challenge 1 The first challenge of policy decomposition is to figure out the static policy that each decomposed code fragment should obey. One straightforward (but incorrect) approach is to derive static policies purely based on policy mutation. For example, Figure 6.1 shows the decomposition of the dynamic policy $P \rightarrow^s S$ on variable $x$ if we follow this approach.

Note that the mutation of sensitivity takes place at Line 4, where the security event $s$ is turned on. We can thus divide the program into two code blocks before and after line 4, shown in
Figure 6.1(b) and Figure 6.1(c) respectively. In the first part, all variables have a fixed level P since variable \( y \) and \( z \) are statically set to P; for variable \( x \), the security event \( s \) is currently false, thus the dynamic policy evaluates to P. In the second part, \( y \) and \( z \) remain P, while \( x \)’s sensitivity changes to S since this code block is after line 4, which triggers the security event \( s \).

However, this straightforward approach is unsound. Return to the example in Figure 6.1. It is apparent that both code blocks A and B are secure according to their static policies (note that the secret variable \( x \) is not used in block B at all); but the original program is insecure (note that the value of \( x \) is leaked to public variable \( z \) when \( x \) is upgraded to level S).

Our observation is that the soundness issue is due to the fact that the straightforward approach fails to consider data dependency between code blocks. In this example, the static policy of code block B should recognize the data dependency from variable \( x \) to variable \( y \) in block A, and as a consequence, treat \( y \) at least as sensitive as \( x \) in block B. That is, the static policy of variable \( y \) should be fixed at level S in code block B, which leads to the rejection of code block B due to an explicit flow from S (the level of \( y \)) to P (the level of \( x \)).

**Challenge 2** In terms of completeness, we must rule out false dependencies in a program while decomposing a dynamic policy. For example, Figure 6.3(a) shows a secure program where the value stored in \( x2 \) is not revealed to public variable \( z \) at Line 5, since by combining both Line 3 and Line 5, \( z \) is equivalent to \( x1 \), a public variable throughout program execution. However, if we derive the static policy on code block B based on the level of \( x2 \) in block B (as shown in Figure 6.3(c)), block B is insecure since it reveals to public variable \( z \) the value of \( y - x2 \), which is derived from variable \( x2 \), a secret to be protected in block B.

We note that the incompleteness issue is not unique for dynamic policy: the same issue exists for purely static policies as well. Consider a static policy where we make \( x2, y \) secret (at level S) and \( x1, z \) public (at level P) throughout program execution. If we check the policy individually on Code Block B, the block is insecure as well due to the flow from S (levels of \( y \) and \( x2 \)) to P (level of \( z \)).
Our observation is that *intransitive* data flows prevent a decomposition to be complete, for example in Figure 6.3, we find data dependency from \(x^2\) to \(y\), and then from \(y\) to \(z\), but eventually there is no data dependency from \(x^2\) to \(z\). When verifying the decomposed blocks, the first block contains data dependency from \(x^2\) to \(y\), and the second block has dependency from \(y\) to \(z\), and naturally the combined results will lead to the conclusion that there is a data dependency from \(x^2\) to \(z\).

**Challenge 3** In general dynamic policies, two opposite assumptions are used for upgrading mutations: *persistent* and *transient*. We observe that decomposition of a persistent policy is infeasible since it requires dynamic knowledge of the previous leakage since it assumes that it is always safe to reveal a *leaked* value. For example, Figure 6.2(a) shows a secure program with \(x\) labeled with a persistent upgrading policy. At line 5, the public variable \(y\) learns the value of \(x^2\) when \(x\) is secret. However, this program is still secure since the same secret is already leaked to public at Line 2, when \(x\) is public. When we decompose the program by mutations as resulted in code blocks in 6.2(b) and 6.2(c), code block B can only be verified as safe only if we also check that the leakage of \(x\) is precisely \(x^2\) in block A. In general, we might need to check leakage in prior blocks for a persistent policy, making the decomposition approach infeasible for persistent policy.

**Key Results** Based on the previous observations, we formalize and prove a few important results regarding decomposing a dynamic policy in this chapter. Note that as shown in Figure 6.2, decomposition of a persistent policy is infeasible since it requires history information to decide the sensitivity of a value. Hence, in the rest of the chapter, we assume a transient policy. We summarize the informal results as follows, and then formalize and prove them in the rest of the chapter.

- For a decomposition to be sound, i.e., rejecting a insecure program, a decomposed policy must soundly encounter for all restrictions on values that a variable depends on.
Variables (Vars) \(x, y, z\)

Expressions (\(E\)) \(e ::= x | n | e \text{ op } e\)

Standard Security Events (\(S\)) \(s\)

Standard Commands \(c ::= \text{skip} | x := e | \text{while} (e) c \mid \text{if} (e) \text{ then } c_1 \text{ else } c_2 | c_1; c_2\)

Commands \(cmd ::= c \mid \text{Event} (s) \mid cmd_1; cmd_2\)

Security Labels (\(B\)) \(b ::= L \mid L \rightarrow^s b\)

Policy Specification \(\Gamma : \text{Vars} \rightarrow \mathbb{B}[\diamond]\)

Policy Type \(\diamond ::= \text{Tran} | \text{Per}\)

**Figure 6.4.** Language Syntax with Security Specification.

- For policy decomposition to be complete, i.e., accepting a secure program, data dependency must be transitive so that the decomposed policy does not introduce false positives due to false data dependency.

- We can construct an equivalent decomposition (both sound and complete) when both the soundness and completeness conditions above are satisfied.

### 6.3 Syntax and Semantics of Dynamic Policy

We first present the syntax of an imperative language with general-purpose dynamic security. Then, we formalize the end-to-end security semantics of the dynamic policy, and how to decompose a transient dynamic policy soundly and completely.

#### 6.3.1 Language Syntax and Security Specification

In this section, we use a simple imperative language with expressive security specification, as shown in Figure 6.4. The language includes the standard commands \(c\) such as assignments, sequential composition, branches and loops. We introduce distinguished security events \(S\). An event \(s \in S\) is similar to a Boolean; we distinguish an event \(s\) and variable \(x\) in the language syntax to ensure that security events can only be set using distinguished commands \(\text{Event} (s)\),
which set \( s \) to \( \text{true} \). We assume that all security events are initialized with \( \text{false} \). The full program consists of standard commands \( c \) and security commands. By syntax, the security commands only show up at the top level of the language; it cannot be placed inside a branch. The restriction is needed to ensure that all sensitivity mutations do not dependent on any (potentially sensitive) runtime program states.

The policy specification is adapted from the general policy defined in Chapter 5; the most interesting part is to define how data sensitivity changes at run time. A label \( b \) can simply be a level set \( L \); in this case, the policy is static: it represents immutable sensitivity throughout program execution. In general, a label has the form of \( L \to s b \) where the trigger condition is a security event \( s \), and \( \to \) specifies a one-time sensitivity mutation when the security event \( s \) evaluates to \( \text{true} \).

The information flow policy on a program is specified as a function from variables \( \text{Vars} \) to security labels \( \mathcal{B} \) and a policy type \( \diamond \). Following the discussion in Chapter 5, a policy type can either be transient \( \text{Tran} \), or persistent \( \text{Per} \). A transient policy is assumed by default.

Compared with the label formalization in Chapter 5, the major difference is on the allowed trigger conditions. In this chapter, the specification only supports one single security event in the trigger and all security events are set at the top level. However, security label in Figure 5.2 allows both security events and program expressions. We introduce the restrictions for simplicity in this Chapter; we plan to study decomposition in the most general setting as future work.

### 6.3.2 Interpretation of Security Specification

The security specification syntax in Figure 6.4 naturally separates a program into a couple of code fragments, separated by security events (note that by syntactical restriction, security events only appear at the top level). Hence, we can compute a static security policy for each command in the source code as formalized in Figure 6.5.
Intuitively, \([\text{cmd}]_b\) computes the (static) interpretation of label \(b\) for each standard command, in the form of \(\{c_1\}^{L_1}\{c_2\}^{L_2}\ldots\{c_n\}^{L_n}\), where \(c_i\) consists of standard commands only, and \(L_i\) is a static level set that specifies the sensitivity of policy \(b\) for command \(c_i\). More specifically, the computation looks at the first security event in the program and tries to match it with the first trigger in the dynamic policy \(b\).

- If the first trigger event matches the first security command event, as shown in the first rule, sensitivity change occurs: \(c_1\) is under static policy \(L_1\), and the rest of commands \(\text{cmd}_2\) is under policy \(b_2\).

- If the events do not match, as shown in the second rule, then the event is irrelevant to the dynamic policy, and thus ignored.

- When a static policy is given, as shown in the third rule, the computation (safely) removes the security command and results in a code block with the same static level set \(L\).

For example, the interpretation of the dynamic policy on \(x\) in Figure 6.1 is computed as:

\[
[y := x \% 2; \text{Event}(s); y := z]_{P \rightarrow S} = \{y := x \% 2\}_P; \{y := z\}_S = \{y := x \% 2\}_P; \{y := z\}_S
\]

That is, in the first code block \(\{y := x \% 2\}\), the sensitivity of \(x\) is \(P\); in the second code block \(\{y := z\}\), the sensitivity of \(x\) is \(S\).
Finally, we collect the interpretation of each variable’s policy and form a local mapping $\gamma$ from each variable $x$ to its corresponding static level set $L$. For example, the dynamic policy $\Gamma \triangleq \{ x : P \rightarrow S, y : P \}$ on the program in Figure 6.1 is interpreted as $\{ y := x \% 2 \}_1^\gamma; \{ y := z \}_2^\gamma$, where $\gamma_1 \triangleq \{ x : P, y : P \}$ and $\gamma_2 \triangleq \{ x : S, y : P \}$.

### 6.3.3 End-to-End Security Semantics

Next, we define the end-to-end security semantics of a dynamic policy $\Gamma$ on a program $cmd$.

To do that, an important component is the visibility of an attacker, i.e., what an attacker can see during program execution. We adopt the standard memory-based security model, where a program execution is modeled as a sequence of configurations $\langle c, m \rangle$, where $c$ is a program and $m$ is memory. Here, we define the evaluation on the standard commands $c$ only since the security commands make no difference to the program semantics and are removed after decomposition. An attacker at level $l$ can only observe part of the memory containing variables with level set $L$ if $l \in L$. We formalize this as a low-equivalence relation on memories:

**Definition 31 (Low equivalence)** We say two memory $m_1, m_2$ are low equivalent at level $l$ according to $\gamma$, denoted as $m_1 \approx_l^\gamma m_2$, if the two memory looks the same to an attacker at level $l$ according to the sensitivity mapping $\gamma$: 

$$m_1 \approx_l^\gamma m_2 \iff \forall x. \ l \in \gamma(x) \Rightarrow m_1(x) = m_2(x)$$

For a code block with static policy, i.e., $\{ c \}^\gamma$, we define noninterference in the standard way: a secret, say $x$, is protected from an attacker, if changing its value in initial memory does not make any difference to the visible information available to the attacker.

**Definition 32 (Noninterference per variable $x$)** For a code block $c$ with a static policy $\gamma$, written as $\{ c \}^\gamma$, we say that it satisfies noninterference for variable $x$ (i.e., $x$ is protected),
denoted as \( x \vdash \{c\}^\gamma \), if we have \( l \not\in \gamma(x) \) and

\[
\forall m_0, m'_0, m_1, m'_1. \ (c, m_0) \rightarrow^* \langle \text{skip}, m_1 \rangle \land \langle c, m'_0 \rangle \rightarrow^* \langle \text{skip}, m'_1 \rangle \land \\
(\forall y. y \neq x \land m_0(y) = m'_0(y)) \Rightarrow (\forall l. l \not\in \gamma(x) \Rightarrow m_1 \approx^\gamma l m'_1)
\]

In other words, the definition above states that no matter what is the initial value of \( x \), as long as all other variables have the same initial values, an attacker at level \( l \), who is not allowed to learn \( x \) according to the policy (i.e., \( l \not\in \gamma(x) \)), always sees the same result from the program.

Next, we extend the definition for dynamic policies:

**Definition 33 (Dynamic Protection)** For a transient dynamic policy \( \Gamma \) that is interpreted on a program \( \text{cmd} \) as \( \llbracket \text{cmd} \rrbracket_\Gamma = \{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n} \), we say that a variable \( x \) is protected, denoted as \( x \vdash \llbracket \text{cmd} \rrbracket_\Gamma \), or equivalently \( x \vdash \{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n} \), if we have

\[
\forall m_0, m'_0, \ldots, m_n, m'_n.

\langle\{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n}, m_0\rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}; \{c_{i+1}\}^{\gamma_{i+1}}; \ldots; \{c_n\}^{\gamma_n}, m_i\rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}, m_n\rangle \land \\
\langle\{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n}, m'_0\rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}; \{c_{i+1}\}^{\gamma_{i+1}}; \ldots; \{c_n\}^{\gamma_n}, m'_i\rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}, m'_n\rangle \land \\
(\forall y. y \neq x \land m_0(y) = m'_0(y)) \Rightarrow (\forall l, i \in [1, n]. l \not\in \gamma_i(x) \Rightarrow m_i \approx^\gamma_i l m'_i)
\]

The definition states that whenever two executions only differ in the initial value of \( x \), at the end of each code block \( \{c_i\}^{\gamma_i} \), the memory states (\( m_i \) and \( m'_i \)) must appear the same (i.e., leaks nothing about \( x \)) to any attacker who is not allowed to learn \( x \) (i.e., \( l \not\in \gamma_i(x) \)). Definition 33 is connected to Definition 32 in the following way:

\[
x \vdash \{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n} \iff \forall i \in [1, n]. x \vdash \{c_1; \ldots; c_i\}^{\gamma_i}
\]

We note that dynamic protection is not a local security property confined within each single
static block \( c_i \); it is an end-to-end security property that should be enforced \textit{up to} each code block, i.e., \( c_1; \ldots; c_i \). The goal of policy decomposition, which we elaborate next, is to check some security properties on each code block (i.e., \( \forall i \in [1, n]. \ x \vdash \{ c_i \}^\gamma_i \)) so that the check succeeds if and only if the global dynamic policy holds (i.e., \( x \vdash \{ c_1 \}^\gamma_1 \ldots \{ c_n \}^\gamma_n \)).

### 6.4 Decomposition of Transient Dynamic Policy

We formalize and prove the main results of the chapter: it is possible to decompose a transient dynamic policy in a sound and complete way.

We start with soundness. Intuitively, a sound decomposition requires that the decomposed policies to be \textit{stronger} than the original policy. It ensures that an insecure program, e.g., Figure 6.1, is rejected by at least one of the decomposed code blocks. As discussed in Section 6.2.2, our observation is that for a decomposition to be sound, the sensitivity on each variable must be stronger than the sensitivity of any variable that it depends on in previous blocks. To prove the result, we first formalize data dependency as follows.

**Definition 34 (Dependency)** We say \( x \) is data-dependent on \( y \) in code block \( c \), denoted as \( x \leftrightarrow_c y \) as follows:

\[
x \leftrightarrow_c y \iff \exists m_1, m_2, m'_1, m'_2. \langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m_2 \rangle \land \langle c, m'_1 \rangle \rightarrow^* \langle \text{skip}, m'_2 \rangle \land \\
(\forall z. z \neq x \land m_1(z) = m'_1(z)) \Rightarrow m'_2(y) \neq m'_2(y)
\]

In other words, a data dependency from variable \( x \) to variable \( y \) is spotted in code \( c \), if we can find one case, where by changing the initial value of \( x \) only, the final value of \( y \) is changed consequently.

To soundly protect the value of \( x \) in \( \{ c_1 \}^\gamma_1 \{ c_2 \}^\gamma_2 \), we must require that a variable \( y \) is protected more restrictively in \( c_2 \) (i.e., \( \gamma_2(y) \subseteq \gamma_2(x) \)), since the smaller a level set is, the more
restrictive its sensitivity is) if \( y \) depends on \( x \) in \( c_1 \) (i.e., \( x \rightarrow_{c_1} y \)). This is formalized as the following theorem; its full proof is shown in Appendix C.

**Theorem 9 (Soundness)** A decomposition is sound if every variable is protected at the same or more restrictive level set than any variable that it depends on.

\[
\forall \Gamma, cmd, c_1, \cdots, c_n, L_1, \cdots L_n.
\]
\[
\left[ cmd \right]_\Gamma = \{c_1\}^{L_1} \wedge \cdots \{c_n\}^{L_n} \wedge (\forall x, y, i. x \rightarrow_{c_1; \cdots; c_{i-1}} y \Rightarrow \gamma_i(y) \subseteq \gamma_i(x)) \Rightarrow
\]
\[
(\forall x, i \in [1, n]. x \vdash \{c_i\}^{\gamma_i}) \Rightarrow (\forall x. x \vdash \left[ cmd \right]_\Gamma)
\]

We note that the sensitivity restriction \( \gamma_i(y) \subseteq \gamma_i(x) \) is posed on the current sensitivity in \( \gamma_i \). This is because it must reflect the most current sensitivity of the secret passed from \( x \) to variable \( y \). For example in Figure 6.1, a data dependency from \( x \) to \( y \) is found in the first block. Thus, in the second block, \( y \) stores information of original \( x \). While the initial value of \( x \) is \( P \), \( x \) in the current block is \( S \). Thus, for this decomposition to be sound, \( y \) should be as least \( S \) (the current sensitivity of \( x \)).

Intuitively, a complete decomposition requires the decomposed policies to be weaker than the original policy. It ensures that a secure program, e.g., Figure 6.3, is accepted by all decomposed code blocks with their corresponding policies. As discussed in Section 6.2.2, our observation is that the key barrier that prevents a decomposition to be complete is false data dependency between code blocks.

To formalize false data dependency, we first introduce intransitive data dependency. For example, data dependency in Figure 6.3 is intransitive, since there is no data dependence from \( x_2 \) to variable \( z \), even though there are true data dependencies from \( x_2 \) to \( y \) and \( y \) to \( z \). More formally, we define transitive data dependence as below.

**Definition 35 (Transitive Dependency)** We say all dependencies in \( \{c_1\}; \{c_2\} \) are transitive,
denoted as $\models \{ c_1 \}; \{ c_2 \}$, if

\[
\forall x, y. (\exists z. x \leftrightarrow c_1 \allowbreak z \land z \leftrightarrow c_2 \allowbreak y) \Rightarrow x \leftrightarrow c_1; c_2 \allowbreak y
\]

This property guarantees that any data dependency considered in each decomposed code block is part of a true data dependency globally.

Putting everything together, a decomposition is sound and complete if it takes into account of all the data dependencies in previous blocks and all data dependencies are transitive. We formalize this as Theorem 10 and the full proof is shown in Appendix C.

**Theorem 10 (Sound and Complete)** A decomposition of transient policy is sound and complete if we have transient data dependencies and the sensitivity of each variable is precisely the same as the combined sensitivity of all the value it depends on.

\[
\forall \Gamma, cmd. \ [cmd]_\Gamma = \{ c_1 \}^{\gamma_1}; \ldots; \{ c_n \}^{\gamma_n} \land \\
(\forall x, y, i. \gamma_i(y) = \bigcap_{x', x' \leftrightarrow c_1 \ldots c_{i-1}; y} \gamma_i(x)) \land \models \{ c_1; \ldots c_{i-1}; \{ c_i \}\} \Rightarrow \\
(\forall x. x \vdash [cmd]_\Gamma) \iff (\forall i, x. x \vdash \{ c_i \}^{\gamma_i})
\]

### 6.5 Related Work

Most existing enforcements of dynamic policies are specific to some kinds/forms of dynamic policy; thus they are hardly applicable for other dynamic policies. To the best of our knowledge, this is the first work that explores how to decompose a general dynamic policy soundly and completely into a sequence of static policies on code blocks. Compared with prior work, the decomposition approach allows us to directly reuse well-studied enforcements of static information flow policy.

The most related work by Hunt and Sands [28] shares a similar mindset, which enforces
a local erasure policy as a flow-sensitive noninterference policy on individual code blocks. However, their language syntax restricts the scope of secrets to specific code blocks, making it more restrictive than the language syntax of this chapter, and their enforcement only applies to erasure policy. While they show the encoding could be sound and complete for each erasure code block individually, there is no completeness result on the enforcement regarding end-to-end security of the whole program.

Security type systems are lightweight static methods of verifying dynamic information flow security [18, 31, 53, 56, 112]. A type system checks a global property by breaking it into a set of necessary conditions on sub-programs, which is to some extent similar to our goal. However, those type systems are not general: they are both policy- and language-specific. Moreover, they mostly ignore completeness; they do not shed light on under which conditions, dynamic information flow policy can be decomposed both soundly and completely.

Another common approach to enforce dynamic policy is to embed enforcements in language syntax and semantics. For example, Paralocks [105, 106] introduces locks and lock operations in the language as part of policy specification. Meanwhile, the language semantics guarantees that the program can only run in the way that the locks allow (i.e., the program will terminate otherwise). Similarly, Chong and Myers [27] introduce special commands as part of the policy specification and at the same time, the enforcement of the policy is achieved by the semantics of those special commands. While these approaches are easy to implement, they are fully specialized and are not reusable in cases where program semantics is policy-agnostic.

6.6 Summary

In this chapter, we demonstrate the possibility of enforcing a dynamic policy by decomposition. We show that for a transient policy, sound and complete policy decomposition is possible when the decomposed static policies traces restrictions on data dependency, and dataflows in the program are transitive. Compared with existing enforcements, the advantage of our
approach is its board applicability: it is not limited to a certain type or formalization of dynamic policy. Moreover, it allows us to leverage existing enforcements on static policies once policy decomposition is done.
Chapter 7 | Future Work

In this chapter, we discuss interesting future directions that can further improve information flow analysis in practice.

7.1 False Alarms

In Chapter 3, we consider imprecision due to data- and path-sensitivity. But information flow analysis still may suffer from other sources of imprecision, such as the presence of insecure dead code and false control-flow dependency. For example, consider the secure program in Figure 7.1 (simplified from an example in [117]) with security labels $s : S$, $p : P$. In this example, although $x$ is updated under a confidential branch condition, both branches result in the same state where $x = 1$; thus, the outcome of $p$ is independent of the value of $s$. However, our analysis in Chapter 3 rejects this program since rule (T-ASSIGN) conservatively assumes that any public variable modified in a confidential branch would leak information via implicit

```plaintext
1  x := 1; y := 1;
2  if (s == 0) then skip
3  else x := y;
4  p := x;
```

Figure 7.1. False Control-Flow Dependency.
flow. Motivated by the type system in [117], a promising direction that we plan to investigate is to incorporate sophisticated static program analyses so that the implicit flows can be ignored for the variables whose values are independent of branch outcomes. Additionally, hybrid information flow monitors (e.g., [48, 117, 118]) are shown to be more precise than static flow-sensitive type systems. We plan to study limitation of each approach and compare their analysis precision as future work.

### 7.2 Dependent Label Inference

In Chapter 4, we present the first framework for designing and checking label inference algorithm for information flow analysis with dependent security labels. There are a few directions to further improve the efficiency of our framework. First of all, it is very promising to obtain a practically efficient encoding for dynamic labels by directly adding the assumptions on dynamic labels directly into the right-hand-side constraints on security levels. Doing so requires extending the solving algorithm for label constraints (e.g., Rehof-Mogensen) to make use of the assumptions on dynamic labels, which is already implemented in Jif [11]. Second, the efficiency of the framework can be further boosted up by designing more sophisticated partitioning algorithms, such as using feedback from the solver to guide the search for predicates. A potential candidate is to use the unsatisfiable core from the current iteration to determine the predicates used in next iteration. However, the challenge is to compute a “minimum” unsatisfiable core, since a trivial core could be all constraints in the current iteration, making all predicates be added for the next iteration.

### 7.3 Dynamic Policies

We present the first formalization framework that allows apple-to-apple comparison between various dynamic policies in Chapter 5. One interesting direction is to investigate semantic
security condition of dynamic information flow methods, especially those use dynamic security labels. Despite the similarity that security levels are mutable, issues such as label channels might be challenging to be incorporate in our formalization framework.

Another future direction is to fully support partial release with expression-level specification. However, doing so is tricky since the expressions might have conflicting specifications. For example, consider a specification $x, y : S$ and $x + y, x - y : P$. It states that the values of $x$ and $y$ are secrets, but the values of $x + y$ and $x - y$ are public. Mathematically, learning the values of $x + y$ and $x - y$ can also reveal the concrete values of $x$ and $y$. Thus, it becomes tricky to define security in the presence of expression-level specification.

In Chapter 6, we present how to decompose a transient policy in a sound and complete way. However, the policy specification is more restrictive than one used in Dynamic Release for simplicity. In particular, the specification only supports one single security event in the trigger and all security events are set at the top level. However, security label in Figure 5.2 allows both security events and program expressions. We plan to develop a static type system to check Dynamic Release in a sound and complete manner as future work.
Chapter 8  |  Conclusion

In this dissertation, we present novel approaches to make classic information flow analysis more practical for real-world applications.

For static policy, we first study the sources of imprecision in existing information flow analysis and propose a flow- and path-sensitive static information flow analysis, consisting of a novel program transformation as well as a dependent security type system that rigorously controls information flow. Compared with existing work, we show that our analysis is strictly more precise than a classic flow-sensitive type system, and it tackles the tricky implicit declassification issue completely at the compile time. The novel design also allows a user to control the analysis precision as desired.

Meanwhile, to reduce annotation burden on programmers to provide the dependent labels required to obtain path-sensitivity, we present the first framework for inferring dependent label used in information flow analysis. The framework models an inference algorithm as an iterative process where each step works on a sound and/or complete derivation of the original constraint set. Evaluation result suggests that our novel algorithms improve performance by orders of magnitude and offers better scalability compared with existing work.

Combining these two components, we believe that our novel analysis makes a case for a lightweight static information flow analysis with improved precision. In practice, with the inference framework and the hybrid inference algorithm, we also remove the major barrier (i.e.,
manually writing down dependent labels) to adopt the analysis.

For dynamic policy, we present the first formalization framework that allows apple-to-apple compassion between various dynamic policies. The comparison sheds light on new insights on existing definitions, such as the difference between transient and persistent policies, as well as motivates Dynamic Release, a new general dynamic policy proposed in this work. We also build a new benchmark for testing and understanding variants of dynamic policies in general. We believe that the contributions are also beneficial for future development of dynamic policies: the semantics framework serves as a roadmap, Dynamic Release clarifies the end-to-end security goal for dynamic policies in general, and the new benchmark can help verify the correctness of a dynamic policy.

To enforce dynamic policies, we propose a decomposition approach that breaks down a dynamic policy into a set of static policies on sub-programs. We show that a transient dynamic policy can be enforced soundly and completely via enforcing a set of static policies on code fragments; the latter is well-studied in the literature. Compared with existing enforcement methods for dynamic policy, the decomposition approach is generally applicable for arbitrary kinds of transient dynamic polices, and is compatible with all existing enforcement methods for static policy.

In summary, we tackle two practical limitations of classic information flow analysis. For static policy, we propose a light-weight approach to improve the precision of the analysis. At the same time, the inference framework on dependent label removes the key barrier that prohibits its potential use. For dynamic policy, we clarify and unify its syntax and semantics in theory and propose a decomposition approach for its enforcement.
Appendix A
Proof on Program Transformation

A.1 Correctness of the Transformation

We first show a few lemmas needed to prove the correctness of the program transformation.

Lemma 8 (Equal Expression) Any transformed expression $e$ evaluates to the same value as in the original program, and the transformation does not introduce fresh variables.

\[ \forall e, A, m, m'. \quad \begin{align*}
    m = m^A & \land \langle e, A \rangle \Rightarrow e \land \langle e, m \rangle \downarrow n \land \langle e, m \rangle \downarrow n' \\
    \Rightarrow n = n' \land \text{Vars}(e) \subseteq A.
\end{align*} \]

Proof. By induction on the structure of the expression $e$:

- Case $e = n$: trivial since $e = n$ and $\text{Vars}(e) = \emptyset$.

- Case $e = x$: trivial since we have $e = A(x)$ by the transformation rule. By the assumption $m = m^A$, we have $m(x) = m^A(x)$. Moreover, $\text{Vars}(e) = A(x) \in A$. 

157
• Case $e = e_1 \text{ op } e_2$: from the transformation, we know that $e$ has the form of $e_1 \text{ op } e_2$. By the induction hypothesis, we have $\langle e_1, m \rangle \Downarrow n_1$, $\langle e_2, m \rangle \Downarrow n_2$, $\langle e_1, m \rangle \Downarrow n'_1$, $\langle e_2, m \rangle \Downarrow n'_2$, and $n_1 = n'_1 \wedge n_2 = n'_2$. Thus, we have $e$ and $e$ evaluate to the same value. Moreover, $\text{Vars}(e) \subseteq \mathcal{A}$ by the induction hypothesis.

□

Lemma 9 (Set-Assignment)

$$\forall \mathcal{A}_1, \mathcal{A}_2, m_1, m_2.
\langle \mathcal{A}_2 := \mathcal{A}_1, m_1 \rangle \Rightarrow \langle \text{skip}, m_2 \rangle
\Rightarrow m_1^{A_1} = m_2^{A_2}$$

Proof. Recall that $\mathcal{A}_2 := \mathcal{A}_1$ is just a shorthand for assigning $v_j \in \mathcal{A}_1$ to $v_i \in \mathcal{A}_2$ for each $v \in \text{Vars}$ when $i \neq j$. Hence, for all $x \in \text{Vars}$ such that $A_1(x) \neq A_2(x)$, we have $m_2^{A_2}(x) = m_1^{A_1}(x)$. For any other variable $x$ (such that $A_1(x) = A_2(x)$), we know its value is not updated. Thus, we have $m_1^{A_1} = m_2^{A_2}$.

□

Proof of Theorem 1: Any transformed program is semantically equivalent to its source:

$$\forall c, \mathcal{C}, m, m', m'', \mathcal{A}, \mathcal{A}'.
\langle c, \mathcal{A} \rangle \Rightarrow \langle c, \mathcal{A}' \rangle \wedge \langle c, m \rangle \Rightarrow \langle \text{skip}, m' \rangle \wedge \langle c, m \rangle \Rightarrow \langle \text{skip}, m' \rangle \wedge m = m^A
\Rightarrow m' = (m')^{A'}.$$  

Proof. By induction on the transformation rules in the form of $\langle c, \mathcal{A} \rangle \Rightarrow \langle c, \mathcal{A}' \rangle$.

• Case $\langle \text{skip}, \mathcal{A} \rangle \Rightarrow \langle \text{skip}, \mathcal{A} \rangle$: trivial since no change is made to the memory $m$ and the active set $\mathcal{A}$. 

158
\( \text{Case } \langle x := e, A \rangle \Rightarrow \langle x := \eta e, A \{ x \mapsto x \} \rangle \): From Lemma 8, we know that given \( \langle e, m \rangle \Downarrow n \) and \( \langle e', m' \rangle \Downarrow n' \), then \( n = n' \). We know from the semantics rule S-AssIGN that \( m'(x) = n \) and \( (m')^A(x) = m(x) = n' \). Thus, we have \( m'(x) = (m')^A(x) \). For all variables other than \( x \), they remain unchanged. Thus, we have \( m' = (m')^A \).

\( \text{Case } \langle [x := e], A \rangle \Rightarrow \langle x := \eta e, A \{ x \mapsto x \} \rangle \): Similar to the previous case.

\( \text{Case } \langle c_1; c_2, A \rangle \Rightarrow \langle c_1; c_2, A_2 \rangle \): By the induction hypothesis, given \( \langle c_1, A \rangle \Rightarrow \langle c_1, A_1 \rangle \), \( \langle c_1, m \rangle \rightarrow^* \langle \text{skip}, m_1 \rangle \) and \( \langle c_1, m \rangle \rightarrow^* \langle \text{skip}, m_1 \rangle \), we have \( m_1 = (m_1)^A \). By the semantics, \( \langle \text{skip}; c_2, m_1 \rangle \rightarrow \langle c_2, m_1 \rangle \rightarrow^* \langle \text{skip}, m_2 \rangle \) and \( \langle \text{skip}; c_2, m_1 \rangle \rightarrow \langle c_2, m_1 \rangle \rightarrow^* \langle \text{skip}, m_2 \rangle \). By the induction hypothesis on \( \langle c_2, A_1 \rangle \Rightarrow \langle c_2, A_2 \rangle \) (given \( m_1 = (m_1)^A \)), we have \( m_2 = (m_2)^A \). Thus, we have \( m' = (m')^A \).

\( \text{Case } \langle \text{if } (e) \text{ then } c_1 \text{ else } c_2, A \rangle \Rightarrow \langle \text{if } (e) \text{ then } (c_1; A_3 := A_1) \text{ else } (c_2; A_3 := A_2), A_3 \rangle \): From Lemma 8, we know \( e \) and \( e \) must evaluate to some value \( n \). Thus, both evaluation must take the same branch. Without losing generality, we consider the case when \( n \neq 0 \).

By the induction hypothesis on the transformation \( \langle c_1, A \rangle \Rightarrow \langle c_1, A_1 \rangle \), we know that given \( \langle c_1, m \rangle \rightarrow^* \langle \text{skip}, m_1 \rangle \) and \( \langle c_1, m \rangle \rightarrow^* \langle \text{skip}, m_1 \rangle \), we have \( m_1 = (m_1)^A \). For the rest of the evaluation \( \langle \text{skip}; A_3 := A_1, m_1 \rangle \rightarrow \langle A_3 := A_1, m_1 \rangle \rightarrow^* \langle \text{skip}, m_3 \rangle \), by Lemma 9, we have \( m_3^A = m_1^A = m_1 \). Thus, we have \( m' = m_1 = (m')^A \).

\( \langle \text{while } (e) c, A \rangle \Rightarrow \langle A_1 := A; \text{while } (e) (c; A_1 := A_2), A_1 \rangle \): The evaluation looks like:

\[
\langle A_1 := A; \text{while } (e) (c; A_1 := A_2), m \rangle
\rightarrow \langle \text{while } (e) (c; A_1 := A_2), m_1 \rangle \rightarrow ...
\]

By Lemma 9, we have \( m_1^A = m^A = m \). We proceed by induction on the number of
iterations being executed in the evaluation:

- Base case: 0 iteration is executed in the original program. It must be true that \( \langle e, m \rangle \downarrow 0 \) and \( m' = m \). Since the transformation requires that \( \langle e, A_1 \rangle \Rightarrow e \), we have \( \langle e, m_1 \rangle \downarrow 0 \) by Lemma 8. Hence, 0 iteration is executed in the transformed program, and \( m' = m_1 \). Hence, \( m'A_1 = m_1'A_1 = m = m' \) in this case.

- Induction case for \( N \) iterations (\( N \geq 1 \)): The evaluation of original \texttt{while} looks like:

\[
\langle \text{while} (e) c, m \rangle \\
\rightarrow \langle \text{if} (e) \text{then} (c; \text{while} (e) c) \text{else skip}, m \rangle \\
\rightarrow \langle c; \text{while} (e) c, m \rangle \\
\rightarrow^* \langle \text{skip; while} (e) c, m_1 \rangle \\
\rightarrow \langle \text{while} (e) c, m_1 \rangle \\
\rightarrow \ldots \text{\(N-1\) iterations...}
\]

The evaluation of the transformed \texttt{while} looks like:

\[
\langle \text{while} (e) (c; A_1 := A_2), m_1 \rangle \\
\rightarrow \langle \text{if} (e) \text{then} (c; A_1 := A_2; \text{while} (e) (c; A_1 := A_2)) \text{else skip}, m_1 \rangle \\
\rightarrow \langle c; A_1 := A_2; \text{while} (e) (c; A_1 := A_2), m_1 \rangle \\
\rightarrow^* \langle \text{skip; A_1 := A_2; while} (e) (c; A_1 := A_2), m_2 \rangle \\
\rightarrow \langle A_1 := A_2; \text{while} (e) (c; A_1 := A_2), m_2 \rangle \\
\rightarrow \langle \text{while} (e) (c; A_1 := A_2), m_3 \rangle \rightarrow \ldots
\]

Here, we know that the transformed program must take the “if” branch due to the
same argument as in the base case. Since $m^A_1 = m$, by (structural) induction hypothesis on $c$ and the transformation rule which requires $\langle c, A_1 \rangle \Rightarrow \langle c, A_2 \rangle$, we have $m_1 = m^A_2$. By Lemma 9, we have $m^A_1 = m^A_2 = m_1$. By the induction hypothesis when the original program runs for $N - 1$ iterations, we have $m^A_1 = m'$ as desired.

\[ \square \]

### A.2 Soundness

We need more definitions before showing the soundness of our analysis. We first use a distinguished label $L$ ("low") to define what is observable to the low observer. Since the lemmas and theorems are valid regardless of what level $L$ is, the propositions proved hold for any label $\ell$ in the security lattice.

As discussed in Section 3.7, we prove the soundness based on the erasure semantics in Figure 3.13. The connection between the erasure and the standard semantics is established by the following lemma:

**Lemma 10**

\[ \forall c, m, m_1, m_2, A. \langle m, c \rangle \rightarrow^* \langle m_1, \text{skip} \rangle \land \langle m, c \rangle \rightarrow_{ER(A)}^* \langle m_2, \text{skip} \rangle \]

\[ \forall \forall v \in A. m_1(v) = m_2(v) \]

**Proof.** We note that the erasure semantics only changes the values of dead variables. Hence, the result is trivial given that the live variable analysis is correct. \[ \square \]

Next, we prove that any well-typed target program under the erasure semantics satisfies the noninterference property. To simplify notation, we will use $\rightarrow$ instead of $\rightarrow_{ER(A)}$ hereafter. To prove soundness, we extend the language syntax and semantics with explicitly marked high
values and commands. Memory is extended to track high values as well. The extension is useful since the low equivalence relation we defined earlier corresponds to the equivalence relation on the marked memories.

By showing the completeness of the extended language, all interesting proof are then conducted on the extended language. To do that, we first prove several useful lemmas, and then show the type system enforces noninterference.

### Extended Language

**Extended syntax** The extended syntax is shown in Figure A.1. We augment memories to map high variables to bracketed results. In a similar way, syntax is augmented to include bracketed results, and bracketed commands. Intuitively, bracketed results represent values from high memory, and bracketed commands represent commands executed in a high pc context (such as in a branch with a high guard).

**Extended Semantics** The operational semantics is augmented to propagate brackets, as shown in Figure A.3, A.4. All rules are extensions to the original grammar except that (S-ASGN) is split into three rules: (S-ASGN1), (S-ASGN2), (S-ASGN3). Moreover, the erasure semantics for the extended language also adds brackets when needed. All rules with brackets work the same way as the normal rules from computational perspective. Brackets are just syntactic markers.
\[
\begin{align*}
\langle [n], m \rangle \Downarrow [n] & \quad \langle e_1, m \rangle \Downarrow [n_1] \quad \langle e_2, m \rangle \Downarrow [n_2] \quad n = n_1 \; \text{op} \; n_2 \\
\langle e_1 \; \text{op} \; e_2, m \rangle \Downarrow [n] & \quad \langle e_1 \; \text{op} \; e_2, m \rangle \Downarrow [n] \\
\langle e_1, m \rangle \Downarrow [n_1] \quad \langle e_2, m \rangle \Downarrow [n_2] & \quad n = n_1 \; \text{op} \; n_2 \\
\langle e_1 \; \text{op} \; e_2, m \rangle \Downarrow [n] & \quad \langle e_1 \; \text{op} \; e_2, m \rangle \Downarrow [n]
\end{align*}
\]

Figure A.3. Extended Semantics: Expressions

\[
\text{erase}(m, x, \eta)(x') = \begin{cases} 
0, & x \in \text{FV}(x') \land x' \not\in \text{L}_A(\eta) \land \mathcal{T}(x', m) \subseteq L \\
[0], & x \in \text{FV}(x') \land x' \not\in \text{L}_A(\eta) \land \mathcal{T}(x', m) \not\subseteq L \\
m(x'), & \text{otherwise}
\end{cases}
\]

\[
\begin{align*}
\text{S-Skip1} & \quad \frac{\langle \text{skip}, m \rangle \rightarrow \langle \text{skip}, m \rangle}{\langle \text{skip}, m \rangle \rightarrow \langle \text{skip}, m \rangle} \\
\text{S-Brack1} & \quad \frac{\langle c, m \rangle \rightarrow \langle c', m' \rangle}{\langle c, m \rangle \rightarrow \langle c', m' \rangle} \\
\text{S-Asgn1} & \quad \frac{\langle e, m \rangle \Downarrow n \quad \mathcal{T}(x, m) \subseteq L \quad m' = m \{ x \mapsto n \}}{\langle x := \eta \; e, m \rangle \rightarrow \langle \text{skip, erase}(m', x, \eta) \rangle} \\
\text{S-Asgn2} & \quad \frac{\langle e, m \rangle \Downarrow n \quad \mathcal{T}(x, m) \not\subseteq L \quad m' = m \{ x \mapsto [n] \}}{\langle x := \eta \; e, m \rangle \rightarrow \langle \text{skip, erase}(m', x, \eta) \rangle} \\
\text{S-Asgn3} & \quad \frac{\langle e, m \rangle \Downarrow [n] \quad m' = m \{ x \mapsto [n] \}}{\langle x := \eta \; e, m \rangle \rightarrow \langle \text{skip, erase}(m', x, \eta) \rangle} \\
\text{S-If1} & \quad \frac{\langle e, m \rangle \Downarrow [n] \quad n \not= 0}{\langle \text{if} \; (e) \; \text{then} \; c_1 \; \text{else} \; c_2, m \rangle \rightarrow \langle [c_1], m \rangle} \\
\text{S-If2} & \quad \frac{\langle e, m \rangle \Downarrow [n] \quad n = 0}{\langle \text{if} \; (e) \; \text{then} \; c_1 \; \text{else} \; c_2, m \rangle \rightarrow \langle [c_2], m \rangle}
\end{align*}
\]

Figure A.4. Extended Semantics: Commands
The extended language requires extra typing rules, shown in Figure A.5. Rule (T-BRACKETEXP) treats bracketed expression as high. Note that this rule requires a security level $\ell$, which by definition cannot depend on any program state. Rule (T-BRACKETCMD) in Figure A.6 is given to support the soundness proof. Bracketed command should be type-checked under a pc label that is not bounded by $L$ in the type system.

**Equivalence on Memories and Commands** We define the equivalence of memories and commands up to label $L$ as in Fig. A.2. Intuitively, bracketed memory and commands are indistinguishable. An equivalence relation $\sim$ is defined on memories such that $m_1 \sim m_2$ if and only if they agree on all low variables, and the rest of variables are all high (with brackets).

The type system enforces an important invariant on the memory: a variable holds bracketed value if and only if the security level of that variable is high. This is formalized as follows:

**Definition 36 (Well-Formedness)** A variable $x$ is well-formed under memory $m$, denoted as $\vdash_m x$ if the following condition holds:

$$T(x, m) \nRightarrow L \iff \exists n. (x, m) \Downarrow [n]$$

A memory $m$ is well-formed, denoted as $\vdash m$ if all variables are well formed under $m$:

$$\vdash m \iff (\forall x \in m. \vdash_m x)$$

**Completeness of the Extended Language** It is then clear that for any two low-equivalent standard memories $m_1, m_2$, there are augmented memories, simply by putting brackets for high
variables, that agree with standard memories on all low variables; and vise versa. Hence, the completeness result (Lemma 11) justifies the noninterference result in the unextended language, by showing that starting from any $m_1 \sim m_2$ that are both well-formed, the resulting memories are still equivalent in the augmented language.

Completeness means that every step in the new semantics can be performed in the unextended semantics (maybe with removal of brackets) and vice versa. More formally, given that $c$ is a command in the extended language, let us use the notation of $\lfloor c \rfloor$ to denote removal of all brackets from $c$ in the obvious way, yielding a command from the original language. Similarly, we define $\lfloor m \rfloor$ to convert memory. Completeness can be expressed as the following lemma.

**Lemma 11 (Completeness of the Extended Language)**

$$
\Gamma, pc \vdash c \land \langle \lfloor c \rfloor, \lfloor m \rfloor \rangle \Rightarrow^* \langle \text{skip}, m' \rangle 
\Rightarrow \exists m''. \langle c, m \rangle \Rightarrow^* \langle \text{skip}, m'' \rangle \land m' = \lfloor m'' \rfloor
$$

**Proof.** By rule induction on each evaluation step.

---

**Soundness Proof**

Next, we prove the soundness of the type system on the extended language language with the erasure semantics. We first introduce a couple of useful lemmas.

**Lemma 12** The type comparison is conservative:

$$
\forall m, \tau_1, \tau_2. \tau_1 \sqsubseteq \tau_2 \Rightarrow \mathcal{V}(\tau_1, m) \sqsubseteq \mathcal{V}(\tau_2, m)
$$

**Proof.** Clear from the lifted definition of $\sqsubseteq$ on (dependent) labels.

**Lemma 13** Low expressions always evaluate to ordinary integers (without brackets) under well-formed memory:

$$
\vdash m \land \Gamma \vdash e : \tau \land \mathcal{V}(\tau, m) \sqsubseteq L \Rightarrow \exists n. \langle e, m \rangle \downarrow n
$$
Proof. By induction on the structure of the expression $e$:

- Case $e = n$: trivial.

- Case $e = [n]$: contradiction to the typing rule (T-BRACKETEXP).

- Case $e = x$: $x$ could be either low or high.
  
  - Case $x$ is low ($T(x, m) \subseteq L$): clear from the definition of $\vdash m$.
  
  - Case $x$ is high ($T(x, m) \not\subseteq L$): contradiction to the assumption $V(\tau, m) \subseteq L$.

- Case $e = e_1 \text{ op } e_2$: From $\Gamma \vdash e : \tau$, we can infer that $\Gamma \vdash e_1 : \tau_1$, $\Gamma \vdash e_2 : \tau_2$ and $\tau = \tau_1 \sqcup \tau_2$. By Lemma 12, $V(\tau_1, m) \subseteq L$ and $V(\tau_2, m) \subseteq L$. Hence by induction hypothesis, $\exists n_1, n_2. \langle e_1, m \rangle \downarrow n_1, \langle e_2, m \rangle \downarrow n_2$. Thus, $\langle e, m \rangle \downarrow n$, where $n = n_1 \text{ op } n_2$ by the semantics.

\[\square\]

Lemma 14 (PC Subsumption)

\[\Gamma, pc \vdash c \wedge pc' \subseteq pc \implies \Gamma, pc' \vdash c\]

Proof. By rule induction on the typing derivation for $c$:

- Case skip: From typing rule (T-SKIP), we know that $\Gamma, pc' \vdash \text{skip}$ for any $pc'$.

- Case $x := \eta e$: From typing rule (T-ASSIGN), we know that $\Gamma \vdash e : \tau$ and $\models P(\bullet \eta) \Rightarrow \tau \sqcup pc \subseteq \Gamma(x)$ and $\forall v \in L_A(\bullet \eta). x \not\in \text{FV}(\Gamma(v))$. Since $pc' \subseteq pc$, we have $\tau \sqcup pc' \subseteq \tau \sqcup pc \subseteq \Gamma(x)$. Thus, we have $\models P(\bullet \eta) \Rightarrow \tau \sqcup pc' \subseteq \Gamma(x)$ and there is no change to the other conditions. So we can derive $\Gamma, pc' \vdash x := \eta e$.

- Case $c_1; c_2$: From typing rule (T-SEQ), we know that $\Gamma, pc \vdash c_1$ and $\Gamma, pc \vdash c_2$. By the induction hypothesis, we have $\Gamma, pc' \vdash c_1$ and $\Gamma, pc' \vdash c_2$. Thus, we can derive $\Gamma, pc' \vdash c_1; c_2$. 

166
• Case if (e) then c₁ else c₂: From typing rule (T-If), we know that \( \Gamma \vdash e : \tau \), \( \Gamma, pc \sqcup \tau \vdash c_1 \) and \( \Gamma, pc \sqcup \tau \vdash c_2 \). Since \( pc' \sqsubseteq pc \), we have \( \tau \sqcup pc' \sqsubseteq \tau \sqcup pc \). So by the induction hypothesis, we have \( \Gamma, pc' \sqcup \tau \vdash c_1 \) and \( \Gamma, pc' \sqcup \tau \vdash c_2 \). Hence, \( \Gamma, pc' \vdash \text{if} (e) \text{ then } c_1 \text{ else } c_2 \).

• Case while (e) c: From typing rule (T-While), we know that \( \Gamma \vdash e : \tau \), \( \Gamma, pc \sqcup \tau \vdash c \). Since \( pc' \sqsubseteq pc \), we have \( \tau \sqcup pc' \sqsubseteq \tau \sqcup pc \). So by the induction hypothesis, we have \( \Gamma, pc' \sqcup \tau \vdash c \). Hence, \( \Gamma, pc' \vdash \text{while} (e) \text{ c} \).

\[\square\]

Lemma 15 (Preservation)

\[\Gamma, pc \vdash c \land \langle c, m_1 \rangle \rightarrow \langle c', m_2 \rangle \land \vdash m_1 \]

\[\implies \vdash m_2 \land \Gamma, pc \vdash c' \]

Proof. By rule induction on the evaluation rules \( \langle c, m_1 \rangle \rightarrow \langle c', m_2 \rangle \):

• Case \( \langle \text{skip}, m_1 \rangle \rightarrow \langle \text{skip}, m_1 \rangle \): trivial.

• Case \( \langle \text{skip}; c, m_1 \rangle \rightarrow \langle c, m_1 \rangle \): Trivial since \( m \) does not change and \( \Gamma, pc \vdash c \) is required in rule (T-SEQ).

• Case \( \langle c_1; c_2, m_1 \rangle \rightarrow \langle c'_1; c_2, m_2 \rangle \): From the assumption, we have \( \langle c_1, m_1 \rangle \rightarrow \langle c'_1, m_2 \rangle \). From typing rule (T-SEQ), we have \( \Gamma, pc \vdash c_i, i \in \{1, 2\} \). So by the induction hypothesis, we have \( \Gamma, pc \vdash c'_1 \) and \( \vdash m_2 \). Hence we can derive \( \Gamma, pc \vdash c'_1; c_2 \).

• Case \( \langle [c], m_1 \rangle \rightarrow \langle [c'], m_2 \rangle \): From typing rule (T-BRACKETCMD), we know that there exists some label \( \tau \) such that \( \tau \not\sqsubseteq L \land pc \sqsubseteq \tau \) and \( \Gamma, \tau \vdash c \). From the assumption, we know that \( \langle c, m_1 \rangle \rightarrow \langle c', m_2 \rangle \). By induction hypothesis, we have \( \Gamma, \tau \vdash c' \) and \( \vdash m_2 \). Thus, we can derive \( \Gamma, pc \vdash [c'] \).

167
• Case \( \langle x :=_{\eta} e, m_1 \rangle \to \langle \text{skip}, \text{erase}(m', x, \eta) \rangle \): We have \( \Gamma, pc \vdash \text{skip} \) trivially. Next, we prove that \( \vdash m_2 \) (in this case, \( m_2 \) is \( \text{erase}(m', x, \eta) \)) by showing that every variable \( x' \) is well-formed under \( m_2 \). To do so, we first note that \( T(x', m_2) = T(x', m') \) since there is no chain of dependence or self dependency.

  – Case \( x \not\in \text{FV}(\Gamma(x')) \): We first show that the level of \( x' \) does not change after the assignment. Since \( x' \) does not depend on \( x \), we have \( T(x', m_1) = T(x', m') \).

    Furthermore, we can infer that \( T(x', m_2) = T(x', m') = T(x', m_1) \).

    By the definition of \( \text{erase} \), we have \( m_2(x') = m'(x') \). So if \( x' \neq x \), we have \( \vdash m_2 x' \) trivially since neither its value (since \( m'(x') = m(x') \)) nor its type is changed after the assignment; otherwise, if \( x' = x \) we have three cases:

    * Case S-Asgn1: We have \( m_2(x) = m'(x) = m_1\{x \mapsto n\}(x) = n \) and \( T(x, m_2) = T(x, m_1) \subseteq L \). Thus, \( \vdash m_2 x \).

    * Case S-Asgn2: We have \( m_2(x) = m'(x) = m_1\{x \mapsto [n]\}(x) = [n] \) and \( T(x, m_2) = T(x, m_1) \not\subseteq L \). Thus, we can derive \( \vdash m_2 x \) since \( x \) is high and is given a bracketed value.

    * Case S-Asgn3: We have \( m_2(x) = m'(x) = m_1\{x \mapsto [n]\}(x) = [n] \). We also know from the typing rule (T-ASSGN) that \( \Gamma \vdash e : \tau, \mid P(\bullet\eta) \Rightarrow \tau \sqcup pc \subseteq \Gamma(x) \). By assumption, \( \langle e, m_1 \rangle \Downarrow [n] \). By Lemma 13, \( \forall (\tau, m_1) \not\subseteq L \). Hence, \( T(x, m_1) \not\subseteq L \) due to Lemma 12. Thus, we have \( T(x, m_2) = T(x, m_1) \not\subseteq L \).

    So \( \vdash m_2 x \) since \( x \) is high and is given a bracketed value.

    Thus, in all three cases, we have \( \vdash m_2 x' \).

  – Case \( x \in \text{FV}(x') \): by the definition of \( \text{erase} \), \( x' \) is erased to 0 or [0] according to \( T(x', m') \). We already showed \( T(x', m') = T(x', m_2) \). Thus, we have \( \vdash m_2 x' \).

• Case \( \langle \text{if} (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle \to \langle c_1, m_1 \rangle \) and \( \langle \text{if} (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle \to \langle c_2, m_1 \rangle : \vdash m_2 \) is trivial since \( m_2 = m_1 \). For types, from the typing rule (T-IF), we
have $\Gamma \vdash e : \tau$, $\Gamma, pc \sqcup \tau \vdash c_1$ and $\Gamma, pc \sqcup \tau \vdash c_2$. Since $pc \sqsubseteq pc \sqcup \tau$, we can derive from Lemma 14 that $\Gamma, pc \vdash c_1$ and $\Gamma, pc \vdash c_2$.

• Case $\langle (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle \rightarrow \langle [c_1], m_1 \rangle$, and $\langle \text{if} (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle \rightarrow \langle [c_2], m_1 \rangle$ is trivial since $m_2 = m_1$. For types, from the typing rule (T-If), we have $\Gamma \vdash e : \tau$, $\Gamma, pc \sqcup \tau \vdash c_1$ and $\Gamma, pc \sqcup \tau \vdash c_2$. From the assumption, we have $\langle e, m_1 \rangle \Downarrow [n]$. By Lemma 13, $\mathcal{V}(\tau, m_1) \not\subseteq L$. Hence, $\mathcal{T}(pc \sqcup \tau, m_1) \not\subseteq L$ due to Lemma 12. Thus, we can derive $\Gamma, pc \vdash [c_1]$, and $\Gamma, pc \vdash [c_2]$ by using $pc \sqcup \tau$ as the “$\tau$” in rule (T-BRACKETCmd).

• Case $\langle \text{while} (e) c, m_1 \rangle \rightarrow \langle \text{if} (e) \text{ then } (\text{while} (e) c) \text{ else } \text{skip}, m_1 \rangle$:

  $\Gamma \vdash m_2$ is trivial since $m_2 = m_1$. By the typing rule (T-WHILE), we have two assumptions $A = \Gamma \vdash e : \tau$ and $B = \Gamma, \tau \sqcup pc \vdash c$. So the program after evaluation can be typed as follows:

  \[
  \begin{array}{c}
  \hline
  A & B \\
  \hline
  \Gamma, \tau \sqcup pc \vdash \text{while} (e) c & \Gamma, \tau \sqcup pc \vdash \text{skip} \\
  \hline
  \Gamma, pc \vdash \text{if} (e) \text{ then } (\text{while} (e) c) \text{ else } \text{skip} \\
  \end{array}
  \]

  \[\square\]

**Lemma 16 (High-Step)** A command that type-checks in a high-pc context only modifies high variables.

$\forall m_1, m_2, c, pc. \mathcal{V}(pc, m_1) \not\subseteq L \land \Gamma, pc \vdash c \land \vdash m_1 \land \langle c, m_1 \rangle \rightarrow \langle c', m_2 \rangle \implies m_1 \sim m_2$

**Proof.** By induction on evaluation rules $\langle c, m_1 \rangle \rightarrow \langle c', m_2 \rangle$:

• Cases $\langle [\text{skip}], m_1 \rangle \rightarrow \langle \text{skip}, m_1 \rangle, \langle \text{skip}; c, m_1 \rangle \rightarrow \langle c, m_1 \rangle, \langle \text{if} (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle \rightarrow \langle c_1, m_1 \rangle, \langle \text{if} (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle \rightarrow \langle c_2, m_1 \rangle,$
\[
\begin{align*}
\langle \text{if } (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle & \rightarrow \langle [c_1], m_1 \rangle, \\
\langle \text{if } (e) \text{ then } c_1 \text{ else } c_2, m_1 \rangle & \rightarrow \langle [c_2], m_1 \rangle, \\
\langle \text{while } (e) \, c, m_1 \rangle & \rightarrow \langle \text{if } (e) \text{ then } (\text{while } (e) \, c) \text{ else } \text{skip}, m_1 \rangle:
\end{align*}
\]

Trivial since \( m_2 = m_1 \).

- Case \( \langle c_1; c_2, m_1 \rangle \rightarrow \langle c'_1; c_2, m_2 \rangle \) : From the type rule (T-SEQ), we have \( \Gamma, pc \vdash c_1 \). By induction hypothesis on the assumption \( \langle c_1, m_1 \rangle \rightarrow \langle c'_1, m_2 \rangle \), we have \( m_1 \sim m_2 \).

- Case \( \langle [c], m_1 \rangle \rightarrow \langle [c'], m_2 \rangle \) : From the evaluation rule, we have \( \langle c, m_1 \rangle \rightarrow \langle c', m_2 \rangle \).

From the typing rule, we have \( \Gamma, \tau \vdash c \) for some \( \tau \) such that \( pc \subseteq \tau \). Since by assumption, \( \mathcal{V}(pc, m_1) \not\subseteq L \), we have \( \mathcal{V}(\tau, m_1) \not\subseteq L \) by Lemma 12. Hence, by induction hypothesis on the evaluation assumption, we have \( m_1 \sim m_2 \).

- Case \( \langle x := y \, e, m_1 \rangle \rightarrow \langle \text{skip}, \text{erase}(m', x) \rangle \) : First, we can infer from the type rule that \( \Gamma \vdash e : \tau, \models \mathcal{P}(\bullet y) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(x) \). Due to the correctness of predicate generator, we have \( \tau \sqcup pc \sqsubseteq \Gamma(x) \). Since \( \mathcal{V}(pc, m_1) \not\subseteq L \), we have \( \mathcal{T}(x, m_1) \not\subseteq L \) by Lemma 12. From assumption \( \vdash m_1 \), we know \( x \) must hold a bracketed value in \( m_1 \). We also know that S-Asgn1 can not be applied since it requires \( \mathcal{T}(x, m_1) \subseteq L \). When S-Asgn2 or S-Asgn3 is applied, we have \( m' = m_1 \{ x \mapsto [n] \} \). Since there is no self-dependence, \( m_2(x) = m'(x) = [n] \). So \( m_2(x) \sim m_1(x) \). Next, we show that for variable \( x' \neq x \), we have \( m_1(x') \sim m_2(x') \):

  - Case \( x \not\in \text{FV}(x') \): by definition of \text{erase}, \( m_2(x') = m'(x') = m_1(x') \). Hence, \( m_1(x') \sim m_2(x') \).

  - Case \( x \in \text{FV}(x') \): by definition of \text{erase}, \( x' \) is erased to 0 or \([0]\) according to \( \mathcal{T}(x', m') \). We already know that \( \mathcal{T}(x, m_1) \not\subseteq L \). Since there is no self-dependence, \( \mathcal{T}(x, m') \not\subseteq L \). Recall that \( x \in \text{FV}(x') \) implies \( \Gamma(x) \subseteq \Gamma(x') \). Hence, by Lemma 12, \( \mathcal{T}(x', m_1) \not\subseteq L \) and \( \mathcal{T}(x', m') \not\subseteq L \). By assumption \( \vdash m_1 \), \( x' \) holds a bracketed value in \( m_1 \). Moreover, by the erasure semantics, \( x' \) is erased to \([0]\) after the assignment. So \( m_1(x') \sim m_2(x') \).
Lemma 17

\[ \forall m_1, m_2, m_1 \sim m_2 \land \langle e, m_1 \rangle \downarrow v_1 \implies \exists v_2. \langle e, m_2 \rangle \downarrow v_2 \land v_1 \sim v_2 \]

**Proof.** By rule induction on the structure of expression \( e \).

- Case \( e = n \): trivial since we have \( \langle e, m \rangle \downarrow n \) for any memory, so \( v_1 = v_2 = n \).
- Case \( e = [n] \): trivial since we have \( \langle e, m \rangle \downarrow [n] \) for any memory, so \( v_1 = v_2 = [n] \).
- Case \( e = x \): From \( m_1 \sim m_2 \), we have \( m_1(x) = m_2(x) \), thus, \( v_1 \sim v_2 \).
- Case \( e = e_1 \text{ op } e_2 \): By the induction hypothesis, we have \( \langle e_1, m_1 \rangle \downarrow v_3, \langle e_1, m_2 \rangle \downarrow v_3', \langle e_2, m_1 \rangle \downarrow v_4, \langle e_2, m_2 \rangle \downarrow v_4' \), and \( v_3 \sim v_3', v_4 \sim v_4' \).
  - If both \( v_3, v_4 \) are non-bracketed values, then \( v_3 = v_3' \) and \( v_4 = v_4' \). Result is trivial.
  - If at least one of \( v_3 \) and \( v_4 \) holds bracketed value, we know that at least one of \( v_3' \) and \( v_4' \) holds a bracketed value. From the semantics we know \( e \) must be evaluated to bracketed values under both \( m_1 \) and \( m_2 \). Thus, \( v_1 \sim v_2 \).

□

Lemma 18 (Unwinding)

\[ \forall c_1, c_2, m_1, m_2, m_3, m_4. \]

\[ \vdash c_1 \land c_2 \land c_1 \sim c_2 \land \vdash m_1 \land \vdash m_2 \land m_1 \sim m_2 \]

\[ \land \langle c_1, m_1 \rangle \rightarrow \langle c_3, m_3 \rangle \]

\[ \implies (\exists c_4, m_4. c_4 \sim c_3 \land \langle c_2, m_2 \rangle \rightarrow^* \langle c_4, m_4 \rangle \land m_3 \sim m_4) \lor (\langle c_2, m_2 \rangle \uparrow \land \exists c. c_2 = [c]) \]

**Proof.** By rule induction on \( \langle c_1, m_1 \rangle \rightarrow \langle c_3, m_3 \rangle \).

171
• Case \(\langle \text{skip}, m_1 \rangle \to \langle \text{skip}, m_4 \rangle\): From \(c_1 \sim c_2\), we know \(c_2\) has the form of \([c_3]\), for some \(c_3\). If \(c_2\) diverges, we are done with \(c = c_5\). Otherwise, we have \(\langle [c_3], m_2 \rangle \to^* \langle [\text{skip}], m_4 \rangle \to \langle \text{skip}, m_4 \rangle\). By induction on the number of steps using Lemma 16, we have \(m_4 \sim m_2 \sim m_1\). So we can choose \(c_4 = \text{skip}\).

• Case \(\langle [c_5], m_1 \rangle \to \langle [c'_5], m_3 \rangle\): From \(c_1 \sim c_2\), we know \(c_2\) has the form of \([c_6]\) for some \(c_6\). Hence, \(c_2 = [c_6] \sim [c'_5]\). Moreover, by Lemma 16, \(m_3 \sim m_1\). Hence, \(m_2 \sim m_3\). Therefore, we can choose \(c_4 = c_2\) and make zero step under \(m_2\).

• Case \(\langle \text{skip}; c_5, m_1 \rangle \to \langle c_5, m_1 \rangle\): Command \(c_2\) must also have the form of \(\text{skip}; c_6\) where \(c_5 \sim c_6\). So \(\langle \text{skip}; c_6, m_2 \rangle \to \langle c_6, m_2 \rangle\) preserves the equivalence on memory and command as required.

• Case \(\langle c_5; c_6, m_1 \rangle \to \langle c'_5; c_6, m_3 \rangle\): Command \(c_2\) must have the form of \(c_7; c_8\) where \(c_5 \sim c_7, c_6 \sim c_8\). We can infer from \(c_5 \sim c_7\) and the evaluation assumption \(\langle c_5, m_1 \rangle \to \langle c'_5, m_3 \rangle\) that \(c_7, m_2 \to^* \langle c'_7, m_5 \rangle\) and \(m_5 \sim m_3\) and \(c'_7 \sim c'_5\). Thus, we can derive \(\langle c_7; c_8, m_2 \rangle \to^* \langle c'_7; c_8, m_5 \rangle\). Therefore, we can choose \(c_4 = c'_7; c_8\) and \(m_4 = m_5\).

• Case \(\langle x :=_\eta e, m_1 \rangle \to \langle \text{skip}, \text{erase}(m'_1, x, \eta) \rangle\): From \(c_1 \sim c_2\), we know \(c_2 = (x :=_\eta e)\), and \(\langle x :=_\eta e, m_2 \rangle \to \langle \text{skip}, \text{erase}(m'_2, x, \eta) \rangle\). Trivially, \(\text{skip} \sim \text{skip}\). Next, we show \(\text{erase}(m'_1, x, \eta) \sim \text{erase}(m'_2, x, \eta)\).

From Lemma 17, we know that \(\langle e, m_1 \rangle \downarrow v_1, \langle e, m_2 \rangle \downarrow v_2\), then \(v_1 \sim v_2\). We know \(m'_1 = m_1\{x \mapsto v_1\}\) and \(m'_2 = m_2\{x \mapsto v_2\}\). Thus, \(m'_1 \sim m'_2\) given \(m_1 \sim m_2\). Since there is no self-dependence, we have \(m_3(x) = m'_1(x) \sim m'_2(x) = m_4(x)\). Next, we show for any \(x' \neq x\), we have \(m_3(x') \sim m_4(x')\):

- Case \(x \not\in \text{FV}(x')\): by definition of \(\text{erase}\), we have \(m_3(x') = m'_1(x') \sim m'_2(x') = m_4(x')\).

- Case \(x \in \text{FV}(x')\): by definition of \(\text{erase}\), \(x'\) is erased to 0 or \([0]\) according to its type \(\mathcal{T}(x', m'_1)\) or \(\mathcal{T}(x', m'_2)\).
* Case S-Asgn1, we have \( \langle e, m_1 \rangle \downarrow n, \mathcal{T}(x, m_1) \not\subseteq L \) and \( m'_1(x) = n \). By Lemma 17 and 13, \( \langle e, m_2 \rangle \downarrow n, \mathcal{T}(x, m_2) \not\subseteq L \). So S-Asgn1 applies under \( m_2 \) as well, and \( m'_2(x) = n \cap m'_1 \sim m'_2 \). When \( x' \) depends on no variable whose level is low under \( m_1 \), we have \( \mathcal{T}(x', m'_1) = \mathcal{T}(x', m'_2) \) since all variables it depends on must be identical under \( m'_1 \) and \( m'_2 \). So \( x' \) will be erased to the same value in this case. Otherwise, say \( x' \) depends on \( y \) such that \( \mathcal{T}(y, m'_1) \not\subseteq L \).

By Lemma 13, \( y \) has a bracketed value under \( m'_1 \). Since \( m'_1 \sim m'_2 \), \( y \) has a bracketed value under \( m'_2 \) as well. By Lemma 13 again, \( \mathcal{T}(y, m'_2) \not\subseteq L \).

Since \( \Gamma(y) \not\subseteq \Gamma(x') \) when \( x' \) depends on \( y \), we have \( \mathcal{T}(x', m'_1) \not\subseteq L \) and \( \mathcal{T}(x', m'_2) \not\subseteq L \) by Lemma 12. Hence, \( x' \) will be erased to \([0]\) under \( m'_1 \) and \( m'_2 \).

* Case S-Asgn2: we have \( \mathcal{T}(x, m_1) \not\subseteq L \). Since \( m_1 \sim m_2 \), we have \( \mathcal{T}(x, m_2) \not\subseteq L \) by Lemma 17 and 13. So S-Asgn2 applies under \( m_2 \) as well. Since there is no self dependence, we have \( \mathcal{T}(x, m'_1) = \mathcal{T}(x, m_1) \not\subseteq L \) and \( \mathcal{T}(x, m'_2) = \mathcal{T}(x, m_2) \not\subseteq L \). Since \( x' \) depends on \( x \), we must have \( \mathcal{T}(x', m'_1) \not\subseteq L \) and \( \mathcal{T}(x', m'_2) \not\subseteq L \). So by the definition of erase, \( m_3(x') = m_4(x') = [0] \).

* Case S-Asgn3: we have \( \langle e, m_1 \rangle \downarrow [n_1] \) for some \( n_1 \). By Lemma 17, \( v_2 = [n_2] \) for some \( n_2 \). So S-Asgn3 applies under \( m_2 \) as well. Moreover, given \( \Gamma \vdash e : \tau \), \( \mathcal{V}(\tau, m_1) \not\subseteq L \) and \( \mathcal{V}(\tau, m_2) \not\subseteq L \) by Lemma 13. We also know from typing rule T-Assgn that \( \models \mathcal{P}(\bullet\eta) \Rightarrow \tau \sqcup pc \subseteq \Gamma(x) \). Due to the correctness of predicate generation, \( \tau \sqcup pc \subseteq \Gamma(x) \). By Lemma 12, \( \mathcal{T}(x, m_1) \not\subseteq L \land \mathcal{T}(x, m_2) \not\subseteq L \). Similar to case S-Asgn2, we know that \( m_3(x') = m_4(x') = [0] \) in this case.

* Case \( \langle \text{if} (e) \text{ then } c_5 \text{ else } c_6, m_1 \rangle \rightarrow \langle c_5, m_1 \rangle \) and \( \langle \text{if} (e) \text{ then } c_5 \text{ else } c_6, m_1 \rangle \rightarrow \langle c_6, m_1 \rangle \): From \( c_1 \sim c_2 \), we know \( c_2 \) must be \( \text{if} (e) \text{ then } c_5 \text{ else } c_6 \). This rule is applied only when \( e \)'s value under \( m_1 \) is not bracketed. By Lemma 17, \( e \)'s value is not bracketed
under $m_2$ and $e$ must evaluate to the same value under $m_1$ and $m_2$. Therefore, $c_1$ and $c_2$ must evaluate using the same rule. We can construct $c_4$ as the corresponding branch taken under $m_1$.

- Case $\langle \text{if } (e) \text{ then } c_5 \text{ else } c_6, m_1 \rangle \rightarrow \langle [c_5], m_1 \rangle$ and $\langle \text{if } (e) \text{ then } c_5 \text{ else } c_6, m_1 \rangle \rightarrow \langle [c_6], m_1 \rangle$. From $c_1 \sim c_2$, we know $c_2$ must be $\text{if } (e) \text{ then } c_5 \text{ else } c_6$. This rule is applied when $\langle e, m_1 \rangle \Downarrow [n_1]$. By Lemma 17, we know that $\langle e, m_2 \rangle \Downarrow [n_2]$. We construct $c_4$ as the branch taken under $m_2$ in one step. According to the semantics, $c_4$ evaluates to either $[c_5]$ or $[c_6]$. That is, both $c_3$ and $c_4$ evaluate to bracket commands. Hence, we have $c_3 \sim c_4$.

- Case $\langle \text{while } (e) \text{ } c_5, m_1 \rangle \rightarrow \langle \text{if } (e) \text{ then } (\text{while } (e) \text{ } c_5) \text{ else } \text{skip}, m_1 \rangle$: From $c_1 \sim c_2$, we know $c_2$ has the form of $\text{while } (e) \text{ } c_5$. Therefore, we construct $c_4$ as $\text{if } (e) \text{ then } (\text{while } (e) \text{ } c_5) \text{ else } \text{skip}$. Hence, $m_2 \sim m_4$ and $c_3 \sim c_4$.

\[ \square \]

**Theorem 11 (Soundness under Erasure Semantics)** Any transforms program that type checks satisfies noninterference:

\[ \forall \ell, m_1, m_2, m_3, m_4, \Gamma, \ell', A'. \]

\[ \Gamma \vdash c \land m_1 \approx^\ell m_2 \land \langle c, m_1 \rangle \rightarrow_{ER(A')}^* \langle \text{skip}, m_3 \rangle \land \langle c, m_2 \rangle \rightarrow_{ER(A')}^* \langle \text{skip}, m_4 \rangle \]

\[ \implies m'_1 \approx^\ell m'_2 \]

**Proof.** By the construction of the extended language, for any particular $\ell$, the relation $m_1 \approx^\ell m_2$ is the same as $\vdash m_1, \vdash m_2, m_1 \sim m_2$ in the extended language. We proceed by induction on the number of steps in the execution under $m_1$.

Case zero step: trivial since $c$ must be $\text{skip}$. 

174
Case $N + 1$ steps: consider the first step of the execution under $m_1$. By Lemma 18, we know either $c$ diverges under $m_2$ or we can make multiple steps under $m_2$ and $m_3 \sim m_4$. By Lemma 15, we also have $\vdash m_3, \vdash m_4$ and the remaining programs $c_1$ and $c_2$ type-checks. Hence, result is true by the induction hypothesis. □

Proof of Theorem 3

$$\forall c, m_1, m_2, m_3, m_4, \ell, A, A', \Gamma .$$

\[
\langle c, A \rangle \Rightarrow \langle c, A' \rangle \land \Gamma \land \Gamma \vdash c \land m_1^A \approx_{\Gamma, A} m_2^A \\
\land \langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m_3 \rangle \land \langle c, m_2 \rangle \rightarrow^* \langle \text{skip}, m_4 \rangle \\
\Rightarrow m_3^A \approx_{\Gamma, A'} m_4^A
\]

Proof. Trivial by Theorem 11 and Lemma 10, which states that the erasure semantics agrees with the standard semantics on active copies in $A'$. □

Proof of Theorem 2

$$\forall c, c, m_1, m_2, m_1', m_2', \ell, \Gamma, A, A' .$$

\[
\langle c, A \rangle \Rightarrow \langle c, A' \rangle \land \Gamma \land \Gamma \vdash c \land m_1 \approx_{\Gamma, A} m_2 \land \\
\langle c, m_1 \rangle \rightarrow^* \langle \text{skip}, m_1' \rangle \land \langle c, m_2 \rangle \rightarrow^* \langle \text{skip}, m_2' \rangle \\
\Rightarrow m_1' \approx_{\Gamma, A'} m_2'
\]

Proof. Trivial by Theorem 3 and Theorem 1, the correctness of program transformation. □
A.3 Enabling Flow-Sensitivity with Program Transformation

To facilitate the proof, we say a typing environment \( \Gamma' \) is an extension of \( \Gamma \), written \( \Gamma \preceq \Gamma' \), if \( \forall x \in \text{Dom}(\Gamma). \Gamma(x) = \Gamma'(x) \). Easy to check that this relation is a partial order on environments (i.e., the relation satisfies reflexivity, antisymmetry and transitivity).

Lemma 19

\[
\langle [c], A \rangle \Rightarrow \langle c, A' \rangle \Rightarrow A' \subseteq (A \cup \text{FVars}(c))
\]

Proof. By induction on the structure of \( c \).

- **skip**: \( A' \subseteq A \cup \text{FVars}(c) \) since \( A = A' \) in this case.

- \( x := e \): we have \( A' \subseteq A \cup \text{FVars}(c) \) since \( A' = \{x\} \cup (A - \{A(x)\}) \).

- \( c_1; c_2 \): by the transformation rule we have \( \langle [c_1], A \rangle \Rightarrow \langle c_1, A_1 \rangle \) and \( \langle [c_2], A_1 \rangle \Rightarrow \langle c_2, A' \rangle \) for some \( A_1 \). By the induction hypothesis, we have

\[
A_1 \subseteq (A \cup \text{FVars}(c_1)) \land A' \subseteq (A_1 \cup \text{FVars}(c_2))
\]

Hence, we have

\[
A' \subseteq (A_1 \cup \text{FVars}(c_1)) \cup \text{FVars}(c_2) \subseteq (A \cup \text{FVars}(c_1)) \cup \text{FVars}(c_2)
\]

Therefore, \( A' \subseteq (A \cup \text{FVars}(c_1; c_2)) \).

- **if** \( (e) \) then \( c_1 \) else \( c_2 \): by the transformation rule, we have \( \langle e, A \rangle \Rightarrow e \) and \( \langle [c_1], A \rangle \Rightarrow \langle c_i, A_i \rangle, i \in \{1, 2\} \). By the induction hypothesis, we have

\[
A_1 \subseteq (A \cup \text{FVars}(c_1)) \land A_2 \subseteq (A \cup \text{FVars}(c_2))
\]
The transformed branch has the form of $c_i; A_3 := A_i$ where $A_3 = \Phi(A_1, A_2)$. By the definition of $\Phi$, $A_3 \subseteq A_1 \cup A_2 \cup FVars(A_3 := A_1; A_3 := A_2) \subseteq A \cup FVars(c_i).

• while $(e) c$: by the transformation rule, we have

$$\langle [c], A \rangle \Rightarrow \langle c_1, A_1 \rangle \Rightarrow \langle c', A_2 \rangle \Rightarrow \langle e, A_1 \rangle \Rightarrow c$$

$$\langle \textbf{while} (e) c, A \rangle \Rightarrow \langle A_1 := A; \textbf{while} (c') (c'; A_1 := A_2), A_1 \rangle$$

By induction on $\langle [c], A \rangle \Rightarrow \langle c_1, A_1 \rangle$, we have $A_1 \subseteq A \cup FVars(c_1)$. For a variable $v$ in $A_1$ but not in $A$, we know that $v \in FVars(A_1 := A)$ since by definition, $A_1 := A$ assigns to any $v \in A_1 \land v \notin A$. Hence, $A_1 \subseteq A \cup FVars(c_i)$.

□

The next lemma shows that the extension of a typing environment $\Gamma$ subsumes $\Gamma'$:

**Lemma 20**

$$\Gamma \preceq \Gamma' \land \Gamma \vdash c \Rightarrow \Gamma' \vdash c$$

**Proof.** We first show $\Gamma \preceq \Gamma' \land \Gamma \vdash e : \tau \Rightarrow \Gamma' \vdash e : \tau$ by induction on the structure of $e$. Then we can prove the lemma by induction on the structure of $c$. □

**Proof of Lemma 2**

$$pc \vdash_{\text{HS}} \Gamma \{ c \} \Gamma' \land \langle [c], A \rangle \Rightarrow \langle c, A' \rangle \Rightarrow \forall v \in Vars. (A(v) = A'(v)) \Rightarrow (\Gamma(v) = \Gamma'(v))$$

**Proof.** By induction on the structure of $c$.

• skip: trivial since $\Gamma = \Gamma'$.
• \( x := e \): when \( v = x \), \( \mathcal{A}'(v) = x_i \neq A(x) \) since \( x_i \) is fresh. So the result is trivially true.

For other variables, \( \Gamma(v) = \Gamma'(v) \) by the HS typing rule (HS-ASSIGN).

• \( c_1; c_2 \): by the transformation rule and HS typing rule, we have \( \langle \llbracket c_1 \rrbracket, \mathcal{A} \rangle \Rightarrow \langle \mathcal{C}_1, A_1 \rangle \) and \( \langle \llbracket c_2 \rrbracket, \mathcal{A} \rangle \Rightarrow \langle \mathcal{C}_2, A' \rangle \) for some \( A_1 \), as well as \( pc \vdash_{HS} \Gamma \{ c_1 \} \Gamma_1 \) and \( pc \vdash_{HS} \Gamma \{ c_2 \} \Gamma' \) for some \( \Gamma_1 \). By Lemma 19, \( \mathcal{A}' \subseteq A_1 \cup \mathcal{FVars}(\mathcal{C}_2) \). So when \( \mathcal{A}(v) = \mathcal{A}'(v) \), it must be true that \( A_1(v) = \mathcal{A}'(v) \) since otherwise, \( \mathcal{A}'(v) \) must be a fresh variable in \( \mathcal{C}_2 \), and hence, cannot appear in \( \mathcal{A} \). Therefore, we have \( \mathcal{A}(v) = \mathcal{A}'(v) = A_1(v) \). By the induction hypothesis, it must be true that \( \Gamma(v) = \Gamma_1(v) = \Gamma'(v) \).

• if \( (e) \) then \( c_1 \) else \( c_2 \): by the HS typing rule, \( pc \vdash_{HS} \Gamma \{ c_1 \} \Gamma_1 \land pc \vdash_{HS} \Gamma \{ c_2 \} \Gamma_2 \land \Gamma' = \Gamma_1 \sqcup \Gamma_2 \). By the transformation rules, \( \langle \llbracket c \rrbracket, \mathcal{A} \rangle \Rightarrow \langle \mathcal{C}_i, A_i \rangle, i \in \{1, 2\} \).

By Lemma 19, \( \mathcal{A}_1 \subseteq \mathcal{A} \cup \mathcal{FVars}(\mathcal{C}_1) \). So when \( \mathcal{A}(v) \neq \mathcal{A}_1(v) \), \( \mathcal{A}_1(v) \) must be a fresh variable generated in \( \mathcal{C}_1 \), and hence, cannot be in \( \mathcal{A}_2 \). By the definition of \( \Phi \), \( \mathcal{A}_3(v) \) must be fresh as well. This contradicts the assumption that \( \mathcal{A}(v) = \mathcal{A}'(v) \). Hence, we have \( \mathcal{A}(v) = \mathcal{A}_1(v) \) (and similarly, \( \mathcal{A}(v) = \mathcal{A}_2(v) \)). So \( \Gamma(v) = \Gamma_1(v) \) by the induction hypothesis. Therefore, \( \Gamma'(v) = \Gamma_1(v) \sqcup \Gamma_2(v) = \Gamma(v) \).

• while \( (e) \) c: By rule (TRSF-WHILE), we have \( \langle \llbracket c \rrbracket, \mathcal{A} \rangle \Rightarrow \langle \mathcal{C}_1, A_1 \rangle \), where \( \mathcal{A}' \) is \( \mathcal{A}_1 \) in this case. Hence, by the assumption, we have \( \mathcal{A}(v) = \mathcal{A}_1(v) \). By rule (HS-WHILE), there is a sequence of environments \( \Gamma'_i, \Gamma''_i \) such that \( pc \sqcup \tau_i \vdash \Gamma'_i \{ c \} \Gamma''_i \). By the induction hypothesis, \( \Gamma''_i(v) = \Gamma'_i(v) \). Since \( \Gamma_0 = \Gamma \) and \( \Gamma_{i+1} = \Gamma_i \sqcup \Gamma''_i \) in rule (HS-WHILE), we can further infer that \( \Gamma_{i+1}(v) = \Gamma_i(v) \). Hence, we have \( \Gamma'(v) = \Gamma_{i+1}(v) = \Gamma_0(v) = \Gamma(v) \).

\[ \square \]

**Lemma 21**

\[ \Gamma \vdash_{HS} e : \tau \land \langle e, \mathcal{A} \rangle \Rightarrow e \land \Gamma = \Gamma^\mathcal{A} \Rightarrow \]

\[ \Gamma \vdash e : \tau \]

178
Proof. By rule induction on the transformation.

Proof of Theorem 4 We prove a lightly stronger version of Theorem 4:

\[
\forall c, \underline{c}, pc, \mathcal{A}, \mathcal{A}', \Gamma, \Gamma', \Gamma_0.
\]

\[
(pc, \Gamma, \mathcal{A})\{[c] \Rightarrow \underline{c}\}(\Gamma', \mathcal{A}') \rightarrow \Gamma \Rightarrow \text{Dom}(\Gamma) \subseteq \mathcal{A} \cup \text{FVars}(\underline{c}) \wedge \Gamma = \Gamma^\mathcal{A} \wedge \Gamma' = \Gamma'^\mathcal{A} \wedge \Gamma_0, pc \vdash \underline{c}
\]

Proof. By induction on the structure of \(c\).

- **skip**: \(\Gamma, pc \vdash \underline{c}\) is trivial since \(\text{skip}\) can be type-checked with any \(pc, \Gamma\). Other conditions are trivial.

- **\(x := e\)**: by the construction rule, we have \(\Gamma'(x) = \tau\) for some \(\tau, \underline{c}\) has the form of \(x_i := \underline{c}, \mathcal{A}'(x) = x_i\) and \(\Gamma = \Gamma_0 \cup \{x_i \mapsto \tau\}\).
  
  - \(\Gamma = \Gamma^\mathcal{A}, \Gamma' = \Gamma'^\mathcal{A}\): Since \(x_i\) is fresh, \(\Gamma = \Gamma^\mathcal{A}\). Moreover, \(\Gamma'^\mathcal{A} = \Gamma\{x_i \mapsto \tau\} = \Gamma'\).
  
  - \(\Gamma, pc \vdash \underline{c}\): let \(\Gamma \vdash_{\text{HS}} e : \tau_0\). By the \(\text{HS}\) typing rule, we have \(\tau = \tau_0 \sqcup pc\). Moreover, we have \(\Gamma \vdash \underline{c} : \tau\) due to Lemma 21 and the fact \(\Gamma = \Gamma^\mathcal{A}\). By the construction, \(\Gamma(x_i) = \tau\). Hence, \(\Gamma, pc \vdash x_i := \underline{c}\).
  
  - \(\text{Dom}(\Gamma) \subseteq \mathcal{A} \cup \text{FVars}(\underline{c})\): By the construction, \(\text{Dom}(\Gamma) = \mathcal{A} \cup \{x_i\}\).

- **\(c_1; c_2\)**: from the construction rule, we have \((pc, \Gamma, \mathcal{A})\{[c_1] \Rightarrow \underline{c}_1\}(\Gamma'', \mathcal{A}'') \rightarrow \Gamma_1\) and \((pc, \Gamma'', \mathcal{A}'')\{[c_2] \Rightarrow \underline{c}_2\}(\Gamma', \mathcal{A}') \rightarrow \Gamma_2\) for some \(\Gamma'', \mathcal{A}'', \Gamma_1\) and \(\Gamma_2\). By the induction hypothesis, we have

  \[
  \Gamma = \Gamma_1^\mathcal{A} \wedge \Gamma'' = \Gamma_1'' \wedge \text{Dom}(\Gamma_1) \subseteq \mathcal{A} \cup \text{FVars}(\underline{c}_1)
  \]

  \[
  \Gamma'' = \Gamma_2'' \wedge \Gamma' = \Gamma_2' \wedge \text{Dom}(\Gamma_2) \subseteq \mathcal{A}'' \cup \text{FVars}(\underline{c}_2)
  \]
\[ \Gamma_1, pc \vdash c_1 \land \Gamma_2, pc \vdash c_2 \]

- \( \Gamma = \Gamma^A \), \( \Gamma' = \Gamma'^A \): By the construction, we have \( \Gamma_1 \preceq \Gamma \). Next, we check \( \Gamma_2 \preceq \Gamma \).

Due to the results above on \( \text{Dom}(\Gamma_1) \) and \( \text{Dom}(\Gamma_2) \), we know that if \( v \in \text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) \), then \( v \in \mathcal{A}'' \) since \( \mathcal{FVars}(c_2) \) contain fresh variables generated in \( c_2 \) (hence, not in \( \mathcal{A} \) and \( \mathcal{C}_1 \)). Therefore, \( \Gamma_1(v) = \Gamma_2(v) \) because \( \Gamma^A_{1''} = \Gamma'' = \Gamma^A_{2''} \) by the induction hypothesis. Therefore, \( \Gamma_2 \preceq \Gamma \).

Moreover, by the definition, \( \Gamma = \Gamma^A_{1} \) is equivalent to \( \Gamma_{\mathcal{A}_1} \preceq \Gamma \). So we have \( \Gamma_{\mathcal{A}_1} \preceq \Gamma \), and hence, \( \Gamma = \Gamma^A \). Similarly, we can prove that \( \Gamma' = \Gamma'^A \).

- \( \mathcal{C} \): we know that \( \Gamma, pc \vdash c_1 \land \Gamma, pc \vdash c_2 \) by Lemma 20. So \( \Gamma, pc \vdash c_1 ; c_2 \).

- \( \text{Dom}(\Gamma) \subseteq \mathcal{A} \cup \mathcal{FVars}(\mathcal{C}) \): by the construction, \( \text{Dom}(\Gamma) = \text{Dom}(\Gamma_1) \cup \text{Dom}(\Gamma_2) \). The result is true since \( \mathcal{A}'' \subseteq \mathcal{A} \cup \mathcal{FVars}(\mathcal{C}_1) \) by Lemma 1 and Lemma 19.

\[ \mathbf{if} \ (e) \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 : \] from the construction rule, we have \( (pc \sqcup \tau, \Gamma, \mathcal{A})\{[c_i] \} (\Gamma_i, \mathcal{A}_i) \leftrightarrow \Gamma_i \) for \( i \in \{1, 2\} \), and \( \Phi(\mathcal{A}_1, \mathcal{A}_2) \Rightarrow \mathcal{A}_3 \), where \( \Gamma \vdash_{\text{HS}} e : \tau \). By the induction hypothesis, we have

\[ \Gamma = \Gamma^A_i \land \Gamma_i = \Gamma^A_{1i} \land \Gamma_i, pc \sqcup \tau \vdash c_i \land \text{Dom}(\Gamma_i) \subseteq \mathcal{A} \cup \mathcal{FVars}(\mathcal{C}_i) \]

- \( \Gamma_1 \preceq \Gamma, \Gamma_2 \preceq \Gamma \): By the construction, we have \( \Gamma_1 \preceq \Gamma \). Next, we check \( \Gamma_2 \preceq \Gamma \).

Since \( \text{Dom}(\Gamma_i) \subseteq \mathcal{A} \cup \mathcal{FVars}(\mathcal{C}_i) \), if \( v \in \text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) \), then it must be true that \( v \in \mathcal{A} \). Since we know that \( \Gamma = \Gamma^A_1 \) and \( \Gamma = \Gamma^A_2 \) from the induction hypothesis, \( \Gamma(v) = \Gamma_1(v) = \Gamma_2(v) \) for such variables. Hence, we have \( \Gamma_2 \preceq \Gamma \).

- \( \Gamma = \Gamma^A, \Gamma' = \Gamma'^A \): We have \( \Gamma = \Gamma^A \) since \( \Gamma = \Gamma^A_1 \) and \( \Gamma_1 \preceq \Gamma \). To check \( \Gamma' = \Gamma'^A \), consider \( v \in \mathcal{A}_3 \cap (\text{Dom}(\Gamma_1) \cup \text{Dom}(\Gamma_2)) \). We use \( v \) to denote \([v]\).

  * When \( v \in \text{Dom}(\Gamma_1) \): we have \( v \in \mathcal{A} \) or \( v \in \mathcal{FVars}(\mathcal{C}_1) \) since \( \text{Dom}(\Gamma_1) \subseteq \mathcal{A} \cup \mathcal{FVars}(\mathcal{C}_1) \).
In the latter case, we have $A_1(v) \neq A_2(v)$ since $A_2 \subseteq A \cup \text{FVars}(\mathcal{E}_2)$ by Lemma 1 and Lemma 19. So $A_3(v)$ must be a fresh variable by the definition of $\Phi$, and hence $v \not\in A_3$. Contradiction.

So $v \in A$. By the assumption that $v \in A_3$ and the definition of the $\Phi$ function, we have $v \in A_1$ and $v \in A_2$. That is, $A(v) = A_1(v) = A_2(v)$. By Lemma 2, we have $\Gamma_1(v) = \Gamma(v) = \Gamma_2(v)$. Hence, $\Gamma'(v) = \Gamma_1(v) \sqcup \Gamma_2(v) = \Gamma(v) = \Gamma(v)$, where the last equation is true since $v \in A \land \Gamma^A = \Gamma$.

* When $v \in \text{Dom}(\Gamma_2)$: same argument as the case above.

Therefore, for any $v \in A_3 \cap (\text{Dom}(\Gamma_1) \cup \text{Dom}(\Gamma_2))$, $\Gamma(v) = \Gamma'(v)$. For other variables, $\Gamma(v) = \Gamma'(v)$ is trivial from the construction. So $\Gamma' = \Gamma^A$.

\[ - \Gamma, pc \vdash c: \text{We know that } \langle e, A \rangle \Rightarrow \mathcal{E} \text{ in the transformation assumption. Given } \Gamma^A = \Gamma, \text{ we have } \Gamma \vdash e : \tau \text{ by Lemma 21. Moreover, } \Gamma, pc \sqcup \tau \vdash \mathcal{E}_2 \text{ since } \Gamma_1 \preceq \Gamma, \Gamma_2 \preceq \Gamma. \text{ Furthermore, both } A_3 := A_1 \text{ and } A_3 := A_2 \text{ type-checks by our construction since: 1) by the definition, the extra assignments only assign to fresh variables in } A_3 \text{ (i.e., } v \not\in A_1 \cap A_2); \text{ 2) for those variables, we have } \Gamma(v) = \Gamma'(v) \text{ by the construction; 3) the HS system ensures that } \Gamma_i(v) \subseteq \Gamma'(v) \text{ for } i \in \{1, 2\} \text{ and } pc \sqcup \tau \subseteq \Gamma_i(v) \text{ for } i = 1 \text{ or } 2 (v \text{ must be assigned to under some branch since } v \not\in A_1 \cap A_2). \text{ Therefore, we have } pc \sqcup \tau \subseteq \Gamma'(v) = \Gamma(v), \text{ and } \Gamma(A_i(v)) = \Gamma_i(v) \subseteq \Gamma'(v) = \Gamma(v). \]

\[ - \text{Dom}(\Gamma) \subseteq A \cup \text{FVars}(\mathcal{E}): \text{ by the construction, } \text{Dom}(\Gamma) = \text{Dom}(\Gamma_1) \cup \text{Dom}(\Gamma_2) \cup A_3. \text{ By Lemma 19, } A_3 \subseteq A \cup \text{FVars}(\mathcal{E}). \text{ Result is true since we also have } \text{Dom}(\Gamma_i) \subseteq A \cup \text{FVars}(\mathcal{E}_i). \]

* While $(e) c$: by the construction rule, we have

\[ \Gamma' \vdash_{\text{HS}} e : \tau, (\tau \sqcup pc, \Gamma', A_1)\{[c] \Rightarrow \mathcal{E}'\}(\Gamma', A_2) \hookrightarrow \Gamma_0 \]
By the induction hypothesis, we have

$$\Gamma' = \Gamma_0^{A_1} \land \Gamma' = \Gamma_0^{A_2} \land \Gamma_0, pc \sqcup \tau \vdash \varepsilon' \land \text{Dom}(\Gamma_0) \subseteq A_1 \cup \text{FVars}(c')$$

- \( \Gamma = \Gamma^A, \Gamma' = \Gamma^{A_1} \): We have \( \Gamma_0 \preceq \Gamma \) by the construction. Given \( \Gamma' = \Gamma_0^{A_1} \), we know that \( \Gamma' = \Gamma^{A_1} \). Next, we show that \( \Gamma = \Gamma^A \).

  * When \( v \in A \cap \text{Dom}(\Gamma_0) \) (we use \( v \) to denote \( [v] \)). Since \( \text{Dom}(\Gamma_0) \subseteq A_1 \cup \text{FVars}(c') \) from the induction hypothesis, we have \( v \in A_1 \) or \( v \) is fresh in \( c' \). But the latter must be false since \( v \in A \). So \( v \in A_1 \).

    From the construction rule assumption, we have \( pc \vdash_{\text{HS}} \Gamma\{\text{while } (e) \ c\} \Gamma' \).

    By Lemma 1, we have \( \langle [[\text{while } (e) \ c]], A \rangle \Rightarrow \langle c, A_i \rangle \). In addition, we just showed that \( v \in A_1 \) and \( v \in A \). Hence, by Lemma 2, \( \Gamma(v) = \Gamma'(v) \).

    Because \( v \in A_1 \) and \( v \in A \), \( A(v) = A_1(v) \). Therefore, \( \Gamma^A(v) = \Gamma(A(v)) = \Gamma(A_1(v)) = \Gamma'(v) \) (since \( \Gamma' = \Gamma^{A_1} \) = \( \Gamma(v) \).

  * When \( v \in A - \text{Dom}(\Gamma_0) \), the result is trivial by the construction.

- \( \Gamma, pc \vdash c \): we have showed that \( \Gamma^{A_1} = \Gamma' \). In the transformation assumption, we have \( \langle e, A_1 \rangle \Rightarrow c \). So by Lemma 21, \( \Gamma, e : \tau \vdash c \). Moreover, we have \( \Gamma, pc \sqcup \tau \vdash c' \) by Lemma 20 and the induction hypothesis \( \Gamma_0, pc \sqcup \tau \vdash c' \). Next, we show that the extra assignments (i.e., \( A_1 := A \) and \( A_1 := A_2 \)) type-check.

    Since \( \Gamma' = \Gamma_0^{A_1} \land \Gamma' = \Gamma_0^{A_2} \) by the induction hypothesis, we have \( \Gamma^{A_1} = \Gamma^{A_2} \). Moreover, by the same argument as in the “if” case, the LHS of \( A_1 := A_2 \) must have a level that is higher than \( pc \sqcup \tau \) in the HS system. So \( \Gamma, pc \sqcup \tau \vdash A_1 := A_2 \).

    Moreover, we know that \( \forall v \in \text{Vars}, \Gamma(v) \subseteq \Gamma'(v) \) in the HS system. Since \( \Gamma^A = \Gamma \) and \( \Gamma^{A_1} = \Gamma' \), we have \( \Gamma, pc \vdash A_1 := A \).

- \( \text{Dom}(\Gamma) \subseteq A \cup \text{FVars}(c) \): by the construction, \( \text{Dom}(\Gamma) = \text{Dom}(\Gamma_0) \cup A \). From the induction hypothesis, we have \( \text{Dom}(\Gamma_0) \subseteq A_1 \cup \text{FVars}(c') \). Hence, \( \text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma) \subseteq A_1 \cup \text{FVars}(c') \).
$A_1 \cup \text{FVars}(\mathcal{L})$. We know from Lemmas 1 and 19 that $A_1 \subseteq A \cup \text{FVars}(\mathcal{L})$. Hence, $\text{Dom}(\Gamma) \subseteq A \cup \text{FVars}(\mathcal{L})$. 

□
Appendix B

Formalizations in Table 5.1

B.1 Paralock

Paralock \[105, 106, 108\] uses locks to formalize the sensitivity of security objects. Paralock uses fine-grained model to encode role-based access control systems. It covers both declassification and revocation of information to a principal in the system. As described in Section 5.3.2, security specification is written as \(\{\Sigma \Rightarrow a; \ldots\}\), where \(\Sigma\) is a lock set and \(a\) is an actor. An actor \(a\) is the base sensitivity entity of the model, which is used to model a lattice level \(L\) in two-point lattice \(\{H, L\}\) in \[108\], and a principal \(p\) in role-base access control system in \[106\].

To formalize Paralock security, an attacker \(A = (a, \Sigma)\) is modeled as an actor \(a\) with a (static) set of open locks \(\Sigma\). To simplify notation, we use \(\Gamma(x, a) = \Sigma\) to denote the fact that \(\{\Sigma \Rightarrow a; \ldots\}\) is part of the security policy of \(x\), otherwise \(\Gamma(x, a) = \top\). With respect to an attacker \(A = (a, \Sigma)\), a variable \(x\) is observable to \(A\) iff \(\Gamma(x, a) \subseteq \Sigma\), meaning that the attacker possesses more opened locks than what’s required in the policy.

To simplified the notations in this work, we extend the output event \(t\) to also record the current open locks. So, for a trace fragment \(\langle c, m \rangle \xrightarrow{b,v,\gamma} \langle c', m' \rangle\), it generates the output event \(t = \langle b, v, \gamma, \Delta \rangle\), where \(\Delta = \text{unlock}(\langle c, m \rangle)\).

Let \(|A|\) be the set of variables that are visible to \(A\), and \([\vec{t}]_A\) be the outputs that are visible
to $A = (a_A, \Sigma_A)$:

$$\|A\| \triangleq \{ x \mid \forall x \in \text{Vars}. \, \Gamma(x, a_A) \subseteq \Sigma_A\}$$

$$\lfloor \vec{t} \rfloor_A \triangleq \lfloor \vec{t} \rfloor_{\lambda b, n, \gamma, \Delta, \Gamma(b, a_A) \subseteq \Sigma_A}$$

Paralock security defines attacker’s knowledge\(^1\) as follows:

$$k_{\text{PL}}(c, m, \vec{t}, A) = \{m' \mid m' \approx_{\|A\|} m$$

$$\land \langle c, m' \rangle \xrightarrow{\vec{t}} \langle c', m'' \rangle \xrightarrow{\vec{t}} \langle c'', m''' \rangle \land \lfloor \vec{t} \rfloor_A = \lfloor \vec{t} \rfloor_A\}$$

Paralock security semantics extends that of gradual release, by treating “unlock” events as releasing events:

**Definition 37 (Paralock Security)** A program $c$ is Paralock secure if for any attacker $A = \langle a, \Sigma \rangle$, the attacker’s knowledge remains unchanged whenever $\text{unlock}(\tau_{[i]}) \subseteq \Sigma_A$:

$$\forall c, m, m', \vec{t}', t', a, \Sigma_A, A, i.$$

$$\langle c, m \rangle \xrightarrow{\vec{t}} \langle c', m' \rangle \xrightarrow{\vec{t}} \langle c'', m''' \rangle \land A = \langle a, \Sigma_A \rangle \land$$

$$\text{unlock}((c'', m''')) \subseteq \Sigma_A$$

$$\Rightarrow k_{\text{PL}}(c, m, \vec{t} \cdot t', A) = k_{\text{PL}}(c, m, \vec{t}, A)$$

We use the memory closure on $A$ for memory that looks the same to attacker $A$:

$$\llbracket m \rrbracket_A \triangleq \{ m' \mid \forall m'. \forall x \in \text{Vars.} \, \Gamma(x, a_A) \subseteq \Sigma_A \Rightarrow m(x) = m'(x)\}$$

\(^1\)We revised the definition for termination insensitivity. We note that Paralock also presents a different termination insensitive policy following ideas from [119]. However, here we follow Gradual Release to define terminated insensitive knowledge by taking a intersection with the initial memory of traces that terminates.
The conversion of Definition 37 to our framework is straightforward.

\[ \sim_{PL} \equiv \{ (\vec{t}_1, \vec{t}_2) \mid \text{\(\vec{t}_1\)} \text{ prefix of } \text{\(\vec{t}_2\)} \} \]

\[ A_{PL} \equiv \begin{cases} \mathcal{K}(c, \vec{t}^{i-1}, \sim_{PL}) \cap [m]_A, & \text{if } \vec{t}^{[i]} \Delta \subseteq \Sigma_A \\ [m]_0, & \text{otherwise} \end{cases} \]

**Lemma 22** With \( \sim_{PL} \) and \( A \equiv A_{PL} \), Definition 25 is equivalent to Definition 37.

### B.2 Equivalence Proof

We first introduce a useful lemma which allows us to rewrite the original knowledge definition in [18] to the knowledge definition \( \mathcal{K} \) in this paper.

**Lemma 23** Let \( k \) be defined as in Equation 5.5.2 and \( \mathcal{K} \) be defined as in Equation 5.5.1, then we have

\[ \forall c, m, L, \Gamma, \vec{t}, M. \]

\[ M \subseteq [m]_{L, \Gamma} \wedge k(c, m, \vec{t}, L, \Gamma) \subseteq M \rightarrow k(c, m, \vec{t}, L, \Gamma) = M \leftrightarrow \mathcal{K}(c, \vec{t}, \sim_{N_1}) \supseteq M \]

**Proof.** By definition, we know

\[ k(c, m, \vec{t}, L, \Gamma) = \mathcal{K}(c, \vec{t}, \sim_{N_1}) \cap [m]_{L, \Gamma} \]

- case \( \rightarrow \): we know

\[ M = k(c, m, \vec{t}, L, \Gamma) = \mathcal{K}(c, \vec{t}, \sim_{N_1}) \cap [m]_{L, \Gamma} \]
Thus, we have $\mathcal{K}(c, \vec{t}, \sim_{NI}) \supseteq M$.

- case $\Leftarrow$: we know

$$
k(c, m, \vec{t}, \Gamma) = \mathcal{K}(c, \vec{t}, \sim_{NI}) \cap \llbracket m \rrbracket_{L, \Gamma}
$$

Thus, we know $k(c, m, \vec{t}, L, \Gamma) \supseteq M$. From assumption $k(c, m, \vec{t}, \Gamma) \subseteq M$, we know $k(c, m, \vec{t}, L, \Gamma) = M$.

So, we have $k(c, m, \vec{t}, L, \Gamma) = M \iff \mathcal{K}(c, \vec{t}, \sim_{NI}) \supseteq M$. □

**Gradual Release**

**Lemma 3.** With $\sim \triangleq \sim_{GR}$ and $A \triangleq A_{GR}$, Definition 25 is equivalent to Definition 26:

$$
\forall c, m, L, i, \vec{t}, (c, m) \leftrightarrow \vec{t} \implies
$$

$$
i \text{ not release event} \implies k(c, m, \vec{t}_{[i]}, L, \Gamma) = k(c, m, \vec{t}_{[i-1]}, L, \Gamma)
$$

$$
\iff \mathcal{K}(c, \vec{t}_{[i]}, \sim_{GR}) \supseteq \llbracket m \rrbracket_{L, \vec{t}_{[i]}, \gamma} \cap \mathcal{K}(c, \vec{t}_{[i-1]}, \sim_{GR})
$$

**Proof.** From the encoding of Gradual Release, we know:

$$
\vec{t}_{[i], \gamma} = \begin{cases} 
\gamma_{\bot}, & \text{i is a release event} \\
\Gamma, & \text{i not a release event}
\end{cases}
$$
• case when \( i \) is a release event: \( \vec{t}_i, \gamma = \gamma_\perp \). From the definition, we know \( \llbracket m \rrbracket_{L, \gamma_\perp} \) returns the singleton set \( \{ m \} \).

From \( \langle c, m \rangle \rightarrow \vec{t} \) and the definition of \( \mathcal{K} \), we know \( \forall j. \ m \in \mathcal{K}(c, \vec{t}_{[j]}, \sim_{GR}) \):

\[
m \in \mathcal{K}(c, \vec{t}_{[i]}, \sim_{GR})
\]

\[
m \in \mathcal{K}(c, \vec{t}_{[i-1]}, \sim_{GR})
\]

\[
\{ m \} = \llbracket m \rrbracket_{L, \vec{t}_{[i]}, \gamma}
\]

Thus, both Definition 25 and 26 are trivially true.

• case when \( i \) is not a release event: \( \vec{t}_0, \gamma = \Gamma \). From the definitions, we know \( \sim_{GR} = \sim_{NI} \).

We know from the monotonicity of the knowledge that:

\[
k(c, m, \vec{t}_{[i-1]}, L, \Gamma) \subseteq \llbracket m \rrbracket_{L, \Gamma}
\]

\[
k(c, m, \vec{t}_{[i]}, L, \Gamma) \subseteq k(c, m, \vec{t}_{[i-1]}, L, \Gamma)
\]

So, we can instantiate Lemma 23 with:

\[
M := k(c, m, \vec{t}_{[i-1]}, L, \Gamma), \quad \vec{t} := \vec{t}_{[i]}
\]

and we get:

\[
k(c, m, \vec{t}_{[i]}, L, \Gamma) = k(c, m, \vec{t}_{[i-1]}, L, \Gamma)
\]

\[\iff \quad \mathcal{K}(c, \vec{t}_{[i]}, \sim_{GR}) \supseteq k(c, m, \vec{t}_{[i-1]}, L, \Gamma)\]

By definition, we know

\[
k(c, m, \vec{t}_{[i-1]}, L, \Gamma) = \llbracket m \rrbracket_{L, \Gamma} \cap \mathcal{K}(c, \vec{t}_{[i-1]}, \sim_{GR})
\]
Thus, when $i$ is not a release event, we have:

$$k(c, m, \vec{t}^{[i]}, L, \Gamma) = k(c, m, \vec{t}^{[i-1]}, L, \Gamma)$$

$$\iff \mathcal{K}(c, \vec{t}^{[i]}, \sim_{GR}) \supseteq [m]_{L, \vec{t}^{[i]}, \gamma} \cap \mathcal{K}(c, \vec{t}^{[i-1]}, \sim_{GR})$$

Therefore, Definition 25 is equivalent to Definition 26. \hfill \square

**Tight Gradual Release**

**Lemma 4.** With $\sim \triangleq \sim_{TGR}$, $\equiv \triangleq \equiv_{TGR}$ and $A \triangleq A_{TGR}$, Definition 25 is equivalent to Definition 27.

$$\forall i. \ 1 \leq i \leq \|\vec{t}\|.$$  

$$( [m]_{L, \Gamma} \cap [m]_{E_i} ) \subseteq k(c, m, \vec{t}^{[i]}, L, \Gamma)$$

$$\iff$$

$$\mathcal{K}(c, \vec{t}^{[i]}, \sim_{TGR}) \supseteq [m]_{L, \vec{t}^{[i]}, \gamma}$$

**Proof.** The encoding limited $E_i$ to a variable set $X_i$, thus, we assumes $E_i = X_i$. From the definition, we know that

$$k(c, m, \vec{t}, L, \Gamma) = \mathcal{K}(c, \vec{t}^{[i]}, \sim_{TGR}) \cap [m]_{L, \Gamma} \quad (B.2.1)$$

From encoding, we know that

$$[m]_{L, \vec{t}^{[i]}, \gamma} = ([m]_{L, \Gamma} \cap [m]_{E_i}) \quad (B.2.2)$$

- case $\implies$: From Equation B.2.2 and the assumption, we know

$$[m]_{L, \vec{t}^{[i]}, \gamma} \subseteq k(c, m, \vec{t}^{[i]}, L, \Gamma)$$
From Equation B.2.1, we know

\[ k(c, m, \vec{t}^{[i]}, L, \Gamma) \subseteq \mathcal{K}(c, \vec{t}^{[i]}, \sim_{TGR}) \]

Therefore, we have \([m]_{L,\vec{t}^{[i]},\gamma} \subseteq \mathcal{K}(c, \vec{t}^{[i]}, \sim_{TGR})\).

• case \(\Leftarrow\): By taking an intersection with \([m]_{L,\Gamma}\) on both side of the assumption, we have:

\[ \mathcal{K}(c, \vec{t}^{[i]}, \sim_{TGR}) \cap [m]_{L,\vec{t}^{[i]},\gamma} \supseteq [m]_{L,\vec{t}^{[i]},\gamma} \cap [m]_{L,\Gamma} \]

Apply Equation B.2.1 to the left and Equation B.2.2 to the right:

\[ k(c, m, \vec{t}^{[i]}, L, \Gamma) \supseteq ([m]_{L,\Gamma} \cap [m]_{E_i}) \cap [m]_{L,\Gamma} = [m]_{L,\Gamma} \cap [m]_{E_i} \]

Thus, we have \(k(c, m, \vec{t}^{[i]}, L, \Gamma) \supseteq ([m]_{L,\Gamma} \cap [m]_{E_i})\).

Therefore, Definition 25 is equivalent to Definition 27. \[\square\]

**According to Policy p**

**Lemma 5.** With \(\sim_{\triangleleft_{AP}}\), \(A \triangleleft_{AP} A\), and outside equivalence \(\equiv_{\triangleleft_{AP}}\), Definition 25 is equivalent to Definition 28.

**Proof.** First, we convert a security levels \(L\) from the Denning’s style to our attacker levels \(l\) as described in Section 5.3, and outputs any intermediate memory of the trace to its variable’s level. That is,

\[ \forall c, m_0, m', c_i, m_i, i, e, \Gamma. \]

\[ \tau = \langle c, m_0 \rangle_{\gamma_0} \rightarrow^{*} \langle c_i, m_i \rangle_{\gamma_i} \rightarrow^{*} m' \land \tau_{[i]} = \langle c_i, m_i \rangle_{\gamma_i} \]

190
\[\langle c, m_0 \rangle \mapsto \vec{t} \]
\[\wedge \vec{t}^{[i]} = \{\langle ch, n, \gamma \rangle \mid ch = e \land n = m_i(e) \land \gamma = \gamma_i \}\]

We note that our normal \(\vec{t}^{[i]}\) returns a single output event, say some \(t = \langle ch, n, \gamma \rangle\). But here we overload \(\vec{t}^{[i]}\) to return a set of output events that output all values on memory \(\tau_{[i]}\). Thus, with all values on memory outputted, we have:

\[\forall c, m_1, m_2, m'_1, m'_2, \tau, \tau', \vec{t}_1, \vec{t}_2. \]
\[\wedge \tau = \langle c, m_1 \rangle \rightarrow m'_1 \wedge \tau' = \langle c, m_2 \rangle \rightarrow m'_2 \]
\[\wedge \langle c, m_1 \rangle \leftrightarrow \vec{t}_1 \wedge \langle c, m_2 \rangle \leftrightarrow \vec{t}_2 \implies \]
\[\tau_{[i]} \approx_t \tau'_{[j]} \iff [\vec{t}^{[i]}]_t = [\vec{t}^{[j]}]_t.\]

Thus, we rewrite Definition 28 in following two-run style:

\[\forall c, m_1, m_2, l, p, \vec{t}_1, \vec{t}_2. \]
\[m_2 \in \llbracket m_1 \rrbracket_p \wedge \langle c, m_1 \rangle \leftrightarrow \vec{t}_1 \wedge \langle c, m_2 \rangle \leftrightarrow \vec{t}_2 \implies \]
\[\exists R. (\forall (i, j) \in R. \vec{t}_{1[i]} \in [\vec{t}_1]^{p,l} \wedge \vec{t}_{2[j]} \in [\vec{t}_2]^{p,l} \]
\[\implies [\vec{t}_1]_t = [\vec{t}_2]_t)\]

We combine the two filters and assume \(R'\) as \(R\) after filtering:

\[\forall c, m_1, m_2, l, p, \vec{t}_1, \vec{t}_2. \]
\[m_2 \in \llbracket m_1 \rrbracket_p \wedge \langle c, m_1 \rangle \leftrightarrow \vec{t}_1 \wedge \langle c, m_2 \rangle \leftrightarrow \vec{t}_2 \implies \]
\[\exists R'. (\forall (i, j) \in R'. ([\vec{t}_1]^{p,l})_{[i]} = ([\vec{t}_2]^{p,l})_{[j]} )\]
With $\mathcal{K}(c, \vec{t}, \sim_{AP})$ unfolded as below:

$$
\mathcal{K}(c, \vec{t}, \sim_{AP}) = \{m_2 \mid \forall m_2, \vec{t}_2. \langle c, m_2 \rangle \leftrightarrow \vec{t}_2 \}
\quad \land \exists R', \forall (i, j) \in R'. ([\vec{t}_{p,l}]_i) = ([\vec{t}_2_{p,l}]_j)
$$

We can further rewrite the definition as follow:

$$
\forall c, m_1, m_2, l, p, \vec{t}_1.

m_2 \in [m_1]_p \land \langle c, m_1 \rangle \leftrightarrow \vec{t}_1 \implies m_2 \in \mathcal{K}(c, \vec{t}_1, \sim_{AP})
$$

That is,

$$
\forall c, m_1, l, p, \vec{t}_1. \langle c, m_1 \rangle \leftrightarrow \vec{t}_1 \land [m_1]_p \subseteq \mathcal{K}(c, \vec{t}_1, \sim_{AP})
$$

We note that only the equivalence relation in $\sim_{AP}$ is $\equiv_{AP}$. The equivalence relation in $\vec{t}^l \equiv \vec{t}$ in Definition 25 in this case is not $\equiv_{AP}$, but $\equiv_{AP} \triangleq \{(\vec{t}_1, \vec{t}_2) \mid \vec{t}_1 = \vec{t}_2\}$.

\[\square\]

**Cryptographic Erasure**

**Lemma 6.** With $\sim_{CE}, \equiv_{CE}$ and $A \equiv A_{CE}$, Definition 25 with adjusted attack model is equivalent to Definition 29.

$$
\forall c, m, \gamma_0, c_i, m_i, \gamma_i, c_n, m_n, \gamma_n, m', \vec{t}_1, \vec{t}_2, l, i, j, n.

\langle c, m \rangle_{\gamma_0} \xrightarrow{\vec{t}_1} \langle c_i, m_i \rangle_{\gamma_i} \xrightarrow{\vec{t}_2} \langle c_n, m_n \rangle_{\gamma_n} \rightarrow^{*} m' \implies \kappa_{CE}(c, L, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} [m]_{L, \gamma_j} \iff \mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{t_n \in \vec{t}_2} [m]_{L, t_n, \gamma}
$$
Proof. We note that in Definition 29, $\gamma_j$ are state policies attached to the configurations, not from the output event $\vec{t}$. According to the definition, $\vec{t}$ does not contain empty events. In Definition 29, it takes $n - j$ steps to generate output sequence $\vec{t}_2$, we know $n - j \geq \|\vec{t}_2\|$. We first show that the right hand side allowance defined using $\gamma_j$ is the same as using state policy from the output sequence $\vec{t}$:

$$k_{\text{CE}}(c, L, \vec{t}_2) \supseteq \bigcap_{i \leq j \leq n} [m]_{L, \gamma_j}$$

$$\iff k_{\text{CE}}(c, L, \vec{t}_2) \supseteq \bigcap_{t_n \in \vec{t}_2} [m]_{L, t_n, \gamma} \quad \text{(B.2.3)}$$

• case $\implies$: Crypto [31] supports only erasure policy (and static policy). That is, the sensitivity of any security entity is monotonically increasing:

$$\forall j \in [i, n], \gamma_j \preceq \gamma_{j+1}$$

From the definition of memory closure, we know:

$$\forall \gamma_1, \gamma_2. \gamma_1 \preceq \gamma_2 \implies [m]_{L, \gamma_1} \subseteq [m]_{L, \gamma_2}$$

Thus, we know:

$$\bigcap_{i \leq j \leq n} [m]_{L, \gamma_j} = [m]_{L, \gamma_i}$$

$$\bigcap_{t_n \in \vec{t}_2} [m]_{L, t_n, \gamma} = [m]_{L, \vec{t}_2[0], \gamma}$$

From the definitions, we know $\vec{t}_2[0], \gamma = \gamma_i$ if $\langle c_i, m_i \rangle\gamma_i$ does not immediately generates an empty output event. Otherwise, if the first non-empty event is generated at configuration
\[ \langle c', m' \rangle_{\gamma', i < i' < n}, \] we know:

\[ [m]_{L, \gamma} \subseteq [m]_{L, \gamma'} = [m]_{L, t_2[i_0], \gamma} \]

We can instantiate Definition 29 for \( i := i' \), and we get:

\[ k_{CE}(c, L, t_2) \supseteq \bigcap_{j \leq i \leq n} [m]_{L, \gamma_j} = [m]_{L, \gamma_i} = [m]_{L, t_2[i_0], \gamma}. \]

Thus, we have \( k_{CE}(c, L, t_2) \supseteq \bigcap_{i \leq j \leq n} [m]_{L, \gamma_j}. \)

• case \( \Leftarrow : \) from \( n - j \geq \|t_2\| \), we know:

\[ \forall t_n \in t_2. \exists j' \in [i, n], \gamma_{j'} = t_n \cdot \gamma \]

\[ \{t_n \cdot \gamma \mid t_n \in t_2\} \subseteq \{\gamma_j \mid i \leq j \leq n\} \]

\[ \bigcap_{t_n \in t_2} [m]_{L, t_n \cdot \gamma} \supseteq \bigcap_{i \leq j \leq n} [m]_{L, \gamma_j}. \]

Thus, we have \( k_{CE}(c, L, t_2) \supseteq \bigcap_{i \leq j \leq n} [m]_{L, \gamma_j}. \)

Therefore, we know Equation B.2.3 is true.

Now we convert a security level \( L \) from Denning's style to our attacker level \( l \) as described in Section 5.3. Let \( [c] \triangleq \{m \mid \exists \overline{t}. \langle c, m \rangle \leftrightarrow \overline{t}\} \) denote the set of memory that terminates. From definition we know:

\[ K(c, \overline{t}, \sim_{CE}) = k_{CE}(c, L, \overline{t}) \cap [c] \]

For the interest of a termination-insensitive policy, we can ignore the difference made by the terminated set \([c]\). Thus, we assumes \( K(c, \overline{t}, \sim_{CE}) = k_{CE}(c, L, \overline{t}). \)

**Forgetful Attacker**

**Lemma 7.** With \( \sim_{FA} \) and \( A \triangleq A_{FA} \), Definition 25 is equivalent to Definition 30.
**Proof.** In the forgetful attacker [32], the sensitivity level is changed by setPolicy command. Recall from our encoding, setPolicy is encoded using security commands and generates a security event, but no output event. So, there is no sensitivity change between the two states that generates an output. That is, for the output event \( t' \) in the trace:

\[
\langle c, m \rangle \overset{t'}{\longrightarrow} * \langle c', m' \rangle \overset{(b,v,\gamma)}{\longrightarrow} * \Rightarrow
\]

We know \( t'.\gamma = \gamma' \) and therefore, we have

\[
[m]_{L,\gamma} = [m]_{L, v, \gamma}
\]

Definition 30 is rephrased as:

\[
\forall c, m, L, i, \vec{t} . \langle c, m \rangle \Rightarrow \vec{t} \Rightarrow
\]

\[
k_{FA}(c, L, \text{Atk}, \vec{t}[i]) \subseteq k_{FA}(c, L, \text{Atk}, \vec{t}[i-1]) \cap \[m\]_{L, \vec{t}[i], \gamma}
\]

By definition, we know:

\[
k_{FA}(c, L, \text{Atk}, \vec{t}) = K(c, \vec{t}, \sim_{FA})
\]

Thus, we know Definition 30 is equivalent to Definition 25. \( \square \)

**Paralock**

**Lemma 22.** With \( \sim_{PL} \) and \( A \triangleq A_{PL} \), Definition 25 is equivalent to Definition 37.

\[
\forall c, m, i, \vec{t}, \vec{t}', i, A . \langle c, m \rangle \Rightarrow \vec{t} \wedge \vec{t}' = \vec{t}[i] \Rightarrow \]

\[
\bar{t}[i].\Delta \subseteq \Sigma_A \Rightarrow
\]

\[
k_{PL}(c, m, \vec{t}[i], A) = k_{PL}(c, m, \vec{t}[i-1], A)
\]

195
\[ \iff \]

\[ \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \supseteq \mathcal{K}(c, \vec{t}^{[i-1]}, \sim_{PL}) \cap [m]_A \]

**Proof.** We omit the case when \( \vec{t}^{[i]}, \Delta \not\subseteq \Sigma_A \) since both definitions are trivially true. By Definition, we know:

\[ \forall j. k_{PL}(c, m, \vec{t}^{[j]}, A) = \mathcal{K}(c, \vec{t}^{[j]}, \sim_{PL}) \cap [m]_A \]

- case \( \implies \): we know:

\[ k_{PL}(c, m, \vec{t}^{[i-1]}, A) = k_{PL}(c, m, \vec{t}^{[i]}, A) \]
\[ k_{PL}(c, m, \vec{t}^{[i]}, A) = \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \cap [m]_A \]
\[ \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \supseteq \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \cap [m]_A \]

Thus, we have

\[ \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \supseteq k_{PL}(c, m, \vec{t}^{[i-1]}, A) \]

With \( k_{PL}(c, m, \vec{t}^{[i-1]}, A) = \mathcal{K}(c, \vec{t}^{[i-1]}, \sim_{PL}) \cap [m]_A \), we get \( \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \supseteq \mathcal{K}(c, \vec{t}^{[i-1]}, \sim_{PL}) \cap [m]_A \).

- case \( \impliedby \): we know:

\[ \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \supseteq \mathcal{K}(c, \vec{t}^{[i-1]}, \sim_{PL}) \cap [m]_A \]
\[ k_{PL}(c, m, \vec{t}^{[i-1]}, A) = \mathcal{K}(c, \vec{t}^{[i-1]}, \sim_{PL}) \cap [m]_A \]

Thus, we have

\[ \mathcal{K}(c, \vec{t}^{[i]}, \sim_{PL}) \supseteq k_{PL}(c, m, \vec{t}^{[i-1]}, A) \]

(B.2.4)
We know $[m]_A$ is the initial knowledge of $A$ before observing any output event. From the monotonicity of Paralock knowledge, we know:

$$[m]_A \supseteq k_{PL}(c, m, \bar{t}_{[i-1]}, A) \quad (B.2.5)$$

By taking an intersection on both side of Equation (B.2.4) and Equation (B.2.5), we have:

$$\begin{align*}
(K(c, \bar{t}_{[i]}, \sim_{PL}) \cap [m]_A) \\
\supseteq (k_{PL}(c, m, \bar{t}_{[i-1]}, A) \cap k_{PL}(c, m, \bar{t}_{[i-1]}, A)) \\
= k_{PL}(c, m, \bar{t}_{[i-1]}, A)
\end{align*}$$

Thus, we have

$$k_{PL}(c, m, \bar{t}_{[i-1]}, A) \subseteq K(c, \bar{t}_{[i]}, \sim_{PL}) \cap [m]_A$$

By Definitions, we have:

$$k_{PL}(c, m, \bar{t}_{[i]}, A) = K(c, \bar{t}_{[i]}, \sim_{PL}) \cap [m]_A$$

Thus, we know:

$$k_{PL}(c, m, \bar{t}_{[i-1]}, A) \subseteq k_{PL}(c, m, \bar{t}_{[i]}, A)$$

From the monotonicity of the Paralock knowledge, we know

$$k_{PL}(c, m, \bar{t}_{[i-1]}, A) \supseteq k_{PL}(c, m, \bar{t}_{[i]}, A)$$

Thus, we have:

$$k_{PL}(c, m, \bar{t}_{[i-1]}, A) = k_{PL}(c, m, \bar{t}_{[i]}, A)$$
Therefore, Definition 25 is equivalent to Definition 37.
Appendix C

Proof in Decomposition

Before we show the proof for Theorem 9 and Theorem 10, we first introduce a useful lemma for the equivalence on per-variable and per-set formalization for noninterference.

**Definition 38 (Noninterference per set X)** for a code block $c$ with a static policy $\gamma$, written as $\{c\}^{\gamma}$, we say that it satisfies noninterference per set $X$ at level $l$, denoted as $X : l \vdash \{c\}^{\gamma}$, if we have $\forall x \in X. l \not\in \gamma(x)$ and

$$\forall m_0, m'_0, m_1, m'_1. \langle c, m_0 \rangle \rightarrow^* \langle \text{skip}, m_1 \rangle \land \langle c, m'_0 \rangle \rightarrow^* \langle \text{skip}, m'_1 \rangle \land$$

$$(\forall y \not\in X. m_0(y) = m'_0(y)) \Rightarrow m_1 \approx^\gamma m'_1$$

**Lemma 24** Given a program $c$ with static policy $\gamma$, per-variable and per-set formalizations are interchangeable.

$$\forall X, c, l, \gamma. (\forall x \in X. l \not\in \gamma(x)) \Rightarrow$$

$$X : l \vdash \{c\}^{\gamma} \iff (\forall x \in X. x \vdash \{c\}^{\gamma})$$

**Proof.** Proof for $\implies$ is trivial since any counterexample for the RHS: $\forall x \in X. x \vdash \{c\}^{\gamma}$ (the initial memroy differs only at a single variable $x$) is also a counterexample for the LHS: $X : \{c\}^{\gamma}$ (initial memory differs only at set $X$). That is, $(\exists x \in X. x \not\vdash \{c\}^{\gamma}) \implies \neg(X : l \vdash \{c\}^{\gamma})$. 

199
Now we prove for \( \exists \) by contradiction, where a counterexample is found for the LHS: 
\[ X : \{e\}^\gamma, \text{ but RHS: } \forall x \in X. \ x \vdash \{e\}^\gamma \text{ hold.} \]

In the counterexample, the two initial memory \( m_1 \) and \( m_2 \) only differ in the variables in \( X \), but in the resulting memory of the two executions \( m_1' \) and \( m_2' \), we know there must exists a variable \( z \) that \( l \in \gamma(z) \land m_1'(z) \neq m_2'(z) \).

Now we construct memory \( m_{c_1} \), for each variable \( x_i \in X \), assuming there are \( n \) variables in \( X \), where for variables \( x_1, \ldots, x_i \), it has the same value of \( m_2 \), but for variable \( x_{i+1}, \ldots, x_n \), it has the same value of \( m_1 \):

\[
\forall j \in [1, i]. \ m_{c_i}(x_j) = m_2(x_j);
\]
\[
\forall j \in (i, n]. \ m_{c_i}(x_j) = m_1(x_j);
\]

From assumption \( \forall x_i \in X. \ x_i \vdash \{e\}^\gamma \), we know if the two executions start with initial memory \( m_{c_i-1} \) and \( m_{c_i} \), since \( m_{c_i-1} \) and \( m_{c_i} \) only differ in variable \( x_i \), their ending memory \( m_{c_i-1}' \) and \( m_{c_i}' \) is low equivalent according to Definition 32: \( \forall l'.l' \notin \gamma(x_i) \Rightarrow m_{c_i-1}' \approx_l' m_{c_i}' \).

From assumption, we have \( \forall i \in [1, n]. \ l \notin \gamma(x_i) \).

Thus, we have \( m_1' \approx_l m_{c_1}' \approx_l \cdots \approx_l m_{c_{i-1}}' \approx_l m_{c_i}' \approx_l \cdots \approx_l m_{c_n}' = m_2' \).

From \( m_1' \approx_l m_2' \), by Definition 31, we know \( \forall z.l \in \gamma(z) \Rightarrow m_1(z) = m_2(z) \), which conflicts with the assumption that \( \exists z. \ l \in \gamma(z) \land m_1'(z) \neq m_2'(z) \).

\[ \square \]

**Proof of Theorem 9**

\[
\forall \Gamma, cmd, c_1, \ldots, c_n, L_1, \ldots, L_n.
\]
\[
[cmd]_{\Gamma} = \{c_1\}^{L_1}; \ldots; \{c_n\}^{L_n} \land (\forall x, y, i. \ x \leftarrow_{c_1; \ldots; c_{i-1}} y \Rightarrow \gamma_i(y) \subseteq \gamma_i(x)) \Rightarrow
\]
\[
(\forall x, i \in [1, n]. \ x \vdash \{c_i\}^{\gamma_i}) \Rightarrow (\forall x. \ x \vdash [cmd]_{\Gamma})
\]

**Proof.** Consider any two terminating executions of the decomposed code \( \{c_1\}^{L_1}; \ldots; \{c_n\}^{L_n} \):

\[
\langle\{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n}, m_0\rangle \rightarrow^* \cdots \rightarrow^* \langle\text{skip}; \{c_{i+1}\}^{\gamma_{i+1}}; \ldots; \{c_n\}^{\gamma_n}, m_i\rangle \rightarrow^* \cdots \rightarrow^* \langle\text{skip}, m_n\rangle
\]
\{c_1\}^{\gamma_1}; \ldots ; \{c_n\}^{\gamma_n}, m_0' \rightarrow^* \cdots \rightarrow^* \langle \text{skip}; \{c_{i+1}\}^{\gamma_{i+1}}; \ldots ; \{c_n\}^{\gamma_n}, m_i' \rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}, m_n' \rangle

From assumption \((\forall x, i \in [1, n]. x \vdash \{c_i\}^{\gamma_i})\) and Lemma 24, we know for each block \(\{c_i\}^{\gamma_i}\) and variable \(x\), we have \(x\) protected locally according to \(\gamma_i\), i.e., \((\forall l \not\in \gamma(x) \Rightarrow m_i \approx^{\gamma_i} m_i')\) as long as we have the two initial memory \(m_{i-1}\) and \(m_i'\) only differ in set \(H_i^x \triangleq \{z \mid \forall z. \gamma_i(z) \subseteq \gamma_i(x)\}\). Intuitively, set \(H_i^x\) contains all the variables that have the same or more restrictive sensitivity than \(x\) in \(\gamma_i\).

From Definition 34, we know when the initial memories \(m_0\) and \(m_0'\) only differ at variable \(x\), the memory \(m_{i-1}\) and \(m_i'\) at the beginning of block \(\{c_i\}^{\gamma_i}\) differs at most at \(\{z \mid \forall z. x \leftrightarrow_{c_{i-1}} z\} \triangleq D_{i-1}^x\). Intuitively, set \(D_{i-1}^x\) contains all the variables that have a dependence from \(x\) till the end of code block \(i\).

From the assumption \(\forall x, y, i. x \leftrightarrow_{c_{i-1}} y \Rightarrow \gamma_i(y) \subseteq \gamma_i(x)\), we know \(\forall x, i. D_{i-1}^x \subseteq H_i^x\).

Putting them together, we have, if the two initial memories \(m_0\) and \(m_0'\) only differ in variable \(x\), then at the beginning of each code block \(\{c_i\}^{\gamma_i}\), the two memories \(m_{i-1}\) and \(m_i'\) differ at most at set \(D_{i-1}^x\). With \(D_{i-1}^x \subseteq H_i^x\) and \(x \vdash \{c_i\}^{\gamma_i}\), we have \(x\) protected according to \(\gamma_i\), \((\forall l \not\in \gamma(x) \Rightarrow m_i \approx^{\gamma_i} m_i')\). That is, by definition \(\forall x. x \vdash [\text{cmd}]_\Gamma\).

\(\square\)

Proof of Theorem 10

\[\forall \Gamma, \text{cmd}. [\text{cmd}]_\Gamma = \{c_1\}^{\gamma_1}; \ldots ; \{c_n\}^{\gamma_n} \land (\forall x, y, i. \gamma_i(y) = \bigcap_{x' \leftrightarrow_{c_{i-1}} c_i} \gamma_i(x')) \land \vdash \{c_1; \ldots c_{i-1}\}; \{c_i\} \Rightarrow (\forall x. x \vdash [\text{cmd}]_\Gamma) \iff (\forall x. x \vdash \{c_i\}^{\gamma_i})\]

Proof. Case \(\iff\) is a trivial application of Theorem 9.

Next, we show the proof for \(\Rightarrow\). Consider any two terminating executions of the decomposed code \(\{c_1\}^{L_1}; \ldots ; \{c_n\}^{L_n}\):

\(\langle \{c_1\}^{\gamma_1}; \ldots ; \{c_n\}^{\gamma_n}, m_0 \rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}; \{c_{i+1}\}^{\gamma_{i+1}}; \ldots ; \{c_n\}^{\gamma_n}, m_i \rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}, m_n \rangle\)
\[
\langle \{c_1\}^{\gamma_1}; \ldots; \{c_n\}^{\gamma_n}, m_0' \rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}; \{c_{i+1}\}^{\gamma_{i+1}}; \ldots; \{c_n\}^{\gamma_n}, m_i' \rangle \rightarrow^* \cdots \rightarrow^* \langle \text{skip}, m_n' \rangle
\]

We prove by contradiction, where \( \forall x. x \vdash [\text{cmd}]_\Gamma \) holds, but there exists a variable \( y \) that is not correctly protected at code block \( c_i \), \( \exists y, i. y \not\in \{c_i\}^{\gamma_i} \). According to Definition 32 and Definition 34, that is, \( \exists y, i, z. \gamma_i(z) \nsubseteq \gamma_i(y) \land y \leftarrow_{c_i} z \). From the assumption \( \forall x. x \vdash [\text{cmd}]_\Gamma \) and Definition 34, we know that \( \forall x. x \leftarrow_{c_1; \ldots; c_i} z \Rightarrow \gamma_i(z) \subseteq \gamma_i(x) \).

From contradiction assumption, we know there exists \( y, z \) such that \( y \leftarrow_{c_i} z \land \gamma_i(z) \nsubseteq \gamma_i(y) \); with the transitive property, we can derive that \( \forall x. x \leftarrow_{c_1; \ldots; c_i} y \Rightarrow x \leftarrow_{c_1; \ldots; c_i} z \).

Putting them together, we have \( \forall x. x \leftarrow_{c_1; \ldots; c_i} y \Rightarrow x \leftarrow_{c_1; \ldots; c_i} \gamma_i(x) \).

From assumption \( \gamma_i(y) = \bigcap_{x'. x' \leftarrow_{c_1; \ldots; c_i} y} \gamma_i(x) \), we know \( \gamma_i(z) \subseteq \gamma_i(y) \subseteq \gamma_i(x) \), which contradicts with \( \gamma_i(z) \nsubseteq \gamma_i(y) \).

\(\square\)
Bibliography


205


207


Vita
Peixuan Li

EDUCATION

Pennsylvania State University
Ph.D. candidate in Computer Science and Engineering, GPA: 3.8/4.0
State College, PA
December 2021

Northeastern University
B.E. in Software Engineering
Shenyang, China
June 2014

EXPERIENCE

Amazon
New York, NY
Applied Scientist Intern
May 2020 - Aug 2020

Max Planck Institute for Software Systems
Saarbrücken, Germany
Research Intern
Jun 2019 - Sep 2019

Microsoft
Redmond, WA
Software Engineer Intern
May 2015 - Aug 2015

PUBLICATIONS


