Secure Information Flow: Analysis and Enforcement

Adedayo Oyelakin Adetoye

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
United Kingdom
April 2009

Abstract

When a computer program requires legitimate access to confidential data, the question arises whether such a program may reveal sensitive information to an unauthorised observer. There is therefore a need to ensure that a program, which processes confidential data, is free of unwanted information flow. This thesis presents a formal framework for the analysis and enforcement of secure information flow in computational systems such as computer programs.

An important aspect of the problem of secure information flow is the development of policies by which we can express intended information release. For this reason *information lattices* and maps on these lattices are presented as models, which capture intuitive notions about *information* and *information flow*. A definition of security is given, based on the lattice formalisation of *information* and *information flow*, that exploits the partial order of the information lattice. The lattice formalisation gives us a uniform way to enforce information security policies under various qualitative and quantitative representations of information.

An *input-output relational model*, which describes how a system transforms its input to publicly observable outputs with respect to a given attacker model, is presented as a primitive for the study of secure information flow. By using the relational model, various representations of information, which are shown to fit into the lattice model of information, are derived for the analysis of information flow under deterministic and nondeterministic system models. A systematic technique to derive the relational model of a system, under a given attacker model, from the operational semantics in a language-based setting, is also presented. This allows the development of information flow analyses parametrised by chosen attacker models.

A flow-sensitive and termination-sensitive static analysis calculus is presented for the analysis of information flow in programs written in a deterministic *While* language with outputs. The analysis is shown to be correct with respect to an attacker model that is able to observe all program outputs and which can determine the termination or nontermination of program execution. The static analysis also detects certain disjunctive information release. A termination-sensitive dependency analysis is developed which demonstrates how, by employing abstract interpretation techniques, other less precise but possibly more efficient information flow analysis may be obtained. The thesis concludes with further examples to highlight various aspects of the information flow analysis and enforcement framework developed.

Contents

List of Figures

List of Tables

Acknowledgements

I must first thank my supervisor, Eike Ritter, who masterfully introduced me to formal research. Thank you for your constant support and for always throwing in the right questions that have helped me immensely to clarify and discipline my thoughts.

I thank Mark Ryan and Volker Sorge, members of my Thesis Group, for all your very helpful suggestions and friendly words of advice.

I am fortunate to have a loving and supporting wife, Nike, who endured with me throughout my academic studies. Thank you darling. Damilola, my beautiful daughter, has been a constant source of pleasant distractions from academic work.

Many friends and colleagues at the school have made my study here a pleasant one. Thank you all.

Finally, I thank my Father and Mother, who made many sacrifices that got me to where I am today.

Chapter 1

Introduction

Today, society is highly-dependent on information and computer networks. The convenience of connecting computers and other personal devices, and the ease with which this can be done, has made it ever more difficult to protect sensitive information from being released in unwanted ways. The security of information release has thus become a very important problem.

Secure information release or *secure information flow* is not a new problem, but the current standard security techniques do not provide a satisfactory solution to it. For example, techniques such as cryptography and operating system access control lists, which can be used to limit access to sensitive information in computer systems, are of little use once information has been decrypted and released to a program which requires *legitimate access* to such information. Such a program can release sensitive information maliciously, or inadvertently due to programming flaws in it. There is a need for mechanisms through which we can specify what information we want to release, and whereby we can check whether a program that has access to such information conforms to the required information

flow specification.

This thesis develops a formal framework whereby we can model information and information flow, which allows us to specify security policies to capture intended information release. The analysis techniques developed allow us to check whether a system that processes sensitive data conforms to the information release specifications in a given policy. By these, we are able to enforce secure information flow requirements, which is achieved by preventing programs from processing data for which they have not been certified as having secure information flow with respect to the release policy of that data.

1.1 Modelling Information Flow

The traditional approach to modelling information flow, or rather, the lack of information flow for the enforcement of security, is through the *noninterference* requirement [GM82]. Noninterference prevents *any* confidential information from propagating to unauthorised observers and is useful in multilevel security systems, where information must not flow from *high* levels of security classifications to *low* levels of security classifications. However, in practice (for example, during authentication, encryption, and when performing statistical analyses), we often have to, or want to release some information in a controlled manner. Noninterference is not suitable under such circumstances. Noninterference as a policy model is of limited use in practice [RMMG01, Vol99a]. There is thus a need for a general model for the specification of what information we intend to release.

In [SS05], a taxonomy of declassification mechanisms is introduced based on *what*, *where*, *when*, and by *whom* information is released. This thesis focuses

on the *what* dimension, where we are interested in regulating *what* information an observer can gain from a system that is processing sensitive data. For this purpose, a lattice-theoretic model of information and information flow is introduced based on the observation that securing what information an observer can gain alludes to a notion of the *level* of information that is considered safe to be released to the observer. Furthermore, information is intuitively *ordered*, whereby we say that one piece of information is *greater* or *more informative* than another one. This suggests an ordering of information, which we exploit in the lattice model of information to define a notion of secure information flow.

The associated partial order of the information lattice captures the intuition of information levels or the information order. This allows us to model the flow of information as maps on the lattice of information, which describe how an observer's knowledge changes when observing a system that is causing information flow. By using only the lattice structure in the definition of information flow and security, we open up the possibility to use the same enforcement technique or mechanism, regardless of the particular representation of information. Although lattice-based techniques are often used in language-based security [SM03a], the lattices are usually of security classes or security types in a multilevel system, rather than lattices of information. The use of lattices as a general model of information has the advantage that it unifies various representations of information under the same model for the enforcement of *what* declassification policies. This makes it possible to use the same enforcement mechanism under different lattice-based information representations.

1.2 Deriving Information Flow

Once a choice of *information representation* has been made, we need to be able to derive the information that a system may release in order to check whether the system is secure with respect to an information flow policy. In this thesis, *information representations* based on *Equivalence Relations* or, more generally, *Partial Equivalence Relations*, as well as *Families of Sets* are considered as *qualitative* representations of information. Under *quantitative* representations, information-theoretic measures are considered as representations of information. These representations are all shown to fit into the lattice model of information.

An extensional input-output *relational model* is presented in the thesis as a primitive for the analysis of information flow, which derives the information that a system releases and links the system's input-output semantics to its information flow properties under a given representation of information. The simple idea is that information released by a system is ultimately linked to how the system transforms its secret inputs to publicly observable outputs. The relational model itself may be defined parametric to a specified attacker model by relating inputs to the outputs which that particular attacker can observe. Using the relational model, analyses of information flow under deterministic and nondeterministic system models are presented. Furthermore, the thesis demonstrates, in a language-based setting, that the relational model-based analysis copes very well with information flows due to nontermination.

1.3 Enforcing Secure Information Flow

Our objective is to enforce secure information flow by ensuring that only programs with secure information flow have access to sensitive data. Thus, given an information flow policy, which specifies our intentions about what information we allow to be released, an enforcement framework is needed that can decide whether a system is safe with respect to that policy. A semantics-based approach to the analysis of information flow, which uses the system's input-output relational model, is proposed in this thesis. We demonstrate how to derive the relational model from the operational semantics in a language-based setting. We also present a static information flow analysis and a dependency analysis for a deterministic imperative *While* language with outputs. The analyses are used to check whether a given program has secure information flow with respect to given policies. By this, we can enforce the security of information flow by granting access only to programs which have secure information flow.

1.4 Overview of Thesis

The objective of this thesis is to study the problem of secure information flow and to develop techniques to model, analyse, and enforce secure information flow in computer systems. The current chapter concludes by introducing some of the mathematical definitions and notations used in the thesis. In Chapter 2, important language-based techniques for the analysis and enforcement of secure information flow are reviewed.

The goal of Chapter 3 is to model the notions of information and information

flow suitable for the definition of information flow policies. The lattice-based definition agrees with the intuitive notions of information ordering and provides the basis for the enforcement of secure information flow, where information release is considered insecure with respect to a policy when it is greater than the levels permitted in the policy. The notion of information levels is captured by the lattice order. This approach has the advantage that the enforcement relies only on the lattice properties, which can be applied independently of the particular representation of information that is used. Representations of information based on partial equivalence relations, families of sets, and information-theoretic characterisations are all shown to fit into the lattice model. Furthermore, Chapter 3 presents an extensional, semantics-based, relational model approach to the analysis of information released by a system, where the relational model describes how the system transforms its inputs to publicly observed outputs. The relational model links the input-output semantics of the system being analysed to the lattice-based representation of information that the system releases.

Chapter 4 shows how to derive the input-output relational model of a system from the operational semantics in a language-based setting. It starts by considering interactive systems formalised as labelled transition systems, where the labels capture what the attacker can observe during each state transition of the system. This provides a formalism for studying attackers with different observational powers. As a concrete example, the *While* language is introduced as a programming language for deterministic systems with interactive outputs and buffered input. A specific *semantic attacker* model, which is able to observe program outputs as prescribed by the standard operational semantics of *While* and can additionally determine whether the program terminates or not, is introduced in Chapter 4 to

illustrate the definition of an attacker model, and the definition of termination-sensitive analyses. Extensions to the core *While* language are also presented to demonstrate analyses of information flow in possibilistic and probabilistic nondeterministic systems.

In Chapter 5, a static analysis of the imperative *While* language is presented. The analysis, which uses PERs on the set of program states as the representation of information, is flow-sensitive and termination-sensitive, and is also capable of detecting certain disjunctive information flows. The static analysis is shown to be sound with respect to the semantic definition of information flow that the semantics attacker gains as defined in Chapter 4.

Since the static analysis using PERs may be computationally prohibitive when the set of states considered is very large, or we may otherwise not want the level of detail of information that may be represented by PERs over program states, Chapter 6 demonstrates how the machinery of abstract interpretation may be used to reduce or simplify the domain over which static analysis is performed, while maintaining correctness. A flow-sensitive and termination-sensitive dependency analysis is presented, which is shown to be an abstract interpretation of the concrete PER-based analysis of Chapter 5. The dependency analysis also identifies some disjunctive dependencies.

Chapter 7 concludes the thesis with examples, which highlight various lessons learnt in the thesis. The chapter reviews the main contributions of the thesis and suggests areas of future work.

1.5 Mathematical Preliminaries

The mathematical developments in this thesis rely on the basic theory of sets and relations [End77, AGM92], as well as principles from ordered sets and maps between them [DP03, GHK+03]. This section briefly reviews the important definitions and results, and introduces some of the notations that we shall use.

Sets, Binary Relations, and Functions

A set $X = \{x_0, x_1, ..., x_n\}$ is a collection of objects $x_0, x_1, ..., x_n$, which are called its elements, and no other elements. The membership relation $\in$, such as $x_0 \in X$, asserts which element belongs to a set. The opposite relation $\notin$ asserts that an object does not belong to a set. The Cartesian product of two sets X and Y is the set of pairs $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$. Sometimes, $X \times X$ is written as X^2.

If X and Y are sets, $X \subseteq Y$ asserts that X is a subset of Y, which means that $x \in X$ implies $x \in Y$. The empty set, which has no element, is denoted by $\varnothing$. The powerset of a set X is $\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$, which is the set of all subsets of X. A family of sets over X is a subset of the powerset $\mathcal{P}(X)$. The family of sets $\{Y_i \subseteq X \mid i \in I\}$ is sometimes denoted by $Y_{i \in I}$ or simply Y_I, where I is an index set. The union and intersection of the family of sets denoted by $Y_{i \in I}$ is given by $\bigcup_{i \in I} X_i$ and $\bigcap_{i \in I} X_i$ respectively.

The set-theoretic difference between the sets X and Y is the set of elements in X, but not in Y, and is denoted by $X \backslash Y = \{x \in X \mid x \notin Y\}$. The set of natural numbers is denoted by $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, and the set of integers is denoted by $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

We write $R \subseteq X \times Y$ to denote a *binary relation* which associates elements of

the set X with elements of the set Y. When $x \in X$ is related to $y \in Y$ by R, we write $x\,R\,y$ or $(x, y) \in R$. The assertion $(x, y) \notin R$ means that x is not related to y by R. The *graph* of the binary relation $R \subseteq X \times Y$ is the set of pairs, which is defined as $\textit{graph}(R) \triangleq \{(x, y) \in X \times Y \mid x\,R\,y\}$.

A binary relation $f \subseteq X \times Y$ is a *function* or *map* if for any $y, y' \in Y$ and $x \in X$, $x\ f\ y$ and $x\ f\ y'$ implies that $y = y'$. The usual notation for functions is to write $f(x) = y$ whenever $x\,f\,y$ holds. It is also customary to write $f : X \to Y$ to say that f is a total function from the set X to the set Y. The set of all total functions from the set X to the set Y is denoted by $[X \to Y]$.

Partially Ordered Sets

A set X is a *partially ordered set* (*poset*) if it is equipped with a binary relation $\leq$, such that for all $x, y, z \in X$ the relation $\leq$ has the following properties:

- $x \leq x$, (reflexivity)
- $x \leq y$ and $y \leq x$ implies $x = y$, (antisymmetry)
- $x \leq y$ and $y \leq z$ implies $x \leq z$. (transitivity)

If the relation $\leq$ on X is not necessarily antisymmetric, then X is a *pre-order* with respect to $\leq$. When we wish to lay emphasis on the order relation of a poset we write $\langle X, \leq \rangle$ to say that the elements of X are partially ordered by $\leq$.

Duality Principle

The duality principle says that there is a corresponding *dual* statement ($S_{\geq}$) to each statement ($S_{\leq}$) about an ordered set $\langle X, \leq \rangle$, which can be obtained by replacing

each occurrence of $\leq$ in $S_{\leq}$ by $\geq$ and vice versa. If $S_{\leq}$ is true in all ordered sets then so also is $S_{\geq}$.

Upper and Lower Bounds

Let $\langle X, \leq \rangle$ be a poset and let $S \subseteq X$. An element $x \in X$ is an *upper bound* of S if for all $s \in S, s \leq x$. A *lower bound* is dually defined. The set of upper bounds of S in X is $S^u \triangleq \{x \in X \mid x \text{ is an upper bound of } S\}$. The set of lower bounds S^ℓ of S in X is dually defined.

An element $x \in S^u$, if it exists, is said to be the *least upper bound* or *supremum* of the set S if for all $y \in S^u, x \leq y$. The *greatest lower bound* or *infimum* of the set S, if it exists, is defined dually on the set S^ℓ of lower bounds. The *down-set* $\downarrow S$ of the set S is defined as $\downarrow S = \{x \in X \mid s \in S, x \leq s\}$.

Joins and Meets

Let $\langle X, \leq \rangle$ be a poset. The *join* of two elements $x, y \in X$, if it exists, is the least upper bound of the set $\{x, y\}^u$ in X. This is usually written as $x \vee y$. Similarly, the *meet* of these elements is written as $x \wedge y$ and is the greatest lower bound of the set $\{x, y\}^\ell$ in X. More generally, for any subset $S \subseteq X$, the join is denoted as $\bigvee S$, and the meet as $\bigwedge S$, both of which are assumed to be elements of the set X whenever they exist. If the elements of $S = \{s_i \mid i \in I\}$ are indexed, an alternative notation is $\bigvee_i s_i$ and $\bigwedge_i s_i$ for the join and meet of S respectively.

Top and Bottom

A poset $\langle X, \leq \rangle$ is said to have a *top* or *greatest* element, written as $\top$ if for all $x \in X, x \leq \top$. Dually, the *bottom* or *least* element $\bot \in X$, when it exists, has the

property that for all $x \in X, \bot \leq x$.

Pointwise Function Ordering

Let X be a set and let $\langle Y, \leq \rangle$ be an ordered set. The order relation $\leq$ on Y induces an order on the set of all maps $[X \to Y]$ from X to Y, which is called the *pointwise order*, which for any $f, f' \in [X \to Y]$ we have $f \leq f'$ iff for all $x \in X, f(x) \leq f'(x)$.

Operators on Partially Ordered Sets

An operator on a poset $\langle X, \leq \rangle$ is a function $f : X \to X$, which may have any of the following properties

- *extensivity*: if $\forall x \in X. \quad x \leq f(x)$
- *reductivity*: if $\forall x \in X. \quad f(x) \leq x$
- *idempotency*: if $\forall x \in X. \quad f(f(x)) = f(x)$
- *monotonicity*: if $\forall x, x' \in X. \quad x \leq x' \implies f(x) \leq f(x')$

Closure operators

An operator on a poset is an *upper closure operator* if it *extensive, monotone* and *idempotent*. A *lower closure operator* is an operator which is *reductive, monotone* and *idempotent*.

Fixpoints and Chain Conditions

Let $\langle X, \leq \rangle$ be an ordered set and let $f : X \to X$ be a map. We say that $x \in X$ is a *fixpoint* of f if $f(x) = x$. The set of fixpoints of f is denoted by $\mathit{fix}(f)$. The ordered set X is said to satisfy the *ascending chain condition* (ACC) if for any given sequence $x_1 \leq x_2 \leq \cdots \leq x_n \leq \cdots$ of elements of X, there exists $k \in \mathbb{N}$ such that $x_k = x_{k+1} = \cdots$. The dual notion to the ACC is the *descending chain condition*.

Lattices and Complete Lattices

A non-empty ordered set $\langle X, \leq \rangle$ is a *lattice* if for all $x, x' \in X$, the join $x \vee x'$ and the meet $x \wedge x'$ exist. If, furthermore, for all $S \subseteq X$, $\bigvee S$ and $\bigwedge S$ exist, X is a *complete lattice*. In order to show that a poset $\langle X, \leq \rangle$ is a complete lattice, it is sufficient to show that $\bigvee S$ exists for arbitrary subsets S of X, because the existence of arbitrary joins guarantees the existence of arbitrary meets [GHK+03].

Chapter 2

Language-based Security

This chapter reviews language-based approaches to the problem of secure information flow. Language-based approaches to security seek to determine or ensure program security by analysing the programming language constructs or by using the language constructs to enforce security.

2.1 Language-based Approach to Security

The term security in this thesis generally refers to the security of information flow in systems. Although language-based approaches have been used for the protection of security other than the confidentiality of information, this thesis focuses on the application of language-based techniques to the problem of secure information flow.

A traditional approach to the protection of resources in computer systems is through the use of *access control*. Access control mechanisms prevent a principal, such as a user program, from having unauthorised access to resources. How-

ever, when a user program requires a *legitimate* access to a resource, such as a database or file, which contains confidential data, how can we ensure that such a program does not reveal sensitive information illegally? A language-based approach to security, which seeks to determine the security of a program by the analysis of the language-based constructs used in the program, is attractive in this regard. The semantics of the language constructs provides us with primitives through which we can understanding what a program does with information. This makes language-based techniques very powerful. Other techniques such as those which introduce new security constructs, for example explicit declassification constructs [MZZ+08, ZM01, Zda04b, MSZ06, AS07, BNR08], into the programming language, to ensure safe release of information also fall under language-based approach to information flow security.

Language-based techniques to security have been used in other areas which are not necessarily related to information-flow security. Examples include language and compiler mechanisms to prevent buffer overflow, which may lead to privilege escalation [NCH+05, CPM+98] vulnerabilities in C programs; and *bytecode verification*, *stack inspection*, and *sandboxing* techniques to protect local resources from networked applications and applets [LY99, Ler03, FG03, Gon99, MF97]. Another language-based approach to security is the *proof-carrying code* [NL97, CLN00] mechanism, which uses the premise that checking that a proof (of software security) is correct is easier to do than directly verifying the security of a software. Under the proof-carrying code approach, the program author generates a proof that his or her software has certain security properties, which the program consumer can easily verify.

For secure information flow, language-based approaches include static typing

systems and dependency analyses [VSI96, VS97, AB04, HS06] where well-typed programs are guaranteed to satisfy a noninterference property, semantics-based analysis of secure information flow [JL00, SS01, GM04], information-theoretic measures of information flows [Den82, CHM02, PHW02], and specialised languages with explicit declassification constructs [MZZ+08, ZM01, Zda04b, MSZ06, AS07, BNR08]. Other language-based approaches to information security include complexity-theoretic analyses [VS00, Lau01, Lau03, BL06] that characterise the security of information based on the complexity of extracting such information from a protected system, and runtime monitor-based approaches where monitors are attached to a program to prevent insecure executions [BD03, GBJS06, SST07, CC08].

2.2 Multilevel security

One of the first approaches, which uses static analysis for the enforcement of noninterference in programs, is due to Denning and Denning [Den76, DD77]. In their work [DD77], a security policy is a pair $\langle S, \rightarrow \rangle$, where S is a finite set of security classes arranged on a lattice, and $\rightarrow \subseteq S \times S$ is a flow relation, which specifies permitted information flow between pairs of security classifications. Objects x and y in a system are assigned security classes $\underline{x}$ and $\underline{y}$ respectively, and information is permitted to flow from x to y if and only if $(\underline{x}, \underline{y}) \in \rightarrow$ (written as $\underline{x} \rightarrow \underline{y}$). The flow relation $\rightarrow$ is reflexive (so that information may flow within the same class) and transitive.

The lattice properties of the set S are exploited to make program certification more efficient. The *least upper bound* operation ($\vee$) and the *greatest lower bound*

operation ($\wedge$) on S are defined so that for any index set I such that for all $i \in I, \underline{x_i} \rightarrow \underline{y}$ then $(\bigvee_{i \in I} \underline{x_i}) \rightarrow \underline{y}$. Similarly, if for all $j \in J, \underline{x} \rightarrow \underline{y_j}$, then $\underline{x} \rightarrow (\bigwedge_{j \in J} \underline{y_j})$. The class $\bigvee_{i \in I} \underline{x_i}$ is viewed as a common class via which information flows from the various classes $\underline{x_i}$ to the class $\underline{y}$, and similarly the class $\bigwedge_{j \in J} \underline{y_j}$ is a common class through which information flows from $\underline{x}$ to the various classes $\underline{y_j}$. Under this framework, information is said to flow from object x to another object y (written as $x \Rightarrow y$), when the information stored in x is transferred to, or is used to derive the information that is transferred to the object y. Information flow is said to be *explicit* if the value assigned to object y is computed directly from the value of the object x such as during the function assignment $y := f(\cdots, x, \cdots)$, or *implicit* when the subprogram that assigns a value to y is executed conditionally on x such as in the program `if(`$x = 0$`)then` $y := 0$ `else` $y := 1$.

A program P is said to be secure when the information flow $x \Rightarrow y$ is *specified* by P only if $\underline{x} \rightarrow \underline{y}$. A *certification mechanism* presented in [DD77] checks this *security condition.* The certification mechanism uses the observation that checking whether a program P specifies the flow $x \Rightarrow y$ can be efficiently done through a static *certification condition* which checks when information *might* flow from x to y. For example, the program `if(`$x \geq 10 \,\&\, x = 9$`)then` $y := 0$ `else skip` suggests that information might flow from x to y because y may be assigned in a control-flow context that is predicated on the value of x. However, no information actually flows from x to y because there is no execution of this program under which the assignment to y is reached. The certification conditions are purely syntactic, and they are computed from the security classes of the objects used in each construct. For example, the certification condition for the assignment $y := x$ requires that the classification of x must be strictly below or at most equal

to the classification of y, that is, $\underline{x} \leq \underline{y}$. For implicit flows such as in the program `if(`$x \leq y$`)then` $z_1 = 0$ `else` $z_2 := 1$, the certification condition is that $\underline{x} \vee \underline{y} \leq \underline{z}_1 \wedge \underline{z}_2$ - that is, the least upper bound of the classifications of x and y is at most equal to the greatest lower bound of the classifications of the assigned variables z_1 and $\underline{z}_2$. The purely syntactic nature of the certification conditions and the use of lattice operation on the security classes of the objects involved makes it possible to certify programs quickly. However, a stronger, but generally undecidable variant of the security condition above is also proposed in [DD77], which says that P is secure if and only if no execution of P results in a flow $x \Rightarrow y$ unless $\underline{x} \rightarrow \underline{y}$.

2.3 Type-based Certification

An established language-based approach for the certification of programs for secure information flow is by the use of a security type system [VSI96, HR98, VS00, Aga00, HS06], where well-typed programs are guaranteed to have a security property. For example, the type system of Volpano, Smith, and Irvine [VSI96] proves the soundness of the lattice-based multilevel-security analysis of [Den76, DD77] as a statement of the noninterference property of the program. By proving the soundness with respect to standard program semantics, well-typed programs are shown in [VSI96] to have the required noninterference property.

Under the type system of [VSI96], program phrases are assigned security types which are drawn from a partially ordered set $\langle S, \leq \rangle$. The primitive types $\tau \in S$ are the so-called "*data types*", which are similar to the security classes of Denning [Den76, DD77]. *Program phrases*, which are expression phrases and

command phrases, are assigned phrase types ρ. Expression phrases may be ordinary program identifiers or memory *locations*. The language does not have input-output primitives, however input is achieved by dereferencing an explicit location and output is achieved by assignment to an explicit location. The syntax of the block-structured language is the following.

$$
\begin{array}{llll}
(\textit{phrases}) & P & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e < e' \\
(\textit{commands}) & c & ::= & x := e \mid c; c' \mid \texttt{if(}e\texttt{)then}\, c\, \texttt{else}\, c' \mid \\
 & & & \texttt{while(}e\texttt{)}\, \texttt{do}\, c \mid \texttt{letvar}\ x := e\ \texttt{in}\ c
\end{array}
$$

Variables are ranged over by x, and l ranges over locations. All expressions are integers, where n is an integer literal, and, 0 and 1 are used as conditional guards. The phrase types are defined as follows.

$$
\begin{array}{llll}
(\textit{data type}) & \tau & ::= & s \\
(\textit{phrase type}) & \rho & ::= & \tau \mid \tau\ var \mid \tau\ cmd
\end{array}
$$

The metavariable s ranges over of security classes in S, and the type $\tau\ var$ is the type of variables and locations, whereas the type $\tau\ cmd$ is the type of commands. Under the proposed type system, typing judgements have the form

$$\Lambda; \Gamma \vdash P : \rho. \tag{2.1}$$

The finite maps Λ and Γ, which are respectively called *location* and *identifier* typing, assign types to locations and identifiers. These maps may be updated as

usual, where

$$\Gamma[x:\rho](x') \triangleq \begin{cases} \rho & \text{if } x = x' \\ \\ \Gamma(x') & \text{otherwise.} \end{cases}$$

The judgement of (2.1) means that the phrase P has type ρ under the assumption that Λ and Γ respectively prescribe types for the locations and the free identifiers in P. If the phrase is an expression e, with the typing judgement τ, the intuition is that e contains expressions at level τ or lower. However, if the phrase is a command c the type $\tau\ cmd$ means that c only assigns to variables at level τ or higher. These properties, called the *simple security* and the *confinement* properties are enforced by the typing rules. The full typing rules and the subtyping conditions are given in Figure 2.1 and Figure 2.2 respectively. Using both the simple security and the confinement properties of the typing system, well-typed programs can be shown to satisfy a noninterference property.

In order to ensure that explicit information flows are secure, the typing rule for the assignment statement requires that the variable (x) assigned to, and the assigned expression (e), must agree on their security levels (τ). An upward flow is still allowed during assignment since the type of e can be coerced up by the subtyping rule. Conditional commands are made secure with respect to implicit information flow by requiring that the type of the conditional guard and the branch(es) must agree. As the subtyping judgements of Figure 2.2 shows, the subtyping rule for commands is *antimonotone*, since if $\tau \subseteq \tau'$, and if a command phrase is judged to have a type $\tau'\ cmd$, then that command phrase only assigns to variables which are judged to have type τ' or higher.Thus, an even stronger statement is that the

$$\text{(INT)} \quad \Lambda;\Gamma \vdash n : \tau$$

$$\text{(VAR)} \quad \Lambda;\Gamma \vdash x : \tau\ var \quad \text{if } \Gamma(x) = \tau\ var$$

$$\text{(VARLOC)} \quad \Lambda;\Gamma \vdash l : \tau\ var \quad \text{if } \Lambda(x) = \tau\ var$$

$$\text{(ARITH-op)} \quad \frac{\Lambda;\Gamma \vdash e : \tau,\ \Lambda;\Gamma \vdash e' : \tau}{\Lambda;\Gamma \vdash e\ op\ e' : \tau} \quad op \in \{+, -, =, <\}$$

$$\text{(R-VAL)} \quad \frac{\Lambda;\Gamma \vdash e : \tau\ var}{\Lambda;\Gamma \vdash e : \tau}$$

$$\text{(ASSIGN)} \quad \frac{\Lambda;\Gamma \vdash x : \tau\ var,\ \Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash x := e : \tau\ cmd}$$

$$\text{(COMPOSE)} \quad \frac{\Lambda;\Gamma \vdash c : \tau\ cmd,\ \Lambda;\Gamma \vdash c' : \tau\ cmd}{\Lambda;\Gamma \vdash c; c' : \tau\ cmd}$$

$$\text{(IF)} \quad \frac{\Lambda;\Gamma \vdash e : \tau,\ \Lambda;\Gamma \vdash c : \tau\ cmd,\ \Lambda;\Gamma \vdash c' : \tau\ cmd}{\Lambda;\Gamma \vdash \texttt{if(}e\texttt{)then}\ c\ \texttt{else}\ c' : \tau\ cmd}$$

$$\text{(WHILE)} \quad \frac{\Lambda;\Gamma \vdash e : \tau,\ \Lambda;\Gamma \vdash c : \tau\ cmd}{\Lambda;\Gamma \vdash \texttt{while(}e\texttt{)}\ \texttt{do}\ c : \tau\ cmd}$$

$$\text{(LETVAR)} \quad \frac{\Lambda;\Gamma \vdash e : \tau,\ \Lambda;\Gamma[x : \tau\ var] \vdash c : \tau'\ cmd}{\Lambda;\Gamma \vdash \texttt{letvar}\ \ x := e\ \texttt{in}\ \ c : \tau'\ cmd}$$

Figure 2.1: *The Volpano-Smith-Irvine Typing rules*

command assigns to variables at τ or higher, and hence $\tau'\ cmd \subseteq \tau\ cmd$. This fact is exploited for upward flows from the guard to the branches, which allows branches typed at a higher level to be predicated on a lower guard, by either coerc-

$$\text{(BASE)} \quad \frac{\vdash \tau \leq \tau'}{\vdash \tau \subseteq \tau'}$$

$$\text{(REFLEX)} \quad \vdash \rho \subseteq \rho$$

$$\text{(TRANS)} \quad \frac{\vdash \rho \subseteq \rho', \quad \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''}$$

$$\text{(CMD}^{-}\text{)} \quad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau'\ cmd \subseteq \tau\ cmd}$$

$$\text{(SUBTYPE)} \quad \frac{\Lambda; \Gamma \vdash P : \rho, \quad \vdash \rho \subseteq \rho'}{\Lambda; \Gamma \vdash P : \rho'}$$

Figure 2.2: *The Volpano-Smith-Irvine Subtyping rules*

ing the type of the branches downwards using the antimonotonicity of commands, or by using the usual (upward) coercion subtyping rule for the conditional guard. Various extensions to the programming language and the type system have been proposed [SV98, VS00, Smi01, Smi03, Smi06].

2.4 Dependency Analysis

Another static approach to checking whether a program satisfies noninterference is via a dependency analysis of variables in the program [ABHR99, AB04, HS06]. This is based on the premise that in a setting where the attacker can only observe the final values of public variables on program termination, the static independency of the final value of a public variable on the initial secret values stored in other variables of a program is a static approximation of the noninterference property of that program.

Amtoft and Banerjee [AB04] presented a framework for the static enforcement

of noninterference policies via a variable-independency analysis. The analysis is based on an abstract interpretation of program traces that makes explicit the independencies between program variables. Being a data-flow analysis, the inference logic deems more program secure than security type systems such as [VSI96]. For example, under the [VSI96], if we have $\Gamma(h) = H$ and $\Gamma(l) = L$ for some variables h and l, and where $L < H$ the program, the program $l := h; l = 0$ is not typable because the first statement directly assigns a higher security-typed expression to a variable that has a lower security type. Although the program contains an insecure subprogram, however the program as a whole is secure because the high content of l is overwritten by the constant 0 during the second assignment before the attacker can observe the content of l (on termination). This property is called *flow-sensitivity*, where the order of commands matters. Such a secure composition of insecure programs is detectable in type systems of [AB04, HS06], for example, but the type system of [VSI96] cannot detect this. This is because the type system of [VSI96] does not take into account the order of program execution, and is therefore flow-insensitive.

The inference rules of [AB04] are presented in a Hoare-like logic, which derives independencies between program variables on termination of a program fragment, given the independencies before the execution of that fragment. In the previous example, l becomes dependent on h after the first assignment, but becomes independent after the second assignment (written as $[l\#h]$, which means that the value of variable l is *independent* of the initial value of variable h). The analysis of [AB04] conservatively extends the type system of [VSI96], where well-typed programs in the system of [VSI96] satisfy the invariant $[l\#h]$ in the system of [AB04], which asserts that, on program termination, the value of an L-typed l

variable is independent of the initial value of an H-typed variable h.

Judgements in the system of [AB04] are of the form

$$G \vdash \{T_1^\#\}\ P\ \{T_2^\#\}. \tag{2.2}$$

Given the set $\mathbf{Var}$ of variables, the sets $T_1^\#, T_2^\# \in \mathbf{Independ} = \mathcal{P}(\mathbf{Var} \times \mathbf{Var})$ are sets of variable independencies, and $G \subseteq \mathbf{Var}$ is an *implicit-flow context* describing the set of variables, values of which the program control flow might depend on. The set G is used to eliminate implicit information flows. The meaning of (2.2) is now that if the independencies in $T_1^\#$ hold *before* the program P is executed under the control context G, then, provided that P terminates, the independencies in $T_2^\#$ hold *after* the execution of P. The intended meaning of any $[z\#x] \in T_2^\#$ is that the *final* value of z after executing P is independent of the *initial* value of x from the computation preceding P.

The independency inference logic of [AB04] is shown in Figure 2.3. In the rules $FV(e) \subseteq \mathbf{Var}$ is the set of free variables of the expression e and the partial order relation $\preceq$ on sets of independencies is defined as the reverse subset inclusion order on independencies: $T_1^\# \preceq T_2^\#$ iff $T_2^\# \subseteq T_1^\#$.

An equivalent derivation to [AB04] was presented by Hunt and Sands [HS06], which is based on standard semantics. As opposed to the approach of [AB04], which computes *independencies* of variables, the inference rules of [HS06] computes variable *dependencies* directly when the lattice of dependency is chosen to be the powerset lattice $\mathcal{P}(\mathbf{Var})$ of variables. The dependency result of [HS06] is the De-Morgan's dual of the independency computation of [AB04], and both type systems enforce a partial correctness noninterference property for well-typed

$$\text{[Assign]} G \vdash \{T_0^\#\}\, x := e\, \{T^\#\} \quad \text{if } \forall [y\#w] \in T^\#.\ x \neq y \implies [y\#w] \in T_0^\#,\ x = y \implies w \notin G \wedge \forall z \in FV(e).[z\#w] \in T_0^\#$$

$$\text{[Seq]} \frac{G \vdash \{T_0^\#\}\, c\, \{T_1^\#\} \quad G \vdash \{T_1^\#\}\, c'\, \{T_2^\#\}}{G \vdash \{T_0^\#\}\, c; c'\, \{T_2^\#\}}$$

$$\text{[If]} \frac{G_0 \vdash \{T_0^\#\}\, c\, \{T^\#\} \quad G_0 \vdash \{T_0^\#\}\, c'\, \{T^\#\}}{G \vdash \{T_0^\#\}\, \texttt{if(}e\texttt{)then}\, c\, \texttt{else}\, c'\, \{T^\#\}} \quad \text{if } G \subseteq G_0 \text{ and } w \notin G_0 \implies \forall z \in FV(e).[z\#w] \in T_0^\#$$

$$\text{[While]} \frac{G_0 \vdash \{T^\#\}\, c\, \{T^\#\}}{G \vdash \{T^\#\}\, \texttt{while}(e)\, \texttt{do}\, c\, \{T^\#\}} \quad \text{if } G \subseteq G_0 \text{ and } w \notin G_0 \implies \forall z \in FV(e).[z\#w] \in T^\#$$

$$\text{[Sub]} \frac{G_1 \vdash \{T_1^\#\}\, c\, \{T_2^\#\}}{G_0 \vdash \{T_0^\#\}\, c\, \{T_3^\#\}} \quad \text{if } G_0 \subseteq G_1 \text{ and } T_0^\# \preceq T_1^\# \text{ and } T_2^\# \preceq T_3^\#$$

Figure 2.3: *Amtoft-Banerjee Independency Logic*

programs. The algorithmic version of the dependency type system of [HS06] is presented in Chapter 6, where it is compared with a dependency analysis proposed in this thesis.

2.5 Equational Characterisation

Joshi and Leino [JL00] proposed a semantics-based equational characterisation of the noninterference property of a system, via a special program construct HH (also known as "*havoc on* h", which stands for the "destruction" of the *high* portion of memory). Under this approach, in the simplest case, it is assumed that program memory is partitioned to two, namely, the *high-security* portion h (which is a tuple over the domains of high-security variables) and the *low-security* por-

tion l (which is a tuple over the domains of low-security variables). Furthermore, it is also assumed that the attacker has the program sources and can observe the l-portion of the memory before and after program execution, but cannot directly observe the h-portion. The intention is that, in a secure program, information should not flow from the h-portion of the memory to the l-portion, although the reverse flow is permitted.

A relational semantics is used to describe the properties of programs such that for any program P, $\sigma \langle P \rangle \sigma'$ signifies that there is an execution of P from the initial state σ to the final state σ'. A special "looping state" ∞ is also considered, which has the property that for all programs P, $\infty \langle P \rangle \infty$ holds, preventing any program from exiting the looping state. For any variable x, the evaluation of x at the looping state ∞ is $\infty(x) = \perp_x$, which is taken to be a special value outside the domain of x. The semantics of HH is defined such that for all states σ, σ', we have $\sigma \langle HH \rangle \sigma'$ iff $\sigma(l) = \sigma'(l)$.

Using the program HH, the security property of a program P is formalised as a total correctness program equivalence as follows.

A program P is secure iff

$$HH; P; HH =_l P; HH. \tag{2.3}$$

The relation $=_l$ in this definition may be viewed as comparing only the l-portions of the memory. The intuition behind definition (2.3) is that regardless of the initial values in the h-portion of the memory in the two programs, as long as the initial l-portion is the same, then the final l-portions of memory agree. The trailing HH on both sides of $=_l$ means that we do not care about the final values in the h-

portions of memory, since it cannot be observed by the attacker. If programs are replaced by their relational semantics in (2.3) and ; stands for the relation composition operator, then the relation $=_l$ is simply the equality of relations. This simple semantic definition captures the noninterference property of the program P because the initial h-portion of the memory is noninterfering with the final result of the l-portion of memory whenever P is secure. This definition has motivated the work of [BGM07], and is related to the PER-based semantic definition of [SS01].

2.6 PER Model of Information Flow

It is a well-known fact in language-based information security that the notion of noninterference in security is closely related to the notion of (in)dependencies. Motivated by the work of Hunt, which used PERs to construct the abstract interpretation of strictness properties in higher-order functional programs [Hun91a, Hun91b], and, in particular, to model dependencies in *binding time analyses* [HS91], Sabelfeld and Sands proposed a PER model for the analysis of secure information flow [SS01, Sab01]. The PER model, which is semantics-based, was shown in [SS01] to generalise the semantic characterisation of security of [JL00].

The idea behind the PER-based characterisation is the observation that the *variation* (or lack of it) in the publicly observable output of a program relative to the *variation* in the private input to the program can be modelled by PERs on the output and input domains of the program respectively. By fixing all other inputs in a deterministic program, if no variation is observable in a given public output under all variations of a confidential input, then there is no information flow from that input to that output via the program. This intuition is captured by

PER-transformer relation, $\twoheadrightarrow$ (defined below), between PERs over the program's input and output domains.

For simplicity, it is assumed that program inputs and outputs are partitioned to two parts, namely, a *high-security* part (whose domain is the set H, say), which is not publicly observable. The second part is a *low-security* part (whose domain is the set L) that is observable publicly only before and after, but not during, program execution. Programs (P) whose denotations are maps of the form $[\![P]\!] : H \times L \to H \times L$ are considered. Now let $\mathrm{PER}(S)$ be the set of all PERs over the set S, and for any two PERs $R_1 \in \mathrm{PER}(S_1)$ and $R_2 \in \mathrm{PER}(S_2)$, define the PER $R_1 \bullet R_2$ such that for any $(s_1, s_2), (s_1', s_2') \in S_1 \times S_2$, $(s_1, s_2)\ R_1 \bullet R_2\ (s_1', s_2')$ iff $s_1\, R_1\, s_1'$ and $s_2\, R_2\, s_2'$. The security property of the program P is described in terms of its denotational semantics, such that if $Q, Q' \in \mathrm{PER}(H)$ and $R, R' \in \mathrm{PER}(L)$, then $(\!|P|\!) : (Q \bullet R) \twoheadrightarrow (Q' \bullet R')$ holds iff for all $(h, l), (h', l') \in H \times L$

$$(h,l)\ Q \bullet R\ (h',l') \implies [\![P]\!](h,l)\ Q' \bullet R'\ [\![P]\!](h',l'). \tag{2.4}$$

The intuition behind (2.4) is that under the PERs $Q \bullet R$ and $Q' \bullet R'$ defined respectively over the input and output domains of P, any pair of inputs that is *indistinguishable* by $Q \bullet R$ (that is, pairs of inputs that are are related by this PER), lead to outputs of P that are also indistinguishable by the PER $Q' \bullet R'$. The definition is compositional so that for two programs P and P' we have

$$\frac{(\!|P|\!) : A \twoheadrightarrow B, \quad (\!|P'|\!) : B \twoheadrightarrow C}{(\!|P; P'|\!) : A \twoheadrightarrow C}$$

Let $id_S, all_S \in \mathrm{PER}(S)$ be PERs over S defined such that for all $s, s' \in S$,

$s\ id_S\ s'$ iff $s = s'$ and $s\ all_S\ s'$. Using (2.4) the noninterference security condition for the program P is now the following.

The program P is secure iff

$$(\!|P|\!) : (all_H \bullet id_L) \rightarrowtriangle (all_H \bullet id_L). \tag{2.5}$$

This means that if an attacker can observe the value of the public input only (the id_L part of the PER over the input domain), then for each possible value of the public output (the id_L part of the PER over the output domain), all the values of the secret input are possible (the all_H part of the PER over the input domain). That is, if the public input is fixed, any variation in the secret input is not observable by the attacker in a secure program. This is a statement of the noninterference requirement of [GM82]. The security definition of (2.5) is termination-sensitive by requiring that the termination properties of a secure program must not be influenced by the values of secret inputs. By defining the PERs over appropriate powerdomains, the definition of (2.5) is shown to also describe the security properties of nondeterministic systems.

2.7 Abstract Noninterference

Giacobazzi and Mastroeni introduced in [GM04, GM05] a notion of abstract noninterference as a semantic description of the information released by a program based on standard techniques from abstract interpretation [CC77, CC79]. The core idea is that instead of observing the concrete values of the public input and output data in a program, the attacker is modelled as an *abstract interpretation*

that can observe only the properties of these data, that is, the abstract semantics of the program. By weakening the observational capability of the attacker so that the attacker is only able to observe *properties* of data, the noninterference requirement can be weakened since an otherwise offending program under noninterference may become safe in the presence of an attacker that cannot observe public input and output precisely.

The concrete domain C is taken to be the powerset lattice of concrete program values with the subset order relation. As usual, the concrete domain is partitioned to two sets H and L, which are the domains of confidential and public values respectively. Let $\mathit{uco}(\mathcal{P}(C))$ be the set of all *upper closure operators* on the ordered set $\langle \mathcal{P}(C), \subseteq \rangle$, the abstract domains are based on upper closure operators $\eta, \rho \in \mathit{uco}(\mathcal{P}(L))$ and $\phi \in \mathit{uco}(\mathcal{P}(H))$, which are defined over the concrete domain of program values. Under this framework, the attacker is modelled as a pair of abstractions $\langle \eta, \rho \rangle$, where η and ρ model respectively the attacker's observational power over the public input and output values. The concrete semantics of the program P is formalised using *angelic denotational semantics*, which associates an input-output function, $[\![P]\!] : H \times L \to H \times L$, with P and ignores nontermination. Furthermore, the observation of (public) values occur at the beginning of program execution and on program termination. To slightly simplify the notations, we shall denote the concrete semantics of P as a map $[\![P]\!] : H \times L \to L$, throwing away the H projection of state on termination, which is not used. Additionally, for singleton sets we shall write $\eta(l)$ instead of $\eta(\{l\})$ for the image of $\{l\}$ under η. A program P is said to satisfy the *narrow abstract noninterference*

(NANI), written as $[\eta]P(\rho)$, when for all $h, h' \in H$ and $l, l' \in L$

$$\eta(l) = \eta(l') \implies \rho([\![P]\!](h,l)) = \rho([\![P]\!](h',l')). \tag{2.6}$$

The intuition behind definition (2.6) is that if the attacker can only observe the properties η and ρ respectively of the public input and public output, then no information about the secret input flows via P whenever $[\eta]P(\rho)$ holds.

A problem with this definition is the so-called notion of *deceptive flows*, where a program that fails to satisfy the NANI property may still not reveal any information about secrets. To see why, let the set of *even*, *odd*, *positive* (including 0), and *negative* integers be respectively defined as $\textbf{\textit{Even}} \triangleq \{2i \mid i \in \mathbb{Z}\}$, $\textbf{\textit{Odd}} \triangleq \{2i+1 \mid i \in \mathbb{Z}\}$, $\textbf{\textit{Pos}} \triangleq \{i \in \mathbb{Z} \mid i \geq 0\}$, $\textbf{\textit{Neg}} \triangleq \{i \in \mathbb{Z} \mid i < 0\}$. Now suppose $h \in H \triangleq \mathbb{Z}$ and $l \in L \triangleq \mathbb{Z}$ and consider the program $l := l \times h^2$ under the *parity* and *sign* abstraction pair $\langle \eta, \rho \rangle \triangleq \langle \textbf{\textit{Par}}, \textbf{\textit{Sgn}} \rangle$, which are given by their set of fixpoints[1] $\mathit{fix}(\textbf{\textit{Par}}) = \{\mathbb{Z}, \textbf{\textit{Even}}, \textbf{\textit{Odd}}, \varnothing\}$ and $\mathit{fix}(\textbf{\textit{Sgn}}) = \{\mathbb{Z}, \textbf{\textit{Pos}}, \textbf{\textit{Neg}}, \varnothing\}$. If an attacker can only observe the parity of l before executing this program and its sign afterwards, then that attacker cannot gain any information about h since the sign of h has been destroyed in the final value of l by taking the square of h. However, the property $[\textbf{\textit{Par}}]l := l \times h^2(\textbf{\textit{Sgn}})$ does not hold. This is due to ***Par***-indistinguishable l-input values that are ***Sgn***-distinguishable causing the "deceptive flow". To eliminate this flow, a check is performed instead on the *set of outputs*, with a fixed η-property on the input. This is denoted as $(\eta)P(\rho)$, which

[1] Closure operators are completely determined by their set of fixpoints.

holds if for all $h, h' \in H$ and $l \in L$

$$\rho(\bigcup_{l' \in \eta(l)} \{[\![P]\!](h, l')\}) = \rho(\bigcup_{l' \in \eta(l)} \{[\![P]\!](h', l')\}). \tag{2.7}$$

The definition of NANI, can further be weakened to allow information flow about secret inputs. This information flow about secret is specified by the upper closure operator $\phi \in \mathit{uco}(\mathcal{P}(H))$ on secrets. The resulting notion is called *abstract noninterference* (ANI), written as $[\eta]P(\phi \rightsquigarrow [\![\rho)$, which holds if for all $h, h' \in H$ and $l \in L$

$$\rho(\bigcup_{\substack{h_1 \in \phi(h) \\ l' \in \eta(l)}} \{[\![P]\!](h_1, l')\}) = \rho(\bigcup_{\substack{h_2 \in \phi(h') \\ l' \in \eta(l)}} \{[\![P]\!](h_2, l')\}). \tag{2.8}$$

The meaning of the ANI definition of (2.8) is that under the fixed attacker model $\langle \eta, \rho \rangle$, the attacker cannot gain the information characterised by the upper closure operator ϕ. The idea is that by fixing l to the property η (to eliminate "deceptive flows") and evaluating P under all variations of h that are constrained by the property ϕ, the attacker observing the ρ property of the public output cannot see any difference. This is referred to as "*declassified ANI via blocking*" in [Mas05], since the property ϕ cannot be observed. A related notion, called "*declassified ANI via allowing*", *allows* the property ϕ to be observed. This is denoted as $(\eta)P(\phi \implies \rho)$ and is defined as $\forall h, h' \in H$ and $\forall l \in L$

$$\phi(h) = \phi(h') \implies \rho(\bigcup_{l' \in \eta(l)} \{[\![P]\!](h, l')\}) = \rho(\bigcup_{l' \in \eta(l)} \{[\![P]\!](h', l')\}). \tag{2.9}$$

Under this notion, we only check that the attacker cannot observe a difference

under pairs of h-values with the same ϕ property. Since the attacker may be able to observe a difference in ρ when $\phi(h) \neq \phi(h')$ information about ϕ *may* flow.

2.8 Language-based declassification

Another language-based technique for the enforcement of secure information flow uses explicit *declassification* constructs that are added to the programming language, so that intentional release of information may only be performed by using a declassification construct. This approach has been well studied [MZZ+08, ZM01, SM03b, CM04, Zda04b, MSZ06, AS07, BNR08].

Zdancewic, Myers, and Sabelfeld introduced a notion of *robust declassification* [ZM01, Zda04b, MSZ06], which features a language-based declassification construct for the controlled release of information. However, the provision of an information downgrading construct raises the question of whether the declassification mechanism can be exploited by attackers to launder information. A notion of *robustness* ensures the safety of the declassification mechanism so that neither attacker-injected values nor attacker-inserted code can be used to control *what* information is released, or *whether* information is released. This means that, due to robustness, an *active* attacker, which can both modify and observe a system cannot gain more information than a *passive* attacker that can merely observe the system.

The security model is based on a lattice $\mathcal{L} \triangleq \mathcal{L}_C \times \mathcal{L}_I$ derived from the product of a *confidentiality* policy lattice $\mathcal{L}_C$ and an *integrity* policy lattice $\mathcal{L}_I$, which are used to reason about both the confidentiality and integrity of data as well as the integrity of code in the system. The notion of integrity is a dual notion of con-

fidentiality. High-integrity data (and code) are trusted and are assumed not to be under the control of attackers, whereas low-integrity ones are not trusted and are assumed to be under the control of the attacker. The lattice $\mathcal{L}$ is partially ordered by $\sqsubseteq$, and attackers are assigned security levels such that an attacker A, characterised by its level $\ell_A \in \mathcal{L}$, may only view information at confidentiality level[2] $\pi_1(\ell_A)$ and below on the confidentiality lattice $\mathcal{L}_C$. Furthermore, this attacker can only modify data at integrity level $\pi_2(\ell_A)$ and above on the integrity lattice $\mathcal{L}_I$. Under this framework, a typing environment, $\Gamma : \mathbf{Var} \to \mathcal{L}$, assigns security types to variables. Expression types are derived by taking the least upper bound of the types of the free variables of that expression. With the exception of the declassification expression, the programming language is largely standard as shown in Figure 2.4.

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid e_1 \mathit{op}\, e_2 \mid \textbf{\textit{declassify}}(e, \ell) \\
c & ::= & \texttt{skip} \mid x := e \mid c_1; c_2 \mid \\
 & & \texttt{if(}e\texttt{)then}\, c_1\, \texttt{else}\, c_2 \mid \texttt{while}\,(e)\,\texttt{do}\, c
\end{array}
$$

Figure 2.4: *Language-based Declassification*

The operation *op* stands for the usual arithmetic and boolean operations on expressions and $\ell \in \mathcal{L}$ is a security level. The declassification expression ***declassify***(e, ℓ) has the same operational semantics as the expression e. However, ***declassify***(e, ℓ) allows the security level of e to be declassified to the level $\ell \in \mathcal{L}$. Thus, the declassification mechanism is used to control the security level of information, which is checked statically, and is intended to have no semantic effect on pro-

[2]The notation $\pi_i(\cdot)$ is the i^{th} projection, and the confidentiality lattice is arranged from top to bottom with the highest confidentiality at the top, whereas the integrity lattice is arranged with the lowest integrity at the top.

gram execution. Since information may flow from variable x to another variable y only if $\Gamma(x) \sqsubseteq \Gamma(y)$, the choice of lattice $\mathcal{L}$ and the typing environment Γ specifies a *security policy*. The security framework is formalised as a type system so that well-typed programs satisfy the robustness property. The full type system is shown in Figure 2.5.

$$\frac{}{\Gamma \vdash n : \ell} \qquad \frac{\Gamma(x) = \ell}{\Gamma \vdash x : \ell}$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma \vdash e' : \ell}{\Gamma \vdash e \; op \; e' : \ell} \qquad \frac{\Gamma \vdash e : \ell \quad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell'}$$

$$\frac{}{\Gamma, pc \vdash \texttt{skip}} \qquad \frac{\Gamma \vdash e : \ell \quad \ell \sqcup pc \sqsubseteq \Gamma(x)}{\Gamma, pc \vdash x := e}$$

$$\frac{\Gamma \vdash e : \ell' \quad \ell \sqcup pc \sqsubseteq \Gamma(x) \quad \pi_2(\ell) = \pi_2(\ell') \quad pc, \ell' \in \{\ell'' \in \mathcal{L} \mid \pi_2(\ell_A) \not\sqsubseteq \pi_2(\ell'')\}}{\Gamma, pc \vdash x := \textbf{\textit{declassify}}(e, \ell)} \qquad \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c_1 \quad \Gamma, \ell \sqcup pc \vdash c_2}{\Gamma, pc \vdash \texttt{if}(e)\texttt{then}\; c_1 \; \texttt{else}\; c_2} \qquad \frac{\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup pc \vdash c}{\Gamma, pc \vdash \texttt{while}\,(e)\,\texttt{do}\; c}$$

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

Figure 2.5: *Robustness Typing Rules*

The typing system is fairly straightforward and is parametric to the environment Γ and the attacker level ℓ_A, against which the typable program is robust. The $pc \in \mathcal{L}$ level is used to rule out implicit flow of information and also to ensure that the attacker cannot control whether declassification can take place or not. The important rule is the typing judgement for the assignment $x := \textbf{\textit{declassify}}(e, \ell)$, where the expression e with type ℓ' is to be declassified to the level ℓ. For this to be successful, the security level of the assigned variable x must be at least $\ell \sqcup pc$,

ensuring that no implicit information flows to x and that it cannot be corrupted by a lower integrity data. Furthermore, it is required that declassification should not change the integrity of the declassified information ($\pi_2(\ell) = \pi_2(\ell')$), and that the "attacker", which has control over whether the declassification expression is executed (the pc part) and which might have tainted the data in e (the ℓ' part) must have an integrity level that is strictly greater than the integrity level of the declassified expression e so that $pc, \ell' \in \{\ell'' \in \mathcal{L} \mid \pi_2(\ell_A) \not\sqsubseteq \pi_2(\ell'')\}$.

2.9 Information-theoretic Characterisation

Qualitative definitions of information flow describe what information is released only in a possibilistic sense. That is, they specify whether it is *possible* or not that certain information may be released by a system, but they do not usually capture the notion of how *likely* it is for that information flow to occur. While it may be possible that certain information may be released by a system, it may be extremely unlikely that such information flow may occur. Quantitative measures of information flow, in particular, information-theoretic characterisation can capture a sense of how likely it is for information to flow in the amount of information released.

In cases where the semantics of a system is characterised by probability distributions, information-theoretic measures of information flow can be particularly useful. Even in cases where the semantics of a system is deterministic, but where the choice of inputs to the system is governed by probability distributions, it is still possible to apply information-theoretic techniques to characterise the information release. The basic model of security under quantitative characterisations is similar to the qualitative definitions such as the traditional noninterference def-

inition. However, instead of checking whether information may flow as is done under noninterference, quantitative approaches seek to assign a quantity to the amount of information that flows. A system that has no probabilistic information flow, for example, will also satisfy the standard noninterference requirement.

One of the earliest application of information theory to information flow in a language-based setting is by Denning [Den82]. Since then, the use of quantitative techniques, especially information theory, for information flow has been an active area of research [CHM02, PHW02, Low02, ABG04, CHM05, CMS05, Bac05, OCC06, Mal07, Smi07, CHM07, AP08].

In [CHM05] an analysis technique is presented, which computes an upper bound of the amount of information released in programs written in a deterministic imperative language with a looping construct. The analysis of [CHM05] has two parts. Firstly, a Use-Definition Graph (UDG) [Muc97, NNH99] of the program is extracted from the program source, which will be used to guide the quantitative analysis. Secondly, a quantitative analysis which assigns upper bounds to the amount of information flow along paths of the UDG is then performed.

Since the probability distribution of the low program input may be in the control of the attacker, it is assumed that the attacker chooses input values to maximise the leakage of information. Each program point, corresponding to a node on the UDG, is assigned a random variable X (which may be a tuple of variables), where $P(X^n = x)$ is the probability that X takes on the value x at the node n of the UDG. Two distinguished nodes are identified, namely, the program *entry* and *exit* nodes, denoted respectively as ι and ω. So, X^ι and X^ω correspond to the random variable X at program entry and exit respectively. The main idea is that in a deterministic program, once the variation in the low input has been accounted

for, any variation that is observed in the public output must be due to variations in the secret input. Hence, the leakage of information about secret input to a variable X at the exit node, denoted as $\mathcal{L}^{\omega}(X)$, is defined as

$$\mathcal{L}^{\omega}(X) = p(\omega)\, \mathcal{H}(X^{\omega} \mid L^{\iota}) \tag{2.10}$$

The measure $\mathcal{H}(X^{\omega} \mid L^{\iota})$ is the *conditional entropy* of the random variable X^{ω} given another random variable L^{ι}, and $p(\omega)$ is the probability of reaching the exit node. The variable L, as usual, stands for the low part of the memory. Thus, L^{ι} is the random variable representing the low input at the program entry point. For programs which always terminate we have $p(\omega) = 1$. The analysis of information flow itself is parametric to the program point, and in (2.10), ω may be replaced by any arbitrary node n to compute the information flow into X at that point. A more recent work [CHM07], by the authors of [CHM05], uses a syntax-directed approach to the analysis, which quantifies the amount of information released, as opposed UDGs.

Information-theoretic approaches, in general, rely on having probability measures in order to perform the analysis, and conservative assumptions usually have to be made. Like most language-based approaches to the analysis of information flow, where the attacker is assumed to supply inputs at the beginning of program execution and can only observe the final results at the end of program execution, the model of the analysis in [CHM05] is *buffered*. However, many practical programs are *interactive*, which may accept inputs and produce outputs at any point during the program execution. Program interaction introduces additional information flow problems. Quantitative analyses that consider program interactions

include [ABG04, OCC06, Bac05, AP08]. In addition to interactions, nontermination issues are also important when modelling the information released by a program. *Termination-insensitive* analyses ignore information release due to nontermination and may admit insecure programs, which release information during diverging traces. When program interactions are involved, arbitrary amount of information may be leaked through nontermination channels. The problem of information leakage in termination-insensitive analyses is studied in [AHS08].

Chapter 3

Lattice Model of Information and Information Flow

The phrase "*secure information flow*" alludes to an understanding of the notions of *information* and *information flow*. In this chapter, we present a lattice-theoretic model of information and information flow and define a notion of security using the lattice model of information for the enforcement of *what* declassification policies.

In order to check whether a system, or its model, conforms to an information flow policy, we need to analyse its information flow properties. For this purpose an extensional input-output *relational model* is presented as a primitive for the semantic analysis of information flow in both deterministic and nondeterministic systems. By using the relational model, various representations of information, suitable for the characterisation of the information flow, are derived. The derived information representations are all shown to fit into the lattice model of information. Later on, in Chapter 4, we show how to derive the relational model of a

system from the operational semantics under a given attacker model in a language-based setting.

3.1 Modelling Information and Information Flow

A fundamental property of information is the intuitive notion of *information levels*, where we say that one piece of information is *greater* or *more informative* than another. For example, information about an integer secret which reveals that it is a positive even integer is more informative than another one which only reveals that the secret is a positive integer. This suggests an ordering of information, which we shall exploit in our information model and security definition. For this reason we shall model information as *lattices*, where the associated *partial order* captures the notion of information levels. This lattice-based definition of security falls under the *what* dimension of declassification as proposed by [SS05], because it regulates the *level* of information or *what* information that we want to release.

3.1.1 A Lattice Model of Information

We consider information as elements of a complete lattice $\mathcal{I}$, such that a piece of information in $\mathcal{I}$ describes what may be learnt about secrets and such that the lattice *partial order* and *join operation* respectively model the notions of *ordering* and *combination* of information. The ordering of $\mathcal{I}$ captures when one information is greater than or equal to another, and it is closely related to the notion of information combination where the combination of a lesser information with a greater one yields the greater information.

Definition 3.1.1 (Information Lattice)**.** *Any complete lattice* $\langle \mathcal{I}, \sqsubseteq, \sqcup \rangle$ *is a* lattice of information.

In the lattice $\langle \mathcal{I}, \sqsubseteq, \sqcup \rangle$ of information, the partial order $\sqsubseteq$ models the relative degree of informativeness of the elements of $\mathcal{I}$, and the join operation $\sqcup$ models the combination of information in $\mathcal{I}$. The idempotency, commutativity, and associativity properties of the join operation agree with natural intuitions about information because idempotency says that the combination of a piece of information with itself should yield the same information [Koh03]. Similarly, the commutativity and associativity properties respectively agree with the intuitions that the order and grouping of information combination should not matter to the end result. Furthermore, for any $s, s' \in \mathcal{I}$, the lattice property, $s \sqsubseteq s'$ iff $s \sqcup s' = s'$, agrees with the idea that the combination of a lesser information with a greater one yields the greater information, where $s \sqsubseteq s'$ means that the information s is less than or at most equal to s'.

3.1.2 Information Flow

We shall define *information flow* to model how the knowledge of an observer changes due to information release. Under this model, information flow is defined as a function which transforms knowledge on a given lattice $\mathcal{I}$ of information. Hence, if $f : \mathcal{I} \to \mathcal{I}$ is an *information flow function*, then for any initial knowledge $s \in \mathcal{I}$ that the observer might have before observing the system which causes the information flow f, $f(s)$ describes the final information that this observer might gain after observing the system. To describe the observer's knowledge after receiving new information released by a system, the information flow function

must have certain properties identified in the following definition.

Definition 3.1.2 (Information flow)**.** *Let $\langle \mathcal{I}, \sqsubseteq \rangle$ be a lattice of information. An* information flow function *$f : \mathcal{I} \to \mathcal{I}$ on the lattice $\mathcal{I}$ is an extensive and monotone function. Define $\mathcal{F}lows \triangleq \{f : \mathcal{I} \to \mathcal{I} \mid f$ is extensive and monotone$\}$ to be the set of all information flows on $\mathcal{I}$.*

Similarly to the properties of the lattice of information, the properties of information flow functions $f \in \mathcal{F}lows$ are intuitive. Firstly, the extensivity property, which means that for all $s \in \mathcal{I}, s \sqsubseteq f(s)$ shows that the observer's knowledge may only *increase* by observing the system causing information flow. Secondly, the monotonicity requirement means that the greater the initial knowledge of the observer before observing the system that is releasing information, the greater the final knowledge afterwards.

The set $\mathcal{F}lows$ of all information flows on the lattice $\mathcal{I}$ of information itself is a complete lattice under the pointwise ordering of functions because $\mathcal{I}$ is a complete lattice. The least element of the resulting lattice $\mathcal{F}lows$ is the identity map, $id_{\mathcal{I}}$, on $\mathcal{I}$. This is easily shown because if there exists $f \in \mathcal{F}lows$ such that $f \sqsubseteq id_{\mathcal{I}}$, then by the pointwise order we have that for all $s \in \mathcal{I}$, $f(s) \sqsubseteq id_{\mathcal{I}}(s) = s$. This means that $f(s) \sqsubseteq s$, and since f is extensive, we have that $f(s) = s$ by the antisymmetry of $\sqsubseteq$. Hence, $f = id_{\mathcal{I}}$. The least element $id_{\mathcal{I}} \in \mathcal{F}lows$ of the set of information flows is the lattice model equivalent of the notion of noninterference (that is, lack of information flow) since the attacker *cannot change* its knowledge via $id_{\mathcal{I}}$.

3.2 Information Flow Policies

An *information flow policy*, or simply a *flow policy* or *policy*, is a statement of the (information flow) security requirements for a system. We define policies, which can be used to regulate what information is allowed to flow through a system.

Definition 3.2.1 (*What* Policies)**.** *Let $\langle \mathcal{I}, \sqsubseteq \rangle$ be a lattice of information and let $\mathcal{F}lows$ be the set of information flows over this lattice. An information flow policy with respect to the lattice $\mathcal{I}$ is a* subset *$\mathscr{P}$ of $\mathcal{F}lows$.*

An information flow $f \in \mathcal{F}lows$ is said to be permitted *or* allowed *by a policy $\mathscr{P} \subseteq \mathcal{F}lows$ iff there exists a flow function $f' \in \mathscr{P}$ such that $f \sqsubseteq f'$. Consequently, a policy $\mathscr{P}$ is* fully non-trivial *if all elements of $\mathscr{P}$ are maximal in $\mathscr{P}$.*

The ordering $f \sqsubseteq f'$ between information flow functions in this definition is the usual pointwise ordering of functions induced by the partial order $\sqsubseteq$ on the lattice $\mathcal{I}$. This partial ordering of information flow functions is used to control, or regulate, the level of information that we allow a system to release because the elements of $\mathscr{P}$ set lattice upper bounds on the information flow that the attacker is allowed to receive when the information release is permitted by the policy $\mathscr{P}$. The policy model of Definition 3.2.1 falls under the *what* dimension of declassification according to the taxonomy of [SS05, SS07].

3.2.1 Information Flow Policy Patterns

This section highlights some information flow patterns under the proposed policy model.

Noninterference Policy

By far the most studied type of information flow policy is the noninterference policy, originally introduced by [GM82], which says that

> "*one group of users, using a certain set of commands, is* noninterfering *with another group of users if what the first group does with those commands has no effect on what the second group of users can see*".

This requirement abstractly describes a system property, which implies the lack of information flow, via the system in question, from secret inputs issued by one (high security) group to public outputs observed by the other (low security) group. Thus, the noninterference requirement is an information flow policy for a system (or more precisely, its model), that prevents *any* flow of information from secret inputs to public outputs. Under our lattice-based policy model, noninterference corresponds to the policy $\{id_{\mathcal{I}}\}$, which is the identity map over the lattice $\mathcal{I}$ of information about secrets. This abstractly describes the fact that the attacker cannot benefit by observing a system which *satisfies*[1] this policy since any flow that is permitted by this policy cannot be greater than $id_{\mathcal{I}}$. The intuition is that for all information $s \in \mathcal{I}$ in the lattice of information about secrets, representing the attacker's initial knowledge, we have that the attacker's final knowledge, $id_{\mathcal{I}}(s) = s$, after observing the system remains the same. Furthermore, since $id_{\mathcal{I}}$ is the least element of the lattice $\mathcal{F}lows$ of information flows, the baseline status of the noninterference policy $\{id_{\mathcal{I}}\}$ is clear.

Although the noninterference model is very simple, it is however too strong to be useful in practice [RMMG01, Vol99a]. Policies that allow deliberate (but controlled) release of information are necessary.

[1] The formal definition of what it means, when a system satisfies a given policy, is given in section 3.3.

Unconditional Release Policy

We may wish to have partial (but unconditional) release of information $s' \in \mathcal{I}$ but not more in a system. The pattern for this under the lattice model is captured by the policy $\{f \mid \forall s \in \mathcal{I}, f(s) = s' \sqcup s\}$, which permits an attacker to learn at most s' (s in the definition being the attacker's initial knowledge). Since any flow $f' \in \mathcal{F}lows$ which is permitted by this policy has the property that $f' \sqsubseteq f$, this means that any information that the attacker gains from the system that is strictly greater than s' is what the attacker could already derive by the combination of the initial knowledge of the attacker and the declassified information s'. However, if the attacker's initial knowledge is less than s' the greatest information that the attacker is allowed to gain by this policy is s'.

A scenario where such a policy is necessary is during password authentication, where we wish to release unconditionally the information about the equality or not of the stored password and the user-supplied password. If the attacker knows the user supplied password (for example, by supplying a guess) then the attacker (by combining its initial knowledge with the outcome of the authentication attempt) either learns the stored password (in the case of a successful authentication) or learns what it is not (if the authentication attempt fails). However, if the attacker only observes the result of the authentication without knowing the supplied password (issued, for example, by another user), the most that the attacker can learn, depending on the outcome of the authentication attempt, is whether the user-supplied and the stored passwords match or not - which is exactly the information that we have declassified.

Conditional Release Policy

Another scheme is the *conditional release* pattern, where information (s') is released based on having some initial knowledge (s''). This is modelled by the policy $\{f\}$ where $\forall s \in \mathcal{I}, f(s) = s' \sqcup s$ if $s'' \sqsubseteq s$, and $f(s) = s$ otherwise. Under this scheme, the attacker gains some information on the condition that the attacker has at least a given initial information s''. A scenario where such a policy is needed is during decryption in a symmetric key system, where the plaintext may be learnt (the knowledge s') only when the decryption key is known (the knowledge s'').

Disjunctive Release Policy

Another pattern, called *disjunctive flow policy* - after the disjunctive flow pattern of [SS05], is the policy specified by fully non-trivial policies $\mathscr{P}$, where $|\mathscr{P}| \geq 2$. Take, for example, the disjunctive flow policy $\{f, f' \mid f \not\sqsubseteq f', f' \not\sqsubseteq f\} \subseteq \mathcal{F}lows$. This policy permits at most one of f or f' to be released but not *both* at the same time. It is clear that the notion of disjunctive information flow is only meaningful for incomparable information and information flows, because whenever two information are comparable then the greater already contains the lesser information. An information flow $f'' \in \mathcal{F}lows$ is permitted by the disjunctive policy $\{f, f' \mid f \not\sqsubseteq f', f' \not\sqsubseteq f\}$, when f'' is smaller than or equal to at most one of f and f' - since f and f' are incomparable. Also, a flow $f'' \sqsupseteq f \sqcup f'$, which contains both f and f' is not permitted since there is no such $f_1 \in \{f, f' \mid f \not\sqsubseteq f', f' \not\sqsubseteq f\}$ for which $f'' \sqsubseteq f_1$.

3.3 Secure Information Flow

Let us now define a notion of security, which uses the lattice of information to formalise when the information released by a system is secure.

Definition 3.3.1 (Security Condition)**.** *Let P be a program modelling a system, and let $\langle \mathcal{I}, \sqsubseteq \rangle$ be a lattice of information, and let $\mathcal{F}lows$ be the set of all information flows with respect to the lattice $\mathcal{I}$. Furthermore, let $\mathscr{P} \subseteq \mathcal{F}lows$ be an information flow policy; and let $[\![P]\!]^{\mathcal{I}} \subseteq \mathcal{F}lows$ be a subset of $\mathcal{F}lows$, called the* information flow property *of the system modelled by P. The system modelled by P* satisfies*, and is said to be* secure *with respect to the policy $\mathscr{P}$ iff for all $f \in [\![P]\!]^{\mathcal{I}}$ there exists $f' \in \mathscr{P}$ such that $f \sqsubseteq f'$.*

Intuitively, this definition says that the program P, or the system it models, is secure (with respect to the policy $\mathscr{P}$) iff every flow $f \in [\![P]\!]^{\mathcal{I}}$ that is caused by P is permitted by the policy ($\exists f' \in \mathscr{P}$ such that $f \sqsubseteq f'$). The partial order $\sqsubseteq$ regulates the level of information that we wish to release. This extensional view of policy enforcement abstractly describes, in terms of the information lattice order, what information flows are permitted in the system.

In the remainder of this chapter we shall show how to derive the information released by a system from an input-output relational model of the system, providing us with a way to check whether the system has secure information flow. Later on, in Chapter 4, we shall show how to derive this input-output relational model from the operational semantics in a language-based setting, and show, for a given program P, how to define $[\![P]\!]^{\mathcal{I}}$ under various representations of the lattice $\mathcal{I}$ of information.

3.4 System Models and Information Representation

In the following sections we shall formalise the information released by a system by using an input-output *relational model*, which describes how the system transforms its inputs to publicly observable outputs. The relational model captures the input-output semantics of the system through a *relation* which associates the public outputs that the attacker may observe with the inputs which generate them. We shall derive, from the relational model, various representation of information, which are shown to fit into the lattice model of information. The relational model technique is applicable to the analysis of information flow under *deterministic* system models as well as the more general *nondeterministic* system models.

Definition 3.4.1 (Relational System Model)**.** *The input-output* relational model *of a system is defined as a relation $S \subseteq \mathbf{\Sigma} \times \mathcal{V}$, over the set $\mathbf{\Sigma}$ of the system's inputs and the set $\mathcal{V}$ of observable outputs of the system according to an attacker model, where for all input $\sigma \in \mathbf{\Sigma}$ and possible output $v \in \mathcal{V}$, $\sigma\ S\ v$ holds iff the system can produce the output v when supplied with the input σ. The system model is said to be* deterministic *if S is a function from $\mathbf{\Sigma}$ to $\mathcal{V}$, otherwise it is said to be* nondeterministic.

Using the relational model primitive defined above, we shall develop information representation suitable for the analysis of information release in deterministic and, or nondeterministic systems. We assume that the model S of the system is both input-total and output-total, that is, $\mathbf{\Sigma} = \{\sigma \mid v \in \mathcal{V}, \sigma\ S\ v\}$ and $\mathcal{V} = \{v \mid \sigma \in \mathbf{\Sigma}, \sigma\ S\ v\}$.

3.5 Information Flow in Deterministic Systems

Under the relational model, *deterministic* systems are modelled by (total) *functions* of the form $f : \Sigma \to \mathcal{V}$ from an *input space* Σ (representing the set of all possible inputs to the system) to an *output space* $\mathcal{V}$ (representing the set of all publicly observable outputs that the system can generate), such that for any input $\sigma \in \Sigma$ supplied to the system in question, $f(\sigma) \in \mathcal{V}$ is what the attacker publicly observes. The system is deterministic, with respect to the attacker's view, because f is a function and thus the output observed by the attacker is *unique* for every input supplied to the system.

Suppose Σ is the set of all secret values that can be supplied to a system modelled by $f : \Sigma \to \mathcal{V}$, then the system is said to be noninterfering if for all $\sigma, \sigma' \in \Sigma$, $f(\sigma) = f(\sigma')$. That is, the public output of this system that the attacker sees is fixed regardless of the chosen secret input to the system, as required by the noninterference definition of [GM82]. It is thus clear when we say that another system modelled by the function $g : \Sigma \to \mathcal{V}'$ releases more information than the one modelled by f if there exists at least an input pair $\sigma, \sigma' \in \Sigma$ such that $g(\sigma) \neq g(\sigma')$. In other words, there are some runs of the system modelled by g which can be distinguished by observing the output, whereas no run of the system modelled by f can be distinguished based on the observed output. How do we then represent more generally that a deterministic system releases *more information* than another one?

3.5.1 An Equivalence Relation Representation of Information

In the example above, the reason why g releases more information than f can be explained by the relative granularity of the equivalence classes of the kernels of the two functions. The *kernel* of any function $f : \Sigma \to \mathcal{V}$, is an *equivalence relation* (κ_f) over Σ which relates a pair of elements in Σ iff they have same image under f. Since any pair of input values $\sigma, \sigma' \in \Sigma$ that are related by the kernel κ_f produce the same output under f, then we say that the inputs σ and σ' are *indistinguishable* under the system modelled by f because the attacker cannot tell which of the two was supplied to the system based on observed output. Using this idea, we can describe the information released by a deterministic system that is modelled by the function $f : \Sigma \to \mathcal{V}$ via its kernel:

$$\forall \sigma, \sigma' \in \Sigma, \sigma \; \kappa_f \; \sigma' \iff f(\sigma) = f(\sigma'). \tag{3.1}$$

The finer the partition of Σ under κ_f, the more the information that is revealed by the system that f models. In the following, we will sometimes simply refer to the deterministic system modelled by a function f as "system f".

We say that a system $g : \Sigma \to \mathcal{V}'$ releases more information than another system $f : \Sigma \to \mathcal{V}$ iff $\kappa_g \subseteq \kappa_f$, where κ_g and κ_f are respectively the kernels of g and f. Using the definition of function kernels, this property can be equivalently stated as follows.

A deterministic system $g : \Sigma \to \mathcal{V}'$ releases more information than another deterministic system $f : \Sigma \to \mathcal{V}$ iff for all $\sigma, \sigma' \in \Sigma$

$$g(\sigma) = g(\sigma') \implies f(\sigma) = f(\sigma'). \tag{3.2}$$

This definition simply says that if g cannot distinguish a pair of inputs, neither can f. Notice the fact that this definition does not rely on the sets $\mathcal{V}$ and $\mathcal{V}'$ because intuitively we care only about the ability of a system to distinguish its inputs, that is, how it partitions its domain. It is easy to see that if the systems f and g both release the same amount of information, then they are equal up to an isomorphism of their output representations. Thus, the systems modelled by $f : \Sigma \to \mathcal{V}$ and $g : \Sigma \to \mathcal{V}'$ release the same information if there exists a set isomorphism ι from the range of f to that of g such that $\iota \circ f = g$. The information released jointly by two systems f and g processing independently the same inputs can be modelled by another system $(f, g) : \Sigma \to \mathcal{V} \times \mathcal{V}'$ given by $(f, g)(\sigma) = (f(\sigma), g(\sigma))$ and whose kernel is the equivalence relation $\kappa_f \cap \kappa_g$ [LR93].

3.5.2 Lattice of Equivalence Relations

The authors of [LR93] first proposed an equivalence relation model as a way to describe the security properties of systems. Under the equivalence relation representation of information, two elements in the domain of an equivalence relation R are said to be *indistinguishable* if they are related by R. Alternatively, we say that a pair of elements in the domain of R are *distinguishable* when they are *not related* by R. This leads to a lattice of information, represented by equivalence relations, based on the ability to distinguish elements of a set.

Definition 3.5.1 (Lattice of Equivalence Relations [LR93])**.** *Let Σ be a set, and let $ER(\Sigma)$ be the set of all equivalence relations over Σ. Define an information order relation over $R, R' \in ER(\Sigma)$ such that $R \sqsubseteq R'$ iff for all $\sigma, \sigma' \in \Sigma, \sigma\ R'\ \sigma' \implies \sigma\ R\ \sigma'$.*

The combination *of the information modelled by the equivalence relations R and R' is given by the join $R \sqcup R'$ of the two relations, which is defined such that for all $\sigma, \sigma' \in \Sigma, \sigma\ (R \sqcup R')\ \sigma'$ iff $\sigma\ R\ \sigma'$ and $\sigma\ R'\ \sigma'$. The join operation naturally extends to subsets $\mathcal{R} \subseteq ER(\Sigma)$ such that for all $\sigma, \sigma' \in \Sigma, \sigma \bigsqcup \mathcal{R}\ \sigma'$ iff for all $R \in \mathcal{R}, \sigma\ R\ \sigma'$.*

It should be noted that the order relation $\sqsubseteq$ on equivalence relations is the *reverse subset inclusion* ($\supseteq$) order on relations, and is thus the dual of the traditional ordering of relations that is based on subset inclusion of their graphs. Consequently, the join operation $\sqcup$ on equivalence relations corresponds to set intersections $\cap$.

As demonstrated above, the lattice of equivalence relation models information release in deterministic systems. Furthermore, the ordering of equivalence relations by their information content forms a complete lattice of information.

Proposition 3.5.2. *The partially ordered set $\langle ER(\Sigma), \sqsubseteq, \sqcup \rangle$ is a complete lattice.*

Proof. Standard. □

Under the equivalence relation representation, the greatest information is the identity relation ($id \in \mathrm{ER}(\Sigma)$) on the set Σ since by definition it distinguishes any pair of elements in Σ that are not the same, relating an element to itself only. The identity equivalence relation represents complete knowledge. An example of a system which releases this kind of information is one that simply reveals its input, such as the system $g : \Sigma \to \Sigma$ defined as $\forall \sigma \in \Sigma, g(\sigma) = \sigma$. At the other extreme, the least element of the lattice of equivalence relation is the "*for all*" relation $all \in \mathrm{ER}(\Sigma)$, defined as $\forall \sigma, \sigma' \in \Sigma, \sigma\ all\ \sigma'$. This relation represents no information since it relates all elements of the set and thus cannot distinguish any

of them. An example of a system with such an information flow property, that is, which releases no information is the constant function whose kernel is *all*. We shall be referring to the equivalence relations *id* and *all* defined over some set (which will hopefully be clear from the context) throughout the thesis.

We can extend this basic idea to partial equivalence relations (PERs) on the set of system inputs. PERs are particularly useful in the analysis of composite systems, providing us with the additional ability to specify the knowledge that a secret does not belong to a given set. This cannot be stated naturally with equivalence relations since they are, by definition, reflexive. The simple generalisation to PERs gives us some expressive powers, which we briefly illustrate.

Suppose $f : \mathbf{\Sigma} \to \mathcal{V}$ and $g : \mathbf{\Sigma} \to \mathcal{V}'$ are functions modelling two deterministic systems, where $\mathcal{V}$ and $\mathcal{V}'$ are disjoint and $\Sigma \subseteq \mathbf{\Sigma}$. Let us define another system model which makes a choice between f and g (depending on whether the input belongs to the set Σ or not), which is given by $\between_{\Sigma}(f, g) : \mathbf{\Sigma} \to \mathcal{V} \cup \mathcal{V}'$ and defined such that for any $\sigma \in \mathbf{\Sigma}$,

$$\between_{\Sigma}(f, g)(\sigma) \triangleq \begin{cases} g(\sigma) & \text{if } \sigma \in \Sigma \\ f(\sigma) & \text{otherwise.} \end{cases}$$

It is easy to see that the information released by $\between_{\Sigma}(f, g)$ is in general not $\kappa_g \sqcup \kappa_f$ - the join of the kernels of g and f. This is because the choice restricts the domains of the two subsystems modelled by f and g. This example in fact demonstrates a kind of *conditional* information release, where g releases information only about inputs in Σ and f releases information about inputs in $\mathbf{\Sigma} \backslash \Sigma$ (a property which we shall use in the analysis of information flow in conditional statements in Chap-

ter 5). By dropping the reflexivity requirement, we can precisely describe the information flow of this system by two PERs $R \cap \kappa_g$ and $\overline{R} \cap \kappa_f$, where R and $\overline{R}$ are respectively the PERs $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R \, \sigma'$ iff $\sigma, \sigma' \in \Sigma$ and $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; \overline{R} \, \sigma'$ iff $\sigma, \sigma' \in \mathbf{\Sigma} \backslash \Sigma$. The PER R requires that any pair of inputs must belong to the set Σ, and $\overline{R}$ requires that the inputs must belong to the complement of Σ. Hence the PER $R \cap \kappa_g$ models the information that can distinguish all inputs that g can distinguish subject to the constraint that the input belongs to Σ (that is, an output from g tells us that the input is *not* in $\mathbf{\Sigma} \backslash \Sigma$). The PER $\overline{R} \cap \kappa_f$ has similar information interpretation. Note that the overall information flow of the system $\curlywedge_\Sigma(f, g)$ is an equivalence relation given by the union of the two disjoint PERs $(R \cap \kappa_g) \cup (\overline{R} \cap \kappa_f)$. It should be noted that if $\mathcal{V}$ and $\mathcal{V}'$ are not disjoint, such that for some $\sigma \in \Sigma$ and $\sigma' \in \mathbf{\Sigma} \backslash \Sigma$ we have $g(\sigma) = f(\sigma')$, then the equivalence relation $(R \cap \kappa_g) \cup (\overline{R} \cap \kappa_f)$ will be greater than the information actually released by $\curlywedge_\Sigma(f, g)$.

3.5.3 A PER Representation of Information

Partial equivalence relations generalise equivalence relations by dropping the reflexivity requirement. This leads to a more general representation of information based on partial equivalence relations on a set,whereby we can also express when a secret does not belong to a set - the set of elements not in the domain of the PER.

Definition 3.5.3 (Set of PERs)**.** *Let $\mathbf{\Sigma}$ be a set. Define $\mathit{PER}(\mathbf{\Sigma})$ to be the set of all partial equivalence relations over the set $\mathbf{\Sigma}$. The domain of definition of a PER R on $\mathbf{\Sigma}$ is given by dom$(R) \triangleq \{\sigma \in \mathbf{\Sigma} \mid \sigma \; R \, \sigma\}$.*

A PER R is reflexive on its domain of definition since for any $\sigma, \sigma' \in \mathbf{\Sigma}$ such

that $\sigma \ R \ \sigma'$ holds, then $\sigma' \ R \ \sigma$ holds by symmetry and, thus $\sigma \ R \ \sigma$ holds by transitivity. Similarly to equivalence relations, we say that a PER R over $\mathbf{\Sigma}$ models information (or more precisely, ignorance) by *indistinguishability* of elements in $\mathbf{\Sigma}$. Thus, if $R \in \text{PER}(\mathbf{\Sigma})$ describes the information or the knowledge of an attacker, then that attacker cannot distinguish two elements $\sigma, \sigma' \in \mathbf{\Sigma}$ if $\sigma \ R \ \sigma'$ holds (this may be read as, σ is indistinguishable from σ' via information R). The information modelled by R describes what elements of the set *dom*(R) are indistinguishable by an attacker. All elements in the set $\mathbf{\Sigma}\backslash$*dom*$(R)$ are considered not possible in the world described by the information R.

Using PERs to describe information

Let us further illustrate the use of PERs for information representation. Consider three PERs on the set $\mathbb{Z}$ of integers, representing different levels of information about an integer secret as follows. The first one is the equivalence relation ***Par*** defined as: $\forall n, m \in \mathbb{Z}, n \ \textbf{\textit{Par}} \ m \iff n \mod 2 = m \mod 2$. This describes the knowledge of parity because it can only distinguish two integer values when they have different parities. The second one is the equivalence relation *id*, which is defined over integers and relates an integer to itself only. This models the ability to distinguish between any two different integers and therefore contains more information than ***Par***. The third one is the PER $id^{\mathbb{N}}$ which is defined as $\forall m, n \in \mathbb{Z}, m \ id^{\mathbb{N}} \ n \iff n = m, n \in \mathbb{N}$. This PER models the fact that the integer values must be natural numbers (the knowledge that the integer secret cannot have a negative value), in addition to the ability to distinguish between any two such integers. Thus, $id^{\mathbb{N}}$ contains more information than *id* because it limits the set of possibilities to natural numbers. It is clear, in a computational sense, that

an attacker which knows beforehand that a certain subset of the domain of secret values is not possible needs to do less work in searching for that secret than one that does not know beforehand.

3.5.4 Lattice of PERs

The interpretation of PERs as a representation of information content suggests an information order of PERs. The intuition is that the information content of a PER R' is greater than that of another PER R if R' distinguishes at least all that R can distinguish and the domain of R' is contained in the domain of R.

Definition 3.5.4 (Lattice of PERs)**.** *Let Σ be a set. Define the order relation $\sqsubseteq$ on partial equivalence relations over Σ such that for any $R, R' \in \mathit{PER}(\Sigma)$, $R \sqsubseteq R'$ iff for all $\sigma, \sigma' \in \Sigma$, $\sigma\ R'\ \sigma' \implies \sigma\ R\ \sigma'$. The associated join operation $\sqcup$ on $\mathit{PER}(\Sigma)$ is defined as $\sigma\ (R \sqcup R')\ \sigma'$ iff $\sigma\ R\ \sigma'$ and $\sigma\ R'\ \sigma'$. More generally, for any subset $\mathcal{R} \subseteq \mathit{PER}(\Sigma)$ define the join of $\mathcal{R}$ as the PER $\bigsqcup\mathcal{R}$, such that for all $\sigma, \sigma' \in \Sigma$, $\sigma \bigsqcup\mathcal{R}\ \sigma'$ iff $\forall R \in \mathcal{R}$, $\sigma\ R\ \sigma'$.*

Since PERs are reflexive on their domains, $R \sqsubseteq R'$ implies $\mathit{dom}(R') \subseteq \mathit{dom}(R)$. Note that, similarly to equivalence relations, the partial order $\sqsubseteq$ on PERs is the reverse subset inclusion order on relations and that $\sqcup$ corresponds to set intersections on the graph of relations. The ordering of PERs by their information content forms a complete lattice of information.

Proposition 3.5.5. *The partially ordered set $\langle \mathit{PER}(\Sigma), \sqsubseteq, \sqcup \rangle$ is a complete lattice.*

Proof. The proof is similar to the proof of the completeness of the lattice of equivalence relations. □

Partitions of a PER

We refer to the *partition* of the set $\mathit{dom}(R)$ by the PER $R \in \mathrm{PER}(\mathbf{\Sigma})$, which describes the information about the elements of $\mathbf{\Sigma}$ as modelled by R. This partitioning is defined as the family of sets, which we denote by

$$[\mathbf{\Sigma}]_R \triangleq \{\{\sigma' \in \mathbf{\Sigma} \mid \sigma \; R \; \sigma'\} \mid \sigma \in \mathit{dom}(R)\}. \tag{3.3}$$

If R is an equivalence relation, then $[\mathbf{\Sigma}]_R$ is the set of *equivalence classes* of R. Similarly to the standard notation for equivalence classes, we write $[\sigma]_R$ for the *equivalence class* of the PER R that $\sigma \in \mathit{dom}(R)$ belongs to. This is defined as

$$[\sigma]_R \triangleq \{\sigma' \in \mathbf{\Sigma} \mid \sigma \; R \; \sigma'\}. \tag{3.4}$$

Furthermore, like the membership property of equivalence classes in an equivalence relation, if two elements of $\mathbf{\Sigma}$ are related by a PER R then they belong to the same equivalence class of R.

Proposition 3.5.6. *Let R be a PER over a set $\mathbf{\Sigma}$, then for any $\sigma, \sigma' \in \mathbf{\Sigma}$, $\sigma \; R \; \sigma' \implies [\sigma]_R = [\sigma']_R$.*

Proof. Straightforward. □

In terms of information described by PERs, this means that if a pair of values are indistinguishable via the knowledge described by a PER, then those values belong to the same equivalence class of the PER.

3.5.5 PERs and Disjunctive Information

We shall show in this section that we can represent certain disjunctive information with PERs, contrary to a conjecture in [SS05] that disjunctive properties may not be expressed by PERs. Disjunctive information modelling can be useful in applications, where we want to express the fact that at most one of two pieces of information is released in a system during a run of the system. For example, we might want to express the fact that a symmetric encryption module which accepts a parameter to release either the key or ciphertext releases only the key or the ciphertext to the recipient (depending on the choice of the release parameter), but not both at the same time. Firstly, we define the property of a PER when it reveals at most one of two pieces of information.

Definition 3.5.7 (Disjunctive Information)**.** *Let V be a set and let $R_1, R_2 \in \mathit{PER}(V)$ be PERs over V representing some information, and let $R = R_1 \sqcup R_2$ be a PER, which represents a combination of the information modelled by R_1 and R_2. We say that the PER $R \in \mathit{PER}(V)$ contains the* disjunctive information *R_1 and R_2 iff any pair of elements in V that is not related by R_1 is related by R_2, and any pair of elements in V that is not related by R_2 is related by R_1.*

This definition requires that whenever R_1 has knowledge about a pair of values (that is, can distinguish the pair because it is not related by R_1) then R_2 does not have the knowledge, and vice versa. Thus, $R = R_1 \sqcup R_2$ has the knowledge about a pair of values if that knowledge comes from at most one of R_1 or R_2. More formally, this means that for any $(v, v') \in V^2$ such that $(v, v') \notin R$, then either $(v, v') \notin R_1$ and $(v, v') \in R_2$, or $(v, v') \in R_1$ and $(v, v') \notin R_2$. Since the information in R_1 and R_2 are mutually exclusive then R contains the disjunctive

information R_1 and R_2 - revealing information that comes from at most one of the two PERs about any pair of values in V. This is illustrated as follows.

For any PER $R \in \text{PER}(V)$ over V, let the *ignorance set* of R be given by its graph $\mathit{graph}(R) = \{(v, v') \in V^2 \mid v \, R \, v'\}$. On one hand, the set $\mathit{graph}(R)$ is called the "ignorance set" of R because it is the set of pairs in V^2 that R cannot distinguish. On the other hand, let $\overline{\mathit{graph}(R)} = V^2 \backslash \mathit{graph}(R)$ be the *knowledge set* of R - representing the set of pairs in V^2 that R can distinguish. Clearly, for any PER, the knowledge and the ignorance sets are disjoint. It is also easy to see that for any $R, R_1, R_2 \in \text{PER}(V)$ such that $R = R_1 \sqcup R_2$, we have that $\mathit{graph}(R) = \mathit{graph}(R_1) \cap \mathit{graph}(R_2)$, and that $\overline{\mathit{graph}(R)} = \overline{\mathit{graph}(R_1)} \cup \overline{\mathit{graph}(R_2)}$. Now let $A = \overline{\mathit{graph}(R_1)}$ and $B = \overline{\mathit{graph}(R_2)}$, and assume that R contains *disjunctive information* R_1 and R_2 according to Definition 3.5.7, then $\overline{\mathit{graph}(R)} = (A \cap B) \cup (A \backslash B) \cup (B \backslash A) = (A \backslash B) \cup (B \backslash A)$. This is because by the disjunctive information requirement $(v, v') \in A \implies v \, R_2 \, v'$, and by the partitioning property of the knowledge and ignorance sets of a PER, $v \, R_2 \, v' \implies (v, v') \notin B$. Similarly, for B, $(v, v') \in B \implies v \, R_1 \, v' \implies (v, v') \notin A$. Thus, A and B are disjoint sets. Therefore, whenever R can distinguish a pair of values (that is, $(v, v') \in \overline{\mathit{graph}(R)} = (A \backslash B) \cup (B \backslash A)$), that pair is distinguishable by at most one of R_1 or R_2.

To show an example, suppose X and Y and Z are sets, which are mutually disjoint, and such that $V = X \cup Y \cup Z$. Now define PERs $R_1, R_2, R \in \text{PER}(V)$ such that for all $v, v' \in V$, $v \, R_1 \, v' \iff (v, v') \in (X \cup Y)^2 \cup (X \cup Z)^2$, and $v \, R_2 \, v' \iff (v, v') \in (Y \cup Z)^2$, and $R = R_1 \sqcup R_2$. It is easy to see that R_1 can distinguish elements of Y from those of Z, since it relates no such pairs. However, R_2 can distinguish any pair of elements in X, and elements of X from

those of Z, and also elements of X from those of Y. Therefore, R, which relates a pair of elements in V if and only if the pair belongs to Y^2 or Z^2 can distinguish elements of Y from elements of Z (information that comes from R_1 precisely); and, disjunctively, can also distinguish any pair or elements in X, and elements of X from those of Z, and elements of X from those of Y (information that comes precisely from R_2).

The idea of disjunctive information can be extended to PERs on maps (or tuples), where we want to express the idea that a PER reveals information about at most one of two elements in the domain of the function (or at most about one of two indices, when we consider tuples). Assume that $\mathbf{Var}$ is a set of variables, and for each variable $x \in \mathbf{Var}$, let V_x be the set of all the possible values of x. Now let $\mathbf{\Sigma} = [\mathbf{Var} \to \bigcup_{x \in \mathbf{Var}} V_x]$ be the set of all functions from variables to values, such that for any $\sigma \in \mathbf{\Sigma}$ and $x \in \mathbf{Var}, \sigma(x) \in V_x$ is the x-image of the function σ. We shall refer to $\sigma \in \mathbf{\Sigma}$ as a *state*.

Definition 3.5.8. *Let $Z \subseteq \mathbf{Var}$ be a set of variables and let $\Sigma_0 \subseteq \mathbf{\Sigma}$. Define the operation* havoc *with the signature havoc : $\mathcal{P}(\mathbf{Var}) \times \mathcal{P}(\mathbf{\Sigma}) \to \mathcal{P}(\mathbf{\Sigma})$ as*

$$\mathit{havoc}Z(\Sigma_0) \triangleq \{\sigma' \in \mathbf{\Sigma} \mid \sigma \in \Sigma_0, \forall y \in \mathbf{Var} \backslash Z, \sigma(y) = \sigma'(y)\}.$$

Suppose $x \in \mathbf{Var}$ and let $X = \{x\}$, we say that the set $\Sigma \subseteq \mathbf{\Sigma}$ is *dense* with respect to the values of x if $\mathit{havoc}X(\Sigma) = \Sigma$. That is, for any state $\sigma \in \Sigma$, all the possible values of x are already present in the set Σ since $\mathit{havoc}X(\{\sigma\}) \subseteq \Sigma$. We can now define when a PER over $\mathbf{\Sigma}$ contains disjunctive information about two variables in $\mathbf{Var}$.

Definition 3.5.9. *Let $R \in \mathit{PER}(\mathbf{\Sigma})$ be a PER over $\mathbf{\Sigma}$ and let $x, y \in \mathbf{Var}$ be variables*

such that $X = \{x\}$ *and* $Y = \{y\}$. *The PER* R *contains disjunctive information about* x *and* y *iff for all* $\sigma \in dom(R)$

$$havocX([\sigma]_R) \neq [\sigma]_R \implies havocY([\sigma]_R) = [\sigma]_R$$

and

$$havocY([\sigma]_R) \neq [\sigma]_R \implies havocX([\sigma]_R) = [\sigma]_R.$$

This definition requires that any equivalence class of R that is not dense with respect to the values of the variable x (that is, R has some information about x in that equivalence class) must be dense with respect to y. Similarly, if an equivalence class of R contains any information with respect to y, it must not contain any information about x. This definition is a specialisation of Definition 3.5.7 by considering each equivalence class of R as a join two PERs on values, each of which may contain information about the values of x or y, but not both at the same time.

To illustrate this, suppose $\mathbf{Var} = \{h_1, h_2, l\}$ such that h_1 and h_2 are two integer secrets and l is a boolean public variable. Let $R \in \text{PER}(\mathbf{\Sigma})$ be a PER over $\mathbf{\Sigma}$ such that R reveals the parity of h_1 whenever l is chosen to be $\mathbf{tt}$ but reveals the value of h_2 whenever l is chosen to be $\mathbf{ff}$. This is defined as $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R \; \sigma'$ iff $\sigma(h_1) \; mod \; 2 = \sigma'(h_1) \; mod \; 2, \sigma(l) = \sigma'(l) = \mathbf{tt}$ or $\sigma(h_2) = \sigma'(h_2), \sigma(l) = \sigma'(l) = \mathbf{ff}$. The PER R reveals disjunctive information about h_1 (its parity) or h_2 (its value) for any pair of states $\sigma, \sigma' \in \mathbf{\Sigma}$. Take, for example, the equivalence class of $\sigma \in dom(R)$ where $\sigma(l) = \mathbf{ff}$, any variation in the value of h_2 alone is distinguishable by R, whereas, variations in the value of h_1 alone are indistinguishable by R in that equivalence class. Similarly, for any $\sigma \in dom(R)$ where $\sigma(l) = \mathbf{tt}$, h_1

either has odd or even parity in the equivalence class $[\sigma]_R$, and varying only the parity of h_1 is distinguishable by R in this equivalence class, whereas, R does not distinguish any variation in the value of h_2 alone.

It is worth noting that for PERs over states $\mathbf{\Sigma}$ that are not necessarily disjoint, but which contain disjunctive information according to Definition 3.5.9, we can create another PER which preserves the disjunctive information by taking disjoint unions. To illustrate, assume that the PERs $R_1, R_2 \in \mathrm{PER}(\mathbf{\Sigma})$ both contain disjunctive information about variables $x, y \in \mathbf{Var}$. Then we can define another PER R preserving the disjunctive information as follows. Let $z \notin \mathbf{Var}$ be a variable which has two possible values (it does not matter what the values are), for example, let $V_z = \{0, 1\}$, and let $\mathbf{\Sigma}_z \triangleq [(\mathbf{Var} \cup \{z\}) \to \bigcup_{x \in \mathbf{Var} \cup \{z\}} V_x]$ be a domain extension (of maps in $\mathbf{\Sigma}$ by z). Define the PER $R \in \mathrm{PER}(\mathbf{\Sigma}_z)$ over $\mathbf{\Sigma}_z$ as $\forall \sigma, \sigma' \in \mathbf{\Sigma}_z, \sigma \, R \, \sigma'$ iff $\sigma \, R_1 \, \sigma', \sigma(z) = \sigma'(z) = 0$ or $\sigma \, R_2 \, \sigma', \sigma(z) = \sigma'(z) = 1$. The PER R contains disjunctive information about x and y.

3.6 Information Flow in Nondeterministic Systems

In the following sections we shall consider representations of information for nondeterministic system models. The nondeterministic system model generalises the deterministic one because the public output that is observed is not necessarily unique for each input to the system. Firstly, we propose a qualitative representation of information for nondeterministic system models, which is based on families of sets that generalises the PER representation of information presented earlier. Secondly, we then present a quantitative representation, which uses probability measures and information theory to describe the attacker's knowledge (or

more precisely, uncertainty) about the inputs.

3.7 A Qualitative Representation

We propose a qualitative representation of information, based on families of sets, to model *possibilistic* information flow to the attacker. We say the model is possibilistic because, given the output observation of the attacker, it reveals whether certain inputs are *possible*, as opposed to how likely it is for the inputs to generate the public observation. However, the quantitative information representation presented in section 3.8 additionally accounts for the likelihood, using probabilities, of an input to generate a given output.

3.7.1 Possibilistic Information Representation

Let us start by motivating the use of families of sets as a representation of information under a nondeterministic system model. Consider a system, whose relational model is given by $S \subseteq \mathbf{\Sigma} \times \mathcal{V}$. We can describe the information that the attacker gains on observing the output $v \in \mathcal{V}$ of the system by the inverse image of v under S. The inverse image $S^{-1}(v) = \{\sigma \in \mathbf{\Sigma} \mid \sigma\ S\ v\}$ of v represents the set of all *possible* inputs that can produce the output v in the system modelled by S, and thus describes the attacker's uncertainty about the inputs given the observation of v. It is thus easy to see that the family of sets $\{S^{-1}(v) \mid v \in \mathcal{V}\}$ models the uncertainties of the attacker under the observation of individual outputs of the system modelled by S. In the special case that S models a deterministic system, in which case S is a function, it is clear that $\{S^{-1}(v) \mid v \in \mathcal{V}\}$ corresponds to the set of equivalence classes of the kernel of the function S, which uniquely identifies the equivalence

relation over Σ used to describe the information released in the previous sections. In this sense, the family of sets representation generalises the PER representation.

However, unlike the deterministic model, where for any $v, v' \in \mathcal{V}$, $v \neq v'$ implies $S^{-1}(v) \cap S^{-1}(v') = \varnothing$, the inverse images are not necessarily disjoint under a nondeterministic model since the outputs resulting from any given input may not necessarily be unique. This leads to another avenue of information release in nondeterministic systems. The property that the nondeterministic system modelled by S does not necessarily partition its domain introduces the possibility that an attacker might gain further information by repeated execution of the system under a fixed input. To illustrate this, suppose $S \subseteq \Sigma \times \mathcal{V}$ models a nondeterministic system, where $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ and $\mathcal{V} = \{v_1, v_2\}$ and where the graph of the relation S is given by $\mathit{graph}(S) = \{(\sigma_1, v_1), (\sigma_2, v_1), (\sigma_2, v_2), (\sigma_3, v_2)\}$. The model is nondeterministic since the input σ_2 can produce outputs v_1 or v_2. By observing an output v_1 the attacker learns that the input must be one of σ_1 and σ_2, as suggested by $S^{-1}(v_1) = \{\sigma_1, \sigma_2\}$. Similarly, on observing the output v_2, the attacker learns that the input is in the set $S^{-1}(v_2) = \{\sigma_2, \sigma_3\}$. However, if under a fixed input the attacker observes outputs v_1 and v_2 in different runs of the system, then the attacker confirms that the input to the system must be σ_2 - derived by taking the intersection $S^{-1}(v_1) \cap S^{-1}(v_2)$. This avenue of information leakage is not available under the deterministic system model since for a fixed input, the output of the system always remains the same. This leads us to a definition of information based on families of sets.

3.7.2 Lattice of Possibilistic Information

In order to account for the possible refinement of knowledge by repeatedly running a nondeterministic system under fixed input, the families of sets, which represent the information that the attacker derives by observing the outputs, must be closed under set intersection.

Definition 3.7.1 (Lattice of possibilistic information)**.** *Let* $\Sigma_J = \{\Sigma_j \subseteq \mathbf{\Sigma} \mid j \in J\}$ *be a family of subsets of* $\mathbf{\Sigma}$ *indexed by some set* J*. Define the operation* $\langle\langle\cdot\rangle\rangle$ *on families of subsets of* $\mathbf{\Sigma}$ *as* $\langle\langle\Sigma_J\rangle\rangle \triangleq \bigcup_{K\subseteq J}\{\bigcap \Sigma_K\}$*, which closes the family under intersections. Define the possibilistic information set over* $\mathbf{\Sigma}$ *as* $\mathit{FAM}(\mathbf{\Sigma}) \triangleq \{\langle\langle\Sigma_J\rangle\rangle | \Sigma_J$ *is a family of subsets of* $\mathbf{\Sigma}\}$ *to represent information contained in families of subsets of* $\mathbf{\Sigma}$*. For any* $\Sigma_J, \Sigma_K \in \mathit{FAM}(\mathbf{\Sigma})$ *define the join operation as* $\Sigma_J \sqcup \Sigma_K \triangleq \langle\langle\Sigma_J \cup \Sigma_K\rangle\rangle$ *and define the partial order* $\sqsubseteq$ *to be the subset ordering of families in* $\mathit{FAM}(\mathbf{\Sigma})$*.*

The intuition behind the partial ordering $\Sigma_J \sqsubseteq \Sigma_K$, for some $\Sigma_J, \Sigma_K \in \mathit{FAM}(\mathbf{\Sigma})$, is that every information token $X \in \Sigma_J$ is also present in Σ_K. Thus, from the relational model $S \subseteq \mathbf{\Sigma} \times \mathcal{V}$, we can derive the information that an attacker gains from the induced family of sets $\Sigma_{\mathcal{V}} = \{S^{-1}(v) \mid v \in \mathcal{V}\}$, describing the attacker's uncertainty under various observations of the outputs of the system modelled by S. The information that the attacker can gain is then described by the family $\langle\langle\Sigma_{\mathcal{V}}\rangle\rangle$, whose minimal elements identify minimal subsets of the inputs in $\mathbf{\Sigma}$ that can produce any given output under repeated execution of the system with fixed inputs.

We note that for any $V \subseteq \mathcal{V}$, the set $\bigcap \Sigma_V$ can be empty if there is no common input for which all outputs in V can be produced. Note also that by definition

$\Sigma_{\mathcal{V}} \subseteq \langle\langle \Sigma_{\mathcal{V}} \rangle\rangle$ due to the singleton subsets of $\mathcal{V}$, since for any $v \in \mathcal{V}$, $S^{-1}(v) = \bigcap S^{-1}(v) \in \langle\langle \Sigma_{\mathcal{V}} \rangle\rangle$. Furthermore, under the powerset lattice of $\boldsymbol{\Sigma}$ with the usual subset ordering, which is a complete lattice, the intersection $\bigcap F$ of any family F of subsets of $\boldsymbol{\Sigma}$ exists uniquely. In particular, for the empty family, we have $\bigcap \varnothing = \boldsymbol{\Sigma}$ and hence, $\langle\langle \varnothing \rangle\rangle = \{\boldsymbol{\Sigma}\}$. This has intuitive meaning because $\boldsymbol{\Sigma}$, which is the set of *all inputs*, rules out no possibility and therefore represents lack of information. Thus, $\langle\langle \varnothing \rangle\rangle$, which represents the information released by a system which produces no output agrees with the intuition that it cannot cause information flow.

The ordering of possibilistic information over inputs, $\langle \mathit{FAM}(\boldsymbol{\Sigma}), \sqsubseteq, \sqcup \rangle$, forms a complete lattice.

Theorem 3.7.2. *The ordered family of sets $\langle \mathit{FAM}(\boldsymbol{\Sigma}), \sqsubseteq, \sqcup \rangle$ over $\boldsymbol{\Sigma}$, representing the set of possibilistic information, is a complete lattice.*

Proof. Since the relation $\sqsubseteq$ over $\mathit{FAM}(\boldsymbol{\Sigma})$ is the subset inclusion order on sets, it is clear that $\langle \mathit{FAM}(\boldsymbol{\Sigma}), \sqsubseteq \rangle$ is a partially ordered set. In order to show that $\mathit{FAM}(\boldsymbol{\Sigma})$ is a complete lattice, it is sufficient to show that arbitrary joins exist [GHK+03].

We first show that $\sqcup$ is the relevant join operation over $\mathit{FAM}(\boldsymbol{\Sigma})$ with respect to the partial order $\sqsubseteq$. Specifically, we want to show that for any $\Sigma_J, \Sigma_K \in \mathit{FAM}(\boldsymbol{\Sigma})$, $\Sigma_J \sqsubseteq \Sigma_K$ iff $\Sigma_J \sqcup \Sigma_K = \Sigma_K$.

- Suppose $\Sigma_J \sqsubseteq \Sigma_K$, that is, $\Sigma_J \subseteq \Sigma_K$. Hence $\Sigma_J \cup \Sigma_K = \Sigma_K$, and since $\Sigma_K \in \mathit{FAM}(\boldsymbol{\Sigma})$ is already closed under intersections, $\Sigma_J \sqcup \Sigma_K = \langle\langle \Sigma_K \rangle\rangle = \Sigma_K$.

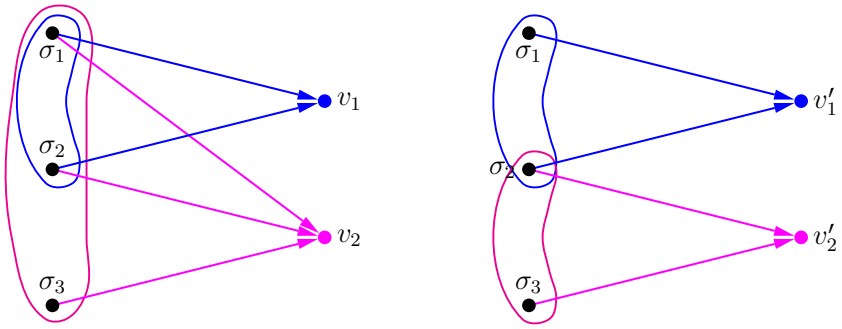
- Now assume that $\Sigma_J \sqcup \Sigma_K = \Sigma_K$. By the definition of the join operation on $\mathit{FAM}(\boldsymbol{\Sigma})$, we have $\Sigma_K = \langle\langle \Sigma_J \cup \Sigma_K \rangle\rangle = \langle\langle \Sigma_J \cup \Sigma_K \rangle\rangle \cup \langle\langle \Sigma_J \rangle\rangle$. Since

$\Sigma_J \in \mathit{FAM}(\mathbf{\Sigma})$ then $\Sigma_J = \langle\langle \Sigma_J \rangle\rangle$. Hence $\Sigma_J \sqcup \Sigma_K = \Sigma_K$ implies $\Sigma_K = \langle\langle \Sigma_J \cup \Sigma_K \rangle\rangle \cup \Sigma_J \implies \Sigma_J \subseteq \Sigma_K$. That is, $\Sigma_J \sqsubseteq \Sigma_K$.

This shows the necessary relationship between the join operation and the partial order over $\mathit{FAM}(\mathbf{\Sigma})$. It now remains to be shown that arbitrary joins exist in $\mathit{FAM}(\mathbf{\Sigma})$. Let $F = \{\Sigma_J \mid J \in \mathcal{J}\} \subseteq \mathit{FAM}(\mathbf{\Sigma})$ be an arbitrary subset of $\mathit{FAM}(\mathbf{\Sigma})$, where for any $J \in \mathcal{J}$, Σ_J is a family of subsets of $\mathbf{\Sigma}$. It is clear from the definition that $\bigsqcup F = \langle\langle \bigcup_{J \in \mathcal{J}} \Sigma_J \rangle\rangle \in \mathit{FAM}(\mathbf{\Sigma})$, since $\bigcup_{J \in \mathcal{J}} \Sigma_J$ is a family of subsets of $\mathbf{\Sigma}$. □

To illustrate how the lattice $\mathit{FAM}(\mathbf{\Sigma})$ describes the relative information released by two systems, consider two nondeterministic systems modelled by the relations $S \subseteq \mathbf{\Sigma} \times \mathcal{V}$ and $S' \subseteq \mathbf{\Sigma} \times \mathcal{V}'$, where $\mathbf{\Sigma} = \{\sigma_1, \sigma_2, \sigma_3\}$ and $\mathcal{V} = \{v_1, v_2\}$ and $\mathcal{V}' = \{v_1', v_2'\}$. Suppose the graphs of the relations S and S' are respectively given by $\mathit{graph}(S) = \{(\sigma_1, v_1), (\sigma_2, v_1), (\sigma_1, v_2), (\sigma_2, v_2), (\sigma_3, v_2)\}$ and $\mathit{graph}(S') = \{(\sigma_1, v_1'), (\sigma_2, v_1'), (\sigma_2, v_2'), (\sigma_3, v_2')\}$. The set of inverse images under S and S' are respectively given by $\Sigma_{\mathcal{V}} = \{\{\sigma_1, \sigma_2\}, \mathbf{\Sigma}\}$ and $\Sigma_{\mathcal{V}'} = \{\{\sigma_1, \sigma_2\}, \{\sigma_2, \sigma_3\}\}$. The situation is illustrated in Figure 3.1, where each squiggle contains the set of inputs which produce a given output and represents the inverse image of that output. Intuitively, the set $\Sigma_{\mathcal{V}}$ (of the inverse images under S) has more uncertainty (and thus less information) than the set $\Sigma_{\mathcal{V}'}$ (of the inverse images under S') since for each output $v \in \mathcal{V}$ of the system modelled by S there is a corresponding output $v' \in \mathcal{V}'$ of the other system for which $S'^{-1}(v') \subseteq S^{-1}(v)$. The greater information released by the system modelled by S' is confirmed by the fact that $\langle\langle \Sigma_{\mathcal{V}} \rangle\rangle = \{\{\sigma_1, \sigma_2\}, \mathbf{\Sigma}\} \sqsubseteq \langle\langle \Sigma_{\mathcal{V}'} \rangle\rangle = \{\{\sigma_1, \sigma_2\}, \{\sigma_2, \sigma_3\}, \{\sigma_2\}, \mathbf{\Sigma}\}$, which means that by fixing the input to the system S' the attacker can learn (in addition to what may be learnt under S) when the input to the system modelled by S' belongs only

to the set $\{\sigma_2, \sigma_3\}$ or $\{\sigma_2\}$. The knowledge $\{\sigma_2, \sigma_3\}$ is gained by observing v_2' in S' - which eliminates the possibility of σ_1 as the input, as opposed to the knowledge $\mathbf{\Sigma}$ on observing v_2 in S, which eliminates no possibility. Furthermore, by fixing the input it is possible to isolate the input σ_2 in the input space of S', which is the only input that can produce both v_1' and v_2'. These additional information cannot be derived under S.



Model of system S Model of system S'

Figure 3.1: *Information flow under two nondeterministic systems*

The qualitative representations of information (equivalence relations, PERs, and families of sets) presented above for the general nondeterministic system answer the question of whether a given input is possible when an output is observed. This however does not address the question of how likely, in particular, what the probability is for such an input to have been chosen. For systems which exhibit probabilistic nondeterminism, it may be possible to derive the probability that a certain input has been selected based on the observation of a given output. Thus, by observing the pattern of the outputs, an attacker may reduce his or her uncertainty about the inputs by deriving the probabilities for selecting inputs to the system based on the pattern of outputs. This view of information flow result-

ing from a change in the attacker's uncertainty about inputs to a system (which is modelled by probability distributions over the input space) lends itself to an information-theoretic analysis presented next.

3.8 A Quantitative Representation

Under the qualitative representations of information flow presented earlier, given the relational model $S \subseteq \mathbf{\Sigma} \times \mathcal{V}$ of a system, an attacker on observing an output $v \in \mathcal{V}$ thinks it *possible* that the input $\sigma \in \mathbf{\Sigma}$ may have been supplied to the system whenever $(\sigma, v) \in S$ - although it might be extremely unlikely that the input σ generates the output v. We consider *probabilistic systems*, which have probability distributions associated with the occurrence of their inputs and outputs and derive a quantitative measure, based on Shannon's information theory, which describes the level of uncertainty of the attacker induced by the system's probabilistic input-output dependency. For the quantitative probabilistic analysis that we consider in this thesis, we assume that both the set $\mathbf{\Sigma}$ and $\mathcal{V}$ are *finite*.

3.8.1 Probability Measures and Entropy

We start by presenting standard definitions from probability and information theory, and introduce some notations that we shall use in the analysis.

Definition 3.8.1 (σ-Algebra [Hal03])**.** *The set $\mathcal{F}$ of subsets of $\mathbf{\Sigma}$ is an* algebra *over $\mathbf{\Sigma}$ if it contains $\mathbf{\Sigma}$ and is closed under set union and complementing, so that if $\Sigma, \Sigma' \in \mathcal{F}$ then so are $\Sigma \cup \Sigma', \overline{\Sigma} = \mathbf{\Sigma} \backslash \Sigma \in \mathcal{F}$. A* σ-algebra *is closed under complementing and countable union, so that if $\Sigma_1, \Sigma_2, \ldots \in \mathcal{F}$ then $\bigcup_i \Sigma_i \in \mathcal{F}$.*

A probability space *over* $\mathbf{\Sigma}$ *is a triple* $\langle \mathbf{\Sigma}, \mathcal{F}, \mu \rangle$*, where* $\mathcal{F}$ *is an algebra over* $\mathbf{\Sigma}$*, and* $\mu : \mathcal{F} \to [0,1]$ *called a* probability measure *is a map to the closed real interval* $[0,1]$ *such that*

- $\mu(\mathbf{\Sigma}) = 1$
- $\mu(\Sigma \cup \Sigma') = \mu(\Sigma) + \mu(\Sigma')$ *for any disjoint* $\Sigma, \Sigma' \in \mathcal{F}$.

Any algebra is also closed under intersection since by De Morgan's duality we have that $\Sigma \cap \Sigma' = \overline{\overline{\Sigma} \cup \overline{\Sigma'}}$ for any pair $\Sigma, \Sigma' \in \mathcal{F}$. A set $\Sigma \in \mathcal{F}$ is called an *event*. Since the set $\mathbf{\Sigma}$ that we shall consider for the probabilistic information analysis is assumed to be finite, we have that $\mathcal{F}$ is always a σ-algebra. Furthermore, we shall always take $\mathcal{F}$ to be the powerset $\mathcal{P}(\mathbf{\Sigma})$.

Definition 3.8.2 (Probability Measures)**.** *For any finite set* $\mathbf{\Sigma}$ *considered, define* $\mathcal{F} \triangleq \mathcal{P}(\mathbf{\Sigma})$ *to be an algebra over* $\mathbf{\Sigma}$*. Furthermore, define the set of all probability measures over* $\mathbf{\Sigma}$ *to be* $\mathscr{M}(\mathbf{\Sigma}) \triangleq \{\mu \mid \langle \mathbf{\Sigma}, \mathcal{F}, \mu \rangle$ *is a probability space over* $\mathbf{\Sigma}\}$.

For any family $\Sigma_J \subseteq \mathcal{F}$ whose elements are pairwise disjoint it can be inductively shown that

$$\mu(\bigcup_{j \in J} \Sigma_j) = \sum_{j \in J} \mu(\Sigma_j).$$

This property is referred to as *finite additivity*. In the following, since the algebra $\mathcal{F} = \mathcal{P}(\mathbf{\Sigma})$ is the powerset of the finite set $\mathbf{\Sigma}$, it is sufficient to define μ for singleton subsets of $\mathbf{\Sigma}$ because we can derive the probability (using the finite additivity property) for any other event $\Sigma \in \mathcal{F}$ as

$$\mu(\Sigma) = \sum_{\sigma \in \Sigma} \mu(\{\sigma\}).$$

We shall often omit the braces for singleton events and simply write $\mu(\sigma)$, for brevity, instead of $\mu(\{\sigma\})$.

Conditional Probability

We shall use the notion of *conditional probability* to describe how an attacker's observation of a system's outputs affects the attacker's initial uncertainty about the inputs to the system. This is because information flow occurs when the attacker is able to reduce his or her uncertainty about inputs based on the observation of the system output.

Suppose $\mathbf{\Sigma}$ and $\mathcal{V}$ are respectively the sets of inputs and outputs of a system whose relational model is $S \subseteq \mathbf{\Sigma} \times \mathcal{V}$. We assume that both $\mathbf{\Sigma}$ and $\mathcal{V}$ are finite. Let $\mu \in \mathscr{M}(\mathbf{\Sigma} \times \mathcal{V})$ be a probability measure describing the probability of event $E \in \mathcal{F}$ (where $\mathcal{F} = \mathcal{P}(\mathbf{\Sigma} \times \mathcal{V})$) occurring. For any singleton event $\{(\sigma, v)\} \in \mathcal{F}$, we write $\mu(\sigma, v) \triangleq \mu(\{(\sigma, v)\})$ for the *joint probability* of input σ and output v occurring under the system in question.

For any $\sigma \in \mathbf{\Sigma}$, define $E_\sigma \triangleq \{(\sigma, v) \mid v \in \mathcal{V}\}$. Then the *marginal probability* of the input value σ occurring is given by $\mu(E_\sigma) = \sum_{v \in \mathcal{V}} \mu(\sigma, v)$. We shall simply denote this probability as $\mu(\sigma) \triangleq \mu(E_\sigma)$. Similarly, for any $v \in \mathcal{V}$, define $E_v \triangleq \{(\sigma, v) \mid \sigma \in \mathbf{\Sigma}\}$, the marginal probability of v is given by $\mu(v) \triangleq \mu(E_v)$.

Now suppose $E, E' \in \mathcal{F}$ are events, then the *conditional probability* that E occurs given that E' has occurred is written as $\mu(E \mid E')$, this is given by [Ros06]

$$\mu(E \mid E') \triangleq \frac{\mu(E \cap E')}{\mu(E')} \text{ if } \mu(E') > 0. \tag{3.5}$$

For a given $E' \in \mathcal{F}$, $\mu(\cdot \mid E')$ is also a probability measure [Ros06]. For any input

$\sigma \in \mathbf{\Sigma}$ and output $v \in \mathcal{V}$, we shall write the conditional probability that σ was selected given that v was observed as

$$\mu(\sigma \mid v) \triangleq \mu(E_\sigma \mid E_v) = \frac{\mu(\sigma, v)}{\mu(v)} \text{ if } \mu(v) > 0. \tag{3.6}$$

This definition follows directly from (3.5). Whenever $\mu(v) = 0$, the conditional probability $\mu(\sigma \mid v)$ is undefined. The conditional probability, $\mu(v \mid \sigma)$, that the output v is produced given that input σ was selected is similarly defined.

Random Variables

Assume that $\mu \in \mathscr{M}(\mathbf{\Sigma} \times \mathcal{V})$ is a probability measure over $\mathbf{\Sigma} \times \mathcal{V}$ as define above, we shall designate two random variables $X_{\mathbf{\Sigma}}$ over $\mathbf{\Sigma}$ and $X_{\mathcal{V}}$ over $\mathcal{V}$ respectively to represent occurrence of inputs in $\mathbf{\Sigma}$ and outputs in $\mathcal{V}$. For any $\sigma \in \mathbf{\Sigma}$, $X_{\mathbf{\Sigma}} = \sigma$ means that the random variable $X_{\mathbf{\Sigma}}$ takes on the value of σ, and the probability of this happening (written $\mu(X_{\mathbf{\Sigma}} = \sigma)$) is $\mu(E_\sigma) = \mu(\sigma)$, which as shown above is the marginal probability of selecting σ. Similarly, for any $v \in \mathcal{V}$, the probability of $X_{\mathcal{V}} = v$ occurring is $\mu(v)$. When we consider probability measures $\mu \in \mathscr{M}(\mathbf{\Sigma})$ over a set $\mathbf{\Sigma}$ alone, we shall use μ interchangeably for both the probability measure and the random variable over $\mathbf{\Sigma}$ induced by this measure.

Entropy

The notion of *entropy* describes, quantitatively, the degree of uncertainty encoded in a random variable. Suppose μ is a probability measure over $\mathbf{\Sigma}$, this induces a random variable that we also denote by μ, which takes on a value $\sigma \in \mathbf{\Sigma}$ with a

probability of $\mu(\sigma)$. The entropy of μ is defined as

$$\mathcal{H}(\mu) \triangleq \sum_{\sigma \in \mathbf{\Sigma}} \mu(\sigma) \log \left(\frac{1}{\mu(\sigma)} \right). \tag{3.7}$$

The logarithm in definition (3.7) is traditionally to the base 2 and the measurement unit is *bit*. Furthermore, whenever $\mu(\sigma) = 0$ then $\mu(\sigma) \log \frac{1}{\mu(\sigma)}$ is conventionally taken to be 0, which is reasonable since $\lim_{x \to 0^+} x \log x = 0$.

The value of $\mathcal{H}(\mu)$ measures the degree of uncertainty over the space $\mathbf{\Sigma}$ as described by μ. We shall use this measure to describe the information that an attacker gains by computing the difference in the attacker's uncertainty at two points in time: before and after the observation of outputs. For example, if by the observation of outputs the attacker whose uncertainty is encoded by the probability measure μ becomes less uncertain - represented by another probability measure μ', then the information gained can be characterised by the quantity $\mathcal{H}(\mu) - \mathcal{H}(\mu')$.

We note two properties of the entropy measure [Mac03, Sha48]. Suppose μ is a probability measure over the nonempty finite set $\mathbf{\Sigma}$ which has n elements, we have

- $\mathcal{H}(\mu) \geq 0$, and $\mathcal{H}(\mu) = 0$ when there exists a $\sigma \in \mathbf{\Sigma}$ for which $\mu(\sigma) = 1$ (uncertainty is minimised when an event becomes certain).
- $\mathcal{H}(\mu) \leq \log(n)$, and $\mathcal{H}(\mu) = \log(n)$ when $\mu(\sigma) = \frac{1}{n}$ for all $\sigma \in \mathbf{\Sigma}$ (uncertainty is maximised when all events are equally likely).

Conditional Entropy and Mutual Information

Now consider a probability measure $\mu \in \mathscr{M}(\mathbf{\Sigma} \times \mathcal{V})$ over $\mathbf{\Sigma} \times \mathcal{V}$, and the induced random variables $X_{\mathbf{\Sigma}}$ ($\forall \sigma \in \mathbf{\Sigma}$, $\mu(X_{\mathbf{\Sigma}} = \sigma) = \mu(\sigma)$) and $X_{\mathcal{V}}$ ($\forall v \in \mathcal{V}$, $\mu(X_{\mathcal{V}} = v) = \mu(v)$) respectively defined over $\mathbf{\Sigma}$ and $\mathcal{V}$. The entropy of the random variable $X_{\mathbf{\Sigma}}$ is given as

$$\mathcal{H}(X_{\mathbf{\Sigma}}) = \sum_{\sigma \in \mathbf{\Sigma}} \mu(\sigma) \log \left(\frac{1}{\mu(\sigma)} \right)$$

and similarly the entropy of the random variable $X_{\mathcal{V}}$ is

$$\mathcal{H}(X_{\mathcal{V}}) = \sum_{v \in \mathcal{V}} \mu(v) \log \left(\frac{1}{\mu(v)} \right).$$

The *conditional entropy* of the random variable $X_{\mathbf{\Sigma}}$, given the observation of the event $X_{\mathcal{V}} = v$ for some $v \in \mathcal{V}$ is defined as

$$\mathcal{H}(X_{\mathbf{\Sigma}} \mid X_{\mathcal{V}} = v) \triangleq \sum_{\sigma \in \mathbf{\Sigma}} \mu(\sigma \mid v) \log \left(\frac{1}{\mu(\sigma \mid v)} \right).$$

We shall write $\mathcal{H}(X_{\mathbf{\Sigma}} \mid X_{\mathcal{V}} = v)$ simply as $\mathcal{H}(X_{\mathbf{\Sigma}} \mid v)$, and it represents the uncertainty which remains about $X_{\mathbf{\Sigma}}$ when $X_{\mathcal{V}} = v$ is observed. For any $\sigma \in \mathbf{\Sigma}$, $\mathcal{H}(X_{\mathcal{V}} \mid \sigma)$ is similarly defined.

The average, or *expected conditional entropy* of $X_{\mathbf{\Sigma}}$ given $X_{\mathcal{V}}$ is defined as

$$\mathcal{H}(X_{\mathbf{\Sigma}} \mid X_{\mathcal{V}}) \triangleq \sum_{v \in \mathcal{V}} \mu(v) \mathcal{H}(X_{\mathbf{\Sigma}} \mid v).$$

The *mutual information* between the random variables $X_{\mathbf{\Sigma}}$ and $X_{\mathcal{V}}$ describes how much of information about $X_{\mathbf{\Sigma}}$ is encoded in $X_{\mathcal{V}}$ and vice versa. The mutual

information between $X_{\mathbf{\Sigma}}$ and $X_{\mathcal{V}}$ is defined as

$$I(X_{\mathbf{\Sigma}}; X_{\mathcal{V}}) \triangleq \mathcal{H}(X_{\mathbf{\Sigma}}) - \mathcal{H}(X_{\mathbf{\Sigma}} \mid X_{\mathcal{V}}). \tag{3.8}$$

Intuitively, $I(X_{\mathbf{\Sigma}}; X_{\mathcal{V}})$ measures the remaining uncertainty about $X_{\mathbf{\Sigma}}$ after $X_{\mathcal{V}}$ is known. This measure is always positive because $\mathcal{H}(X_{\mathbf{\Sigma}}) \geq \mathcal{H}(X_{\mathbf{\Sigma}} \mid X_{\mathcal{V}})$, and equality occurs when $X_{\mathbf{\Sigma}}$ and $X_{\mathcal{V}}$ are *independent* [Sha48]. The mutual information measure is used in our analysis to determine the information that flows from the inputs to the outputs of a system. The use of this measure to characterise information flow is not new [CHM07]. We now show that the information contained in random variables can be arranged on a lattice by using the entropy measure to describe the relative amount of information that they contain.

3.8.2 Lattice of Probabilistic Information

In this section we show that the set $\mathscr{M}(\mathbf{\Sigma})$ of probability measures over $\mathbf{\Sigma}$ can be arranged on a lattice based on Shannon's entropy measure of their relative quantitative information contents. The entropy measure, by design [Sha48], is not sensitive to permutations of the probabilities assigned to events under a given probability measure. For example, since it is immaterial which particular event becomes certain when we have the least entropy of zero, it is clear that there is no *unique* probability measure which maximises information release. This fact suggests that an order on probability measures based on Shannon's entropy will only be a preorder. However, this does not pose a serious technical difficulty as we can move to a partial order over equivalence classes of probability measures with the same entropy.

Definition 3.8.3 (Lattice of probability measures)**.** *Let $\mu, \mu' \in \mathscr{M}(\Sigma)$ be probability measures over a finite set Σ. Define a preorder $\preceq$ on $\mathscr{M}(\Sigma)$ as $\mu \preceq \mu'$ iff $\mathcal{H}(\mu') \leq \mathcal{H}(\mu)$.*

Now define the equivalence relation θ over $\mathscr{M}(\Sigma)$ as $\mu \,\theta\, \mu'$ iff $\mu \preceq \mu'$ and $\mu' \preceq \mu$, and define $\mathscr{M}_\theta(\Sigma) \triangleq \{[\mu]_\theta \mid \mu \in \mathscr{M}(\Sigma)\}$ to be the set of equivalence classes of the relation θ over $\mathscr{M}(\Sigma)$. Define a partial order $\sqsubseteq$ on $\mathscr{M}_\theta(\Sigma)$, which for any $\mu, \mu' \in \mathscr{M}(\Sigma)$ is given by $[\mu]_\theta \sqsubseteq [\mu']_\theta$ iff $\mu \preceq \mu'$. The join operation on $\mathscr{M}_\theta(\Sigma)$ is defined as usual such that $[\mu]_\theta \sqcup [\mu']_\theta = [\mu']_\theta$ iff $[\mu]_\theta \sqsubseteq [\mu']_\theta$.

The partially ordered set $\langle \mathscr{M}_\theta(\Sigma), \sqsubseteq, \sqcup \rangle$, which we call the *lattice of Shannon's information measures*, is a complete lattice as will be shown shortly. From an information-theoretic point of view, if $\mu \preceq \mu' \preceq \mu$ holds, the amount of information that two attackers whose uncertainties are described by the probability measures μ and μ' have is the same. However, since the relation $\preceq$ is not antisymmetric this does not necessarily mean that μ and μ' are the same. The reason is that the computation of entropy does not distinguish between mere permutations of probabilities of events over which a probability measure is defined [Sha48]. For example, if $\Sigma = \{\sigma, \sigma'\}$ such that $\mu(\sigma) = 1$ and $\mu(\sigma') = 0$, whereas $\mu'(\sigma) = 0$ and $\mu'(\sigma') = 1$, we have $\mathcal{H}(\mu) = \mathcal{H}(\mu') = 0$. In fact, for any pair of probability measures on this set where $\mu_1(\sigma) = \mu_2(\sigma')$ and $\mu_1(\sigma') = \mu_2(\sigma)$ it is easy to see that $\mathcal{H}(\mu) = \mathcal{H}(\mu')$. As entropy merely quantifies the degree of uncertainty in elements of $\mathscr{M}(\Sigma)$, we achieve partial ordering by moving to a set whose canonical elements are the equivalence classes of the relation $\theta \triangleq \preceq \cap \succeq$ which relates probability measures with equal entropy. This technique is standard, and it allows us to obtain a partially ordered set $\langle \mathscr{M}_\theta(\Sigma), \sqsubseteq \rangle$. The quantitative information content of the lattice of Shannon's information measures, $\langle \mathscr{M}_\theta(\Sigma), \sqsubseteq, \sqcup \rangle$, over Σ forms a

complete lattice.

Theorem 3.8.4. *Let Σ be a finite set. The ordered set $\langle \mathscr{M}_\theta(\Sigma), \sqsubseteq, \sqcup \rangle$ is a complete lattice.*

Proof. It is clear that $\sqsubseteq$ is a partial order. Now take any pair $[\mu]_\theta, [\mu']_\theta \in \mathscr{M}_\theta(\Sigma)$ for some $\mu, \mu' \in \mathscr{M}(\Sigma)$. The join, and the meet operation, which is dually defined, exist uniquely and are well defined. Hence $\mathscr{M}_\theta(\Sigma)$ is a lattice. From the properties of entropy and the fact that Σ is finite we know that there is a greatest element of $\mathscr{M}_\theta(\Sigma)$ corresponding to the entropy of 0 and there also exists a least element corresponding to the entropy $\mathcal{H}(\mu_\perp)$, where $\mu_\perp$ is the probability measure which assigns equal probabilities to all $\sigma \in \Sigma$. In particular, since entropy is continuous over probability measures [Sha48], $\langle \mathscr{M}_\theta(\Sigma), \sqsubseteq \rangle$ is lattice isomorphic to the closed real interval $I = \langle [0, \log(|\Sigma|)], \geq \rangle$ (which is bounded above and below respectively by the entropy measures 0 and $\log(|\Sigma|)$) via the isomorphism $\iota([\mu]_\theta) \triangleq \mathcal{H}(\mu)$. Hence, $\langle \mathscr{M}_\theta(\Sigma), \sqsubseteq \rangle$ is a complete lattice. □

3.8.3 Deriving Probabilistic Information Flow

We shall now apply the information-theoretic definitions above to the analysis of information flow using the relational model of systems. Here, in addition to the relational model $S \subseteq \Sigma \times \mathcal{V}$, we also make use of a joint probability measure $\mu \in \mathscr{M}(\Sigma \times \mathcal{V})$ over system's input-output domain characterising how the system transfers probabilistic information from its inputs to its outputs.

Definition 3.8.5. *Let $S_P \subseteq \Sigma \times \mathcal{V}$ be the relational model of a system P over its set Σ of inputs and set $\mathcal{V}$ of its outputs, both of which are finite. In addition, let $\hat{\mu} \in \mathscr{M}(\Sigma \times \mathcal{V})$ be a probability measure over $\Sigma \times \mathcal{V}$ such that for all $\sigma \in \Sigma$ and*

$v \in \mathcal{V}$, $\hat{\mu}(\sigma, v)$ *is the joint probability of supplying input* σ *to* P *and producing the output* v.

Let $\mu \in \mathscr{M}(\Sigma)$ *to be a probability measure over* Σ *describing an attacker's initial uncertainty over the input space such that for any* $\sigma \in \Sigma$, $\mu(\sigma) = \sum_{v \in \mathcal{V}} \hat{\mu}(\sigma, v)$. *Similarly, let* $\mu' \in \mathscr{M}(\mathcal{V})$, *such that for any* $v \in \mathcal{V}$, $\mu'(v) = \sum_{\sigma \in \Sigma} \hat{\mu}(\sigma, v)$ *is the marginal probability of observing output* v *in* P. *Furthermore, let the conditional probability measure* $\mu_v \in \mathscr{M}(\Sigma)$ *be the attacker's uncertainty about the inputs after observing output* $v \in \mathcal{V}$ *which for any* $\sigma \in \Sigma$ *is given by* $\mu_v(\sigma) \triangleq \hat{\mu}(\sigma \mid v)$.

The quantitative information flow via P *to an attacker whose initial uncertainty about the input is described by* μ *is given by*

$$I_{\langle P,\mu \rangle} \triangleq \mathcal{H}(\mu) - \sum_{v \in \mathcal{V}} \mu'(v)\, \mathcal{H}(\mu_v).$$

The information $I_{\langle P,\mu \rangle} = I(X_{\Sigma}; X_{\mathcal{V}})$ released by P is the *mutual information* (see (3.8)) between the random variable X_{Σ} induced by the probability measures μ over the input space Σ and random variable $X_{\mathcal{V}}$ induced by the probability measure μ' over the output space $\mathcal{V}$. Let us illustrate Definition 3.8.5 with examples.

Suppose the secret $h \in \{0, 1, 2, 3\}$ is a parameter to the program P and that P reveals the parity of h by producing an output $h \, mod \, 2$. Now suppose that h is chosen with uniform probability over its set of possible values. Then the relational model of P is given by $S_P \subseteq \Sigma \times \{0, 1\}$, where the 2-bit state Σ is represented by the set $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$ where for all $i, \sigma_i = [h \mapsto i]$ and we have $graph(S_P) = \{(\sigma_0, 0), (\sigma_1, 1), (\sigma_2, 0), (\sigma_3, 1)\}$. Since all inputs are equally likely, we have $\forall i, \mu(\sigma_i) = \frac{1}{4}$. Furthermore, because P is deterministic, for all $(\sigma, v) \in S_P$ the joint probability σ and v is $\hat{\mu}(\sigma, v) = \frac{1}{4}$. Hence we can compute the marginal

probabilities $\mu'(v)$ of the outputs $v \in \{0,1\}$, which are $\mu'(0) = \sum_{\sigma \in \mathbf{\Sigma}} \hat{\mu}(\sigma, 0) = \frac{1}{2}$, and $\mu'(1) = \frac{1}{2}$. The conditional probabilities $\mu_v(\sigma) = \hat{\mu}(\sigma \mid v)$ are as follows:

$$\mu_0(\sigma_0) = \frac{\hat{\mu}(\sigma_0, 0)}{\mu'(0)} = \frac{1}{2} = \mu_0(\sigma_2) \text{ and } \mu_0(\sigma_1) = \mu_0(\sigma_3) = 0.$$

Similarly,

$$\mu_1(\sigma_1) = \mu_1(\sigma_3) = \frac{1}{2} \text{ and } \mu_1(\sigma_0) = \mu_1(\sigma_2) = 0.$$

The information released is $I_{\langle P,\mu \rangle} = \mathcal{H}(\mu) - (\frac{1}{2}\mathcal{H}(\mu_0) + \frac{1}{2}\mathcal{H}(\mu_1) = 1$. The 1-bit information that is derived corresponds to the knowledge gained by the attacker, which now knows whether the parity of h is even or not. Under the same setup, where h is chosen with uniform probability, if we consider the program P_2 which reveals h directly, we now have the information released to be $I_{\langle P_2,\mu \rangle} = 2$ bits. This is clear, since the attacker completely learns the secret h. However, if we consider under the same setting, a program P_3, which produces a constant output regardless of the choice of h, the information flow is $I_{\langle P_3,\mu \rangle} = 0$ bits. This is also intuitive because the output of P_3 is independent of h. Definition 3.8.5 applies also to nondeterministic system models, which we illustrate next.

In this example, we introduce a construct $[]_p$ for probabilistic nondeterminism. Informally, the semantics of the probabilistic construct $c_1 \; []_p \; c_2$ is to execute the subprogram c_1 with a probability of p and to execute program c_2 with a probability of $1 - p$, where $0 < p < 1$. The formal semantics of a programming language featuring this construct is presented in Chapter 4. Now, consider the nondeterministic program $P_4 \triangleq P \; []_{.8} \; P_5$ which accepts a parameter $h \in \{0,1,2,3\}$ and which is made up of two deterministic subprograms: P - introduced above, which reveals the parity of h, and P_5 which produces output 2

when $h = 0$ and produces output 3 otherwise - hence P_5 reveals whether $h = 0$ or not. Now suppose, as in the previous example, that h is chosen with equal probability as the input to P_4. The set of possible outputs of P_4 is $\mathcal{V}_4 = \{0, 1, 2, 3\}$ and thus its relational model is given by $S_{P_4} \subseteq \mathbf{\Sigma} \times \mathcal{V}_4$, where *graph*$(S_{P_4}) = \{(\sigma_0, 0), (\sigma_0, 2), (\sigma_1, 1), (\sigma_1, 3), (\sigma_2, 0), (\sigma_2, 3), (\sigma_3, 1), (\sigma_3, 3)\}$. Using the fact that for all $i = 0, 1, 2, 3$, the probability of choosing the input σ_i is $\mu(\sigma_i) = \frac{1}{4}$, then the input-output joint probabilities is computed from P_4 as $\mu(\sigma_0, 0) = \mu(\sigma_1, 1) = \mu(\sigma_2, 0) = \mu(\sigma_3, 1) = \frac{1}{5}$ and $\mu(\sigma_0, 2) = \mu(\sigma_1, 3) = \mu(\sigma_2, 3) = \mu(\sigma_3, 3) = \frac{1}{20}$. From these we obtain the marginal probabilities of the outputs, where $\mu'(i)$ is the probability of producing output i as: $\mu'(0) = \mu'(1) = \frac{2}{5}$ and $\mu'(2) = \frac{1}{20}$ and $\mu'(3) = \frac{3}{20}$. Additionally, by applying the definitions, the conditional probabilities ($\mu_i(\sigma_j)$) that σ_j was chosen as the input given the observation of the output i are the following: $\mu_0(\sigma_0) = \mu_0(\sigma_2) = \mu_1(\sigma_1) = \mu_1(\sigma_3) = \frac{1}{2}$ and $\mu_2(\sigma_0) = 1$ and $\mu_3(\sigma_1) = \mu_3(\sigma_2) = \mu_3(\sigma_3) = \frac{1}{3}$, and for every other i and j, we have $\mu_i(\sigma_j) = 0$. The quantitative information released by P_4 is given by $I_{\langle P_4, \mu \rangle} = \mathcal{H}(\mu) - \sum_{v \in \mathcal{V}_4} \mu(v)\mathcal{H}(\mu_v) \approx 0.9623$. Although the result 0.9623 bit is not very intuitive, there is some logic behind the value. Considered independently, the program P_5 reveals about 0.8113 bit of information about h, and we have already shown earlier that P reveals 1 bit of information about h. However, in P_4, P is executed 80% of the time and P_5 is executed in the remaining 20%. Thus, the information released about h in P agrees with the semantics and comes from the fact that $0.8 \times I_{\langle P, \mu \rangle} + 0.2 \times I_{\langle P_5, \mu \rangle} \approx 0.9623$. This captures a sense of the frequency or the weighted rate of information release by the two subprograms.

Summary We have developed a lattice-based policy model for *what* declassification policies in this chapter. Useful policy patterns were identified under the lattice-based policy model. The lattice model of information is shown to capture natural intuitions about information and information flow. An input-output relational model was presented as a theoretical primitive for the analysis of information flow. Using the relational model, various representations of information suitable for the analysis of information flow in deterministic and nondeterministic systems were developed and shown to fit into the lattice model of information.

In Chapter 4 we shall show how to derive the relational model, in language-based settings, from the operational semantics of programming languages. The analyses, which are performed parametric to an attacker model, allow us to study information gained by the chosen attacker. The attackers are assumed to have a specification (such as the algorithm or protocol that the system implements, or the program source code) of the system being attacked so that, given any input, the attacker can work out the possible output(s) that the system can produce. These assumptions describe, for example, the malicious code scenario where the attacker is possibly the author of the program that processes sensitive data.

Chapter 4

Information Flow in Computational Systems

The goal of this chapter is to demonstrate how to develop the relational model introduced in the previous chapter from the operational semantics in a language-based setting, and to show how to derive a system's information flow property from this relational model, which is defined parametric to a given attacker's observational power.

To provide a concrete language-based setting, a simple *While* language with *output* is introduced as an imperative core language for studying systems with output *interactions*. The operational semantics of this core language is presented. An illustrative *semantic attacker* model is presented to demonstrate the definition of attacker models for information flow analyses. The resulting analyses of information flow with respect to the semantic attacker model demonstrate how termination-sensitive analyses may be developed under the relational model.

4.1 Operational Semantics and Observational Power

We consider *computational systems*, which process confidential data supplied as part of their inputs and which may produce publicly observable outputs. In order to model what an attacker may learn by observing such a system, we need to be able to specify what the attacker can observe about the system's operation. This is referred to as the *attacker model* and is defined by the attacker's *observational power*, which describes what the attacker can see about the execution of the system. We shall formalise the attacker's observational power with respect to the operational semantics of the system being analysed to enable us to derive the system's information flow property, relative to this attacker model.

A standard way to model the operational semantics of a computational system is by using a *transition systems*. We shall use a *labelled transition system* to model the operational semantics, where the labels in the transition relation of the transition system describe what the attacker sees during a transition of the system in question. Labelled transition systems are powerful tools for describing, at different levels of abstractions, the aspects of a system's operation that are relevant to an analysis. Well known examples of operational description of systems include the structural operational semantics (SOS) that is also called the "small-step" semantics [Plo81], and evaluation relations (also called "big-step SOS" or the "natural semantics" [Kah87]), which are used to formalise the semantics of programs. It is also common to describe interactions of a system with (or its *effects* [NNH99] on) its environment by using labels in the transition system [Mil99, AFV01, PAK02].

4.1.1 Labelled Transition Systems and Interaction

A transition system is a pair $(\mathscr{C}, \twoheadrightarrow)$, where $\mathscr{C}$ is a set of system *configurations* or states, and the binary relation $\twoheadrightarrow \subseteq \mathscr{C} \times \mathscr{C}$ defines valid transitions between configurations. The reflexive, transitive closure of $\twoheadrightarrow$, written as $\twoheadrightarrow^*$, is defined as $s_0 \twoheadrightarrow^* s_n$ iff there is a sequence of transitions $s_0 \twoheadrightarrow s_1 \twoheadrightarrow \cdots \twoheadrightarrow s_n$, for some $s_0, s_1, \cdots, s_n \in \mathscr{C}$. The transition system $(\mathscr{C}, \twoheadrightarrow)$ is said to be *deterministic* when the resulting configuration of every transition is uniquely determined by the initial configuration, that is, for all $s, s', s'' \in \mathscr{C}$, $s \twoheadrightarrow s'$ and $s \twoheadrightarrow s''$ means that $s' = s''$, otherwise the transition system is said to be *nondeterministic*.

We shall consider systems which have output *interactions*, which may occur at any point during the run of the system, and which may be observable by an attacker. *Interactive systems* [Mil99] are traditionally modelled as as *labelled transition systems* (or *automaton* [HU79]), where labels capture the interactions between the system and its environment. A labelled transition system is a triple $(\mathscr{C}, \longrightarrow, \mathscr{A})$, which may be viewed as a transition system that is augmented with a set $\mathscr{A}$ of *labels* (also referred to as *actions*) to capture the system's interactions. The transition relation in a labelled transition system is a ternary relation $\longrightarrow \subseteq \mathscr{C} \times \mathscr{A} \times \mathscr{C}$. If for some action $a \in \mathscr{A}$ and configurations $s, s' \in \mathscr{C}$ we have that $(s, a, s') \in \longrightarrow$, then the system is said to make a transition from configuration s to s' with the effect a. This is often written as $s \xrightarrow{a} s'$. A labelled transition system $(\mathscr{C}, \longrightarrow, \mathscr{A})$ is deterministic if for all $s, s', s'' \in \mathscr{C}$ and $a \in \mathscr{A}$, $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ means that $s' = s''$, otherwise it is nondeterministic [Mil99].

4.1.2 Attacker Models

In this section we shall formalise attacker models via a notion of *observational power* that describes what an attacker sees during a run of a labelled transition system. Suppose $\mathscr{T} = (\mathscr{C}, \longrightarrow, \mathscr{A})$ is a labelled transition system which formalises the semantics of a given computational system. In order to model what the attacker can see about this system's operation, we can define a function $\mathscr{O} : \mathscr{C} \times \mathscr{A} \times \mathscr{C} \to \mathscr{A}_{\mathscr{O}}$ on the transition relation, called the observational power, which models what the attacker sees during each transition of the system $\mathscr{T}$. The set $\mathscr{A}_{\mathscr{O}}$ may be chosen arbitrarily, elements of which represent what the attacker actually sees. Thus, the observational power $\mathscr{O}$ can be defined such that for all $(s, a, s') \in \longrightarrow$, where $s, s' \in \mathscr{C}$ and $a \in \mathscr{A}$ there exists $a' \in \mathscr{A}_{\mathscr{O}}$ such that $\mathscr{O}(s, a, s') = a'$. The intention is that, while the action a in (s, a, s') captures the interaction of $\mathscr{T}$ in the original sense, $\mathscr{O}$ rewrites this as a' to describe what the attacker *actually sees*. In the simplest case, $\mathscr{O}$ is just an identity function on actions, which results in an attacker model that is interacting with the system as originally prescribed by the operational semantics. However, we may want to model more powerful attacker's which can observe *internal* system actions, such as an attacker running a program in a debugger, where internal actions can be observed; or a less powerful attacker which is interacting only with a part of the system where otherwise externally visible actions in some system parts are invisible under the attacker model, for example, in a client-server application where the attacker in question can only observe the system's interactions that are visible on the attacker's client.

The observational power $\mathscr{O}$ allows us to specify precisely (by transforming

system interactions) how an attacker interacts with the system. This gives us a general tool to model different kinds of attackers which can interact with a system in non-standard ways. The introduction of $\mathscr{O}$ induces a view $\mathcal{T}_{\mathscr{O}} = (\mathscr{C}, \longrightarrow_{\mathscr{O}}, \mathscr{A}_{\mathscr{O}})$ of the system $\mathcal{T} = (\mathscr{C}, \longrightarrow, \mathscr{A})$, where $\mathcal{T}_{\mathscr{O}}$ is a labelled transition system, which describes $\mathcal{T}$ as the attacker with the observational power $\mathscr{O}$ sees it. Thus, if the transition relation $\longrightarrow_{\mathscr{O}} \subseteq \mathscr{C} \times \mathscr{A}_{\mathscr{O}} \times \mathscr{C}$ of the transition system $\mathcal{T}_{\mathscr{O}}$ is defined as $(s, a', s') \in \longrightarrow_{\mathscr{O}}$ iff $(s, a, s') \in \longrightarrow$ and $\mathscr{O}(s, a, s') = a'$, then the induced transition system $\mathcal{T}_{\mathscr{O}}$ is capable of describing every possible transition in the semantics of $\mathcal{T}$ because $\mathscr{O}$ is totally defined over all (s, a, s') related by $\longrightarrow$. Observational powers $\mathscr{O}_A$ and $\mathscr{O}_B$ defined over a system's transition relations can be compared according to their relative powers, where the attacker modelled by $\mathscr{O}_B$ is said to be at least as powerful as the attacker modelled by $\mathscr{O}_A$ if there exists a function f such that $\mathscr{O}_A = f \circ \mathscr{O}_B$, that is, the observational power function $\mathscr{O}_A$ has less variety than $\mathscr{O}_B$.

We may also describe an attacker's observational power over a system at the *trace level* rather than only at the level of the individual transitions in the transition system as shown above. A trace of the labelled transition system $\mathcal{T} = (\mathscr{C}, \longrightarrow, \mathscr{A})$ is a sequence of transitions, written as $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ where for all n, $(s_n, a_n, s_{n+1}) \in \longrightarrow$, and $s_0 \in \mathscr{C}_i$ is a configuration chosen from a distinguished set of starting configurations $\mathscr{C}_i$ of the system modelled by $\mathcal{T}$. Such an observational power, $obs(\cdot)$, is thus a map from the set of traces of $\mathcal{T}$ to the set of observed sequences. Observational powers $obs_A(\cdot)$ and $obs_B(\cdot)$ over a system's traces may also be arranged according to their relative degrees of power, where $obs_B(\cdot)$ is said to be at least as powerful as $obs_A(\cdot)$ if there exists a function f such that $obs_A(\cdot) = f \circ obs_B(\cdot)$.

In the next section we shall show how to derive the relational model of a system, given the relevant transition system and a model of the attacker's observational power over the transition system.

4.1.3 Deriving the Relational Model

In Chapter 3 we showed how to derive the information released by a system from its relational model which associates the system's input with the output(s) that the attacker observes. We now show how such a relational model can be derived for the system described by the labelled transition system $(\mathscr{C}, \longrightarrow, \mathscr{A})$ under the observational power $\mathscr{O}$ of a given attacker model. This leads to a configuration $\mathcal{TS} = (\mathscr{C}, \longrightarrow, \mathscr{A}, \mathscr{O})$, which refers to the original labelled transition system $(\mathscr{C}, \longrightarrow, \mathscr{A})$ augmented with the attacker model described by $\mathscr{O}$. The induced $\mathcal{TS}$ essentially defines a new transition system that is similar to the definition of $\mathcal{T}_{\mathscr{O}}$ from $\mathcal{T}$ in the previous section. It is also clear that the original labelled transition system $(\mathscr{C}, \longrightarrow, \mathscr{A})$ is a special case of $\mathcal{TS}$, which is obtained when $\mathscr{O}$ is an identity on $\mathscr{A}$.

Definition 4.1.1 (Deriving the Relational Model)**.** *Let $\mathcal{TS} = (\mathscr{C}, \longrightarrow, \mathscr{A}, \mathscr{O})$ be a labelled transition system under the observational power $\mathscr{O}$, and let $\mathscr{C}_i \subseteq \mathscr{C}$ be the set of all the initial configurations of $\mathcal{TS}$. Define the set of all finite and infinite traces of $\mathcal{TS}$ as seen by $\mathscr{O}$ to be*

$$T_{\mathcal{TS}} \triangleq \{s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots | s_0 \in \mathscr{C}_i . \forall n \geq 0, (s_n, a'_n, s_{n+1}) \in \longrightarrow, \mathscr{O}(s_n, a'_n, s_{n+1}) = a_n\}.$$

Furthermore, for any $s_0 \in \mathscr{C}_i$ let $t_{s_0} \subseteq T_{\mathcal{TS}}$ be the set of all traces starting at the initial configuration s_0. Now define the set of observations, given the starting con-

figuration s_0 *as* $\mathit{obs}(t_{s_0}) \triangleq \{a_0 a_1 \cdots \mid s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \in t_{s_0}\}$*, modelling the possible interactions of* $\mathcal{TS}$ *with the attacker whose observational power is* $\mathscr{O}$ *when* $\mathcal{TS}$ *is started at the configuration* s_0*. The set of all observations of the system is given by* $\mathcal{V} = \bigcup_{s \in \mathscr{C}_i} \mathit{obs}(t_s)$ *and the* relational model *of this system induced by* $\mathscr{O}$ *is now the relation* $S \subseteq \mathscr{C}_i \times \mathcal{V}$ *whose graph is*

$$\mathit{graph}(S) = \{(s, a) \mid s \in \mathscr{C}_i, a \in \mathit{obs}(t_s)\}.$$

This definition provides us with a general tool for describing information flow in deterministic and nondeterministic systems. The element $a = a_0 a_1 \cdots \in \mathit{obs}(t_{s_0})$ is the sequential juxtaposition of the individual observations $a_0, a_1, \cdots$ of the trace $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \in t_{s_0}$. Since the system may be nondeterministic it is clear that $\mathit{obs}(t_s)$ may not be a singleton set and, therefore, S may not be a function from configurations to sequences of actions. The relation S abstractly describes the interaction a of the attacker with the system (modelled by $\mathcal{TS}$), given the initial configuration s of the system, whenever s is related to a by S.

To illustrate the definitions above in a more concrete setting, the next section presents the *While* language as a language-based instantiation for the analysis of information flow in deterministic systems.

4.2 The *While* Language

We now present the imperative *While* language, shown in Figure 4.1, upon which our analysis of information flow in the next chapter shall be based. The operational semantics of this language is fairly standard [NNH99, Win93], and it has been

used as the core imperative language in many language-based security settings. An addition to this language is the *write* construct for program output, which is used to model output interaction of a system with its environment. The semantics of the *write* statement is the same as that of the *output* statement of [GBJS06]. Information flow in interactive programs is recently gaining more attention in language-based security [Bac05, OCC06, GBJS06, AS07, AHSS08]. Since interactive programs are common in practice, the study of the effect of interaction on information flow is important to give us a more realistic account of information flow in real systems.

$$c ::= \texttt{skip} \mid z := e \mid \texttt{write}\, e \mid c; c \mid \texttt{if}(b)\texttt{then}\, c\, \texttt{else}\, c \mid \texttt{while}\,(b)\,\texttt{do}\, c.$$

Figure 4.1: *The* While *Language with Output*

4.2.1 *While* Expressions and Program States

We consider only *boolean* and *integer* expressions in *While* programs, but the analysis techniques developed can be extended to other data types in a fairly straightforward manner. Boolean-valued expressions (ranged over by b) and integer-valued expressions respectively evaluate to values in the set $\mathbb{B} \triangleq \{\mathbf{tt}, \mathbf{ff}\}$ and $\mathbb{Z} \triangleq \{\cdots, -2, -1, 0, 1, 2, \cdots\}$. Accordingly, we have standard data types $\tau \in \{\mathit{bool}, \mathit{int}\}$ whose denotations are sets, and are given by $[\![\mathit{bool}]\!] \triangleq \mathbb{B}$ and $[\![\mathit{int}]\!] \triangleq \mathbb{Z}$. The set $\mathbf{Exp}$ (ranged over by e and b) of all expressions considered are constructed in the standard way by using arithmetic and boolean operators. Furthermore, program variables are taken from the set $\mathbf{Var}$, which is ranged over by x, y, z, h and l, using

subscripts when necessary. We also refer to the function $FV : \mathbf{Exp} \to \mathcal{P}(\mathbf{Var})$, which denotes the set of *free variables* in a given expression.

Program states $\sigma \in \mathbf{\Sigma}$ are finite maps from variables to values. The *evaluation* of expression $e \in \mathbf{Exp}$ at a state $\sigma \in \mathbf{\Sigma}$ is summarised as $\sigma(e)$, and for any expression e and state σ considered it is assumed that $FV(e) \subseteq \mathit{dom}(\sigma)$, where $\mathit{dom}(\sigma)$ is the domain of definition of σ. Furthermore, the evaluation of an expression is assumed to have no *side effect* on the program state.

4.2.2 *While* Commands

The set of *While* commands is denoted by $\mathbf{Com}$. As Figure 4.1 shows, simple commands include the standard *skip* statement and *assignment* statement, as well as the *write* statement (used for program output). Other program command constructors include the conditional *if* statement construct - for choice, the conditional *while* statement construct - for iteration or looping, and the composition constructor (;) for sequential composition of programs. The operational semantics of *While* is presented next.

4.2.3 The Operational Semantics of *While*

In this section we present the operational semantics of the *While* language. The analysis that will be performed is based on an attacker model which can observe output interactions. Thus, in order to formalise this interaction, we introduce two basic types of actions: $\varepsilon, \mathit{out}(\cdot) \in \mathscr{A}$, where ε stands for *internal action* that is not ordinarily observable from the environment and $\mathit{out}(v)$ stands for the *output* of the value v to the system's environment where v can be observed by the attacker.

The operational semantics of *While* is specified by transition relations between *command* and *expression configurations*. A command configuration is a pair $\langle c, \sigma \rangle \in \mathbf{Com} \cup \{\cdot\} \times \mathbf{\Sigma}$, which represents a command c to be executed at the state σ. When there are no more commands to execute, a special *terminal command configuration*, $\langle \cdot, \sigma \rangle$, indicates the termination of the program in the state $\sigma \in \mathbf{\Sigma}$. Similarly, an expression configuration is a pair $\langle e, \sigma \rangle \in \mathbf{Exp} \times \mathbf{\Sigma}$, which evaluates the expression e at the state σ. Since the evaluation of an expression does not have side effect on program states and the evaluation operation itself is taken to be an internal action, the evaluation relation for the expression e at the state σ is summarised as $\langle e, \sigma \rangle \xrightarrow{\varepsilon} \langle \sigma(e), \sigma \rangle$, where $\sigma(e)$ is the value of e at σ.

Assignments modify program states, and in anticipation of this, we use the standard definition of state update for some program state $\sigma \in \mathbf{\Sigma}$ and $z_1, z_2 \in dom(\sigma)$ which updates the variable z_1 in state σ with a value v that is taken from the data type of z_1 as follows

$$\sigma[z_1 \mapsto v](z_2) \triangleq \begin{cases} v & \text{if } z_1 = z_2 \\ \sigma(z_2) & \text{otherwise.} \end{cases} \tag{4.1}$$

The full operational semantics of *While* is presented in Figure 4.2. This semantics is fairly standard, with the exception of the transition rule for *write* statements. This rule says that *write* does not modify program state, but that the observer can see the value of the evaluated expression when the statement is executed. The label $a \in \mathscr{A}$ represents a program action which an attacker may be able to observe.

Let us illustrate how this can be used to describe how an attacker might reason about the possible starting state of a deterministic program. Consider the pro-

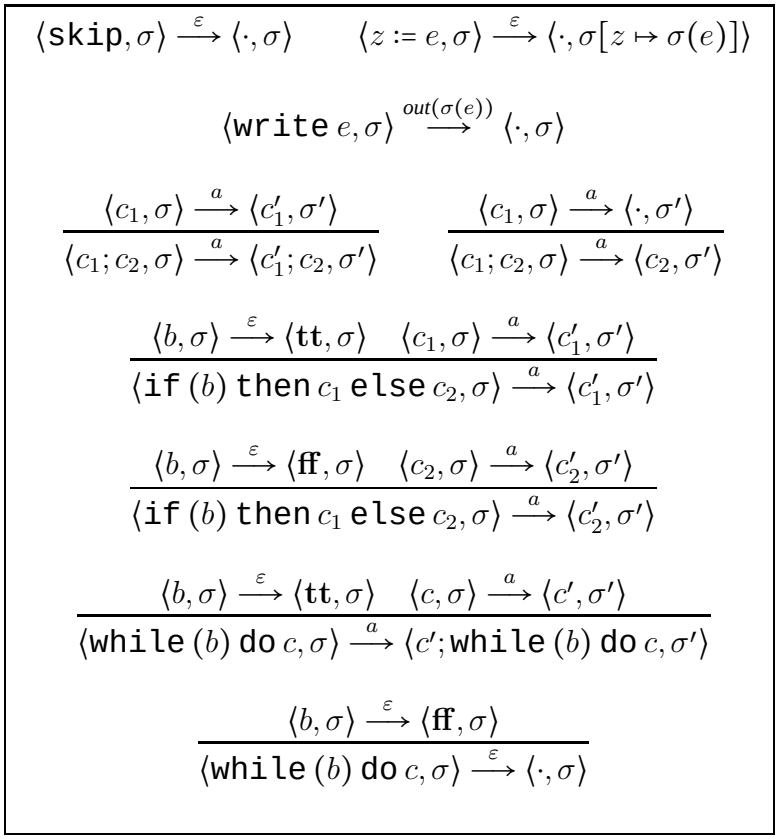


Figure 4.2: *Operational semantics of* While

gram in Figure 4.3, where the attacker can see the program outputs via the *write* statements, but does not know the value of the secret integer input x.

```
if (x<10) then
    write 1;
else
    write 2;

if (x=15) then
    write 1;
else
    write 2;
```

Figure 4.3: *Reasoning about program secrets*

We denote the sequence of output values of this program as tuples. For example, the observation for the trace $\langle P, \sigma \rangle \xrightarrow{out(1)} \langle P', \sigma \rangle \xrightarrow{out(2)} \langle \cdot, \sigma \rangle$ is denoted by $\langle 1, 2 \rangle$. Thus, for any chosen starting state, this program produces an output in the set $\mathcal{V} = \{\langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle\}$, which the attacker can use to reason about possible starting states as follows.

- The output sequence $\langle 1, 2 \rangle$, correspond to two individual outputs from the *write* statements in the *then* branch of the first *if* statement and the *else* branch of the second *if* statement. Using the operational semantics, the attacker can derive the fact that both $x < 10$ and $x \neq 15$ hold. The output $\langle 1, 2 \rangle$ is produced by all traces through the indicated conditional branches and these traces have a starting state in the set $\Sigma_{12} = \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) < 10\} \cap \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) \neq 15\} = \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) < 10\}$, where the attacker learns that the starting value of x is less than 10.

- Similarly, the output sequence $\langle 2, 1 \rangle$ corresponds to the set of traces starting

at the states in $\Sigma_{21} = \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) \geq 10\} \cap \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) = 15\} = \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) = 15\}$. Here, the attacker learns that the value of x is 15.

- Finally, the output sequence $\langle 2, 2\rangle$ corresponds to the set of traces starting at the states in $\Sigma_{22} = \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) \geq 10\} \cap \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) \neq 15\} = \{\sigma \in \mathbf{\Sigma} \mid \sigma(x) \neq 15, \sigma(x) \geq 10\}$. Here, the attacker learns that x is greater than or equal to 10 but is not 15.

This program can be described by the functional model $f : \mathbf{\Sigma} \to \mathcal{V}$ mapping the program's starting state to the produced output. The graph of f is given by $\mathit{graph}(f) = \{(\sigma_{12}, \langle 1, 2\rangle), (\sigma_{21}, \langle 2, 1\rangle), (\sigma_{22}, \langle 2, 2\rangle) \mid \sigma_{12} \in \Sigma_{12}, \sigma_{21} \in \Sigma_{21}, \sigma_{22} \in \Sigma_{22}\} \subseteq \mathbf{\Sigma} \times \mathcal{V}$. Using definition (3.1), we obtain the information flow released by this program as the kernel κ_f of f, which is given by

$$\begin{aligned} \forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; \kappa_f \; \sigma' \quad \text{iff} \quad & \sigma(x), \sigma'(x) < 10, \\ & \text{or } \sigma(x) = \sigma'(x) = 15, \\ & \text{or } \sigma(x), \sigma'(x) \in \{n \in \mathbb{Z} \mid 15 \neq n \text{ and } n \geq 10\}. \end{aligned}$$

The equivalence relation κ_f over $\mathbf{\Sigma}$ describes the information that the attacker gains by observing the outputs of the program in Figure 4.3. This leads us to the definition of information flow properties of programs based on the operational semantics.

4.3 Semantic Information Flow Property

In this section, the information released by a *While* program P is defined based on the observational power of a semantic attacker. This information, which is an

equivalence relation over the program states is used to define the program's information flow property in section 4.3.2. The termination properties of the definition are discussed in section 4.3.3.

4.3.1 The Semantic Attacker Model

We now define an attacker model that will be used throughout this thesis. We refer to this attacker model as the *semantic attacker*, because its definition is based on the standard operational semantics of *While*, with the exception of its ability to determine nontermination. The ability of the semantic attacker to determine nontermination appears to be strong and deserves some explanation. Nontermination is usually not modelled in language-based security because nontermination is not observable. However, modelling information flow due to nontermination is important because in practice, for example, in a hostile code scenario, where the attacker is probably the author of the program, or otherwise has knowledge of the program code, the attacker may be able to determine when the program will not terminate without actually observing it. It has been recently shown that modelling the ability to "observe" nontermination is important, especially for interactive programs, because it is possible to leak arbitrary amount of information via nontermination channels [AHSS08].

The observational power of the semantic attacker is given by the function $obs(\cdot)$ from traces to observations, which is defined below. The semantic attacker cannot observe internal actions ε produced by a program, but can observe all other actions including whether the program terminates or not. For example, *skip* and *assignment* statements, which only generate the internal action ε cannot be ob-

served by the semantic attacker.

Let Σ be the set of all states of the *While* program P. A trace of P starting at the state $\sigma_0 \in \Sigma$ is said to be *terminating* or *finite* if there exists a natural number n such that $\langle P, \sigma_0\rangle \xrightarrow{a_0} \langle P_1, \sigma_1\rangle \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} \langle \cdot, \sigma_n\rangle$, otherwise the trace is said to be *infinite* and P *diverges* at σ_0. The notation $\langle P, \sigma\rangle \Downarrow \sigma'$ denotes the termination of the program P in the state $\sigma' \in \Sigma$ when it is executed from the starting state $\sigma \in \Sigma$. The notation $\langle P, \sigma\rangle \Uparrow$ denotes the divergence of P when it is executed from the starting state $\sigma \in \Sigma$. What the semantic attacker sees during a run of the program P is defined based on the standard operational semantics of *While* as follows.

Definition 4.3.1 (Semantic Attacker Model)**.** *Let P be a* While *program and let Σ be the set of all states of P. Furthermore, let $\xrightarrow{\varepsilon}{}^*$ be the reflexive, transitive closure of $\xrightarrow{\varepsilon}$. Define $t_{\langle P,\sigma\rangle} \triangleq \langle P, \sigma\rangle \xrightarrow{\varepsilon}{}^* \langle P_0, \sigma_0\rangle \xrightarrow{a_0} \langle P_0', \sigma_0'\rangle \xrightarrow{\varepsilon}{}^* \langle P_1, \sigma_1\rangle \xrightarrow{a_1} \cdots$ to be a canonical representation of P's trace starting from the state $\sigma \in \Sigma$ such that for all i we have $\varepsilon \neq a_i \in \mathscr{A}$. What the semantic attacker sees during the trace $t_{\langle P,\sigma\rangle}$ is given by the observational power function obs$(t_{\langle P,\sigma\rangle})$ on traces defined as*

$$\textit{obs}(t_{\langle P,\sigma\rangle}) \triangleq \begin{cases} \langle a_0, a_1, \cdots, \uparrow\rangle & \textit{if } P \textit{ diverges at } \sigma \\ \\ \langle a_0, a_1, \cdots, \downarrow\rangle & \textit{otherwise.} \end{cases} \tag{4.2}$$

Define the set of all traces of P as

$$T_P \triangleq \{t_{\langle P,\sigma\rangle} \,|\, \sigma \in \Sigma\}. \tag{4.3}$$

Finally, define the equivalence relation on program states induced by the se-

mantic attacker's observation as

$$\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \lfloor T_P \rfloor \sigma' \iff \mathit{obs}(t_{\langle P,\sigma \rangle}) = \mathit{obs}(t_{\langle P,\sigma' \rangle}). \tag{4.4}$$

The definition of *obs*$(\cdot)$ formalises the idea that the semantic attacker cannot observe $\xrightarrow{\varepsilon}$ transitions. For nonterminating traces, the token $\uparrow$ is introduced which, in addition to the sequence of outputs observed on the trace, signals the divergence of the program. Similarly, the token $\downarrow$ signals the termination of the program. By accommodating possible knowledge of nontermination, the definition of *obs*$(\cdot)$ allows us to properly account for information flow in the presence of program divergence as demonstrated in section 4.3.3. When P diverges, the sequence $a_0, a_1, \cdots$ of observed output may or may not be finite. However, when P terminates, this sequence is always finite.

Distinguishability of traces and therefore input states is based on the inequality of sequences of observable actions. Two sequences $a_0, a_1, \ldots$ and $a'_0, a'_1, \ldots$ are equal iff for all i we have $a_i = a'_i$. Since the attacker distinguishes between input states by observing differences in program traces, any pair of traces of P starting at states $\sigma, \sigma' \in \mathbf{\Sigma}$ is indistinguishable to the semantic attacker if $\mathit{obs}(t_{\langle P,\sigma \rangle}) = \mathit{obs}(t_{\langle P,\sigma' \rangle})$. This definition induces the equivalence relation $\lfloor T_P \rfloor$ on the set of states that on one hand relates any pair of states that the attacker cannot distinguish, and thus describes the information released by P. On the other hand, any pair of states $\sigma, \sigma' \in \mathbf{\Sigma}$ that is not related by $\lfloor T_P \rfloor$ has the property that $\mathit{obs}(t_{\langle P,\sigma \rangle}) \neq \mathit{obs}(t_{\langle P,\sigma' \rangle})$, and hence can be distinguished by the attacker. Thus, $\lfloor T_P \rfloor$ is the smallest equivalence relation, based on the standard operational semantics, for which any pair of state that it relates cannot be distinguished by the

semantic attacker.

4.3.2 Defining the Information Flow Property

We shall define the *semantic information flow property* of a program P from $\lfloor T_P \rfloor$. Firstly, we highlight the link between the definition of $\lfloor T_P \rfloor$ and the kernel-based equivalence relation definition of information release presented in Section 3.5.1. Let $\mathcal{V} = \{obs(t_{\langle P,\sigma \rangle}) \mid \sigma \in \mathbf{\Sigma}\}$ and let $f_P : \mathbf{\Sigma} \to \mathcal{V}$ be the functional model (since P is deterministic) of P such that for any $\sigma \in \mathbf{\Sigma}$, $f_P(\sigma) = obs(t_{\langle P,\sigma \rangle})$. It is easy to see that the equivalence relation $\lfloor T_P \rfloor$ is the kernel of the function f_P and thus describes the information released by P. Now, let $\mathcal{I}$ be the lattice $\text{PER}(\mathbf{\Sigma})$, then the information flow property of the program P can be defined as the singleton set

$$[\![P]\!]^{\mathcal{I}} = \{f \mid \forall R \in \text{PER}(\mathbf{\Sigma}), f(R) = R \sqcup \lfloor T_P \rfloor\}. \tag{4.5}$$

This definition describes how the semantic attacker's knowledge changes by observing the program P. Furthermore, if $F \subseteq \mathcal{F}lows$ is an information flow policy defined over the lattice of PERs, the program P satisfies the policy F if there exists $f' \in F$ which allows the flow $f \in [\![P]\!]^{\mathcal{I}}$ caused by P: that is, $f \sqsubseteq f'$. However, the definition of information flow property does not have to be a singleton set as we shall show next.

Non-singleton Information Flow Properties

We may want to separate the traces of the system described by the program P based on some considerations. For example, we might want to ensure that information is not released in certain parts of a program - say in the subprogram

executed when authentication fails. We can thus partition the set of traces of the program to those in which the authentication fails and those in which it succeeds, and compute the information flow properties separately over these traces. More generally, let us assume that J is an index set of the traces of P such that for any $j \in J$, $T_P^j \subseteq T_P$ is the set of traces of P identified under j, and where $T_P = \bigcup_{j \in J} T_P^j$. We can define the information flow restricted to the traces indexed by $j \in J$ as

$$\forall \sigma, \sigma' \in \Sigma, \sigma \left\lfloor T_P^j \right\rfloor \sigma' \iff t_{\langle P,\sigma \rangle}, t_{\langle P,\sigma' \rangle} \in T_P^j \text{ and } \mathit{obs}(t_{\langle P,\sigma \rangle}) = \mathit{obs}(t_{\langle P,\sigma' \rangle}). \quad (4.6)$$

The PER $\left\lfloor T_P^j \right\rfloor$ describes the information released by the parts of the system indexed by j. It is easy to see that $\left\lfloor T_P^j \right\rfloor$ is not necessarily an equivalence relation, but a PER over Σ since it is only defined for traces identified by j. Under this view we can define the information flow property of P as the set of flows

$$[\![P]\!]^{\mathcal{T}} = \{ f_j \mid j \in J. \forall R \in \text{PER}(\Sigma), f_j(R) = R \sqcup \left\lfloor T_P^j \right\rfloor \}. \quad (4.7)$$

For any $j \in J$, the relationship between $\left\lfloor T_P^j \right\rfloor$ and $\lfloor T_P \rfloor$ is the fact that $\left\lfloor T_P^j \right\rfloor$ only talks about an aspect of the information $\lfloor T_P \rfloor$ - the information released by the whole system. Specifically, for any $j \in J$, we have that $\forall \sigma, \sigma' \in \Sigma, \sigma \lfloor T_P \rfloor \sigma'$ iff $\sigma \left\lfloor T_P^j \right\rfloor \sigma'$ or $\mathit{obs}(t_{\langle P,\sigma \rangle}) = \mathit{obs}(t_{\langle P,\sigma' \rangle})$.

4.3.3 Termination Properties

Let us demonstrate the termination properties of the definition of $\lfloor T_P \rfloor$ - the information released by the program P as defined in (4.4). Firstly, let us define a program *loop* $\triangleq$ `while (tt) do skip`, which is an infinite loop. For diverging

programs, such as *loop*, (4.2) ensures that they are distinguished from terminating ones by the insertion of a $\uparrow$ or $\downarrow$ symbol. Consider the following program P_A = `if(`$h = 0$`)then skip else` *loop*, which either terminates or enters an infinite loop. The equivalence relation $\lfloor T_{P_A} \rfloor$ on states relates a pair of states only if they agree on the value of h to be 0 or if $h \neq 0$ in both states. That is, the program reveals the fact that $h = 0$ or not. This result is consistent with the information gained by the attacker which knows the source code of this program and can observe program termination. To see how we arrive at $\lfloor T_{P_A} \rfloor$, first consider the trace through the *then* branch which produces no observation. Hence, for any $\sigma \in \Sigma$ such that $\sigma(h) = 0$, we have that $obs(t_{\langle P_A, \sigma \rangle})$ is the sequence $\langle \downarrow \rangle$. On the other hand, P_A diverges on input states where $h \neq 0$ and hence for all $\sigma \in \Sigma$ such that $\sigma(h) \neq 0$, $obs(t_{\langle P_A, \sigma \rangle}) = \langle \uparrow \rangle$ is the single element sequence. By this we arrive at the information flow of P_A as for all $\sigma, \sigma' \in \Sigma, \sigma \lfloor T_{P_A} \rfloor \sigma'$ iff $\sigma(h) = \sigma'(h) = 0$ or $\sigma(h) \neq 0 \neq \sigma'(h)$.

Consider another program $P_B = P; loop$, where P is an arbitrary *While* program which always terminates. Thus P_B always diverges because of the trailing *loop* program, which always diverges. Since P always terminates, the termination properties of P_B is independent of the choice of secret input and intuitively P_B should release no more information than P. It is easy to see that the information flow of P is preserved in $\lfloor T_{P_B} \rfloor$ since all the traces of P are preserved and the observation of the attacker is only changed by appending $\uparrow$ to the end of the observations made in P. That is for all $\sigma \in \Sigma, obs(t_{\langle P_B, \sigma \rangle}) = \langle obs(t_{\langle P, \sigma \rangle}), \langle \uparrow \rangle \rangle$. Thus, because of the isomorphism between what is observed in P and P_B when started from any given state, we have that for all $\sigma, \sigma' \in \Sigma, obs(t_{\langle P, \sigma \rangle}) = obs(t_{\langle P, \sigma' \rangle})$ iff $obs(t_{\langle P_B, \sigma \rangle}) = obs(t_{\langle P_B, \sigma' \rangle})$. Hence, $\lfloor T_P \rfloor = \lfloor T_{P_B} \rfloor$.

Again, consider the program P_C = *loop*; P where P is a *While* program. Although, P_C has a trailing program P which might ordinarily reveal some information, the leading *loop* program prevents P from being executed and intuitively P_C should thus not reveal any information. The semantic analysis shows this because for all $\sigma \in \Sigma$ the trace of P_C is $\langle P_C, \sigma\rangle \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \langle P_C, \sigma\rangle \xrightarrow{\varepsilon} \cdots$. Hence, for all $\sigma \in \Sigma, \mathit{obs}(t_{\langle P_C,\sigma\rangle}) = \langle\uparrow\rangle$ and therefore $\lfloor T_{P_C} \rfloor$ = *all* is the equivalence relation which relates all states in Σ, demonstrating that the attacker gains no information by executing P_C.

The following lemma shows that for any given *While* program P, $\lfloor T_P \rfloor$ distinguishes states under which P terminates from those under which P diverges. By this partitioning we know that whenever a pair of states is related by $\lfloor T_P \rfloor$, P either terminates under both states or diverges under both states.

Lemma 4.3.2. *Let Σ be the set of all states of the* While *program P and let the set of starting states under which P terminates be $\Sigma_{\Downarrow} = \{\sigma \in \Sigma \mid \sigma' \in \Sigma, \langle P, \sigma\rangle \Downarrow \sigma'\}$ and let the set of starting states under which P diverges be $\Sigma_{\Uparrow} = \Sigma \backslash \Sigma_{\Downarrow}$. Then, for all $\sigma, \sigma' \in \Sigma, \sigma \lfloor T_P \rfloor \sigma'$ implies $\sigma, \sigma' \in \Sigma_{\Downarrow}$ or $\sigma, \sigma' \in \Sigma_{\Uparrow}$.*

Proof. The proof follows easily from the definition of $\lfloor T_P \rfloor$. □

4.3.4 Noninterference

We can state the noninterference property of a program P's secret input in terms of $\lfloor T_P \rfloor$. Let P be a *While* program and let $\mathbf{Var}$ and Σ be its set of variables and states respectively. Furthermore, let $H \subseteq \mathbf{Var}$ be the set of variables containing

secret inputs to P. Then P is noninterfering with respect to H variables iff

$$\forall \sigma \in \mathbf{\Sigma}, \mathit{havocH}([\sigma]_{\lfloor T_P \rfloor}) = [\sigma]_{\lfloor T_P \rfloor}. \tag{4.8}$$

This definition requires every equivalence class of $\lfloor T_P \rfloor$ to be dense with respect to H-values (see definition 3.5.8). That is, in any equivalence class of $\lfloor T_P \rfloor$ every variation of H values is present for any given state, and thus no H value can be distinguished from another.

It is useful to see how (4.8) compares with the standard definition of noninterference. Let $L = \mathbf{Var} \backslash H$ be the set of public variables, and for any $\sigma \in \mathbf{\Sigma}$ let $\sigma_{\downarrow L}$ be the projection of σ to L. The noninterference property of the program P is defined as (see [SM03a]):

$$\forall \sigma, \sigma' \in \mathbf{\Sigma}. \sigma_{\downarrow L} = \sigma'_{\downarrow L} \implies \mathit{obs}(t_{\langle P,\sigma \rangle}) = \mathit{obs}(t_{\langle P,\sigma' \rangle}). \tag{4.9}$$

That is, the attacker's observation (low-view) is invariant whenever the L-inputs are fixed (the requirement for fixed L-inputs is to factor out output variations due to the low inputs). This property is captured in (4.8) because $\forall \sigma, \sigma' \in \mathbf{\Sigma}$

$$\begin{aligned} \mathit{havocH}([\sigma]_{\lfloor T_P \rfloor}) = [\sigma]_{\lfloor T_P \rfloor} \quad &\iff \quad \sigma_{\downarrow L} = \sigma'_{\downarrow L} \implies \sigma' \in [\sigma]_{\lfloor T_P \rfloor} \\ &\text{(by the definition of } \mathit{havocH}(\cdot)) \\ &\iff \quad \sigma_{\downarrow L} = \sigma'_{\downarrow L} \implies \mathit{obs}(t_{\langle P,\sigma \rangle}) = \mathit{obs}(t_{\langle P,\sigma' \rangle}) \\ &\text{(by definition of } \sigma \lfloor T_P \rfloor \sigma') \end{aligned} .$$

Thus, (4.8) is a statement of noninterference of H-inputs to P, with respect to the semantic attacker model.

4.4 Other Semantic Definitions of Information Flow

In this section we shall compare our definition of information flow in the deterministic case with the definition of [SS01] that uses PERs to describe information flow, and the *event-based* definition of [AS07] that has a similar definition to our trace-based observational semantics. We also compare our attacker model with the abstract interpretation-based model of the attacker in [GM04].

4.4.1 The PER Security Model

In [SS01, Sab01] the security property of a program is presented as a transformation of PERs by a function, which represents the (Scott-style) denotation of that program. The approach does not have an explicit notion of interaction, but program output is achieved by assignment to special *low* parts of the memory, which may be observed on program termination. The same effect can be achieved under our framework by inserting *write* statements at the end of the program to print out the values in the *low* portions of the memory, modelling the fact that the (*low*) attacker can observe these values on program termination. Alternatively, we can obtain the attacker model by defining an observational power function, which can observe the *low* part of memory at the end of program execution. We shall take the first approach.

Let the function $f : A \to B$ be the denotation of the program P where A and B are sets. Furthermore, let $R \in \text{PER}(A)$ and $Q \in \text{PER}(B)$ be PERs over A and B respectively. The security property of this program is described as a PER

transformer, written as $(\!|f|\!) : R \multimap Q$, which holds iff

$$\forall a, a' \in A, a\ R\ a' \implies f(a)\ Q\ f(a'). \tag{4.10}$$

Intuitively, this definition says that subject to the constraint R on the input space, the output of the program is indistinguishable under Q. Now let $A = A_1 \times \ldots \times A_n$ be a set product, so that for all $i \in [1, n]$ we have $a_i, a_i' \in A_i$ and $R_i \in \text{PER}(A_i)$. The definition easily extends to tuples of relations $(R_i)_{i \in [1,n]}$, which we write as the relation $R_1 \bullet \ldots \bullet R_n$ and defined as

$$(a_1, \ldots, a_n)\ R_1 \bullet \ldots \bullet R_n\ (a_1', \ldots, a_n') \iff \forall i, a_i\ R_i\ a_i'. \tag{4.11}$$

Noninterference Security Condition

Now suppose that the program states is partitioned into a high-security half (H) and a low-security half (L) so that the set of states is the product $\Sigma = H \times L$ and $\mathit{all} \in \text{PER}(H)$ and $\mathit{id} \in \text{PER}(L)$. A program whose denotation is $f : \Sigma \to \Sigma$ is said to be secure iff

$$(\!|f|\!) : (\mathit{all} \bullet \mathit{id}) \multimap (\mathit{all} \bullet \mathit{id}) \tag{4.12}$$

The statement of (4.12) requires that information does not leak from the high part of program state to the low part since from (4.10) this means that

$$\forall h, h' \in H, l \in L.\ f(h, l)\ \mathit{all} \bullet \mathit{id}\ f(h', l).$$

Thus, if the low part of the input state is fixed, regardless of the initial value of the high part of state the low part of the output state remains fixed, making the value of the low output invariant under any pair of high inputs. This definition also extends to nonterminating programs, but with the additional requirement that the termination property of the program is not influenced by H inputs. Thus (4.12) is a statement of the noninterference for the program P whose denotation is f. By fixing the L inputs of P, we effectively say that the attacker can observe the initial *low* values, thereby factoring out variations in the final *low* values that might be caused by a variation of L inputs.

We can achieve this effect under our setting by inserting *write* statements. Let the set of *low* variables of the program P be $\{l_1, \ldots, l_n\}$ and define a program $P_L \triangleq \texttt{write}\, l_1; \cdots; \texttt{write}\, l_n$, which leaks the L-projection of states. Then we derive another program $P' = P_L; P; P_L$, which reveals the value of the L-portion of the memory to the attacker before the execution of P and on termination. Using (4.4) the information released is given by $\lfloor T_{P'} \rfloor$ defined as

$$\forall \sigma, \sigma' \in \Sigma, \sigma \left\lfloor T_{P'} \right\rfloor \sigma' \quad \Longleftrightarrow \quad \mathit{obs}(t_{\langle P', \sigma \rangle}) = \mathit{obs}(t_{\langle P', \sigma' \rangle})$$

Thus, any pair of states that is related by $\lfloor T_{P'} \rfloor$ has the property that they agree on their L-projections, and P' must either terminate in both states to the same L-values or P' (that is, P) must diverge under *both* states.

By its definition, $\lfloor T_{P'} \rfloor$ *computes* the information flow of the program P about H since it is the least equivalence relation over secrets for which the observation of the attacker is invariant whenever the input states are related by it. In particular, if f is the denotation of the program P, then for any PER $R \bullet \mathit{id}$ such

that $(\!|f|\!) : R \bullet \mathit{id} \rightarrowtriangle \mathit{all} \bullet \mathit{id}$ holds, we have also that $\lfloor T_{P'} \rfloor \sqsubseteq R \bullet \mathit{id}$. Hence, $\lfloor T_{P'} \rfloor$ captures the security property of P.

Partial Information Release

For some PERs $R, Q \in \mathrm{PER}(\Sigma)$ over the states Σ of the program P whose denotation is f, the specification $(\!|f|\!) : R \rightarrowtriangle Q$ may be considered as a statement of the security property of the program P, that is, a policy, which P satisfies. Thus, we say that P satisfies the policy $(\!|\cdot|\!) : R \rightarrowtriangle Q$, whenever $(\!|f|\!) : R \rightarrowtriangle Q$. For any $R, R' \in \mathrm{PER}(\Sigma)$ such that $R \sqsubseteq R'$, it follows by definition that $(\!|f|\!) : R \rightarrowtriangle Q \implies (\!|f|\!) : R' \rightarrowtriangle Q$ since $R \sqsubseteq R'$ means that $\forall \sigma, \sigma' \in \Sigma, \sigma\ R'\ \sigma' \implies \sigma\ R\ \sigma'$. In other words, if P satisfies a stronger policy ($(\!|\cdot|\!) : R \rightarrowtriangle Q$) then it also satisfies a weaker one $(\!|\cdot|\!) : R' \rightarrowtriangle Q$. Given the set of states $\Sigma = H \times L$, the general policy schema of [SS01] is $(\!|\cdot|\!) : R \bullet \mathit{id} \rightarrowtriangle \mathit{all} \bullet \mathit{id}$, which declassifies information R about the secret space H. The *id* part on both sides of $\rightarrowtriangle$ means that the attacker can observe the L portion of states (before and after program execution), and the *all* on the right-hand-side means "*don't care*" since the attacker cannot view the H part of state on termination.

Any deterministic program P trivially satisfies the policy $(\!|\cdot|\!) : \mathit{id} \bullet \mathit{id} \rightarrowtriangle \mathit{all} \bullet \mathit{id}$, the proof of which relies only on the fact that the denotation of P is a function. Thus, *policy refinement* involves finding *more coarse* Rs, that is, those $R \sqsubseteq \mathit{id}$, such that the condition $(\!|f|\!) : R \bullet \mathit{id} \rightarrowtriangle \mathit{all} \bullet \mathit{id}$ is satisfied. Policy refinement is an important problem in language-based security [SM03a, Zda04a, SS07]. By finding the information released about $P' = P_L; P; P_L$ as defined above, which corresponds to the observational capability of the attacker model in the noninterference definition of [SS01], our semantic definition $\lfloor T_{P'} \rfloor$ derives the most refined policy

that P satisfies. Thus, $\lfloor T_{P'} \rfloor$ is the smallest equivalence relation over the set of states, which relates every pair of initial states of P with the same L-input, such that whenever P is executed under this pair, the same L-output is produced on termination. That is, $(\!|f|\!) : \lfloor T_{P'} \rfloor \rightarrow\!\!\!\!\triangleright \mathit{all} \bullet \mathit{id}$ holds for all starting states under which P terminates. Furthermore, for any PER $R \bullet \mathit{id}$ such that $R \bullet \mathit{id} \sqsubset \lfloor T_{P'} \rfloor$, P does not satisfy $(\!|\cdot|\!) : R \bullet \mathit{id} \rightarrow\!\!\!\!\triangleright \mathit{all} \bullet \mathit{id}$. This is easy to see because $R \bullet \mathit{id} \sqsubset \lfloor T_{P'} \rfloor$ means that there exists a pair of states $\sigma, \sigma' \in \Sigma$ which is related by $R \bullet \mathit{id}$ but that is not related by $\lfloor T_{P'} \rfloor$. Since the states σ and σ' are not related by $\lfloor T_{P'} \rfloor$, this either means that P terminates in both states to different values of the L-projection of state, or that P diverges in exactly one of the states. Under these two scenarios the attacker observes a difference and can thus distinguish σ from σ'.

4.4.2 Gradual Release

In [AS07] the notion of *gradual release* is introduced as a policy framework for declassification, encryption and key release. The language setting is the standard core imperative *While* language with an explicit declassification construct for expressing declassification policies for secrets. The language is interactive because assignment to *low* variables can be observed as well as the assignment of declassified expressions. As a result, the operational semantics is event-based and is similar to our labelled-transition system approach. Gradual release is enforced by a standard type system similar to [VSI96] with the additional requirement that declassification may not be performed within conditional statements or loops with a high guard. We highlight how the knowledge representation and the computation of knowledge compares with our approach.

The attacker's knowledge (or rather, uncertainty) is represented as a *set* of states, which represents the possible values of the initial program memory. This is similar to our representation of information with PERs over the initial program states. However, since PERs generalise sets [Hun91a], the information under the gradual release approach can be modelled as a PER. Specifically, for any set $\Sigma \subseteq \mathbf{\Sigma}$, which represents the attacker's knowledge, there is a PER $R_X \in \text{PER}(\mathbf{\Sigma})$, which models this knowledge, defined such that for all $\sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_X \; \sigma'$ iff $\sigma, \sigma' \in X$. This makes it possible to encode each instantiation of gradual-release knowledge as a PER. Furthermore, the monotonicity of knowledge and the gradual nature of information release agrees with the notion of information flow being monotone and extensive. In particular, gradual release requires that the attacker's knowledge may only be refined with time by shrinking the set of states, which represents the knowledge - this is an extensivity property on the PER lattice.

The memory is assumed to be partitioned into two: a *high* (H) part and a low (L) part forming a security lattice $L \sqsubset H$. The operational semantics identifies two types of events $\alpha \in \{\epsilon, \ell\}$, where ϵ is an empty label, which the attacker cannot observe, and ℓ is a *low* event that the attacker can observe. Whenever it is generated, the event ℓ is either the L-projection of state or a special termination event $\downarrow$ signalling program termination. Vectors ($\vec{\ell}$) represent sequences of low events and $\langle P, \sigma \rangle \xrightarrow{\vec{\ell}} {}^*\langle P', \sigma' \rangle$ means that $\langle P', \sigma' \rangle$ is reachable via the execution of P at state σ, generating the sequence $\vec{\ell}$ of low events. Similarly, $\langle P, \sigma \rangle \xrightarrow{\vec{\ell}} {}^*\langle \cdot, \sigma' \rangle$ means that the execution of P at state σ terminates in the state σ' and generates the sequence $\vec{\ell}$.

Let $L(P, \sigma^0_{\downarrow L}) \triangleq \{\vec{\ell} \mid \sigma, \sigma' \in \mathbf{\Sigma}, \sigma^0_{\downarrow L} = \sigma_{\downarrow L}. \langle P, \sigma \rangle \xrightarrow{\vec{\ell}} {}^*\langle \cdot, \sigma' \rangle\}$ and let $\hat{L}(P, \sigma^0_{\downarrow L})$ be the prefix closure of $L(P, \sigma^0_{\downarrow L})$. The *termination-sensitive* knowledge gained

by an attacker of the program P under the observation of $\vec{\ell} \in \hat{L}(P, \sigma^0_{\downarrow L})$ when the low projection of the starting states is σ^0_L is defined as

$$k(P, \sigma^0_{\downarrow L}, \vec{\ell}) \triangleq \{\sigma \mid \sigma, \sigma' \in \mathbf{\Sigma}, \sigma^0_{\downarrow L} = \sigma_{\downarrow L}, \langle P, \sigma\rangle \xrightarrow{\vec{\ell}}{}^{*} \langle P', \sigma'\rangle \vee \langle P, \sigma\rangle \xrightarrow{\vec{\ell}}{}^{*} \langle \cdot, \sigma'\rangle\}. \tag{4.13}$$

We can therefore compute the gradual release knowledge $k(P, \sigma^0_{\downarrow L}, \vec{\ell})$ under our approach by defining an observational power function over traces which maps each partial trace to the sequence $\vec{\ell}$ generated under the definition of [AS07], so that a resulting PER is defined which relates any pair of states $\sigma, \sigma' \in \mathbf{\Sigma}$ iff $\sigma, \sigma' \in k(P, \sigma^0_{\downarrow L}, \vec{\ell})$. However, since the termination-sensitive knowledge can only be computed for a terminating trace, or its prefix, (that is, for $\vec{\ell} \in \hat{L}(P, \sigma^0_{\downarrow L})$) one cannot represent the knowledge gained under nonterminating traces. This excludes a large class of programs, for example, $P; \mathit{loop}$, where P is an arbitrary program. As the analyses in the preceding sections show, program divergence does not pose additional difficulty to our information release definition.

4.4.3 Abstract Noninterference Attacker Model

The abstract noninterference definition of [GM04] introduces attacker models as abstract interpretations, which can observe only properties of data in the concrete domain. The idea is that by weakening the observational power of the attacker on the values of public inputs and outputs of a program, so that the attacker can observe only their *properties*, less restrictive policies, which accept programs that might otherwise be rejected by the standard noninterference definitions can be specified. The concrete domain is partitioned into two sets H and L, which rep-

resent the domain of secret and public values respectively, and state is modelled as tuples in $\Sigma = H \times L$. The attacker is modelled as a pair of abstractions $\langle \eta, \rho \rangle$, where $\eta, \rho \in \mathit{uco}(\mathcal{P}(L))$ are upper closure operators on the powerset lattice of public values ordered by subset inclusion. The closure operators η and ρ model what the attacker can observe about the program's public inputs and outputs respectively. The concrete semantics of the program P is formalised using *angelic denotational semantics*, which associates an input-output function, $[\![P]\!] : \Sigma \to \Sigma$, with P and ignores nontermination. Furthermore, the observation of (public) values occur at the beginning of program execution and on program termination. To slightly simplify the notations, we shall denote the concrete semantics of P as a map $[\![P]\!] : H \times L \to L$, throwing away the H-projection of state on termination, since it is not used.

Our observational model is more general since we place no restriction on the nature of the observational power function, as opposed to the requirement in [GM04], where they must be closure operators. Furthermore, our observational model is not restricted to the observation of values at the beginning and end of program execution. The attacker $\langle \eta, \rho \rangle$ can be obtained under our model by defining an observational power function on traces, such that for any $\sigma, \hat{\sigma} \in \Sigma$, and trace $t_{\langle P,\sigma \rangle} = \langle P, \sigma \rangle \xrightarrow{a} \cdots \xrightarrow{a'} \langle \cdot, \hat{\sigma} \rangle$

$$\mathit{obs}_{\langle \eta,\rho \rangle}(t_{\langle P,\sigma \rangle}) \triangleq \langle \eta(\{\sigma_{\downarrow L}\}), \rho(\{\hat{\sigma}_{\downarrow L}\}) \rangle. \tag{4.14}$$

This definition says that the attacker only observes the η-property of the L-projection of the initial state and the ρ-property of the L-projection of the terminating state of P. Consequently, the information released under this observational model is

the PER $\left\lfloor T_{P_{\langle \eta,\rho\rangle}} \right\rfloor$ over Σ defined such that for any $\sigma, \sigma' \in \Sigma$

$$\sigma \left\lfloor T_{P_{\langle \eta,\rho\rangle}} \right\rfloor \sigma' \iff \mathit{obs}_{\langle \eta,\rho\rangle}(t_{\langle P,\sigma\rangle}) = \mathit{obs}_{\langle \eta,\rho\rangle}(t_{\langle P,\sigma'\rangle}). \tag{4.15}$$

It is thus clear that for any $\sigma, \sigma' \in \Sigma$, such that $t_{\langle P,\sigma\rangle} = \langle P, \sigma\rangle \xrightarrow{a} \cdots \xrightarrow{\hat{a}} \langle \cdot, \hat{\sigma}\rangle$ and $t_{\langle P,\sigma'\rangle} = \langle P, \sigma'\rangle \xrightarrow{a'} \cdots \xrightarrow{\hat{a}'} \langle \cdot, \hat{\sigma}'\rangle$ we have

$$\begin{aligned} \sigma \left\lfloor T_{P_{\langle \eta,\rho\rangle}} \right\rfloor \sigma' \quad &\iff \quad \eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \wedge \rho(\{\hat{\sigma}_{\downarrow L}\}) = \rho(\{\hat{\sigma}'_{\downarrow L}\}) \\ &\implies \quad \eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \implies \rho(\{\hat{\sigma}_{\downarrow L}\}) = \rho(\{\hat{\sigma}'_{\downarrow L}\}). \end{aligned}$$

By this we immediately obtain the narrow abstract noninterference (NANI) definition:

$$[\eta]P[\rho] \iff \forall \sigma, \sigma' \in \Sigma, \eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \implies \rho([\![P]\!](\sigma)) = \rho([\![P]\!](\sigma')).$$

Thus, $\left\lfloor T_{P_{\langle \eta,\rho\rangle}} \right\rfloor$ is the least PER over states for which any pair of states that it relates satisfies NANI in P.

The NANI definition causes what is referred to as "deceptive flows", whereby η-undistinguished public input values cause a variation, which makes P to violate NANI. In order to deal with this problem, abstractions of L values are passed as program parameters and another abstraction $\phi \in \mathit{uco}(\mathcal{P}(H))$ is introduced on the input secret values. This results in the abstract noninterference (ANI) property,

$[\eta]P(\phi \rightsquigarrow [\!]\rho)$, of [GM04], which is defined to hold iff for all $\sigma, \sigma' \in \Sigma$,

$$\eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \implies \rho(\bigcup_{\substack{\sigma'' \in \Sigma, \\ \phi(\sigma''_{\downarrow H}) = \phi(\sigma_{\downarrow H}), \\ \eta(\sigma''_{\downarrow L}) = \eta(\sigma_{\downarrow L})}} \{[\![P]\!](\sigma'')\}) = \rho(\bigcup_{\substack{\sigma'' \in \Sigma, \\ \phi(\sigma''_{\downarrow H}) = \phi(\sigma'_{\downarrow H}), \\ \eta(\sigma''_{\downarrow L}) = \eta(\sigma'_{\downarrow L})}} \{[\![P]\!](\sigma'')\}). \tag{4.16}$$

Let $\sigma \in \Sigma$ be a state, and define the set $\Sigma_\sigma^{\eta,\phi}$ of L-projections of the terminating states of P due to the execution of P from any starting state, which agrees with σ on the η-property of the L-projection and on the ϕ-property of the H-projection to be

$$\Sigma_\sigma^{\eta,\phi} \triangleq \left\{ \hat{\sigma}_{\downarrow L} \;\middle|\; \begin{array}{l} \sigma'' \in \Sigma. \langle P, \sigma'' \rangle \xrightarrow{a} \cdots \xrightarrow{a'} \langle \cdot, \hat{\sigma} \rangle. \\ \eta(\{\sigma''_{\downarrow L}\}) = \eta(\{\sigma_{\downarrow L}\}), \phi(\{\sigma''_{\downarrow H}\}) = \phi(\{\sigma_{\downarrow H}\}). \end{array} \right\} \tag{4.17}$$

Hence, from (4.16), $[\eta]P(\phi \rightsquigarrow [\!]\rho)$ holds iff for all $\sigma, \sigma' \in \Sigma$

$$\eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \implies \rho(\Sigma_\sigma^{\eta,\phi}) = \rho(\Sigma_{\sigma'}^{\eta,\phi}). \tag{4.18}$$

We can obtain this observational model under our framework by defining an observational power function on traces, such that for any $\sigma \in \Sigma$, and terminating trace $t_{\langle P,\sigma \rangle}$

$$obs_{\langle \eta,\phi,\rho \rangle}(t_{\langle P,\sigma \rangle}) \triangleq \left\langle \eta(\{\sigma_{\downarrow L}\}), \rho(\Sigma_\sigma^{\eta,\phi}) \right\rangle \tag{4.19}$$

This definition requires that no public output can be distinguished by ρ for any initial state, which is L-indistinguishable from to σ under η and H-indistinguishable from σ under ϕ. Thus, as usual, the information released under our relational

model is the PER $\lfloor T_{P_{\langle \eta,\phi,\rho\rangle}} \rfloor$ over Σ defined such that for any $\sigma, \sigma' \in \Sigma$

$$\sigma \lfloor T_{P_{\langle \eta,\phi,\rho\rangle}} \rfloor \sigma' \iff \mathit{obs}_{\langle \eta,\phi,\rho\rangle}(t_{\langle P,\sigma\rangle}) = \mathit{obs}_{\langle \eta,\phi,\rho\rangle}(t_{\langle P,\sigma'\rangle}). \tag{4.20}$$

Hence, for all $\sigma, \sigma' \in \Sigma$, we have that

$$\begin{aligned} \sigma \lfloor T_{P_{\langle \eta,\phi,\rho\rangle}} \rfloor \sigma' \quad &\iff \quad \eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \wedge \rho(\Sigma_{\sigma}^{\eta,\phi}) = \rho(\Sigma_{\sigma'}^{\eta,\phi}) \\ &\implies \quad \eta(\{\sigma_{\downarrow L}\}) = \eta(\{\sigma'_{\downarrow L}\}) \implies \rho(\Sigma_{\sigma}^{\eta,\phi}) = \rho(\Sigma_{\sigma'}^{\eta,\phi}). \end{aligned}$$

By this we obtain ANI property $[\eta]P(\phi \rightsquigarrow [\!] \rho)$ and $\lfloor T_{P_{\langle \eta,\phi,\rho\rangle}} \rfloor$ is the least PER over Σ, for which any pair of states that it relates satisfies ANI in P.

4.5 Information Flow in Nondeterministic Systems

In the remainder of this chapter, we shall turn our attention to the application of the operational-semantics based relational model definition of section 4.1 in a nondeterministic language setting. The objective is to demonstrate that, similarly to its use in a deterministic language setting, we can also apply the technique to the analysis of information flow in a nondeterministic language. For this reason we shall add, separately, two simple extensions to the *While* language, which are constructs for *possibilistic nondeterminism* (to obtain *While-ND*) and *probabilistic nondeterminism* (to obtain *While-PND*). The resulting two languages provide us with concrete language-based settings to demonstrate the use of the relational model for the definition of information flow property of a nondeterministic system. We shall use the same semantic attacker's observational model that was introduced earlier for the deterministic *While* language. Later on, in Chapter 5

and Chapter 6, static analyses for the *While* language will be presented, however, the full static analyses of the *While-ND* and *While-PND* are beyond the scope of this thesis.

4.5.1 Possibilistic Nondeterminism

We start by introducing the language *While-ND* (for *While* with NonDeterminism) as a language for (possibilistic) nondeterministic systems. *While-ND* extends the *While* language presented in section 4.2 by adding a nondeterministic construct, $c_1 \,[\!]\, c_2$, which makes an invisible but arbitrary choice in the execution of either the command c_1 or the command c_2. Consequently, the operational semantics of *While-ND* extends that of *While* as shown in Figure 4.4.

$$\langle c_1 \,[\!]\, c_2, \sigma\rangle \xrightarrow{\varepsilon} \langle c_1, \sigma\rangle \qquad \langle c_1 \,[\!]\, c_2, \sigma\rangle \xrightarrow{\varepsilon} \langle c_2, \sigma\rangle$$

Figure 4.4: *Extending* While *with Possibilistic Nondeterminism*

As the semantics shows, nondeterministic choice is an internal action, which is not externally observable to the semantic attacker. In other to address nondeterminism, we extend the definition (4.2) of $obs(\cdot)$ to $obs^*(\cdot)$, which now produces the *set* of all observations that can result from the execution of a *While-ND* program from a given starting state. Let P be a *While-ND* program and let Σ be the set of all states of P. Furthermore, let $t_{\langle P,\sigma\rangle} = \{\langle P, \sigma\rangle \xrightarrow{a_0^i} \langle P_1^i, \sigma_1^i\rangle \xrightarrow{a_1^i} \cdots \mid i \in I_\sigma\}$ be the set of all finite and infinite traces of P resulting from the execution of P at the state $\sigma \in \Sigma$, where I_σ is an index set which identifies all the possible traces of P starting from σ. For any $i \in I_\sigma$, let $t^i_{\langle P,\sigma\rangle} \in t_{\langle P,\sigma\rangle}$ be the i^{th} possible trace of P

starting from σ so that $t_{\langle P,\sigma\rangle} = \{t^i_{\langle P,\sigma\rangle} \mid i \in I_\sigma\}$. Now define $\mathit{obs}^*(\cdot)$, which extends $\mathit{obs}(\cdot)$ (see (4.2)) to set of traces as:

$$\mathit{obs}^*(t_{\langle P,\sigma\rangle}) \triangleq \{\mathit{obs}(t^i_{\langle P,\sigma\rangle}) \mid i \in I_\sigma\}. \tag{4.21}$$

The set of all possible observations arising from the execution of P is given by $\mathcal{V}_P \triangleq \{a \in \mathit{obs}^*(t_{\langle P,\sigma\rangle}) \mid \sigma \in \mathbf{\Sigma}\}$ and the relational model $S_P \subseteq \mathbf{\Sigma} \times \mathcal{V}_P$ of P is given by $\forall \sigma \in \mathbf{\Sigma}, \sigma\ S_P\ a \iff a \in \mathit{obs}^*(t_{\langle P,\sigma\rangle})$. We can now define the possibilistic information released by the program P as follows.

Let $S_P \subseteq \mathbf{\Sigma} \times \mathcal{V}_P$ be the relational model of the nondeterministic While-ND program P such that $\Sigma_{\mathcal{V}_P} \triangleq \{S_P^{-1}(a) \mid a \in \mathcal{V}_P\}$. The information released by P under the possibilistic model is given by

$$[\![P]\!] \triangleq \langle\langle \Sigma_{\mathcal{V}_P} \rangle\rangle. \tag{4.22}$$

This definition is based on the possibilistic information flow definition of section 3.7. It is straightforward to see that definition (4.22) is a natural extension of the deterministic definition (4.4). In particular, if P is a deterministic *While* program then $[\![P]\!]$ is a partitioning of states, such that for any $\sigma \in \mathbf{\Sigma}$, $[\sigma]_{\lfloor T_P \rfloor} \in [\![P]\!]$. Similarly to the definition (4.5) of semantic information flow in the deterministic setting, but now taking the lattice $\mathcal{I}$ to be $\mathit{FAM}(\mathbf{\Sigma})$, we can define the semantic information flow property of a *While-ND* program P as

$$[\![P]\!]^{\mathcal{I}} \triangleq \{f \mid \forall \langle\langle \Sigma_J \rangle\rangle \in \mathit{FAM}(\mathbf{\Sigma}), f(\langle\langle \Sigma_J \rangle\rangle) = \langle\langle \Sigma_J \rangle\rangle \sqcup [\![P]\!]\}. \tag{4.23}$$

This definition describes how the attacker's knowledge on the lattice $\mathit{FAM}(\mathbf{\Sigma})$ is

transformed by the program P. Let us further illustrate how $[\![P]\!]$ captures the information released by the nondeterministic *While-ND* program P.

Suppose the integer (secret) h is a parameter to the nondeterministic program P = `if(`$h = 0$`)then skip` [] *loop* `else skip`. This program may either terminate or loop indefinitely when the secret value h is chosen to be zero. Thus, it is easy to see that the attacker may learn the value of h to be zero when the program fails to terminate. The set of possible observation of P is given by $\mathcal{V}_P = \{\langle\downarrow\rangle, \langle\uparrow\rangle\}$ where $\langle\downarrow\rangle$ corresponds to the observation during the terminating traces and $\langle\uparrow\rangle$ corresponds to the observation of the diverging trace. If we represent the set of program states as $\Sigma = \{(n) | n \in \mathbb{Z}\}$, then the relational model of P, is $S_P \subseteq \Sigma \times \mathcal{V}_P$ whose graph is given by $\{((0), \langle\uparrow\rangle), ((n), \langle\downarrow\rangle) | n \in \mathbb{Z}\}$. Thus, we have the following inverse images: $S_P^{-1}(\langle\uparrow\rangle) = \{(0)\}$ and $S_P^{-1}(\langle\downarrow\rangle) = \Sigma$, and $[\![P]\!] = \{\{(0)\}, \Sigma\}$ reflecting the fact that the attacker can learn when the secret value is zero.

Consider another program P_A = `if(`$h = 0$`)then skip` [] *loop* `else` *loop*. In this case the observation of *termination* reveals to the attacker that the value of the integer secret h is zero. The analysis is similar to that of P, but now we have ***graph***$(S_{P_A}) = \{((0), \langle\downarrow\rangle), ((n), \langle\uparrow\rangle) \mid n \in \mathbb{Z}\}$ and $S_{P_A}^{-1}(\langle\downarrow\rangle) = \{(0)\}$ and $S_{P_A}^{-1}(\uparrow) = \Sigma$. Thus, $[\![P_A]\!] = \{\{(0)\}, \Sigma\}$, and it is intuitive that P_A should release the same information as P.

Consider the program P_B = `if(`$h = 0$`)then skip` [] *loop* `else skip` [] *loop* which may or may not terminate regardless of the chosen value of h. Intuitively, this program should not reveal any information to the attacker as its behaviour is independent of the choice of h. This is confirmed as follows: ***graph***$(S_{P_B}) = \{((n), \langle\downarrow\rangle), ((n), \langle\uparrow\rangle) | n \in \mathbb{Z}\}$. Thus, $S_{P_B}^{-1}(\langle\downarrow\rangle) = S_{P_B}^{-1}(\langle\uparrow\rangle) = \Sigma$. This means that $[\![P_B]\!] = \{\Sigma\}$, confirming the fact that the attacker learns nothing by observing the

execution of P_B.

Now suppose that P_C is a *While-ND* program that always terminates. Similarly to the analysis under deterministic programs, the information flow of P_C is preserved in the program $P' = P_C; \mathit{loop}$. This is easy to see because there is an isomorphism between $\mathcal{V}_{P_C}$ and $\mathcal{V}_{P'}$, which appends $\uparrow$ to all $a \in \mathcal{V}_{P_C}$ such that $\forall \sigma \in \boldsymbol{\Sigma}, \sigma\, S_{P_C}\, \langle a \rangle \iff \sigma\, S_{P'}\, \langle a, \uparrow \rangle$, where $S_{P_C} \subseteq \boldsymbol{\Sigma} \times \mathcal{V}_{P_C}$ and $S_{P'} \subseteq \boldsymbol{\Sigma} \times \mathcal{V}_{P'}$ are respectively the relational models of P_C and P'. Hence, we have $[\![P_C]\!] = [\![P']\!]$.

Finally, let P_D be a *While-ND* program such that $P' = \mathit{loop}; P_D$. Like the deterministic analysis, this program reveals no information since for all $\sigma \in \boldsymbol{\Sigma}, \sigma\, S_{P'} \langle \uparrow \rangle$ holds and hence $[\![P']\!] = \{\boldsymbol{\Sigma}\}$.

Possibilistic Noninterference

We can also state a noninterference property under the possibilistic setting to capture when an attacker cannot learn anything about secret inputs by observing the public output. The basic idea is that the choice of secret values should not affect the information that can be deduced by the attacker.

Let $H \subseteq \mathbf{Var}$ be the set of secret-containing variables, a nondeterministic *While-ND* program P has no possibilistic information flow iff

$$\forall \Sigma \in [\![P]\!],\ \mathit{havoc} H(\Sigma) = \Sigma. \tag{4.24}$$

The link between the nondeterministic definition (4.24) and the deterministic one (4.8) is clear since if P is a deterministic program each $\Sigma \in [\![P]\!]$ corresponds to an equivalence class of $\lfloor T_P \rfloor$. The intuition of (4.24) is also straightforward: if $L = \mathbf{Var} \backslash H$ is the set of public variables, then for any $\sigma \in \Sigma \in [\![P]\!]$ the observations

that led to the knowledge that the secret lies within the set Σ could have been produced by any $\sigma' \in \mathbf{\Sigma}$ such that $\sigma_{\downarrow L} = \sigma'_{\downarrow L}$, since *havoc*$H(\Sigma) = \Sigma$. In other words, that observation is independent of the choice of H-values. Let us illustrate this definition with two more examples.

Consider the program P = `write` $h - h$ [] `write` l, where h and l are respectively the secret and public inputs to P. Intuitively, the attacker cannot learn anything about h since the output of this program is never dependent on the value of h regardless of how the nondeterminism is resolved. This is demonstrated by the analysis of its possibilistic information flow. Suppose $h, l \in \{0, 1\}$ are binary numbers. Now let $\sigma_1 \in \mathbf{\Sigma}$ be the state where $\sigma_1(h) = 0, \sigma_1(l) = 1$ and let $\sigma_3 \in \mathbf{\Sigma}$ be the state where $\sigma_3(h) = 1, \sigma_3(l) = 1$. The graph of the relational model of P is given by *graph*$(S_P) = \{(\sigma, \langle 0, \downarrow\rangle), (\sigma_1, \langle 1, \downarrow\rangle), (\sigma_3, \langle 1, \downarrow\rangle) \mid \sigma \in \mathbf{\Sigma}\}$ where as usual $(\sigma, \langle n, \downarrow\rangle) \in S_P$ means that the output sequence $\langle n, \downarrow\rangle$ can be observed when P is executed at state $\sigma \in \mathbf{\Sigma}$. The inverse images induced by these outputs are $S_P^{-1}(\langle 0, \downarrow\rangle) = \mathbf{\Sigma}$ and $S_P^{-1}(\langle 1, \downarrow\rangle) = \{\sigma_1, \sigma_3\} = \Sigma$. Thus, we have $[\![P]\!] = \{\mathbf{\Sigma}, \Sigma\}$. This program is noninterfering with respect to h-inputs since if we defined $H = \{h\}$, *havoc*$H(\mathbf{\Sigma}) = \mathbf{\Sigma}$ and *havoc*$H(\Sigma) = \Sigma$.

Now consider the program P' = `write` $h - h$ [] `write` l XOR h which can nondeterministically compute the *exclusive OR* of l and h, or print $0 = h - h$. It is easy to see that the h input interferes with the output of this program. In particular, if the attacker observes an output of 1, then the attacker can derive the value of h from the value of l since in that case $h \neq l$. The fact that information flows to the attacker is revealed by the analysis because P' does not satisfy (4.24). Now let $\sigma_1 \in \mathbf{\Sigma}$ be the state where $\sigma_1(h) = 0, \sigma_1(l) = 1$ and let $\sigma_2 \in \mathbf{\Sigma}$ be the state where $\sigma_2(h) = 1, \sigma_2(l) = 0$. We have the graph of the relational model

of P' to be $\mathit{graph}(S_{P'}) = \{(\sigma, \langle 0, \downarrow\rangle), (\sigma_1, \langle 1, \downarrow\rangle), (\sigma_2, \langle 1, \downarrow\rangle) \mid \sigma \in \mathbf{\Sigma}\}$. Thus, $S^{-1}_{P'}(\langle 0, \downarrow\rangle) = \mathbf{\Sigma}$ and $S^{-1}_{P'}(\langle 1, \downarrow\rangle) = \{\sigma_1, \sigma_2\} = \Sigma'$ and $[\![P']\!] = \{\mathbf{\Sigma}, \Sigma'\}$. However, $\mathit{havocH}(\Sigma') = \mathbf{\Sigma} \neq \Sigma'$ shows that information is revealed about h.

4.5.2 Probabilistic Nondeterminism

We now introduce the language *While-PND* (for *While* with Probabilistic Non-Determinism) as a language-based instantiation of probabilistic nondeterministic systems. *While-PND* extends the *While* language presented in section 4.2 by adding a probabilistic construct, $c_1 \,[]_p\, c_2$, which executes c_1 with a probability of p and c_2 with a probability of $1 - p$. In the constructor $[]_p$, we assume that $0 < p < 1$. In order to model how probabilistic choices affect the execution of programs, we extend command configurations with probabilities such that $\langle p, c, \sigma\rangle$ means that the command configuration $\langle c, \sigma\rangle$ is to be executed with a probability of p. We shall call $\langle p, c, \sigma\rangle$ a *probabilistic command configuration*. Similarly to terminal command configurations, $\langle p, \cdot, \sigma\rangle$ represent a *terminal probabilistic command configuration* where there are no more commands to execute. The operational semantics of *While-PND* is shown in Figure 4.5. It shows that the probabilistic choice itself is not observable, although we assume that the attacker knows the program code and therefore knows the probability of making any of the two choices.

Let P be a *While-PND* program and let $\mathbf{\Sigma}$ be the set of states of P. Furthermore, for any $\sigma \in \mathbf{\Sigma}$ we denote by I_σ an index set identifying the set of all finite and infinite traces of P starting at state σ. Since P will be executed for any given starting state, we take the starting probability of P to be 1. The set of all traces of

$$\langle p, \texttt{skip}, \sigma\rangle \xrightarrow{\varepsilon} \langle p, \cdot, \sigma\rangle \qquad \langle p, z := e, \sigma\rangle \xrightarrow{\varepsilon} \langle p, \cdot, \sigma[z \mapsto \sigma(e)]\rangle$$

$$\langle p, \texttt{write}\ e, \sigma\rangle \xrightarrow{out(\sigma(e))} \langle p, \cdot, \sigma\rangle$$

$$\frac{\langle p, c_1, \sigma\rangle \xrightarrow{\alpha} \langle p', c_1', \sigma'\rangle}{\langle p, c_1; c_2, \sigma\rangle \xrightarrow{\alpha} \langle p', c_1'; c_2, \sigma'\rangle} \qquad \frac{\langle p, c_1, \sigma\rangle \xrightarrow{\alpha} \langle p', \cdot, \sigma'\rangle}{\langle p, c_1; c_2, \sigma\rangle \xrightarrow{\alpha} \langle p', c_2, \sigma'\rangle}$$

$$\frac{\langle b, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{tt}, \sigma\rangle \quad \langle p, c_1, \sigma\rangle \xrightarrow{\alpha} \langle p', c_1', \sigma'\rangle}{\langle p, \texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma\rangle \xrightarrow{\alpha} \langle p', c_1', \sigma'\rangle}$$

$$\frac{\langle b, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{ff}, \sigma\rangle \quad \langle p, c_2, \sigma\rangle \xrightarrow{\alpha} \langle p', c_2', \sigma'\rangle}{\langle p, \texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma\rangle \xrightarrow{\alpha} \langle p', c_2', \sigma'\rangle}$$

$$\frac{\langle b, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{tt}, \sigma\rangle \quad \langle p, c, \sigma\rangle \xrightarrow{\alpha} \langle p', c', \sigma'\rangle}{\langle p, \texttt{while}\ (b)\ \texttt{do}\ c, \sigma\rangle \xrightarrow{\alpha} \langle p', c'; \texttt{while}\ (b)\ \texttt{do}\ c, \sigma'\rangle}$$

$$\frac{\langle b, \sigma\rangle \xrightarrow{\varepsilon} \langle \mathbf{ff}, \sigma\rangle}{\langle p, \texttt{while}\ (b)\ \texttt{do}\ c, \sigma\rangle \xrightarrow{\varepsilon} \langle p, \cdot, \sigma\rangle}$$

$$\frac{0 < p' < 1}{\langle p, c_1 \,[\!]_{p'}\, c_2, \sigma\rangle \xrightarrow{\varepsilon} \langle p \times p', c_1, \sigma\rangle} \qquad \frac{0 < p' < 1}{\langle p, c_1 \,[\!]_{p'}\, c_2, \sigma\rangle \xrightarrow{\varepsilon} \langle p \times (1 - p'), c_2, \sigma\rangle}$$

Figure 4.5: *The Operational Semantics of* While-PND

P starting at σ is given by

$$t_{\langle P,\sigma\rangle} \triangleq \{\langle 1, P, \sigma\rangle \xrightarrow{a_0^i} \langle p_1^i, P_1^i, \sigma_1^i\rangle \xrightarrow{a_1^i} \cdots \mid i \in I_\sigma\}. \tag{4.25}$$

Similarly to the *While-ND* case, the index set I_σ identifies the probabilistic choices that are made during the execution of the *While-PND* program P when executed from the state σ. We shall write $t^i_{\langle P,\sigma\rangle} \in t_{\langle P,\sigma\rangle}$ to represent the i^{th} possible trace of P starting at the state σ for some $i \in I_\sigma$. The set of all traces of the *While-PND* program P is given by

$$T_P \triangleq \bigcup_{\sigma\in\mathbf{\Sigma}} t_{\langle P,\sigma\rangle} \tag{4.26}$$

In the probabilistic case, we require that the input space $\mathbf{\Sigma}$ and the output space $\mathcal{V}$ both be finite. Furthermore, we shall consider only programs where the set T_P of all traces of P is finite. We do not consider in this thesis the full generality of an infinite set of probabilistic traces. This, for example, rules out programs, which are infinitely branching on $[\![]\!]_p$.

As usual, using the semantic attacker's observational model (see (4.2)), and similarly to the definition in the *While-ND* case (see 4.21), the extension of $\mathit{obs}(\cdot)$ to set of probabilistic traces is $\mathit{obs}^*(t_{\langle P,\sigma\rangle}) = \{\mathit{obs}(t) \mid t \in t_{\langle P,\sigma\rangle}\}$. Define $\mathcal{V}_P = \{\mathit{obs}(t) \mid t \in T_P\}$ to be the set of all observations that the attacker can make about the *While-PND* program P. The relational model $S_P \subseteq \mathbf{\Sigma} \times \mathcal{V}_P$ of P is, as usual, thus defined as $\forall \sigma \in \mathbf{\Sigma}, \sigma\ S_P\ v \iff v \in \mathit{obs}^*(t_{\langle P,\sigma\rangle})$.

Furthermore, for any $\sigma \in \mathbf{\Sigma}$ and $i \in I_\sigma$, define the limiting probability $\omega(t^i_{\langle P,\sigma\rangle}) = p^i_k$ of the trace $t^i_{\langle P,\sigma\rangle}$ of P, to be the smallest probability, due to some probabilistic

command configuration $\langle p_k^i, P_k^i, \sigma_k^i \rangle$ in $t^i_{\langle P,\sigma \rangle} = \langle 1, P, \sigma \rangle \xrightarrow{a_0^i} \cdots \xrightarrow{a_{k-1}^i} \langle p_k^i, P_k^i, \sigma_k^i \rangle \xrightarrow{a_k^i} \cdots \xrightarrow{a_{j-1}^i} \langle p_j^i, P_j^i, \sigma_j^i \rangle \xrightarrow{a_j^i} \cdots$, such that for all probabilistic command configuration $\langle p_j^i, P_j^i, \sigma_j^i \rangle$ in $t^i_{\langle P,\sigma \rangle}$, where $j > k$, if it exists, we have $p_j^i = p_k^i$. The limiting probability $\omega(t^i_{\langle P,\sigma \rangle})$ is the probability of executing $t^i_{\langle P,\sigma \rangle} \in t_{\langle P,\sigma \rangle}$ whenever σ is chosen. From the operational semantics, the limiting probability exists uniquely in the closed real interval $[0, 1]$, ordered by $\leq$, and is the smallest probability

$$\omega(t^i_{\langle P,\sigma \rangle}) \triangleq \mathit{min}\{p_j^i \mid \langle p_j^i, P_j^i, \sigma_j^i \rangle \in t^i_{\langle P,\sigma \rangle}\}. \tag{4.27}$$

In (4.27), the notation $\langle p_j^i, P_j^i, \sigma_j^i \rangle \in t^i_{\langle P,\sigma \rangle}$ denotes the existence of the probabilistic command configuration $\langle p_j^i, P_j^i, \sigma_j^i \rangle$ in the trace $t^i_{\langle P,\sigma \rangle}$.

In order to compute the quantitative information release to an attacker of a *While-PND* program P we need a probability measure $\mu \in \mathscr{M}(\mathbf{\Sigma})$, which assigns probabilities to the selection of input states $\sigma \in \mathbf{\Sigma}$ of P. We assume that the attacker knows the measure μ so that it represents the attacker's uncertainty about the choice of inputs. From μ we can compute the joint probability, $\hat{\mu} \in \mathscr{M}(\mathbf{\Sigma} \times \mathcal{V}_P)$, which represents the probability of joint occurrence of input-output pairs in the relational model of P. As usual, following the standard convention, for any $\sigma \in \mathbf{\Sigma}$ and $v \in \mathcal{V}_P$, $\hat{\mu}(\sigma \mid v)$ and $\hat{\mu}(v \mid \sigma)$ are conditional probabilities, and we shall denote the marginal probability for the occurrence of v by $\mu'(v)$. These measures are computed using the operational semantics of P and the given probabilities μ over the input space as follows.

Let $T^{\langle \sigma, v \rangle} = \{t \in t_{\langle P,\sigma \rangle} \mid \mathit{obs}(t) = v\}$ be the set of traces of P starting from σ, which produce the output observation v. The conditional probability of producing

the output v when the input state σ is chosen is given by

$$\hat{\mu}(v \mid \sigma) \triangleq \sum_{t \in T\langle \sigma, v \rangle} \omega(t). \tag{4.28}$$

The probability $\hat{\mu}(v \mid \sigma)$ is the sum of the probabilities of traces that produce the output v when P is executed at state σ. Thus, the marginal probability of producing the output v by P is $\mu'(v) = \sum_{\sigma \in \mathbf{\Sigma}} \mu(\sigma) \times \hat{\mu}(v \mid \sigma)$. Using these, we can compute the attacker's uncertainty about the input state $\sigma \in \mathbf{\Sigma}$ given the observation of output $v \in \mathcal{V}_P$ as the conditional probability

$$\hat{\mu}(\sigma \mid v) = \frac{\hat{\mu}(v \mid \sigma) \times \mu(\sigma)}{\mu'(v)}$$

Using Definition 3.8.5 we can now compute the quantitative information released by P. Let us illustrate with some example analyses.

Sample Analyses

Consider the program $P =$ `if(`$h = 7$`)then skip` $[]_{.5}$ *loop* `else skip` and suppose h is an integer secret chosen uniformly from the set $\{0, 1, \ldots, 15\}$ of integers. Hence we can model the set of states as $\mathbf{\Sigma} = \{(n) \mid 0 \leq n \leq 15\}$, where for any input states $\sigma \in \mathbf{\Sigma}$ of P, $\mu(\sigma) = \frac{1}{16}$. Let $\sigma_n \in \mathbf{\Sigma}$ be the state of P where $h = n$. We have two possible traces of P for σ_7, both of which are chosen by the probabilistic constructor $[]_{.5}$ with equal probability of 0.5. One of these traces terminates, but the other does not. Let us label the traces as $t^1_{\langle P, \sigma_7 \rangle}$ for the terminating trace and $t^2_{\langle P, \sigma_7 \rangle}$ for the nonterminating trace. Then we have $t^1_{\langle P, \sigma_7 \rangle} = \langle 1, P, \sigma_7 \rangle \xrightarrow{\varepsilon} \langle .5, \cdot, \sigma_7 \rangle$ and $t^2_{\langle P, \sigma_7 \rangle} = \langle 1, P, \sigma_7 \rangle \xrightarrow{\varepsilon} \langle .5, \texttt{skip}, \sigma_7 \rangle \xrightarrow{\varepsilon} \langle .5, \texttt{skip}, \sigma_7 \rangle \xrightarrow{\varepsilon} \cdots$. Hence,

$obs(t^1_{\langle P,\sigma_7\rangle}) = \langle\downarrow\rangle$, whereas $obs(t^2_{\langle P,\sigma_7\rangle}) = \langle\uparrow\rangle$ - being an infinite trace with no output. Furthermore, $\omega(t^1_{\langle P,\sigma_7\rangle}) = .5$ and $\omega(t^2_{\langle P,\sigma_7\rangle}) = .5$. However, for any $n \neq 7$ there is only one possible trace for σ_n, which is, $t^1_{\langle P,\sigma_n\rangle} = \langle 1, P, \sigma_n\rangle \xrightarrow{\varepsilon} \langle 1, \cdot, \sigma_n\rangle$ and we have $obs(t^1_{\langle P,\sigma_n\rangle}) = \langle\downarrow\rangle$ and $\omega(t^1_{\langle P,\sigma_n\rangle}) = 1$. Hence $\hat{\mu}(\langle\downarrow\rangle \mid \sigma_7) = \frac{1}{2}$ (the probability of observing $\langle\downarrow\rangle$ given that σ_7 was chosen as the input to P) and $\hat{\mu}(\langle\uparrow\rangle \mid \sigma_7) = \frac{1}{2}$, whereas for any other $n \neq 7$ we have that $\hat{\mu}(\langle\downarrow\rangle \mid \sigma_n) = 1$ and $\hat{\mu}(\langle\uparrow\rangle \mid \sigma_n) = 0$. We can thus compute the marginal probabilities of the outputs as $\mu'(\langle\downarrow\rangle) = \frac{1}{16} \times \frac{1}{2} + 15 \times \frac{1}{16} \times 1 = \frac{31}{32}$ and $\mu'(\langle\uparrow\rangle) = \frac{1}{16} \times \frac{1}{2} = \frac{1}{32}$. Thus, for σ_7, we have the conditional probabilities

$$\hat{\mu}(\sigma_7 \mid \langle\downarrow\rangle) = \frac{\hat{\mu}(\langle\downarrow\rangle \mid \sigma_7) \times \mu(\sigma_7)}{\mu'(\langle\downarrow\rangle)} = \frac{1}{31}, \text{ and, } \hat{\mu}(\sigma_7 \mid \langle\uparrow\rangle) = 1.$$

Similarly, for any $\sigma_n \in \mathbf{\Sigma}\backslash\{\sigma_7\}$ we have $\hat{\mu}(\sigma_n \mid \langle\downarrow\rangle) = \frac{2}{31}$ and $\hat{\mu}(\sigma_n \mid \langle\uparrow\rangle) = 0$. Using Definition 3.8.5, the information released by P is given by

$$I_{\langle P,\mu\rangle} = \mathcal{H}(\mu) - \sum_{v \in \mathcal{V}_P} \mu'(v)\mathcal{H}(\mu_v)$$

where for any $\sigma \in \mathbf{\Sigma}, \mu_v(\sigma) = \hat{\mu}(\sigma \mid v)$ is the probability that σ was selected as the input to P given the observation of v, and $\mathcal{H}(\mu)$ and $\mathcal{H}(\mu_v)$ are respectively the entropies of the random variables induced by the probability measures μ and μ_v over $\mathbf{\Sigma}$, which respectively describe the attacker's uncertainty about the input state before and after observing P's execution. Therefore, the information (in *bits*) about h released by P is

$$\begin{aligned} I_{\langle P,\mu\rangle} &= \log(16) - \left(\tfrac{31}{32}\left(\tfrac{1}{31}\log(31) + 15 \times \tfrac{2}{31}\log\left(\tfrac{31}{2}\right)\right) + \tfrac{1}{32} \times 0\right) \\ &\approx 0.1381 \end{aligned}$$

This measure is the average uncertainty lost about the input space, that is, the information gained by the attacker. The measure $\mathcal{H}(\mu) = \log(16)$ is the initial uncertainty that the attacker has about the input space, but after observing divergence in one out of 32 runs, whereby the attacker can identify the input on that run, the average uncertainty that remains about the input space is the measure $\sum_{v \in \mathcal{V}_P} \mu'(v)\mathcal{H}(\mu_v)$, and its difference from $\mathcal{H}(\mu)$ models the information released to the attacker about the inputs to P.

Now consider the program P_A = `if(`$h = 7$`)then skip` $[]_{.5}$ *loop* `else` *loop*, which is similar to P but now swaps `skip` and *loop*. The similarity is that, instead of revealing the input on divergence, termination now signals that the input h to P_A is 7. Intuitively, we should get the same result for P_A that we got for P, because P_A is simply a swapping of probabilities (which the entropy measure is not sensitive to) so that P_A terminates once every 32 times, as opposed to P, which symmetrically diverges once every 32 times. Making the same assumptions and using the similar notations as in the last example, the analysis of P_A is similar to the analysis of P. We now have the following probabilities: $\hat{\mu}(\langle\downarrow\rangle \mid \sigma_7) = \hat{\mu}(\langle\uparrow\rangle \mid \sigma_7) = \frac{1}{2}$ and for all $\sigma \in \mathbf{\Sigma}\backslash\{\sigma_7\}, \hat{\mu}(\langle\uparrow\rangle \mid \sigma) = 1$ and $\hat{\mu}(\langle\downarrow\rangle \mid \sigma) = 0$. Also, $\mu'(\langle\uparrow\rangle) = \frac{31}{32}$ and $\mu'(\langle\downarrow\rangle) = \frac{1}{32}$, and for any $\sigma \in \mathbf{\Sigma}$, $\hat{\mu}(\sigma \mid \langle\downarrow\rangle) = 1$ if $\sigma = \sigma_7$ and $\hat{\mu}(\sigma \mid \langle\downarrow\rangle) = 0$ otherwise. Finally, for any $\sigma \in \mathbf{\Sigma}$, $\hat{\mu}(\sigma \mid \langle\uparrow\rangle) = \frac{1}{31}$ if $\sigma = \sigma_7$ and $\hat{\mu}(\sigma \mid \langle\downarrow\rangle) = \frac{2}{31}$ otherwise. Thus, by applying Definition 3.8.5, the information released by P_A is given by $I_{\langle P_A,\mu\rangle} = I_{\langle P,\mu\rangle}$. The identical result is not surprising because the analysis of P_A is merely a permutation of probabilities due to the observation of $\langle\uparrow\rangle$ and $\langle\downarrow\rangle$ in the analysis of P.

The quantitative information flow obtained when *loop* is appended or prepended to *While-PND* programs is similar in nature to the information flow obtained un-

der the deterministic *While* and the possibilistic nondeterministic *While-ND* under the relational model. Specifically, for any *While-PND* program P that always terminates, the quantitative information flow of P is preserved in the program P_B = P; *loop* since P_B only appends $\uparrow$ to the observations of P without changing the probabilities. Similarly, for any *While-PND* program P, it is easy to see that the probabilistic information flow of P_C = *loop*; P is 0 bits, since the P subprogram is never executed and the output observation $\langle\uparrow\rangle$ of P_C is *independent* of the choice of the input values to P_C.

Summary In this chapter we have studied semantic definitions of information flow in various language-based settings. These definitions demonstrate the use of the input-output relational model and information representations introduced in Chapter 3. The definitions of information flow were given relative to an attacker's observational model, the semantic attacker, to illustrate the definition of attacker models, and to demonstrate the development of information flow analyses that are parametrised by a chosen attacker's observational power. The semantic analyses developed demonstrate that the relational model copes well with issues of nontermination. In the next chapter, we shall develop a static information flow analysis technique for the *While* language, which is based on the observational model of the semantic attacker.

Chapter 5

Information Flow Analysis of *While* Programs

In this chapter we present a static analysis of information flow for *While* programs, using PERs on the set of program states to represent information. The semantics-based static analysis, which is flow-sensitive and termination-sensitive, is shown to be correct with respect to the *semantic attacker* model introduced in the previous chapter. This attacker is not only able to observe program outputs as prescribed by the operational semantics, but can also determine whether the program terminates or not. We shall start by presenting examples to motivate some aspects of information flow analysis captured by the static analysis.

5.1 Motivating Examples

In the following examples the variable h (for *high*) is named to suggest that it might contain sensitive input, and variable l (for *low*) is named to suggest that

it initially contains public input. The variable z is generally used for temporary storage of intermediate computation, and is not a parameter to the program. We shall sometimes present two or more programs within the same figure (usually to compare the programs), separated by a vertical line. When there are two programs, we refer to the leftmost program as the left-hand-side (LHS) program and the other as the right-hand-side (RHS) program.

Example 5.1.1 (*Explicit Information Flow*) The two programs shown in Figure 5.1.1 below both reveal exactly the same information, namely, the value of the secret input h. The RHS program first assigns the secret value of h to another intermediate variable z before printing it to the output. Although the two program implementations are different, the information flow analysis of these programs should produce the same result since they both reveal the same information. In the security literature, information flow from h to z through the assignment statement is called *explicit* information flow [Den76].

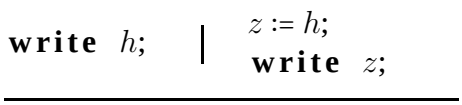


Figure 5.1: *Explicit Information Flow*

Example 5.1.2 (*Implicit Information Flow*) This example demonstrates the idea of *implicit* or *indirect* information flow [Den76]. Assume that both h and z in the programs of Figure 5.2 are boolean-typed variables. The LHS program is a classic example of how information can be propagated implicitly. Although h is not directly assigned to z, the value of h can be learnt indirectly via z because the value assigned to z is determined by the value of h. This information flow from h

to z due to branching is called *control dependence* [Muc97] in compiler analysis. The RHS program has exactly the same information flow, but it is propagated explicitly by the assignment. However, the distinction between implicit and explicit information flow is immaterial in these programs because the binary value of the secret h is revealed in both cases.

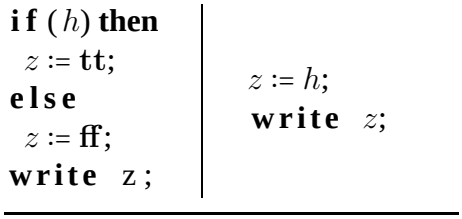


Figure 5.2: *Implicit Flow and a binary-valued Explicit Flow*

Example 5.1.3 (*Implicit Flow Capacity*) Although implicit information flow channels are usually low-capacity transmission channels, but when well-used, implicit information flow can be as potent as the explicit copying of data. Assuming that h is a natural number, the program of Figure 5.3 (inefficiently, through a linear search) copies the secret input to z purely by implicit means.

```
z := 0;
while (h ≠ z) do
        z := z + 1;
write z;
```

Figure 5.3: *Implicit Flows could be as dangerous as Explicit Copying*

Example 5.1.4 (*Implicit Flow - due to assignment or lack of it*) This example demonstrates why not all implicit flows can be detected by only considering one

program trace (or control-flow path) at a time. In Figure 5.4, both programs release the same information about the secret h. More specifically, an output of 0 in both programs indicates that the value of h is 1, whereas an output of 1 indicates that the value of h is *not* 1. It is clear that the particular output value is not important, what matters is the fact that the attacker can determine which branch of the conditional statement has been executed by observing the output values.

```
if (h=1) then          |  z:=0;
 z:=0;                 |  if (h=1) then
else                   |    skip;
 z:=1;                 |  else
write z;               |   z:=1;
                       |  write z;
```

Figure 5.4: *Assignments on all program paths must be considered*

Now suppose that we have a *trace-based monitor* which determines *at runtime* whether information may flow implicitly to a variable by checking whether or not that variable is assigned within the branch of a conditional statement whose guard is predicated on an expression involving secret values. In the LHS program, this monitor will be able to detect the implicit information flow to z from the secret variable h because z is assigned within both branches of the *if* statement. In the RHS program however, this monitor will fail to detect that information flows from h to z on *the trace* through the *then* branch, because as far as the runtime monitor can tell on this trace, z is not assigned within a conditional statement. Information, however, flows to z because it is assigned in at least one branch of the conditional statement. This is a well known problem concerning the use of runtime monitors for the enforcement of information flow security [Vol99b, McL94, SM03a].

The runtime monitor fails to detect the flow because secure information flow is a property of *all* control-flow paths. A runtime-based enforcement monitor for information flow introduced in [GBJS06] uses the result of a static analysis in the enforcement monitor in order to deal with this problem. This highlights the importance of static analysis as a useful technique for information flow protection.

Example 5.1.5 (*Information Flow - due to program interaction or lack of it*) Many useful programs are *interactive* in nature, receiving inputs from the user and generating output as they execute. The *While* language studied in this thesis produces outputs through the *write* construct, which raises the possibility of implicit information flow when output takes place in conditional statements, and also explicit information flow when the attacker observes the result of the evaluation of an expression whose value depends on secrets. The example of Figure 5.5 demonstrates implicit information flow via output interactions.

if ($h = 1$) **then**
  write 1;
else
  write 2;
	if ($h = 1$) **then**
	  write 1;
	else
	  skip;

Figure 5.5: *Program Output, or the lack of it, on all control-flow paths must be considered*

In the LHS program of Figure 5.5, an output of 1 indicates that the value of h is 1, whereas an output of 2 reveals that h is not 1. The RHS program only produces output when the *then* branch is executed, however this single output or the lack of it is sufficient to reveal the same information as in the LHS program to the attacker: an output of 1 in the RHS program reveals h to be 1, and *no output* reveals that h is not 1. This implicit information flow due to lack of output is

similar to the problem of lack of assignment in the RHS program of Figure 5.4.

Example 5.1.6 (*Flow-Sensitivity and Semantics*) This example, adapted from [JL00], demonstrates flow-sensitivity and semantics-related aspects of information flow analyses. Flow-sensitive and semantics-based analyses are usually more precise than flow-insensitive static approximations of information flow, which are commonly used in security type-systems for noninterference such as [VSI96, VS97]. In flow-insensitive security type systems, which are common in language-based security, a variable must be typed as secret whenever it is assigned a value that may be dependent on a secret at any point within the program. While this is the case in the three programs of Figure 5.6, the attacker cannot learn anything about (the secret) h by observing the program output. Flow-sensitive and semantics-based analyses detect this. In the first program, the secret value in z is over-written before it can be used in the output: flow-sensitivity. Similarly, in the second program, the secret value in h is lost before it is assigned to z - which in turn is written to the output. In the third program, although the value of z is computed as a function of h, it is however clear that the final value of z before it is released is the constant 0, which means that the value of the program output is independent of the secret h. Modern type-based analyses such as [HS06], and dependency analyses such as [AB04], which are *flow-sensitive* will detect the security of the first two programs. The analysis that we shall present in this chapter is flow-sensitive and, being semantics-based, will detect that all the three program are safe. In particular, the analysis of the third program demonstrates the semantic properties of our analysis.

Example 5.1.7 (*Dead Code*) This example highlights another aspect of semantics-

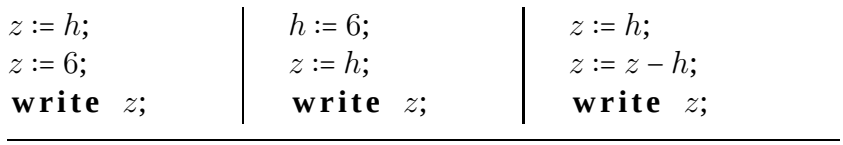


Figure 5.6: *Accuracy: Semantic Analysis against Static Typing*

based analyses, which makes them more accurate than the traditional static-typing systems for information flow. In the LHS program of Figure 5.7, the secret h is assigned to z in the *then* branch, which suggests that the output value might be dependent on the secret input. However, since this branch will never be executed, the output of this program is the constant 0, and hence no information is revealed about h. In the RHS program, the preceding nonterminating loop prevents the execution of the *write* statement, which makes the program safe because the dangerous part will never be executed. The correct information flow in these examples, and similar *dead code* situations, are detected by our analysis.

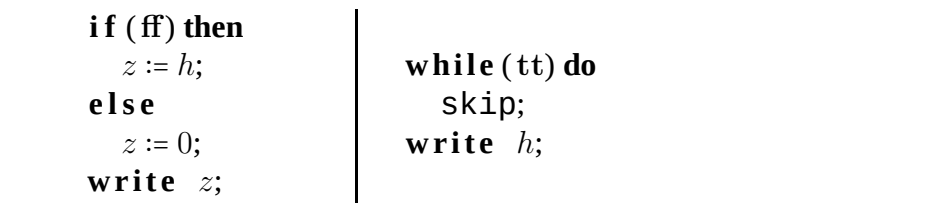


Figure 5.7: *Dead Code and Information Flow*

Example 5.1.8 (*Termination-Sensitivity*) The modelling of information flow due to nontermination (termination channels) or in the presence of nontermination is important because nontermination, especially when combined with program outputs, can be used to reveal arbitrary information about secrets [AHSS08].

On one hand, although the LHS program of Figure 5.8 does not produce any

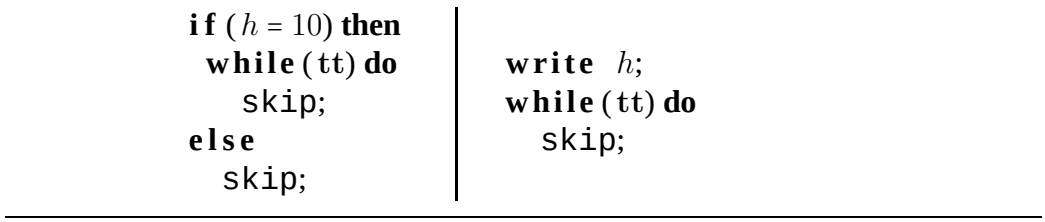


Figure 5.8: *Information Flow in the Presence of Nontermination*

output by a *write* statement, it however has a termination channel via which information about the secret h is transmitted. The termination of this program reveals that $h \neq 10$, whereas nontermination reveals that h is 10. The RHS program, on the other hand, reveals the secret h before diverging. Termination-insensitive analyses, which support program outputs like [GBJS06, AS07], do not model information flow under diverging programs such as the one on the RHS of Figure 5.8. Our analysis can deal with these and similar cases.

Example 5.1.9 (*Disjunctive Information Flow Analysis*) Since a program trace traverses a given control-flow path at a time during the program's execution, it is reasonable to analyse information flow in a way that models this property of program execution. This observation leads to a more accurate analysis of certain *disjunctive information flow*. For example, we might require that at most one of two secrets h_1 or h_2 may be learnt, that is, never *both* at the same time, during the program run. This pattern is akin to the disjunctive information flow property mentioned in [SS05]. We have shown how one may represent disjunctive information with PERs in section 3.5.5. This example demonstrates the usefulness of such a notion.

Consider the programs of Figure 5.9, and assume that l is a public boolean value (possibly chosen by an attacker). It is clear that the first program on the LHS

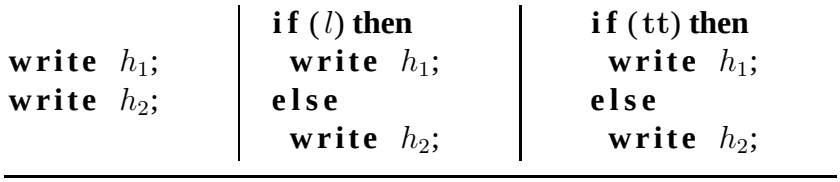


Figure 5.9: *Disjunctive Information Flow*

violates the required disjunctive policy about the release of h_1 and h_2, whereas the other two programs do not. In the second program in the middle, the attacker can learn at most one of either h_1 or h_2 during a run regardless of how l is chosen in this program. One may think of l as the release key to choose between the release of either h_1 or h_2. The third, rightmost, program satisfies the desired policy by definition because it never reveals h_2. Our analysis, by the definition of disjunctive information in PERs, detects the disjunctive properties of information flow about h_1 and h_2 in these three programs.

5.2 Information Flow Analysis with PERs

We have presented, in Chapter 3, the lattice of PERs over a set as a representation of information and we have shown, in Chapter 4, how the semantic information flow property of a *While* program may be defined by using PERs over its set of states. In the remainder of this chapter, we shall develop a static analysis calculus for deriving the program information flow, which uses the lattice of PERs over program states as the representation of information.

In the following, the sets Σ and $\mathbf{Var}$ are taken to be the set of all states and the set of all variables respectively of a *given program* P, which will hopefully be clear from the context. We also assume that the set $\mathbf{Var}$ of the program vari-

ables is partitioned into two; namely, the set $\mathbf{IVar}$ of variables used for program *inputs*, that is, the program's formal parameters, and the set $\mathbf{TVar} \triangleq \mathbf{Var} \backslash \mathbf{IVar}$ of variables, which are not formal parameters but are used for *temporary* intermediate storage during the program's computations. The semantic status of the $\mathbf{IVar}$-$\mathbf{TVar}$ partitioning of variables is established by requiring a property, which preserves the determinism of *While* with respect to the program inputs, namely, that the behaviour of a *While* program must be invariant under fixed $\mathbf{IVar}$-values.

Definition 5.2.1. *Let P be a* While *program. We say that P is* properly-initialised *iff*

1. *for any $\sigma, \sigma' \in \mathbf{\Sigma}$ such that $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}}$, where $\langle P, \sigma \rangle = \langle P, \sigma \rangle \xrightarrow{a_0} \langle P_1, \sigma_1 \rangle \xrightarrow{a_1} \cdots$ and $\langle P, \sigma' \rangle = \langle P, \sigma' \rangle \xrightarrow{a'_0} \langle P'_1, \sigma'_1 \rangle \xrightarrow{a'_1} \cdots$ then for all i, $a_i = a'_i$ and*

2. *there is no assignment to $\mathbf{IVar}$ variables in P.*

We shall consider only properly-initialised programs in the static analysis developed in this chapter. The Definition 5.2.1 of a properly-initialised program P requires that the observable operational behaviour of this program must be fixed from one run to another under fixed input ($\mathbf{IVar}$) values. A way to ensure this property is to initialise properly all $\mathbf{TVar}$ variables before use. A variable is said to be *properly-initialised* if it is defined as a function of the initial values of $\mathbf{IVar}$ variables. This is not an unusual requirement because the initialisation of a variable before use is standard programming practice. The property is required during the static analysis of programs because determinism of the program with respect to its formal parameters is assumed. At the beginning of program execution, the $\mathbf{IVar}$ variables are properly-initialised by definition. But we also require,

to simplify the analysis, that **IVar** variables are not used on the left-hand-side of assignment statements. This is not a serious restriction because assignments to **IVar** variables can be handled by a systematic variable-renaming scheme. The no-assignment-to-**IVar** requirement has the additional benefit that we can simply name a secret input after the **IVar** variable in which it was initially stored.

5.2.1 The Attacker Model

In the static analysis of this chapter, we have assumed the *semantic attacker* model of Chapter 4, which can only observe program outputs through *write* statements as prescribed by the standard operational semantics, and can additionally determine whether the program terminates or not.

5.3 Inducing PERs by Expression Evaluation

Consider the program in Figure 5.10, which reveals the parity of the secret input h. This information is derived by the equivalence relation over states induced by the possible evaluations of the expression $h \, \mathit{mod} \, 2$. This equivalence relation is the "parity of h" relation $\boldsymbol{Par}_h$, such that $\forall \sigma, \sigma' \in \boldsymbol{\Sigma}, \sigma \; \boldsymbol{Par}_h \; \sigma'$ iff $\sigma(h) \; \mathit{mod} \, 2 = \sigma'(h) \; \mathit{mod} \, 2$. Thus, on account of the information revealed by this program alone, any pair of input states, which agree on the parity of h, cannot be distinguished by observing the program's output. Alternatively, we can say that the observer can distinguish two input states of this program, by observing the output, only if those states map h to values with different parities.

Let us now introduce a notation to construct PERs over program states by considering expression evaluations.

```
write (h mod 2);
```

Figure 5.10: *A program revealing the parity of its input.*

Definition 5.3.1 (Inducing PERs by evaluations)**.** *Let $\tau \in \{int, bool\}$ be the program data type of an expression e and let $\phi \in PER(\llbracket\tau\rrbracket)$ be a PER over τ values. Define the PER $e{:}\phi \in PER(\Sigma)$ over states that is induced by the observation of e under the constraint ϕ as*

$$\forall \sigma, \sigma' \in \Sigma. \quad \sigma \ (e{:}\phi) \ \sigma' \iff \sigma(e) \ \phi \ \sigma'(e).$$

Furthermore, define the PERs $\mathbf{T}, \mathbf{F} \in PER(\mathbb{B})$ over booleans such that for any $\forall v, v' \in \mathbb{B}$, $v \ \mathbf{T} \ v' \iff v = v' = \mathtt{tt}$ and $v \ \mathbf{F} \ v' \iff v = v' = \mathbf{ff}$. In the special case when $\phi = id$ is the identity relation over values, we have for any expression e, $\forall \sigma, \sigma' \in \Sigma, \sigma \ (e : id) \ \sigma' \iff \sigma(e) = \sigma'(e)$.

The PER ϕ in $e : \phi$ specifies what values of the expression e an attacker can and cannot distinguish, where $v \ \phi \ v'$ means that the attacker cannot distinguish the pair of values v and v'. More generally, for any $v \in dom(\phi)$, the attacker cannot distinguish any pair of values in $[v]_\phi$. Thus, the attacker cannot distinguish a pair of states σ and σ' by observing their e-values if $\sigma(e), \sigma'(e) \in [v]_\phi$. Additionally, the PER ϕ allows us to specify that certain values are not possible whenever those values are not in the domain of definition of ϕ. This partiality property is used to specify the knowledge of which conditional branch has been taken during the analysis of conditional statements. For example, on entering the *then* branch of the conditional statement $\mathtt{if}\,(b)\,\mathtt{then}\,c_1\,\mathtt{else}\,c_2$, we know statically that b evaluates to the value $\mathtt{tt}$ and hence we can identify the information released about b as

the PER $b : \mathbf{T}$ over states. Since $\mathbf{ff} \notin dom(\mathbf{T})$, the PER $\mathbf{T}$ specifies that the boolean guard could not have evaluated to the value $\mathbf{ff}$ on entering the *then* branch. Consequently, the domain of definition of the PER $b : \mathbf{T}$ identifies exactly only the set of states under which the *then* branch of this conditional statement can be executed and it sets the context of implicit information flow for the analysis of c_1.

5.3.1 Conditional Information Flow

Information flow in a program may be conditional. For example, when a command lies within a conditional statement, the information released by this command becomes conditional on how the boolean guard of the conditional statement evaluates. As a concrete example, consider the program listings of Figure 5.11. It is clear that the information flow caused by the command `write` h in the LHS program is different from that in the program on the RHS. In the LHS program, all possible values of the secret input may be learnt, whereas in the RHS program, `write` h only reveals the value of h whenever that value is 10 (in fact, the RHS program as a whole only reveals whether the value of h is 10 or not). These two *write* statements cause different information flows because of the program context where they occur. In particular, the execution of `write` h in the RHS program is predicated on the condition that $h = 10$ holds.

```
write h;  |  if(h=10)then write h;
          |  else skip;
```

Figure 5.11: *Conditional Information Flow*

Considered independently of the execution context, the PER on states induced by command `write` h is $h : id$, which distinguishes any pair of states with different

values of h.

Now, in the RHS program, the *write* statement is executed only if the boolean conditional guard evaluates to the value $\mathtt{tt}$. This predicate on state is captured by the PER $(h = 10) : \mathbf{T}$ which requires that the value of h must be 10. Thus, the domain of the PER $(h = 10) : \mathbf{T}$ encodes the set of states under which the execution of the *then* branch of the RHS program takes place, capturing the conditionality of execution of the *write* statement. Thus, the PER that is *effectively* induced by the `write` h statement in this conditional context may be computed as $((h = 10) : \mathbf{T}) \sqcup (h : id) = (h = 10) : \mathbf{T}$, which means that h is revealed (by the *write* statement) only if its value is 10. This is as opposed to the LHS program that always reveals the value of h (that is, $h : id$).

For the *else* branch of the RHS program, the information released is modelled by the PER $(h = 10) : \mathbf{F}$, which relates all states where $h \neq 10$ and therefore represents the knowledge that h is not equal to 10. Note that this information is gained by the observer of the RHS program if the program produces *no* output when executed. We can thus represent the information released by the RHS program by the PER obtained by taking the disjoint union $(h = 10) : \mathbf{T} \cup (h = 10) : \mathbf{F}$. This information reveals whether $h = 10$ or not.

5.4 Static Analysis of Information Flow with PERs

In this section we present some PER operations that will be used in the information flow analysis. The analysis itself is based on triples that we refer to as *information configurations*, which provide dynamic semantic contexts for the analysis of programs. Information configurations encode information flows at different points

along a program's control-flow path.

5.4.1 Information Configurations

We are interested in keeping track of three aspects of information flow during analysis. These information are encoded in triples referred to as *information configurations* of the form (E, I, O), where E (for *Explicit*) represents explicit information flows to $\mathbf{TVar}$ variables during assignments, and I (for *Implicit*) represents implicit information flows due to program branching and O (for *Output* or *Observed*) represents the information released due to program outputs and observation.

Definition 5.4.1 (Information configurations)**.** *Let* $\mathcal{I} = \textit{PER}(\mathbf{\Sigma})$ *be the lattice of PERs over the set* $\mathbf{\Sigma}$ *of states of a program* P *and let* $\mathbf{Var}$ *be the set of variables of* P*. Define the set of all information configurations with respect to this program as* $\mathbf{\Phi} \triangleq \mathbf{E} \times \mathbf{I} \times \mathbf{O}$*, where* $\mathbf{E} = [\mathbf{Var} \to \mathcal{I}]$ *and* $\mathbf{I} = \mathbf{O} = \mathcal{I}$*.*

We shall use $(E, I, O), \varphi, \psi \in \mathbf{\Phi}$ for information configurations, and symbols $\Phi, \Psi \subseteq \mathbf{\Phi}$ for sets of information configurations - adding superscripts and/or subscripts when necessary. We shall refer to the first, second, and third projection of an information configuration triple as its E-, I-, and O-component respectively.

5.4.2 Context-based PERs

In order to track the flow of information through a program we shall be constructing PERs, which encode the information released. We shall however be tracking the information released about the formal parameters (elements of the set $\mathbf{IVar}$)

of a program only, throwing away (or *forgetting*) information about other variables (elements of the set $\mathbf{TVar}$). Let us define some operations on PERs and information configurations that we shall use in the static analysis.

Definition 5.4.2 (Forgetting information about variables)**.** *Let $R \in \mathit{PER}(\Sigma)$ be a PER over Σ, and let $Z \subseteq \mathbf{Var}$ be a set of variables. Define $\uparrow_Z R$ such that for any $\sigma, \sigma' \in \Sigma$, $\sigma \uparrow_Z R\, \sigma'$ iff there exist states $\sigma_1, \ldots, \sigma_n \in \Sigma$ and $\sigma''_1, \ldots, \sigma''_{n-1} \in \mathit{dom}(R)$ and $\sigma = \sigma_1, \sigma' = \sigma_n$ such that for all i, $1 \leq i \leq n-1$ implies $\sigma_i, \sigma_{i+1} \in \mathit{havoc}Z([\sigma''_i]_R)$.*

Intuitively, $\uparrow_Z R$ "forgets" the information that R has about the variables in Z since each equivalence class of $\uparrow_Z R$ is dense with respect to the values of variables in Z. That is, for all $\sigma \in \mathit{dom}(\uparrow_Z R)$ we have $[\sigma]_{\uparrow_Z R} = \mathit{havoc}Z([\sigma]_{\uparrow_Z R})$. Let us show that $\uparrow_Z R$ is a PER.

Lemma 5.4.3. *Let $Z \subseteq \mathbf{Var}$ be a set of variables in the domain of states in Σ and let $R \in \mathit{PER}(\Sigma)$ be a PER over Σ. The relation $\uparrow_Z R$ is a PER over Σ.*

Proof. The symmetry of $\uparrow_Z R$ is clear. For transitivity, suppose $\sigma \uparrow_Z R\, \sigma'$ and $\sigma' \uparrow_Z R\, \sigma''$ hold. Then there exist two sequences of states $\sigma_1, \ldots, \sigma_n \in \Sigma$ and $\sigma'_1, \ldots, \sigma'_m \in \Sigma$ such that for all $i = 1, \ldots, n-1$ and $j = 1, \ldots, m-1$ there exist $\sigma^A_i, \sigma^B_j \in \mathit{dom}(R)$ such that $\sigma_i, \sigma_{i+1} \in \mathit{havoc}Z([\sigma^A_i]_R)$ and $\sigma'_j, \sigma'_{j+1} \in \mathit{havoc}Z([\sigma^B_j]_R)$ and $\sigma = \sigma_1$ and $\sigma' = \sigma_n = \sigma'_1$ and $\sigma'' = \sigma'_m$. Thus, transitivity of $\uparrow_Z R$ is clear by concatenating the two sequences of states. □

Definition 5.4.4 (Domain-preserving joins)**.** *Let $R, R' \in \mathit{PER}(\Sigma)$ be PERs over Σ. Define the PER $\mathcal{C}_\Sigma(R)$, which extends the domain of R up to $\Sigma \subseteq \Sigma$ while*

preserving the equivalence classes of R *as*

$$\forall \sigma, \sigma' \in \mathbf{\Sigma}, \quad \sigma \, \mathcal{C}_{\Sigma}(R) \, \sigma' \text{ iff } \sigma \, R \, \sigma' \text{ or } \sigma, \sigma' \in \Sigma \backslash dom(R).$$

Furthermore, let $\Sigma = dom(R) \cup dom(R')$*, define a join operation on PERs* R *and* R'*, which preserves their domains as*

$$R \uplus R' \triangleq \mathcal{C}_{\Sigma}(R) \sqcup \mathcal{C}_{\Sigma}(R').$$

The extension of $\uplus$ *to set of PERs* $\mathcal{R} \subseteq PER(\mathbf{\Sigma})$*, where* $\Sigma = \bigcup_{R \in \mathcal{R}} dom(R)$*, is given by*

$$\biguplus \mathcal{R} \triangleq \bigsqcup_{R \in \mathcal{R}} \mathcal{C}_{\Sigma}(R).$$

The join operation $\uplus$ *has an associated partial order,* $\sqsubseteq$*, defined as*

$$R \sqsubseteq R' \iff R \uplus R' = R'.$$

Let $(E, I, O), (E', I', O') \in \mathbf{\Phi}$ *be information configurations,* $\uplus$ *is extended to information configurations as*

$$(E, I, O) \uplus (E', I', O') \triangleq (E \uplus E', I \uplus I', O \uplus O')$$

where, as usual, $E \uplus E'$ *is the pointwise join of functions. The extension* $\biguplus$ *of* $\uplus$ *to sets of information configurations is done in the usual way. Finally, define an*

order relation on information configurations as

$$(E, I, O) \sqsubseteq (E', I', O') \iff \forall x \in \mathbf{Var}. E(x) \sqsubseteq E'(x), I \sqsubseteq I', O \sqsubseteq O'.$$

The operation $\uplus$ preserves domains of PERs. It also preserves the partitioning, that is, the information content of PERs.

Proposition 5.4.5. *Let $R, R', R'' \in \mathit{PER}(\mathbf{\Sigma})$ be PERs, then*

1. *$\mathit{dom}(R) \subseteq \mathit{dom}(R \uplus R')$,*
2. *for all $\sigma, \sigma' \in \mathit{dom}(R)$, $\sigma \,(R \uplus R')\, \sigma' \implies \sigma \, R \, \sigma'$,*
3. *for any $\sigma \in \mathit{dom}(R)$ and $\sigma' \in \mathbf{\Sigma}$ such that $\sigma' \notin \mathit{dom}(R)$, we have that $(\sigma, \sigma') \notin (R \uplus R')$,*
4. *$\mathit{dom}(R) \subseteq \mathit{dom}(R')$ and $R' \sqsubseteq R''$ implies that $R \sqcup R' \sqsubseteq R \sqcup R''$.*

Proof.

1. It is clear from the definition that $\mathit{dom}(R \uplus R') = \mathit{dom}(R) \cup \mathit{dom}(R')$ and hence that $\mathit{dom}(R) \subseteq \mathit{dom}(R \uplus R')$.

2. Let $\Sigma = \mathit{dom}(R) \cup \mathit{dom}(R')$ and define $\overline{R}$ and $\overline{R'}$ such that $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \overline{R} \sigma' \iff \sigma, \sigma' \in \Sigma \backslash \mathit{dom}(R)$ and $\sigma \; \overline{R'} \; \sigma' \iff \sigma, \sigma' \in \Sigma \backslash \mathit{dom}(R')$. Then we have $R \uplus R' = (R \cup \overline{R}) \sqcup (R' \cup \overline{R'}) = (R \sqcup R') \cup (R \sqcup \overline{R'}) \cup (R' \sqcup \overline{R})$, since $\overline{R} \sqcup \overline{R'} = \varnothing$. Furthermore, since by definition $\sigma, \sigma' \in \mathit{dom}(R) \implies \sigma, \sigma' \notin \mathit{dom}(\overline{R})$, then for any $\sigma, \sigma' \in \mathit{dom}(R), \sigma \; R \uplus R' \; \sigma' \implies \sigma \,(R \sqcup R')\, \sigma'$ or $\sigma \,(R \sqcup \overline{R'})\, \sigma' \implies \sigma \, R \, \sigma'$.

3. From the definition $\sigma_1\ (R \uplus R')\ \sigma_2 \implies \sigma_1\ \mathcal{C}_\Sigma(R)\ \sigma_2 \implies \sigma_1\ R\ \sigma_2$ or $\sigma_1, \sigma_2 \in \Sigma \backslash \mathit{dom}(R)$, where $\Sigma = \mathit{dom}(R) \cup \mathit{dom}(R')$. Since neither $\sigma\ R\ \sigma'$ nor $\sigma, \sigma' \in \Sigma \backslash \mathit{dom}(R)$ holds, then $(\sigma, \sigma') \notin R \uplus R'$.

4. Since $R' \Subset R''$ then $R'' = R' \uplus R''$ and hence $\Sigma'' = \mathit{dom}(R'') = \mathit{dom}(R'') \cup \mathit{dom}(R')$. Define $\overline{R'}$ such that $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ \overline{R'}\ \sigma' \iff \sigma, \sigma' \in \Sigma'' \backslash \mathit{dom}(R')$. Since $\Sigma'' = \mathit{dom}(R'')$ then $\mathcal{C}_{\Sigma''}(R'') = R''$. Thus, $R'' = R' \uplus R'' = \mathcal{C}_{\Sigma''}(R') \sqcup R'' = (R' \sqcup R'') \cup (R'' \sqcup \overline{R'})$. Hence, $R'' \sqcup R = R \sqcup R' \sqcup R''$ because $\overline{R'} \sqcup R = \varnothing$ since $\mathit{dom}(R) \subseteq \mathit{dom}(R')$, and, R' and $\overline{R'}$ are disjoint by definition. Since $R'' \sqcup R = R \sqcup R' \sqcup R''$, then $R \sqcup R' \sqsubseteq R \sqcup R''$.

□

It is clear that $\Subset$ is a partial order on PERs because $\uplus$ is idempotent, commutative, and associative. The resulting lattice is also complete. The induced order on $\mathbf{\Phi}$ in turn makes the set of information configurations a complete lattice.

Theorem 5.4.6. *The set* $\mathit{PER}(\mathbf{\Sigma})$ *of PERs over* $\mathbf{\Sigma}$ *is partially ordered by* $\Subset$*, and* $\langle \mathit{PER}(\mathbf{\Sigma}), \Subset, \uplus \rangle$ *forms a complete lattice*

Proof. The partial order proof is straightforward. We shall now show that for any $\mathcal{R} \subseteq \mathrm{PER}(\mathbf{\Sigma})$ and $R' \in \mathrm{PER}(\mathbf{\Sigma})$ such that for all $R \in \mathcal{R}, R \Subset R'$, we have that $\biguplus \mathcal{R} \Subset R'$. Let $\Sigma = \mathit{dom}(R')$, then we know by (1) of proposition 5.4.5 that for all $R \in \mathcal{R}, \mathit{dom}(R) \subseteq \Sigma$, since $R \Subset R'$, and hence that $\mathit{dom}(\biguplus R) \subseteq \Sigma$. Now, by definition, $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma (\biguplus \mathcal{R} \uplus R') \sigma' \iff \sigma (\bigsqcup_{\forall R \in \mathcal{R}} \mathcal{C}_\Sigma(R) \sqcup R')\, \sigma' \iff \sigma\ R'\ \sigma'$ since for all $R \in \mathcal{R}, \sigma\ R'\ \sigma' \implies \sigma\ \mathcal{C}_\Sigma(R)\ \sigma'$ by the fact that $R \Subset R'$, which means that $\mathcal{C}_\Sigma(R) \sqcup R' = R'$. Therefore $\biguplus \mathcal{R} \uplus R' = R'$, which shows the desired property. □

Theorem 5.4.7. *The partially ordered set* $\langle \mathbf{\Phi}, \sqsubseteq, \sqcup \rangle$ *of information configurations is a complete lattice.*

Proof. For this proof it is sufficient to show that for any arbitrary $\Phi \subseteq \mathbf{\Phi}$ the join $\bigsqcup^{+} \Phi$ exists in $\mathbf{\Phi}$ [GHK+03]. As usual, $\bigsqcup^{+} \Phi = (E', I', O')$ is the information configuration defined as $\forall x \in \mathbf{Var}, E'(x) = \bigsqcup^{+} \{E(x) \mid (E, I, O) \in \Phi\}$ and $I' = \bigsqcup^{+} \{I \mid (E, I, O) \in \Phi\}$ and $O' = \bigsqcup^{+} \{O \mid (E, I, O) \in \Phi\}$. The proof is immediate since from Theorem 5.4.6 $\langle \mathrm{PER}(\mathbf{\Sigma}), \sqsubseteq, \sqcup \rangle$ is a complete lattice. □

Lemma 5.4.8. *Let $\mathbf{\Sigma}$ be the set of all states, which are maps from $\mathbf{Var}$ to values. Suppose $\Sigma, \Sigma' \subseteq \mathbf{\Sigma}$ and that $Z \subseteq \mathbf{Var}$ and let $R, R' \in \mathit{PER}(\mathbf{\Sigma})$. Then we have the following properties:*

1. *Let $X, Y \subseteq \mathbf{Var}$ such that $X \cup Y = Z$, then $\mathit{havoc}Z(\Sigma) \cup \mathit{havoc}Z(\Sigma') = \mathit{havoc}Z(\Sigma \cup \Sigma')$, and $\mathit{havoc}X(\mathit{havoc}Y(\Sigma)) = \mathit{havoc}Z(\Sigma)$.*
2. *The operator $\mathit{havoc}Z(\cdot)$ is an upper closure operator on the powerset lattice $\langle \mathcal{P}(\mathbf{\Sigma}), \subseteq \rangle$ with respect to the subset inclusion order.*
3. *The following identities hold*
   (a) *$\mathit{havoc}Z(\Sigma) \cup \mathit{havoc}Z(\Sigma') = \mathit{havoc}Z(\mathit{havoc}Z(\Sigma) \cup \mathit{havoc}Z(\Sigma'))$.*
   (b) *$\mathit{havoc}Z(\Sigma) \cap \mathit{havoc}Z(\Sigma') = \mathit{havoc}Z(\mathit{havoc}Z(\Sigma) \cap \mathit{havoc}Z(\Sigma'))$.*
   (c) *$\mathit{havoc}Z(\Sigma) \backslash \mathit{havoc}Z(\Sigma') = \mathit{havoc}Z(\mathit{havoc}Z(\Sigma) \backslash \mathit{havoc}Z(\Sigma'))$.*
4. *For all $\sigma \in \mathit{dom}(\uparrow_Z R)$ we have $[\sigma]_{\uparrow_Z R} = \mathit{havoc}Z([\sigma]_{\uparrow_Z R})$.*
5. *For any $X, Y \subseteq \mathbf{Var}$ we have $\uparrow_X \uparrow_Y R = \uparrow_Y \uparrow_X R = \uparrow_{X \cup Y} R$.*
6. *$\uparrow_Z R \sqcup \uparrow_Z R' = \uparrow_Z (\uparrow_Z R \sqcup \uparrow_Z R')$.*
7. *$\uparrow_Z R \uplus \uparrow_Z R' = \uparrow_Z (\uparrow_Z R \uplus \uparrow_Z R')$.*
8. *$R \sqsubseteq R' \implies \uparrow_Z R \sqsubseteq \uparrow_Z R'$.*

Proof. See Appendix A. □

We shall now define some operations on information configurations.

Definition 5.4.9 (Semantic Sets and PERs)**.** *Let $(E, I, O) \in \mathbf{\Phi}$ be an information configuration, define the* semantic set *of (E, I, O), which represents the set of*

program states modelled by the configuration (E, I, O) *to be*

$$\mathit{dom}((E, I, O)) \triangleq \bigcap_{z \in \mathbf{Var}} \mathit{dom}(E(z)) \cap \mathit{dom}(I) \cap \mathit{dom}(O).$$

Define the PER $\mathit{PER}((E, I, O))$*, which encodes this set such that*

$$\forall \sigma, \sigma' \in \mathbf{\Sigma}, \quad \sigma \; \mathit{PER}((E, I, O)) \; \sigma' \iff \sigma, \sigma' \in \mathit{dom}((E, I, O)).$$

Information configurations provide contexts under which the information released by program commands and expression evaluations may be constructed. The information released by a subprogram of a given program is constrained by the possible set of states, as prescribed by the semantic set of an information configuration, that reaches that subprogram. The information released by the observation of the evaluation of an expression in a given program context is defined as follows.

Definition 5.4.10 (Information released in a context)**.** *Let* $\mathbf{\Sigma}$ *be the set of all states, which are maps from* $\mathbf{Var}$ *to values, and let* $\mathbf{IVar} \subseteq \mathbf{Var}$ *such that* $\mathbf{TVar} = \mathbf{Var} \backslash \mathbf{IVar}$*. Furthermore, let* $(E, I, O) \in \mathbf{\Phi}$ *be an information configuration, and let* e *be an expression of type* τ *such that* $FV(e) \subseteq \mathbf{Var}$*, and let* $\phi \in \mathit{PER}(\llbracket \tau \rrbracket)$ *be a PER over* τ*.*

Define the PER $\mathit{flow}(e : \phi, (E, I, O))$ *over* $\mathbf{\Sigma}$ *to be*

$$\mathit{flow}(e : \phi, (E, I, O)) \triangleq \uparrow_{\mathbf{TVar}} (e : \phi \sqcup R_E \sqcup I)$$

where R_E *is defined such that* $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_E \; \sigma'$ *iff* $\sigma, \sigma' \in \mathit{dom}\left(\bigsqcup_{z \in FV(e)} E(z)\right)$*.*

When $\mathbf{IVar}$ is the set of input variables to a *While* program P, this definition constructs a PER *flow*$(e : \phi, (E, I, O))$, which represents the information released about the formal parameters of P by the observation of the evaluation e subject to the constraint ϕ over its evaluation. The PERs R_E and I provide a context, specifying what states are possible as prescribed by the information configuration (E, I, O) (see Definition 5.4.9), for the evaluation of e. The PER R_E places a constraint on the possible values of the free variables of e, and the PER I is a constraint on the possible values of the program parameters that cause the control-flow to reach the evaluation of e due to program branching. The definition of *flow*$(\cdot, \cdot)$ only computes the information released about the program's formal parameters by throwing away (via the operation $\uparrow_{\mathbf{TVar}}(\cdot)$) the information encoded about variables the variables of $\mathbf{TVar}$, which are not formal parameters of P. With these definitions in hand, we can now present the static analysis rules for the analysis of information flow in *While* programs.

5.5 The Information Flow Rules

The algorithmic information flow rules for the static analysis of information flow in a given *While* program is presented in Figure 5.12. The analysis rules are defined parametric to a given program P, where the sets $\mathbf{Var}, \mathbf{IVar}, \mathbf{TVar}, \mathbf{\Sigma}$, and $\mathbf{\Phi}$ are defined with respect to this program. For some $(E, I, O) \in \mathbf{\Phi}$ and subprogram c of P, the analysis of c is specified as a transformation of information configurations: $(E, I, O)\ c\ (E', I', O')$. The information configuration (E, I, O) is referred to as the *pre-configuration* or *precondition* for the analysis of c and (E', I', O') is referred to as the *post-configuration* or *postcondition* of the analy-

sis. The pre-configuration (E, I, O) provides a semantic and information context for the analysis, where the semantic set *dom*$((E, I, O))$ represents the starting set of states for the analysis of c, and O is the attacker's knowledge before the execution of c. Similarly, *dom*$((E', I', O'))$ contains the set of states under which c terminates and O' represents the attacker's knowledge after observing the execution of c. In particular, E and E' respectively keep track of the values of program variables and their dependencies on input parameters before and after the execution of c. Program branching information, specifically, the values of the program input parameters under which a given program point is reached are encoded in the I-part of the information configuration, setting the context for the analysis of subprograms of conditional statements.

We shall explain each of the analysis rules in the remainder of this section. The first two rules are straightforward. The [SKIP] rule shows that the `skip` command does not modify information configurations and therefore causes no information flow, and the sequential composition rule ([SEQNC]) shows that the analysis is compositional. Let us now look at the definition of the remaining rules.

5.5.1 Analysis of *write* Statements

The *write* statements in a program cause information to flow directly to the program observer. The semantic attacker model shows that the attacker can observe the output value of a *write* statement as prescribed by the operational semantics, and hence the information released to the attacker is modelled by the identity observational constraint (e : *id*) on the program output. The information flow to the attacker is however subject to the semantic constraints placed on the possi-

$$\text{Let } \varphi = (E, I, O) \in \mathbf{\Phi}.$$

$$\text{[SKIP]}\ \frac{}{\varphi\ \texttt{skip}\ \varphi} \qquad \text{[SEQNC]}\ \frac{\varphi\ c_1\ \varphi' \quad \varphi'\ c_2\ \varphi''}{\varphi\ c_1; c_2\ \varphi''}$$

$$\text{[WRITE]}\ \frac{O' = O \uplus \mathit{flow}(e : \mathit{id}, \varphi)}{\varphi\ \texttt{write}\ e\ (E, I, O')} \qquad \text{[ASSGN]}\ \frac{E' = E[z \mapsto \mathit{aflow}(z := e, \varphi)]}{\varphi\ z := e\ (E', I, O)}$$

$$\text{[IF]}\ \frac{\begin{array}{c} I' = \mathit{flow}(b : \mathbf{T}, \varphi) \quad I'' = \mathit{flow}(b : \mathbf{F}, \varphi) \\ (E, I', O)\ c_1\ \psi' \quad (E, I'', O)\ c_2\ \psi'' \end{array}}{\varphi\ \texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \psi' \uplus_\varphi \psi''}$$

$$\text{[WHL]}\ \frac{\begin{array}{c} \varphi_0 = \varphi \quad (E_n, I_n, O_n) = \varphi_n \quad \varphi_n\ \texttt{if}(b)\texttt{then}\ c\ \texttt{else}\ \texttt{skip}\ \varphi_{n+1} \quad I'_n = \mathit{flow}(b : \mathbf{F}, \varphi_n) \\ \forall x \in \mathbf{Var}.\ \bar{X} = \mathbf{TVar} \backslash \{x\}.\ \sigma\ E'(x)\ \sigma' \iff \exists i \in \mathbb{N},\ \sigma \uparrow_{\bar{X}} (E_i(x) \sqcup I'_i \sqcup b : \mathbf{F})\ \sigma' \\ (E'', I'', O'') = \bigsqcup\!\!\!\!\!+_{i \geq 0} \varphi_i \quad I' = \mathit{flow}(b : \mathbf{F}, (E', I'', O'')) \quad O' = \bigsqcup\!\!\!\!\!+_{i \geq 0} \mathit{flow}(b : \mathit{id}, \varphi_i) \uplus O'' \end{array}}{\varphi\ \texttt{while}\ (b)\ \texttt{do}\ c\ (E', I', O')}$$

Figure 5.12: *Calculus of Information Flow*

ble values of the free variables of the expression (based on previous assignments) and the possible values of the program parameters that cause the control-flow to reach the *write* statement. These semantic constraints are specified by the pre-configuration (E, I, O). Thus, the attacker's knowledge after observing the result of the statement `write` e in the context provided by (E, I, O) is captured by taking a join of the prior knowledge O of the attacker with the released information $\mathit{flow}(e : \mathit{id}, (E, I, O))$ in that context.

To illustrate how the [WRITE] rule captures the information flow to an attacker, consider the example shown in Figure 5.13. Here the attacker wishes to gain access to the values of two secret input parameters h_1 and h_2 by solving equations involving these inputs.

```
write  h1 + h2;
write  h1 − h2;
```

Figure 5.13: *PER joins capture information flow via equation solving*

This example illustrates how PER joins capture reasoning with equations as follows. The PERs induced by the expressions $h_1 + h_2$ and $h_1 - h_2$ respectively are the equivalence relations $R_1 = (h_1 + h_2) : \mathit{id}$ and $R_2 = (h_1 - h_2) : \mathit{id}$, where

- $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_1 \; \sigma' \iff \sigma(h_1 + h_2) = \sigma'(h_1 + h_2)$, and
- $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_2 \; \sigma' \iff \sigma(h_1 - h_2) = \sigma'(h_1 - h_2)$.

In the flow rules, the fact that the attacker learns the precise values of both secrets after observing the output of the two *write* statements is captured by the join $R = R_1 \uplus R_2 = R_1 \sqcup R_2$ (since both R_1 and R_2 are equivalence relations)

which is given by:

$$\begin{aligned} \forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R \, \sigma' \quad &\Longleftrightarrow \quad \sigma(h_1 + h_2) = \sigma'(h_1 + h_2) \text{ and } \sigma(h_1 - h_2) = \sigma'(h_1 - h_2) \\ &\Longleftrightarrow \quad \sigma(h_1) = \sigma'(h_1) \text{ and } \sigma(h_2) = \sigma'(h_2). \end{aligned}$$

This means that two states are indistinguishable to the attacker (via the knowledge modelled by the PER R) if and only if they both agree on the values of both h_1 and h_2. In other words, the attacker learns the values of h_1 and h_2.

5.5.2 Analysis of *if* statements

We know statically what the conditional guard evaluates to when the control flow is passed to one of the branches of an *if* statement. This information is captured in the [IF] rule by constructing a PER representing the set of states that evaluate the boolean guard to the appropriate value on entering that branch. The information thus released (also known as *implicit information flow*) constitute the implicit information contexts under which the branches of the *if* statement are analysed. These implicit contexts are computed from the boolean expression b as *flow*$(b : \mathbf{T}, \varphi)$ and *flow*$(b : \mathbf{F}, \varphi)$ for the *then* and *else* branches respectively, which identify the states in which the boolean guard evaluates to $\mathtt{tt}$ and $\mathbf{ff}$ respectively.

As demonstrated by Example 5.1.4 and Example 5.1.5, in computing the implicit information flows in the branches of a conditional *if* statement, simply looking for *assignment* or *write* statements in each branch of the *if* statement independently of the other branch can cause certain implicit flows to go undetected. This information flow is due to a well-known problem that information flow is not a property of individual execution paths [Vol99b, McL94, SM03a, Sch00]. Such

flows can occur when an attacker observes that certain actions *did not* take place, such as outputs or assignments on a given execution path. Thus, runtime execution monitors [Sch00] cannot detect such flows when execution passes through the control-flow path where the relevant action is missing. However, this is not a problem for static analysis since information about all program paths are available to the analyser - making static analysis more suitable for the analysis of secure information flow. An execution monitor [GBJS06], which is able to deal with this problem uses the result of a static analysis to prevent this problem in the execution monitor. The operation $\uplus$ in the *if* rule, which is given in the following definition, identifies this information flow.

Definition 5.5.1. *Let* $(E, I, O) \in \mathbf{\Phi}$ *be an information configuration and let* b *be a boolean expression an let* c_1 *and* c_2 *be* While *commands, such that* $I' =$ *flow*$(b : \mathbf{T}, (E, I, O))$ *and* $I'' =$ *flow*$(b : \mathbf{F}, (E, I, O))$ *and* $(E, I', O) c_1 (E_1, I_1, O_1)$ *and* $(E, I'', O)\ c_2\ (E_2, I_2, O_2)$.

Let E_1', E_2' *and* I_3 *be defined as:*

$$\forall z \in \mathbf{Var},\ E_1'(z) = \begin{cases} E_1(z) \sqcup I' & \textit{if } E_2(z) \neq E(z) \\ E_1(z) & \textit{otherwise} \end{cases}$$

and,

$$\forall z \in \mathbf{Var},\ E_2'(z) = \begin{cases} E_2(z) \sqcup I'' & \textit{if } E_1(z) \neq E(z) \\ E_2(z) & \textit{otherwise} \end{cases}$$

and,

$$\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ I_3\ \sigma' \iff \sigma, \sigma' \in \textit{dom}(I_1) \cup \textit{dom}(I_2)$$

The post-configuration of the conditional statement `if` (b) `then` c_1 `else` c_2*,*

with respect to the pre-configuration (E, I, O)*, is given by*

$$(E_1, I_1, O_1) \uplus_{(E,I,O)} (E_2, I_2, O_2) \triangleq (E_1', I_3, O_1) \uplus (E_2', I_3, O_2).$$

The implicit context in the post-configuration of the conditional *if* statement is a PER representing the union of the domains of the post-configurations of the branches. This makes the information in the implicit context local to the relevant branches, otherwise this can result in the so-called *label creep* [SM03a] - a condition where the security type of the "program counter" monotonically increases due to conditional statements. The implicit context also keeps track of the dependency of reaching a particular program point on the value of inputs, and, as will be seen in the analysis of *while* statements, input states leading to program divergence are removed from the implicit context of the *while* statement post-configuration.

5.5.3 Analysis of Assignment Statements

When an assignment takes place in a program context, information is encoded directly in the assigned variable (by virtue of the assigned expression) and/or indirectly (by virtue of the implicit context under which the assignment takes place). However, since assignment also changes program state, we define a *transposition* operation on PERs over states that models the semantic effect of assignments on states.

Definition 5.5.2 (PER transposition and assignment)**.** *Let* R *be a PER over* Σ *and* $z \in \mathbf{TVar}$*. Define the transposition of* R *by the assignment* $z := e$ *as the binary relation* $\overset{z:=e}{\rightsquigarrow} R$ *such that for any* $\sigma, \sigma' \in \Sigma$*,* $\sigma \overset{z:=e}{\rightsquigarrow} R\ \sigma'$ *iff there exist states* $\sigma_1, \ldots, \sigma_n \in \Sigma$

and $\sigma''_1, \ldots \sigma''_{n-1} \in$ *dom*(R) *and* $\sigma = \sigma_1, \sigma' = \sigma_n$ *such that for all* i, $1 \leq i \leq n-1$ *implies* $\sigma_i, \sigma_{i+1} \in \{\hat{\sigma}[z \mapsto \hat{\sigma}(e)] \mid \hat{\sigma} \in [\sigma''_i]_R\}$.

Now let $(E, I, O) \in \mathbf{\Phi}$ *be an information configuration and let* e *be an expression and let* $z \in \mathbf{TVar}$. *Furthermore, let* $R = e : id \sqcup R_E \sqcup I$, *where* R_E *is the PER defined such that* $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_E \; \sigma'$ *iff* $\sigma, \sigma' \in dom\left(\bigsqcup_{y \in FV(e)} E(y)\right)$ *and let* $\bar{Z} = \mathbf{TVar} \backslash \{z\}$. *Define the information released to* z *by the assignment* $z := e$ *under the pre-configuration* (E, I, O) *to be*

$$aflow(z := e, (E, I, O)) \triangleq \uparrow_{\bar{Z}} \overset{z:=e}{\rightsquigarrow} R \,.$$

The intention behind PER transposition is to transfer the relational structure (information content) of R to another PER $\overset{z:=e}{\rightsquigarrow} R$, which updates state in lockstep with the semantic effect of the assignment $z := e$. The information flow to z due to the assignment $z := e$ is then computed as the information released by the evaluation of e in the context (E, I, O), and the state change is reflected by the PER transposition. In the definition of *aflow*$(z := e, (E, I, O))$, only the value of z is retained out of all the $\mathbf{TVar}$ variables and the values of other $\mathbf{TVar}$ variables are "forgotten" (that is, $\uparrow_{\bar{Z}} \overset{z:=e}{\rightsquigarrow} R$, where $\bar{Z} = \mathbf{TVar} \backslash \{z\}$). Thus, in the resulting information configuration of the assignment rule, where $E' = E[z \mapsto aflow(z := e, (E, I, O))]$, $E'(z)$ keeps track only of the value of z and its dependency on the program's formal parameters $\mathbf{IVar}$. Let us show that the transposition of a PER is also a PER.

Lemma 5.5.3. *Let* $R \in PER(\mathbf{\Sigma})$ *be a PER, then the transposition* $\overset{z:=e}{\rightsquigarrow} R$ *of* R *is also a PER.*

Proof. The symmetry of $\overset{z:=e}{\rightsquigarrow} R$ is clear. For transitivity, suppose $\sigma \overset{z:=e}{\rightsquigarrow} R \; \sigma'$ and $\sigma' \overset{z:=e}{\rightsquigarrow} R \; \sigma''$

hold, then there exist two sequences of states $\sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$ and $\sigma'_1, \ldots, \sigma'_m \in \mathbf{\Sigma}$ such that for all $i = 1, \ldots, n-1$ and $j = 1, \ldots, m-1$ there exist $\sigma_i^A, \sigma_j^B \in \mathit{dom}(R)$ and $\sigma_i, \sigma_{i+1} \in \{\sigma''[z \mapsto \sigma''(e)] \mid \sigma'' \in [\sigma_i^A]_R\}$ and $\sigma'_j, \sigma'_{j+1} \in \{\sigma''[z \mapsto \sigma''(e)] \mid \sigma'' \in [\sigma_j^B]_R\}$ and $\sigma = \sigma_1$ and $\sigma' = \sigma_n = \sigma'_1$ and $\sigma'' = \sigma'_m$. Thus, transitivity of $\overset{z:=e}{\rightsquigarrow} R$ is clear by concatenating the two sequences of states. □

A Notation. To aid presentation, we shall often represent a PER by its set of equivalence classes. For example, $R \equiv [\mathbf{\Sigma}]_R$ means that $[\mathbf{\Sigma}]_R$ is the set of equivalence classes of R. Recall from section 3.5.3 that this partitioning is defined as $[\mathbf{\Sigma}]_R = \{[\sigma]_R \mid \sigma \in \mathit{dom}(R)\}$ for any $R \in \text{PER}(\mathbf{\Sigma})$. This notation is reasonable because a PER is completely determined by its set of equivalence classes. For example, suppose $\sigma_2 \neq \sigma_1 \neq \sigma_3 \neq \sigma_2$, the PER R defined as $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R \; \sigma' \iff \sigma = \sigma' = \sigma_1$ or $\sigma, \sigma' \in \{\sigma_2, \sigma_3\}$ is written as $R \equiv \{\{\sigma_1\}, \{\sigma_2, \sigma_3\}\}$. The states themselves will be represented by tuples of values.

Sample Analysis

To illustrate the information flow rules presented so far, consider the analysis of the program shown in Figure 5.14. Suppose $\mathbf{IVar} = \{h_1, h_2\}$ and $\mathbf{TVar} = \{l\}$, such that $[\![\tau_{h_1}]\!] = [\![\tau_{h_2}]\!] = \{0, 1\}$ and $[\![\tau_l]\!] = \{0, 1, 2\}$ (where τ_x is the data type of variable x). For brevity, the states are represented by tuples from $[\![\tau_{h_1}]\!] \times [\![\tau_{h_2}]\!] \times [\![\tau_l]\!]$ so that $(1, 0, 1)$ represents the state $\sigma \in \mathbf{\Sigma}$ where $\sigma(h_1) = \sigma(l) = 1$ and $\sigma(h_2) = 0$.

Assume that h_1 and h_2 contain secret values and that l is public, but we want to find out what information is gained about the secret inputs.

We choose a starting configuration, which contains no prior information, and which makes no assumption about the starting state to be (E, I, O), such that $\forall x \in \{h_1, h_2, l\}$, $E(x) = I = O = \mathit{all} \equiv \{\{(i,j,k) \mid i,j \in \{0,1\}, k \in \{0,1,2\}\}\}$. Thus, the attacker ($O$) has no initial knowledge about the inputs h_1 and h_2 since O relates all states. Applying the assignment rule at line 1, we arrive at the configuration (E_1, I, O), where $E_1 = E[l \mapsto R_1]$ and $R_1 \equiv \{\{(0,0,0)\}, \{(0,1,1),(1,0,1)\}, \{(1,1,2)\}\}$. The partitioning of R_1 reflects the fact that, after this assignment, observing the value of l as 0 reveals that $h_1 = h_2 = 0$, a value 1 reveals that either $h_1 = 0, h_2 = 1$ or $h_1 = 1, h_2 = 0$, and a value 2 reveals that $h_1 = h_2 = 1$.

```
1 l := h1 + h2;
2 if (l = 1) then
3   l := l + h1;
4   write l;
5 else
6   skip;
```

Figure 5.14: *Illustrating* assignment, conditional, *and* write *analysis.*

In the *then* branch of the *if* statement, the implicit context is given by $I_1 = \mathit{flow}((l = 1) : \mathbf{T}, (E_1, I, O)) = \uparrow_{\{l\}} R_2 \equiv \{\{(0,1,k),(1,0,k) \mid k \in [\![\tau_l]\!]\}\}$. The PER $R_2 = ((l = 1) : \mathbf{T}) \sqcup I \sqcup R_2' \equiv \{\{(0,1,1),(1,0,1)\}\}$, where $\sigma \; R_2' \; \sigma'$ iff $\sigma, \sigma' \in \mathit{dom}(E_1(l))$, relates only the set of states where $l = 1$. Consequently, the pre-configuration for the *then* branch is (E_1, I_1, O). Applying the assignment rule again on line 3 under the information configuration (E_1, I_1, O), we obtain (E_2, I_1, O), where the PER encoded against l is now given by $E_2(l) \equiv \{\{(0,1,1)\}, \{(1,0,2)\}\}$. This means that by observing the value of l at this point we can determine the value

of both h_1 and h_2, where $l = 1 \implies h_1 = 0, h_2 = 1$ and $l = 2 \implies h_1 = 1, h_2 = 0$. This information is released by the following *write* statement because by starting with the pre-configuration (E_2, I_1, O) and applying the *write* rule on line 4 we obtain the post-configuration (E_2, I_1, O_1), where $O_1 = O \uplus \mathit{flow}(l : \mathit{id}, (E_2, I_1, O)) \equiv \{\{(0,1,k) | k \in [\![\tau_l]\!]\}, \{(1,0,k) | k \in [\![\tau_l]\!]\}, \{(0,0,k), (1,1,k) | k \in [\![\tau_l]\!]\}\}$. The equivalence classes $\{(0,1,k) | k \in [\![\tau_l]\!]\}$ and $\{(1,0,k) | k \in [\![\tau_l]\!]\}$ of O_1 retain the information encoded in l about h_1 and h_2, namely that states with different input values of h_1 and h_2 can be distinguished. The equivalence class $\{(0,0,k), (1,1,k) | k \in [\![\tau_l]\!]\}$ of O_1 comes from O due to the domain-preserving property of $\uplus$ and it reflects the information released about secrets (that $h_1 = h_2$) in the *else* branch of the conditional *if* statement if no output is produced in that branch (which is the case in this example).

Since the *else* branch is a `skip` statement, the post-configuration of this branch remains unchanged, but the assignment to l in the *then* branch means that the l part of the E-component of the postcondition of the *else* branch must be updated (due to the fact that $E_1(l) \neq E_2(l)$). This yields $E_3 = E_1[l \mapsto E_1(l) \sqcup I_2]$, where $I_2 = \mathit{flow}((l = 1) : \mathbf{F}, (E_1, I, O)) \equiv \{\{(0,0,k), (1,1,k) \mid k \in [\![\tau_l]\!]\}\}$ and hence $E_3(l) \equiv \{\{(0,0,0)\}, \{(1,1,2)\}\}$. Thus, the post-configuration of the conditional *if* statement is $(E_2, I, O_1) \uplus (E_3, I, O) = (E_4, I, O_1)$, where $E_4(l) \equiv \{\{(0,0,0)\}, \{(0,1,1)\}, \{(1,0,2)\}, \{(1,1,2)\}\}$ represents the various possible values that l might take based on the choice of the inputs h_1 and h_2, and hence its dependency on the inputs.

Note that, on one hand, the semantic set of the pre-configuration (E, I, O) of the analysis of this program defines the set of possible starting states of the program $\mathit{dom}((E, I, O)) = \mathbf{\Sigma}$, which is the set of all states. On the other hand,

the semantic set of the post-configuration is the set of terminating states of the program, namely, $\mathit{dom}((E_4, I, O_1)) = \{(0,0,0),(0,1,1),(1,0,2),(1,1,2)\}$. We shall prove a semantic correctness property of the analysis in Theorem 5.7.10, which states that the semantic set of the post-configuration contains the set of states under which the program terminates when the program is executed from a starting state chosen from the semantic set of the pre-configuration. Formally, this means that for a given program P and the relevant information configurations φ and φ' used in its analysis, $\varphi\, P\, \varphi' \implies \{\sigma' \mid \sigma \in \mathit{dom}(\varphi), \langle P, \sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi')$.

5.5.4 Analysis of *while* Statements

The *while* rule computes the limit of the monotonically increasing chain $\bigsqcup\!\!\!\!+_{n\geq 0}\, \varphi_i$ over the lattice $\langle \mathbf{\Phi}, \sqsubseteq, \sqcup\!\!\!\!+ \rangle$, induced by the iterative application of the command `if(`b`)then` c `else skip`. This computes the information released during each iteration of the *while* statement `while` (b) `do` c, which begins from the pre-configuration $\varphi = \varphi_0$ of the *while* statement. Being a monotonically increasing chain on a complete lattice, the fixpoint $\bigsqcup\!\!\!\!+_{n\geq 0}\, \varphi_i$ exists [Tar55].

The definition of the post-configuration (E', I', O') of the *while* analysis ensures that only the set of states under which the *while* loop terminates can be used in the analysis of the subsequent statements after the loop. Thus, E' and I' are defined to select only the states where the boolean guard evaluates to $\mathbf{ff}$. Since the attacker model can determine whether the program terminates or not, the definition of O' partitions states, through $b : \mathit{id}$, to those in which the *while* loop terminates or diverges.

To illustrate the *while* analysis rule, consider the program listing in Figure 5.15,

where the attacker performs a linear search on the value of the secret h. In this example, $\mathbf{IVar} = \{h\}$ and $\mathbf{TVar} = \{l\}$. Let us assume that h and l are of integer type τ_h and τ_l respectively, where $[\![\tau_h]\!] = [\![\tau_l]\!] = \mathbb{V} = \{i \in \mathbb{Z} \mid -n \leq i < n\}$ is the set of integers between $-n$ (inclusive) and n (exclusive), and where $n \in \mathbb{N}$ is a natural number. If h is chosen to be a natural number in $\mathbb{V}$, the *while* loop will terminate with the value of l equal to the secret value of h (and will be printed to the output). However, nontermination reveals that h is a negative integer in $\mathbb{V}$. Let us see how this is derived by the analysis.

The Figure 5.16 annotates the program of Figure 5.15 with information configurations at selected milestones to illustrate the analysis. The pre-configurations of the analysis is assumed to be (E, I, O), where $E(h) = E(l) = I = O =$ *all*, and *all* is the PER which relates all program states. The state $\sigma \in \mathbf{\Sigma}$ is represented in Table 5.1 as a pair $(\sigma(h), \sigma(l)) \in [\![\tau_h]\!] \times [\![\tau_l]\!]$. In the table, $\mathbb{V}^+ \triangleq \{i \in \mathbb{Z} | 0 \leq i < n\} \subseteq \mathbb{V}$ is the natural subset of $\mathbb{V}$, and $\mathbb{V}^- \triangleq \mathbb{V} \backslash \mathbb{V}^+$ is the set of negative values that the secret can take. The information configuration $(E_0, I_0, O_0) = (E[l \mapsto R], I, O)$ after the assignment $l := 0$ is the starting configuration for the analysis of the *while* statement and $R \equiv \{\{(j, 0) \mid j \in \mathbb{V}\}\}$. The i^{th} iteration of the *while* analysis starts at the pre-configuration $\varphi_i = (E_i, I_i, O_i)$, and $\varphi_{i+1} = (E_{i+1}, I_{i+1}, O_{i+1})$ is computed as φ_i $(\texttt{if(}h \neq l\texttt{)then}\ l := (l + 1)$ ***mod*** $n\ \texttt{else skip})$ φ_{i+1}. Since there is no *write* statement in the *while* body, we have $O_i = O$ for all i. Similarly, since the *while* body ($l := (l + 1)$ ***mod*** n) itself terminates, we have $I_i = I$ for all i. The post-configuration of the *while* statement is φ', and it is the pre-configuration of the *write* statement.

As Table 5.1 shows, at the n^{th} iteration (and afterwards), the set of initial values for which the program terminates is identified. This set is modelled by

```
l := 0;
while(h ≠ l) do
        l := (l + 1) mod n;
write  l;
```

Figure 5.15: *Linear search using a* while *loop*

```
l := 0;
φ0 = (E0, I0, O0)
     φi = (Ei, Ii, Oi)
while(h ≠ l) do
        l := (l + 1) mod n;
     φi+1
  φ′
write  l;
```

Figure 5.16: *Analysis of the* while *loop*

Analysis iteration (i)	$E_i(l)$
0	$\{\{(j,0) \mid j \in \mathbb{V}\}\}$
1	$\{\{(0,0)\},\ \{(j,1) \mid j \in \mathbb{V}\backslash\{0\}\}\}$
2	$\{\{(k,k)\},\ \{(j,2) \mid j \in \mathbb{V}\backslash\{0,1\}\} \mid k \in \{0,1\}\}$
$\vdots$	$\vdots$
m	$\{\{(k,k)\},\ \{(j,m) \mid j \in \mathbb{V}\backslash\{0,1,\ldots,m-1\}\} \mid k \in \{0,1,\ldots,m-1\}\}$
$\vdots$	$\vdots$
n	$\{\{(k,k)\},\ \{(j,0) \mid j \in \mathbb{V}^-\} \mid k \in \mathbb{V}^+\}$
$n+1$	$\{\{(k,k)\},\ \{(j,1) \mid j \in \mathbb{V}^-\} \mid k \in \mathbb{V}^+\}$
$\vdots$	$\vdots$

Table 5.1: *Analysis of a* while *statement*

the $\{\{(k,k)\} \mid k \in \mathbb{V}^+\}$ equivalence classes of $E_n(l)$. Further iterations after this point is benign because no new state can be produced that has not been previously encountered during the iterative analysis. Thus, $dom(E_n(l)) = dom(E_{n+m}(l))$ for any $m \in \mathbb{N}$.

The post-configuration of the *while* analysis is given by $\varphi' = (E'', I'', O'')$,

where $E''(l) \equiv \{\{(k,k)\} \mid k \in \mathbb{V}^+\}$ and $E''(h) \equiv \{\{(k,m) \mid m \in \mathbb{V}\} \mid k \in \mathbb{V}^+\}$ and $I'' \equiv \{\{(k,m) \mid k \in \mathbb{V}^+, m \in \mathbb{V}\}\}'$ and $O'' \equiv \{\{(k,m) \mid m \in \mathbb{V}\}, \{(j,m) \mid j \in \mathbb{V}^-, m \in \mathbb{V}\} \mid k \in \mathbb{V}^+\}$. The meaning of I'' is that it encodes the set of starting states for which the *while* statement terminates and it sets the context for the analysis of the subsequent commands. The O'' can distinguish between terminating and nonterminating starting states of the program, and can additionally distinguish different terminating traces due to the computation of $\mathit{flow}((h \neq l) : \mathit{id}, \varphi_i)$ at each stage. Notice also that $\mathit{dom}(\varphi') = \{(k,k) \mid k \in \mathbb{V}^+\}$, which is the set of terminating states of the *while* loop.

Finally, by applying the *write* rule to the `write` l statement, using the pre-configuration φ', we obtain the post-configuration (E'', I'', O') where $O' = O'' \uplus \mathit{flow}(l : \mathit{id}, \varphi') \equiv \{\{(k,m) \mid m \in \mathbb{V}\}, \{(j,m) \mid j \in \mathbb{V}^-, m \in \mathbb{V}\} \mid k \in \mathbb{V}^+\}$. Thus, O' reveals the knowledge of h whenever it is positive and also distinguishes states under which the program terminates from those under which it diverges. However, the attacker cannot distinguish between two states that leads to program divergence. More precisely, O' is the PER $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ O'\ \sigma'$ iff $\sigma(h) = \sigma'(h) \in \mathbb{V}^+$ or $\sigma(h), \sigma'(h) \in \mathbb{V}^-$. This agrees with the intuition about the information released by this program.

Note that after the *while* statement, the attacker has already gained the information O' due to the computation of $\mathit{flow}((h \neq l) : \mathit{id}, \varphi_i)$ at each stage. An alternative definition of the *while* rule, which does not suffer from this overap-

proximation, but which only works when the set of states *is finite* is given by

$$\frac{\begin{array}{c}\varphi_0 = \varphi \quad (E_n, I_n, O_n) = \varphi_n \quad \varphi_n \,\texttt{if(}b\texttt{)then}\, c \,\texttt{else skip}\, \varphi_{n+1} \\ F(\varphi_0) \triangleq \varphi_0 \uplus \varphi_1 \quad F(\varphi_{n+1}) \triangleq F(\varphi_n) \uplus \varphi_{n+2} \quad F(\varphi_k) = \mathit{lfp}(F) \quad I' = \mathit{flow}(b : \mathbf{F}, \varphi_k) \\ \forall x \in \mathbf{Var}, E'_k = E_k[x \mapsto E_k(x) \sqcup I'] \quad I'_k = I_k \sqcup I' \quad O'_k = O_k \uplus \mathit{flow}(b : \mathit{id}, \varphi_k)\end{array}}{\varphi \,\texttt{while}\,(b)\,\texttt{do}\, c\,(E'_k, I'_k, O'_k)}$$

This definition requires the existence of a $k \in \mathbb{N}$, after which further iteration of the *while* statement cannot produce any new state. This point is reached at the fixpoint $\mathit{lfp}(F) = \biguplus_{i \geq 0} \varphi_i = F(\varphi_k)$. The post-configuration of the *while* statement is then computed at this point, where the set of states is partitioned by $b : \mathit{id}$ at the k^{th} step only. When applied to the example above, this produces a better result for the *while* analysis, where $O'' \equiv \{\{(k, m) \mid k \in \mathbb{V}^+, m \in \mathbb{V}\}, \{(k, m) \mid k \in \mathbb{V}^-, m \in \mathbb{V}\}\}$. This means that after the *while* loop, the attacker can only distinguish initial states that lead to termination from those under which the loop diverges, but cannot distinguish one state which leads to termination from another one under which the loop also terminates. Using this definition, the attacker's knowledge after the *write* statement is the same as O' derived above.

It should be noted that when *while* statements are used in a program, whose set of states is infinite, the static analysis of Figure 5.12 is not computable in the general case. While the definition of information flow analysis of Figure 5.12 sheds insight into how the analysis of the *while* rule might be performed in this case, abstract interpretation techniques are necessary. The application of abstract interpretation to make information flow analysis more tractable is presented in Chapter 6.

5.6 Static Information Flow Property

We introduced the notion of the *semantic information flow property* of a *While* program in section 4.3 of Chapter 4 as a semantic definition, which describes how an attacker's knowledge is transformed by observing program executions. We now relate this semantic definition to the static analysis of information flow presented in this chapter, showing that the static analysis is sound.

The semantic information flow property of a *While* program P is derived from the equivalence relation $\lfloor T_P \rfloor$, which relates only pairs of input states under which the semantic attacker makes exactly the same observation when P is executed. This information flow property on the lattice $\mathcal{I} = \text{PER}(\mathbf{\Sigma})$ of PERs over the states of P is given by $[\![P]\!]^{\mathcal{I}} = \{f \mid \forall R \in \text{PER}(\mathbf{\Sigma}), f(R) \sqcup \lfloor T_P \rfloor\}$, where f describes how the attacker's knowledge is transformed by observing the program P. We now show that the static analysis of information flow presented in this chapter derives at least the information $\lfloor T_P \rfloor$. Specifically, when the information configuration $(E_\perp, I_\perp, O_\perp)$ (see Definition 5.6.1) is chosen as the pre-configuration of P, and $(E_\perp, I_\perp, O_\perp)\, P\, (E, I, O)$ holds, then we have also that $\lfloor T_P \rfloor \sqsubseteq O$. The definition of the *static information flow property*, $[\![P]\!]^{\mathcal{I}}_{\mathit{static}}$, of P as derived by the static analysis is the following.

Definition 5.6.1 (Static Information Flow Property)**.** *Let P be a* While *program and let $\mathbf{\Sigma}$ and $\mathbf{Var}$ be the set of all states and the set of all variables of P respectively. Define the information configuration $(E_\perp, I_\perp, O_\perp) \in \mathbf{\Phi}$ such that for all $x \in \mathbf{Var}, E_\perp(x) = I_\perp = O_\perp = \mathit{all} \in \mathit{PER}(\mathbf{\Sigma})$. The* static information flow property *of P is defined as $[\![P]\!]^{\mathcal{I}}_{\text{static}} \triangleq \{f \mid (E_\perp, I_\perp, O_\perp)\ P\ (E, I, O), \forall R \in \mathit{PER}(\mathbf{\Sigma}), f(R) = R \sqcup O\}$.*

The information configuration $(E_\perp, I_\perp, O_\perp)$ enjoys a special status because it makes no assumption about the starting state of P since $\mathit{dom}((E_\perp, I_\perp, O_\perp)) = \mathbf{\Sigma}$. Furthermore, the attacker has no prior knowledge about any input to P ($O_\perp = \mathit{all}$) and the initial implicit context $I_\perp = \mathit{all}$ places no constraint on input. Similarly, no constraint is placed on the initial value of variables: $\forall x \in \mathbf{Var}, E_\perp(x) = \mathit{all}$.

The O-component of the post-configuration of the static analysis specifies the information that the attacker might gain from the execution of P. The correctness requirement is therefore that $\lfloor T_P \rfloor \sqsubseteq O$, which means that the information derived by the analysis is at least as much as that gained by the semantic attacker introduced in Chapter 4. The correctness of the static analysis is shown next.

5.7 Correctness of Static Analysis

This section shows the correctness of the static analysis. For the information flow, it shows that the static information flow property derived by the analysis is at least as much information as the semantic information flow property gained by the semantic attacker. Furthermore, with respect to the program semantics, it also shows that the analysis models the transformation of states by the program. We define first, a set of *initial configurations* $\mathbf{\Phi}_{\mathit{init}} \subseteq \mathbf{\Phi}$, elements of which may serve as the pre-configuration of any subprogram during analysis.

Definition 5.7.1. *Define the set $\mathbf{\Phi}_{\mathit{init}} \subseteq \mathbf{\Phi}$ to be the set of all starting configurations, where $\mathbf{\Phi}_{\mathit{init}} \triangleq \{(E, I, O) \in \mathbf{\Phi} \mid \forall z \in \mathbf{TVar}, \bar{Z} = \mathbf{TVar} \backslash \{z\}. \forall y \in \mathbf{IVar}. E(z) = \uparrow_{\bar{Z}} E(z), E(y) = \uparrow_{\mathbf{TVar}} E(y), I = \uparrow_{\mathbf{TVar}} I, O = \uparrow_{\mathbf{TVar}} O\}$.*

It is clear from this definition that $(E_\perp, I_\perp, O_\perp) \in \mathbf{\Phi}_{\mathit{init}}$. A consequence of Definition 5.2.1, where all variables are properly-initialised before use, is that

it confers a property on the PERs constructed during the analysis such that the partitioning of inputs by the PERs, and thus their information content about inputs, is preserved by the operation $\uparrow_Z(\cdot)$, where $Z \subseteq \mathbf{TVar}$ (see (2) of lemma 5.7.3). We identify the properties of such PERs ($\mathcal{R}_{init}$) in Definition 5.7.2 and lemma 5.7.3 highlights some of the consequences.

Definition 5.7.2. *Define* $\mathcal{R}_{\text{init}} \triangleq \{R \in PER(\mathbf{\Sigma}) \mid X \subseteq \mathbf{TVar}.Y = \mathbf{TVar}\backslash X, \forall \sigma, \sigma' \in dom(R), havocY([\sigma]_R) = [\sigma]_R, \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}\}$.

Lemma 5.7.3. *Let* $Z \subseteq \mathbf{TVar}$*, and let* e *be an expression such that* $FV(e) \subseteq \mathbf{Var}$*, and let* $R, R' \in \mathcal{R}_{\text{init}} \subseteq PER(\mathbf{\Sigma})$.

1. *For all* $\sigma \in dom(R)$*,* $havocZ([\sigma]_R) = [\sigma]_{\uparrow_Z R}$.
2. *For all* $\sigma, \sigma' \in dom(R), \sigma \uparrow_Z R\, \sigma' \implies \sigma\, R\, \sigma'$.
3. $R \sqcup R' \in \mathcal{R}_{\text{init}}$.
4. *Let* $X \subseteq \mathbf{TVar}$ *and* $Y = \mathbf{TVar}\backslash X$ *such that* $\forall \sigma, \sigma' \in dom(R), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$ *and* $havocY([\sigma]_R) = [\sigma]_R$*. Furthermore, suppose* $FV(e) \cap \mathbf{TVar} \subseteq X$*. Then for any PER* ϕ *over the values of* e*, we have that* $e : \phi \sqcup R \in \mathcal{R}_{\text{init}}$.

Proof. See Appendix A. □

Lemma 5.7.4. *Let* $(E, I, O) \in \mathbf{\Phi}_{init}$*, and suppose the program* `while` (b) `do` c *does not assign to* $\mathbf{IVar}$ *variables, where all variables are properly-initialised before use. Then there exists a unique* (E', I', O')*, so that* (E, I, O) `while` (b) `do` c (E', I', O') *holds and* $(E', I', O') \in \mathbf{\Phi}_{init}$.

Proof. The derivations in the flow rules are syntax-directed. However, it is not immediately clear that we have the desired property for the *while* rule. In particular, we first note that the limit $\bigsqcup^{\boxplus}_{i\geq 0} \varphi_i$ exists, since $\langle \mathbf{\Phi}, \sqsubseteq^{\boxplus}, \sqcup^{\boxplus} \rangle$ is a complete lattice. It now remains to show that the E-component of the *while* post-configuration is a map from variables to PERs, which have the desired properties.

Firstly, suppose $(E_j, I_j, O_j)\texttt{if(}b\texttt{)then}\, c\, \texttt{else skip}(E_{j+1}, I_{j+1}, O_{j+1})$ holds, for some $(E_j, I_j, O_j) = \varphi_j \in \mathbf{\Phi}_{\mathit{init}}$ during the iterative analysis of the statement $\texttt{while}\,(b)\,\texttt{do}\,c$. Now take any $x \in \mathbf{Var}$. We want to show that the property $(E_{j+1}(x) \sqcup I''_{j+1} \sqcup b : \mathbf{F}) \sqsubseteq (E_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$ holds, where $I''_j = \mathit{flow}(b : \mathbf{F}, \varphi_j)$ and $I''_{j+1} = \mathit{flow}(b : \mathbf{F}, (E_{j+1}, I_{j+1}, O_{j+1}))$. Furthermore, let $I'_j = \mathit{flow}(b : \mathbf{T}, \varphi_j)$ so that the analysis of c is given by $(E_j, I'_j, O_j)\; c\; (E'_{j+1}, I'_{j+1}, O'_{j+1})$ according to the *if* rule. The post-configuration of the *else* branch is therefore (E_j, I''_j, O_j), being a *skip* statement.

We first observe, from Definition 5.5.1, that $\sigma_1\; I_{j+1}\; \sigma_2$ iff $\sigma_1, \sigma_2 \in \mathit{dom}(I''_j) \cup \mathit{dom}(I'_{j+1})$, and hence we have that $I_{j+1} \sqsubseteq I''_j$. Now, since variables are properly-initialised before use, and the $\mathbf{IVar}$ projection of states are not modified then I'_j and I''_j are disjoint PERs. To see why, let $Y = FV(b)$ and define R_{E_j} such that $\sigma\; R_{E_j}\; \sigma'$ iff $\sigma, \sigma' \in \mathit{dom}(\bigsqcup_{y \in Y} E_j(y))$. Since variables are properly-initialised before use, and the $\mathbf{IVar}$ projection of states is not modified, we have that for any $y \in Y$ and i, and for all $\sigma, \sigma' \in \mathit{dom}(E_i(y))$, $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma(y) = \sigma'(y)$ (this is shown in Lemma 5.7.5). Hence, for all $\sigma, \sigma' \in \mathit{dom}(R_{E_j})$, $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow Y} = \sigma'_{\downarrow Y} \implies \sigma(b) = \sigma'(b)$. Hence, by the contrapositive, $\sigma(b) \neq \sigma'(b) \implies \sigma_{\downarrow \mathbf{IVar}} \neq \sigma'_{\downarrow \mathbf{IVar}}$. This means that the PERs, $R_A = b : \mathbf{T} \sqcup I_j \sqcup R_{E_j}$ and $R_B = b : \mathbf{F} \sqcup I_j \sqcup R_{E_j}$ are disjoint since R_A restricts the domain of R_{E_j} to those where b is *true*, whereas R_B restricts the domain of R_{E_j} to those where

b is *false*. Hence $I'_j = \uparrow_{\mathbf{TVar}} R_A$ and $I''_j = \uparrow_{\mathbf{TVar}} R_B$ are disjoint PERs, since the operation $\uparrow_{\mathbf{TVar}}(\cdot)$ does not modify the **IVar** projection of states. Now according to Definition 5.5.1, for the merging of the post-configuration of the conditional *if* statement, we have that for any $z \in \mathbf{Var}$, $E_{j+1}(z) = E_j(z) \uplus E_j(z) = E_j(z)$ if $E_j(z) = E'_{j+1}(z)$, or $E_{j+1}(z) = (E_j(z) \sqcup I''_j) \uplus E'_{j+1}(z)$ if $E_j(z) \neq E'_{j+1}(z)$. In the latter case, we further note that since $E'_{j+1}(z)$ has been modified within the branch c, which is predicated on the implicit context I'_j, then $\mathit{dom}(E'_{j+1}(z)) \subseteq \mathit{dom}(I'_j)$ and hence $(E_j(z) \sqcup I''_j)$ and $E'_{j+1}(z)$ are disjoint PERs, since $I'_j \sqcup I''_j = \varnothing$. Hence, in this case, $E_{j+1}(z)$ is simply the disjoint union of PERs: namely that, $E_{j+1}(z) = (E_j(z) \sqcup I''_j) \cup E'_{j+1}(z)$ by the definition of $\uplus$. Thus, in both cases, we have that $E_{j+1}(z) \sqsubseteq E_j(z) \sqcup I''_j$.

Now for the proof of $(E_{j+1}(x) \sqcup I''_{j+1} \sqcup b : \mathbf{F}) \sqsubseteq (E_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$, there are two cases to consider according to Definition 5.5.1, which depends on whether $E'_{j+1}(x)$ is different from $E_j(x)$ or not.

- Case 1: Suppose $E_j(x) = E'_{j+1}(x)$. In this case we have, according to Definition 5.5.1 that $E_{j+1}(x) = E_j(x) \uplus E_j(x) = E_j(x)$. It thus remains only to show that $I''_{j+1} \sqsubseteq I''_j$ in the case. Now define $R_{E_{j+1}}$ such that $\sigma \; R_{E_{j+1}} \; \sigma'$ iff $\sigma, \sigma' \in \mathit{dom}(\bigsqcup_{y \in FV(b)} E_{j+1}(y))$ and let $R_C = b : \mathbf{F} \sqcup I_{j+1} \sqcup R_{E_{j+1}}$. Hence, $I''_j = \uparrow_{\mathbf{TVar}} R_C$. Since for all $z \in \mathbf{Var}$, $E_{j+1}(z) \sqsubseteq E_j(z) \sqcup I''_j$, then it follows that $R_{E_{j+1}} \sqsubseteq R_{E_j} \sqcup I''_j$. Now, since $I''_j = \uparrow_{\mathbf{TVar}} R_B$, then $I''_j \sqsubseteq R_B$ by definition, which means also that $I_{j+1} \sqsubseteq R_B$ since $I_{j+1} \sqsubseteq I''_j$. Therefore, $R_{E_{j+1}} \sqcup b : \mathbf{F} \sqsubseteq R_{E_j} \sqcup I''_j \sqcup b : \mathbf{F} \sqsubseteq R_{E_j} \sqcup R_B \sqcup b : \mathbf{F} = R_B$, and hence $R_{E_{j+1}} \sqcup I_{j+1} \sqcup b : \mathbf{F} = R_C \sqsubseteq R_B$. Since $R_C \sqsubseteq R_B$, then by applying (8) of lemma 5.4.8, we have $\uparrow_{\mathbf{TVar}} R_C \sqsubseteq \uparrow_{\mathbf{TVar}} R_B$, that is, $I''_{j+1} \sqsubseteq I''_j$. This shows the first case.

- Case 2: Suppose $E_j(x) \neq E'_{j+1}(x)$. Then, as shown above, $E_{j+1}(x)$ is the disjoint union $E_{j+1}(x) = (E_j(x) \sqcup I''_j) \cup E'_{j+1}(x)$, hence $E_{j+1}(x) \sqsubseteq E_j(x) \sqcup I''_j$. Therefore, $E_{j+1} \sqcup I''_{j+1} \sqcup b : \mathbf{F} \sqsubseteq (E_j(x) \sqcup I''_j) \sqcup I''_{j+1} \sqcup b : \mathbf{F} = E_j(x) \sqcup I''_j \sqcup b : \mathbf{F}$, since $I''_{j+1} \sqsubseteq I''_j$. This shows the desired property.

Now define $\bar{X} = \mathbf{TVar} \backslash \{x\}$. Since we have that $(E_{j+1}(x) \sqcup I''_{j+1} \sqcup b : \mathbf{F}) \sqsubseteq (E_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$, then by applying (8) of lemma 5.4.8 we obtain the fact that $\uparrow_{\bar{X}}(E_{j+1}(x) \sqcup I''_{j+1} \sqcup b : \mathbf{F}) \sqsubseteq \uparrow_{\bar{X}}(E_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$. Hence, for the *while* rule, we have that for any $j, k \in \mathbb{N}$ such that $j \leq k$, then $\uparrow_{\bar{X}}(E_k(x) \sqcup I''_k \sqcup b : \mathbf{F}) \sqsubseteq \uparrow_{\bar{X}}(E_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$ by the transitivity of $\sqsubseteq$. Thus, for any $x \in \mathbf{Var}$, such that $\sigma \; E'(x) \; \sigma'$ and $\sigma' \; E'(x) \; \sigma''$ hold, we know that there exist $j, k \in \mathbb{N}$, such that $\sigma \uparrow_{\bar{X}}(E_j(x) \sqcup I''_j \sqcup b : \mathbf{F}) \; \sigma'$ and $\sigma' \uparrow_{\bar{X}}(E_k(x) \sqcup I''_k \sqcup b : \mathbf{F}) \; \sigma''$. If $j \leq k$, then we know from above that $\sigma \uparrow_{\bar{X}}(E_j(x) \sqcup I''_j \sqcup b : \mathbf{F}) \sigma' \implies \sigma \uparrow_{\bar{X}}(E_k(x) \sqcup I''_k \sqcup b : \mathbf{F}) \sigma'$, and therefore $\sigma \uparrow_{\bar{X}}(E_k(x) \sqcup I''_k \sqcup b : \mathbf{F}) \; \sigma''$ holds also. That is, $\sigma \; E'(x) \; \sigma''$ holds. Since $\uparrow_{\bar{X}}(E_k(x) \sqcup I''_k \sqcup b : \mathbf{F})$ is a PER, then so also is $E'(x)$. Furthermore, since $\bar{X} = \mathbf{IVar} \backslash \{x\}$, it is easy to see that the post-configuration $(E', I', O') \in \mathbf{\Phi}_{init}$. □

Because of the property that all variables in a program P are properly-initialised before use in P, and the fact that the $\mathbf{IVar}$ projection of states are not modified by P, on termination, the value of a variable that is properly-initialised during the execution of P is determined only by the value of the inputs to P.

Lemma 5.7.5. *Suppose* $(E_1, I_1, O_1) \in \mathbf{\Phi}_{init}$ *is an information configuration of the* While *program* P*, which does not use* $\mathbf{IVar}$ *variables on the left-hand-side of assignment and where all variables are properly-initialised before use. If* $(E_1, I_1, O_1) \; P \; (E_2, I_2, O_2)$ *holds, then for any variable* $x \in \mathbf{Var}$*, which has*

been properly-initialised in P *we have that* $E_2(x) \in \mathcal{R}_{\text{init}}$*, and for all* $\sigma, \sigma' \in$ *dom*$(E_2(x)), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma(x) = \sigma'(x)$.

Proof. The proof proceeds by induction on the derivation tree of $(E_1, I_1, O_1)P(E_2, I_2, O_2)$ according to the information flow rules. The E-component of information configurations are modified only during assignments, and in the analysis of conditional *if* and *while* statements.

We shall first show that the desired property holds after assignment statements. Let $(E, I, O)\, x := e\, (E', I', O')$ be the analysis of the assignment $x := e$ in P. Furthermore, let $X = \{x\}$ and let $\overline{X} = \mathbf{TVar} \backslash X$. Then $E' = E[x \mapsto \mathit{aflow}(x := e, (E, I, O))]$, where $\mathit{aflow}(x := e, (E, I, O)) = \uparrow_{\overline{X}} \overset{x:=e}{\rightsquigarrow} R$, and $R = e : id \sqcup I \sqcup R_E$, and $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\, R_E\, \sigma' \iff \sigma, \sigma' \in \mathit{dom}(\bigsqcup_{z \in FV(e)} E(z))$. Now let $Z = FV(e) \cap \mathbf{TVar}$. Since $\mathbf{TVar}$ variables are properly-initialised before use, then we have by the induction hypothesis that for any $z \in Z, E(z) \in \mathcal{R}_{init}$ and for all $\sigma, \sigma' \in \mathit{dom}(E(z)), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma(z) = \sigma'(z)$. Thus, for all $\sigma, \sigma' \in \mathit{dom}(R_E), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow Z} = \sigma'_{\downarrow Z}$, and therefore for any $\sigma, \sigma' \in \mathit{dom}(R), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow Z} = \sigma'_{\downarrow Z}$ since $\mathit{dom}(R) \subseteq \mathit{dom}(R_E)$.

Now by definition $\mathit{dom}(\overset{x:=e}{\rightsquigarrow} R) = \{\sigma[x \mapsto \sigma(e)] \mid \sigma \in \mathit{dom}(R)\}$, therefore for all $\sigma, \sigma' \in \mathit{dom}(\overset{x:=e}{\rightsquigarrow} R), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma(x) = \sigma'(x)$ because all states in the domain of R, which agree on the $\mathbf{IVar}$ projection also agree on the Z projection, and hence on the evaluation of e - since $FV(e) \subseteq \mathbf{IVar} \cup Z$. Now let $R' = \overset{x:=e}{\rightsquigarrow} R$. We know by (4) of lemma 5.4.8, for all $\sigma \in \mathit{dom}(\uparrow_{\overline{X}} R'), \mathit{havoc}\overline{X}([\sigma]_{\uparrow_{\overline{X}} R'}) = [\sigma]_{\uparrow_{\overline{X}} R'}$. Since $x \notin \overline{X}$ and $\mathbf{IVar} \cap \overline{X} = \varnothing$, then $\mathit{havoc}\overline{X}(\mathit{dom}(\overset{x:=e}{\rightsquigarrow} R)) = \mathit{dom}(\uparrow_{\overline{X}} \overset{x:=e}{\rightsquigarrow} R)$ does not modify the $\mathbf{IVar}$ or x projections of states in $\mathit{dom}(\overset{x:=e}{\rightsquigarrow} R)$, it is thus clear that $\uparrow_{\overline{X}} \overset{x:=e}{\rightsquigarrow} R \in \mathcal{R}_{init}$. Hence, $E'(x) = \uparrow_{\overline{X}} \overset{x:=e}{\rightsquigarrow} R \in \mathcal{R}_{init}$ and for all $\sigma, \sigma' \in \mathit{dom}(E'(x)), \sigma_{\downarrow \mathbf{IVar}} =$

$\sigma'_{\downarrow\mathbf{IVar}} \implies \sigma(x) = \sigma'(x)$.

During the analysis of `if` (b) `then` c_1 `else` c_2, starting from the pre-configuration (E, I, O), the post-configuration of c_1 is derived as $(E, I', O)\ c_1\ (E_{c_1}, I_{c_1}, O_{c_1})$, where $I' = \mathit{flow}(b : \mathbf{T}, (E, I, O))$. Similarly, the post-configuration of c_2 is obtained as $(E, I'', O)\ c_2\ (E_{c_2}, I_{c_2}, O_{c_2})$, where $I'' = \mathit{flow}(b : \mathbf{F}, (E, I, O))$. For any variable x, both $E_{c_1}(x)$ and $E_{c_2}(x)$ have the desired property by applying the induction hypothesis to c_1 and c_2 respectively. Let $(E', I', O') = (E_{c_1}, I_{c_1}, O_{c_1}) \uplus_{(E,I,O)} (E_{c_2}, I_{c_2}, O_{c_2})$ be the post-configuration of the *if* statement. Now suppose that x is assigned in c_1, then $\mathit{dom}(E_{c_1}(x)) \subseteq \mathit{dom}(I')$ since x is assigned in c_1, which is in the scope of an implicit context, whose domain is smaller than the domain of I'. Similarly, if x is assigned within c_2, then $\mathit{dom}(E_{c_2}(x)) \subseteq \mathit{dom}(I'')$. Now since variables are properly-initialised before use, then for all $y \in FV(b), E(y) \in \mathcal{R}_{\mathit{init}}$, and for all $\sigma, \sigma' \in \mathit{dom}(E(y))$, $\sigma_{\downarrow\mathbf{IVar}} = \sigma'_{\downarrow\mathbf{IVar}} \implies \sigma(y) = \sigma'(y)$ since $\mathbf{IVar}$ variables are not assigned to and the value of variables in $\mathbf{TVar} \cap FV(b)$, being properly-initialised, are functions of $\mathbf{IVar}$ projection of states. If we now define R_E such that $\sigma\ R_E\ \sigma'$ iff $\sigma, \sigma' \in \mathit{dom}(\bigsqcup_{y \in FV(b)} E(y))$, where $Y = FV(b) \cap \mathbf{TVar}$, then for all $\sigma, \sigma' \in \mathit{dom}(R_E)$, $\sigma_{\downarrow\mathbf{IVar}} = \sigma'_{\downarrow\mathbf{IVar}} \implies \sigma_{\downarrow Y} = \sigma'_{\downarrow Y} \implies \sigma(b) = \sigma'(b)$, which by the contrapositive means that $\sigma(b) \neq \sigma'(b) \implies \sigma_{\downarrow\mathbf{IVar}} \neq \sigma'_{\downarrow\mathbf{IVar}}$. Hence, $I' = \mathit{flow}(b : \mathbf{T}, (E, I, O))$ and $I'' = \mathit{flow}(b : \mathbf{F}, (E, I, O))$ are disjoint PERs, since $\sigma \in \mathit{dom}(b : \mathbf{T} \sqcup I \sqcup R_E)$ and $\sigma' \in \mathit{dom}(b : \mathbf{F} \sqcup I \sqcup R_E)$ implies $\sigma(b) \neq \sigma'(b)$, which in turn means that $\sigma_{\downarrow\mathbf{IVar}} \neq \sigma'_{\downarrow\mathbf{IVar}}$. Since $\mathit{dom}(I') = \mathit{havoc}\mathbf{IVar}(\mathit{dom}(b : \mathbf{T} \sqcup I \sqcup R_E))$ and $\mathit{dom}(I'') = \mathit{havoc}\mathbf{IVar}(\mathit{dom}(b : \mathbf{F} \sqcup I \sqcup R_E))$, both of which do not modify $\mathbf{IVar}$ variables, $I' \sqcup I'' = \varnothing$. Hence, $E_{c_1}(x)$ and $E_{c_2}(x)$ are disjoint PERs, and therefore $E'(x) = E_{c_1}(x) \uplus E_{c_2}(x)$ has the desired property.

Now suppose that x is assigned in only one branch, say c_1 (the other case for c_2 is symmetrical). Then, again, $\mathit{dom}(E_{c_1}(x)) \subseteq \mathit{dom}(I')$. There are two cases to consider according to Definition 5.5.1. Either $E_{c_1}(x) = E(x)$, in which case $E'(x) = E(x) = E_{c_1}(x)$, and therefore $E'(x)$ has the desired property by the induction hypothesis on c_1. In the second case, $E_{c_1}(x) \neq E(x)$, in which case $E'(x) = E_{c_1}(x) \uplus (E(x) \sqcup I'')$ and since x must be properly-initialised, it must be initialised before control is passed to the *else* branch, and hence $E(x)$ has the desired property, which means that $E'(x)$ has the desired property since $E_{c_1}(x)$ and $E(x) \sqcup I''$ are disjoint PERs.

In the iterative analysis of the command $\texttt{while}\,(b)\,\texttt{do}\,c$, the E-component of the post-configuration is computed from a sequence $(E'_0, I'_0, O'_0), (E'_1, I'_1, O'_1), \cdots$, where for all $i \geq 0$, (E'_i, I'_i, O'_i) $\texttt{if(}b\texttt{)then}\ c\ \texttt{else skip}$ $(E'_{i+1}, I'_{i+1}, O'_{i+1})$, and hence by induction on $\texttt{if(}b\texttt{)then}\ c\ \texttt{else skip}$, $E'_i(x)$ has the desired property for all i. The E-component of the post-configuration of the *while* statement is computed such that $\bar{X} = \mathbf{TVar}\backslash\{x\}, \forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ E'(x)\ \sigma' \iff \exists j \in \mathbb{N}, \sigma{\uparrow_{\bar{X}}}(E'_j(x) \sqcup I''_j \sqcup b : \mathbf{F})\sigma'$, where for any $k \in \mathbb{N}$, $I''_k = \mathit{flow}(b : \mathbf{F}, (E'_k, I'_k, O'_k))$. However, for any $j \in \mathbb{N}$, $E'_j(x) \in \mathcal{R}_{\mathit{init}}$ and $\forall \sigma, \sigma' \in \mathit{dom}(E'_j(x)), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma(x) = \sigma'(x)$ by the induction hypothesis, it is thus clear that this property is preserved in ${\uparrow_{\bar{X}}}(E'_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$ since ${\uparrow_{\bar{X}}}(\cdot)$ does not modify the $\mathbf{IVar} \cup \{x\}$ projection of states. Furthermore, ${\uparrow_{\bar{X}}}(E'_j(x) \sqcup I''_j \sqcup b : \mathbf{F}) \in \mathcal{R}_{\mathit{init}}$ by definition. We have already shown in lemma 5.7.4 that for all $j, k \in \mathbb{N}$, such that $j \leq k$, then ${\uparrow_{\bar{X}}}(E'_k(x) \sqcup I''_k \sqcup b : \mathbf{F}) \sqsubseteq {\uparrow_{\bar{X}}}(E'_j(x) \sqcup I''_j \sqcup b : \mathbf{F})$. Thus, for any $\sigma, \sigma' \in \mathit{dom}(E'(x))$, there exist $j, k \in \mathbb{N}$, such that $k \geq j$ and $\sigma \in \mathit{dom}({\uparrow_{\bar{X}}}(E'_j(x) \sqcup I''_j \sqcup b : \mathbf{F}))$ and $\sigma' \in {\uparrow_{\bar{X}}}(E'_k(x) \sqcup I''_k \sqcup b : \mathbf{F})$. Since $\mathit{dom}({\uparrow_{\bar{X}}}(E'_j(x) \sqcup I''_j \sqcup b : \mathbf{F})) \subseteq {\uparrow_{\bar{X}}}(E'_k(x) \sqcup I''_k \sqcup b : \mathbf{F})$ and ${\uparrow_{\bar{X}}}(E'_k(x) \sqcup I''_k \sqcup b : \mathbf{F}))$ has the desired property, then we are done. $\square$

We now show that composing the analysis of sequential programs correctly approximates the information released by the sequenced program.

Proposition 5.7.6. *Let $P = P_1; P_2$ be a* While *program. Define the PER $\lfloor T_{P_1 \bullet P_2} \rfloor$ to be $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \lfloor T_{P_1 \bullet P_2} \rfloor \sigma'$ iff $\sigma \lfloor T_{P_1} \rfloor \sigma'$ and if $\langle P_1, \sigma \rangle \Downarrow \sigma_1$ and $\langle P_1, \sigma' \rangle \Downarrow \sigma_1'$ then $\sigma_1 \lfloor T_{P_2} \rfloor \sigma_1'$. Then we have $\lfloor T_P \rfloor \sqsubseteq \lfloor T_{P_1 \bullet P_2} \rfloor$.*

Proof. Take any $\sigma, \sigma' \in \mathbf{\Sigma}$ such that $\sigma \lfloor T_{P_1 \bullet P_2} \rfloor \sigma'$ holds. Then, $\sigma \lfloor T_{P_1} \rfloor \sigma'$ holds, which by lemma 4.3.2 means that either P_1 terminates under both states σ and σ' or that it diverges under both states, and that the attacker makes the same observation on the traces of the two states, that is, $\mathit{obs}(t_{\langle P_1, \sigma \rangle}) = \mathit{obs}(t_{\langle P_1, \sigma' \rangle})$.

In the first case, suppose that P_1 diverges under both σ and σ', then we have $\mathit{obs}(t_{\langle P_1, \sigma \rangle}) = \mathit{obs}(t_{\langle P_1, \sigma' \rangle}) = \mathit{obs}(t_{\langle P, \sigma \rangle}) = \mathit{obs}(t_{\langle P, \sigma' \rangle})$ since the trailing subprogram P_2 of P cannot be executed due to the divergence of P_1, and hence $\sigma \lfloor T_P \rfloor \sigma'$ holds.

Now suppose P_1 terminates under both σ and σ', then $\sigma \lfloor T_{P_1 \bullet P_2} \rfloor \sigma'$ implies that $\sigma \lfloor T_{P_1} \rfloor \sigma'$ holds, and there exist $\sigma_1, \sigma_1' \in \mathbf{\Sigma}$ such that $\langle P_1, \sigma \rangle \Downarrow \sigma_1$ and $\langle P_1, \sigma' \rangle \Downarrow \sigma_1'$ and $\sigma_1 \lfloor T_{P_2} \rfloor \sigma_1'$ holds. Thus, $\mathit{obs}(t_{\langle P_1, \sigma \rangle}) = \mathit{obs}(t_{\langle P_1, \sigma' \rangle})$ and $\mathit{obs}(t_{\langle P_2, \sigma_1 \rangle}) = \mathit{obs}(t_{\langle P_2, \sigma_1' \rangle})$, and therefore, $\mathit{obs}(t_{\langle P, \sigma \rangle}) = \mathit{obs}(t_{\langle P, \sigma' \rangle})$, which means that $\sigma \lfloor T_P \rfloor \sigma'$ holds. Thus, $\lfloor T_P \rfloor \sqsubseteq \lfloor T_{P_1 \bullet P_2} \rfloor$. □

Lemma 5.7.7. *Let $P = P_1; P_2$ be a* While *program, which does not modify the* **IVar***-projection of states. Define the PER $\lfloor T_{P_1 \bullet P_2} \rfloor$ to be $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \lfloor T_{P_1 \bullet P_2} \rfloor \sigma'$ iff $\sigma \lfloor T_{P_1} \rfloor \sigma'$ and if $\langle P_1, \sigma \rangle \Downarrow \sigma_1$ and $\langle P_1, \sigma' \rangle \Downarrow \sigma_1'$ then $\sigma_1 \lfloor T_{P_2} \rfloor \sigma_1'$. Furthermore, let $\Sigma, \Sigma_1 \subseteq \mathbf{\Sigma}$ such that $\{\sigma' \mid \sigma \in \Sigma, \langle P_1, \sigma \rangle \Downarrow \sigma'\} \subseteq \Sigma_1$. Define R_Σ and R_{Σ_1} as the PERs $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_\Sigma \; \sigma' \iff \sigma, \sigma' \in \Sigma$ and $\sigma \; R_{\Sigma_1} \; \sigma' \iff \sigma, \sigma' \in \Sigma_1$. Suppose $O_1, O_2 \in \mathit{PER}(\mathbf{\Sigma})$ are PERs such that $\Sigma \subseteq \mathit{dom}(O_1)$ and $O_1 \Subset O_2$ and $\uparrow_{\mathbf{TVar}} O_2 = O_2$,*

and $R_\Sigma \sqcup \lfloor T_{P_1} \rfloor \sqsubseteq R_\Sigma \sqcup O_1$ *and* $R_{\Sigma_1} \sqcup \lfloor T_{P_2} \rfloor \sqsubseteq R_{\Sigma_1} \sqcup O_2$. *Then*

1. $R_\Sigma \sqcup \lfloor T_{P_1 \bullet P_2} \rfloor \sqsubseteq R_\Sigma \sqcup O_2$,
2. $R_\Sigma \sqcup \lfloor T_P \rfloor \sqsubseteq R_\Sigma \sqcup O_2$.

Proof.

1. Define the set of all states under which the subprogram P_1 terminates to be $\boldsymbol{\Sigma}_\Downarrow = \{\sigma \in \boldsymbol{\Sigma} \mid \langle P_1, \sigma \rangle \Downarrow \sigma'\}$ and define $f : \boldsymbol{\Sigma}_\Downarrow \to \boldsymbol{\Sigma}$ to model this transformation of states by P_1 such that for all $\sigma \in \boldsymbol{\Sigma}_\Downarrow$, $f(\sigma) = \sigma'$ if $\langle P_1, \sigma \rangle \Downarrow \sigma'$ - which maps a starting state under which P_1 terminates to the terminating state. Now define the equivalence relation $R \in \text{PER}(\boldsymbol{\Sigma})$ such that $\forall \sigma_1, \sigma_2 \in \boldsymbol{\Sigma}, \sigma_1 \; R \; \sigma_2$ iff $\sigma_1, \sigma_2 \in \boldsymbol{\Sigma} \backslash \boldsymbol{\Sigma}_\Downarrow$ and $\sigma_1, \sigma_2 \in \boldsymbol{\Sigma}_\Downarrow \implies f(\sigma_1) \lfloor T_{P_2} \rfloor f(\sigma_2)$. Hence, we have $\lfloor T_{P_1 \bullet P_2} \rfloor = \lfloor T_{P_1} \rfloor \sqcup R$.

   Hence we wish to show that $R_\Sigma \sqcup \lfloor T_{P_1} \rfloor \sqcup R \sqsubseteq R_\Sigma \sqcup O_2$, that is, for all $\sigma, \sigma' \in \Sigma, \sigma \; O_2 \; \sigma' \implies \sigma \; (\lfloor T_{P_1} \rfloor \sqcup R) \; \sigma'$. Now suppose $\sigma, \sigma' \in \Sigma$, such that $\sigma \; O_2 \; \sigma'$ holds. It then follows that $\sigma \lfloor T_{P_1} \rfloor \sigma'$ holds because $R_\Sigma \sqcup \lfloor T_{P_1} \rfloor \sqsubseteq R_\Sigma \sqcup O_1$ and by (4) of proposition 5.4.5 $R_\Sigma \sqcup O_1 \sqsubseteq R_\Sigma \sqcup O_2$, since $\Sigma = \mathit{dom}(R_\Sigma) \subseteq \mathit{dom}(O_1)$ and $O_1 \Subset O_2$. It now remains to show that $\sigma \; R \; \sigma'$ also holds. But $\sigma \lfloor T_{P_1} \rfloor \sigma'$ implies $\sigma, \sigma' \in \boldsymbol{\Sigma}_\Downarrow$ or $\sigma, \sigma' \in \boldsymbol{\Sigma} \backslash \boldsymbol{\Sigma}_\Downarrow$ by lemma 4.3.2. We now show that $\sigma \; R \; \sigma'$ holds under these two possibilities.

   - Suppose $\sigma, \sigma' \in \boldsymbol{\Sigma} \backslash \boldsymbol{\Sigma}_\Downarrow$, then $\sigma \; R \; \sigma'$ holds by definition.
   - Now suppose $\sigma, \sigma' \in \boldsymbol{\Sigma}_\Downarrow$. Since P does not modify the $\mathbf{IVar}$-projection of states, then for any $\hat{\sigma} \in \boldsymbol{\Sigma}_\Downarrow, f(\hat{\sigma}) \in \mathit{havoc}\mathbf{TVar}(\{\hat{\sigma}\})$. Hence, since $\uparrow_{\mathbf{TVar}} O_2 = O_2$, then by (4) of lemma 5.4.8, $\hat{\sigma} \in \mathit{dom}(O_2) \implies$

havoc$\mathbf{TVar}([\hat{\sigma}]_{O_2}) = [\hat{\sigma}]_{O_2}$. Therefore, $\sigma\, O_2\, \sigma' \implies f(\sigma)\, O_2\, f(\sigma')$. Now $\sigma, \sigma' \in \Sigma \cap \mathbf{\Sigma}_{\Downarrow}$ implies $f(\sigma), f(\sigma') \in \Sigma_1$ and hence $f(\sigma) R_{\Sigma_1} f(\sigma')$. Furthermore, since $f(\sigma) O_2 f(\sigma')$ holds then we know that $f(\sigma) \lfloor T_{P_2} \rfloor f(\sigma')$ holds because $R_{\Sigma_1} \sqcup \lfloor T_{P_2} \rfloor \sqsubseteq R_{\Sigma_1} \sqcup O_2$. Therefore, $\sigma\, R\, \sigma'$ holds.

This shows the required property: $R_\Sigma \sqcup \lfloor T_{P_1 \bullet P_2} \rfloor \sqsubseteq R_\Sigma \sqcup O_2$.

2. The proof is immediate since by proposition 5.7.6 $\lfloor T_P \rfloor \sqsubseteq \lfloor T_{P_1 \bullet P_2} \rfloor$, hence we have that $R_\Sigma \sqcup \lfloor T_P \rfloor \sqsubseteq R_\Sigma \sqcup \lfloor T_{P_1 \bullet P_2} \rfloor \sqsubseteq R_\Sigma \sqcup O_2$.

□

Lemma 5.7.8. *Let* $W \triangleq$ `while` (b) `do` c *be a* while *statement, and let* $C \triangleq$ `if(`b`)then` c `else skip`*. Define the set of starting states under which only the outer* while *loop diverges (excluding those under which* c *diverges) as* $\mathbf{\Sigma}^W_{\Uparrow} \triangleq \{\sigma_0 \in \mathbf{\Sigma} \mid \forall i \in \mathbb{N}, \langle C, \sigma_i \rangle \Downarrow \sigma_{i+1}, \sigma_{i+1}(b) = \mathbf{tt}\}$.

Now define the PER $\overline{W} \in \mathit{PER}(\mathbf{\Sigma})$ *such that for all* $\sigma_0, \sigma'_0 \in \mathbf{\Sigma}, \sigma_0\, \overline{W}\, \sigma'_0$ *iff (*$\sigma_0, \sigma'_0 \in \mathbf{\Sigma} \backslash \mathbf{\Sigma}^W_{\Uparrow}$ *or* $\sigma_0, \sigma'_0 \in \mathbf{\Sigma}^W_{\Uparrow}$*) and* $\sigma_0 \lfloor T_C \rfloor \sigma'_0$ *and* $\forall i, \langle C, \sigma_i \rangle \Downarrow \sigma_{i+1}, \langle C, \sigma'_i \rangle \Downarrow \sigma'_{i+1} \implies \sigma_{i+1} \lfloor T_C \rfloor \sigma'_{i+1}$*. Then we have* $\lfloor T_W \rfloor \sqsubseteq \overline{W}$.

Proof. The proof shows that $\overline{W}$ contains at least as much information as released by the *while* statement W. The PER $\overline{W}$ requires that an indistinguishable pair of states must be stepwise indistinguishable for each possible iteration step of W, and that the pair must both either belong to the set $\mathbf{\Sigma} \backslash \mathbf{\Sigma}^W_{\Uparrow}$ or $\mathbf{\Sigma}^W_{\Uparrow}$ - distinguishing states in which the outer *while* loop terminates from those in which it diverges. The proof is similar to that of proposition 5.7.6 by considering the sequence of the program C as an unwinding of W into its iteration steps. In the following σ_i (resp. σ'_i) is derived by i-step application of C to $\sigma_0 \in \mathbf{\Sigma}$ (resp. $\sigma'_0 \in \mathbf{\Sigma}$).

From lemma 4.3.2 we know that for any $\sigma, \sigma' \in \mathbf{\Sigma}$, such that $\sigma \lfloor T_C \rfloor \sigma'$ holds then C diverges under both of σ and σ', or terminates under both states. Now take any $\sigma_0, \sigma_0' \in \mathbf{\Sigma} \backslash \mathbf{\Sigma}_{\Uparrow}^{W}$ such that $\sigma_0 \, \overline{W} \sigma_0'$ holds. Then by lemma 4.3.2, there are two cases to consider, namely, when W terminates under both states and when it diverges in both. Suppose that W terminates under σ_0 and σ_0', then for all $i \geq 0$, $\langle C, \sigma_i \rangle \Downarrow \sigma_{i+1}$ and $\langle C, \sigma_i' \rangle \Downarrow \sigma_{i+1}'$ and $\sigma_i \lfloor T_C \rfloor \sigma_i'$ and $\sigma_{i+1} \lfloor T_C \rfloor \sigma_{i+1}'$ hold. Thus, the attacker's observation under the traces of W starting at σ_0 and σ_0' is the same, that is, $\mathit{obs}(t_{\langle W, \sigma_0 \rangle}) = \mathit{obs}(t_{\langle W, \sigma_0' \rangle})$. Now suppose that W diverges under both σ_0 and σ_0', then by definition of $\overline{W}$ and by lemma 4.3.2 the pair of traces must diverge on the same iteration of W and the traces must be indistinguishable at each iteration step. That is, there exist $i, j \in \mathbb{N}$ such that for all $i \leq j$, $\sigma_i \lfloor T_C \rfloor \sigma_i'$, but c diverges under both σ_j and σ_j'. Thus, $\mathit{obs}(t_{\langle W, \sigma_0 \rangle}) = \mathit{obs}(t_{\langle W, \sigma_0' \rangle})$. Hence, for all $\sigma, \sigma' \in \mathbf{\Sigma} \backslash \mathbf{\Sigma}_{\Uparrow}^{W}, \sigma \, \overline{W} \, \sigma' \implies \sigma \lfloor T_W \rfloor \sigma'$.

Now take any $\sigma_0, \sigma_0' \in \mathbf{\Sigma}_{\Uparrow}^{W}$ such that $\sigma_0 \overline{W} \sigma'$ holds. Then, for all $i \in \mathbb{N}$, we have that $\sigma_i \lfloor T_C \rfloor \sigma_i'$ by the definition of $\overline{W}$, and since W diverges under both traces we have $\mathit{obs}(t_{\langle W, \sigma_0 \rangle}) = \mathit{obs}(t_{\langle W, \sigma_0' \rangle})$. Thus, for all $\sigma, \sigma' \in \mathbf{\Sigma}_{\Uparrow}^{W}, \sigma \, \overline{W} \, \sigma' \implies \sigma \lfloor T_W \rfloor \sigma'$.

Therefore, for all $\sigma, \sigma' \in \mathbf{\Sigma}, \sigma \, \overline{W} \, \sigma' \implies \sigma \lfloor T_W \rfloor \sigma'$, and hence $\lfloor T_W \rfloor \sqsubseteq \overline{W}$.

□

Lemma 5.7.9. *Let* $(E, I, O) \in \mathbf{\Phi}_{\mathit{init}}$ *such that* $(E, I, O) \; P \; (E', I', O')$. *Then* $\mathit{dom}(I') \subseteq \mathit{dom}(I)$.

Proof. We note that for any $(E, I, O) \in \mathbf{\Phi}_{\mathit{init}}$ and any expression e and PER ϕ over values of e, we have $\mathit{dom}(\mathit{flow}(e : \phi, (E, I, O))) \subseteq \mathit{dom}(I)$. This is because $\mathit{flow}(e : \phi, (E, I, O)) = \uparrow_{\mathbf{TVar}}(e : \phi \sqcup I \sqcup R_E)$ (see the details in Definition 5.4.10). Hence, $\mathit{dom}(\mathit{flow}(e : \phi, (E, I, O))) = \mathit{havoc}\mathbf{TVar}(\mathit{dom}(e : \phi \sqcup R_E \sqcup I))$. That is,

$\mathit{dom}(\mathit{flow}(e : \phi, (E, I, O))) \subseteq \mathit{havoc}\mathbf{TVar}(\mathit{dom}(e : \phi \sqcup R_E)) \cap \mathit{havoc}\mathbf{TVar}(\mathit{dom}(I)) =$ $\mathit{havoc}\mathbf{TVar}(\mathit{dom}(e : \phi \sqcup R_E)) \cap \mathit{dom}(I)$ since $I = \uparrow_{\mathbf{TVar}} I$. Hence, we have that $\mathit{dom}(\mathit{flow}(e : \phi, (E, I, O))) \subseteq \mathit{dom}(I)$.

The proof proceeds by induction on the derivation tree of each command in the information flow rules. The rules for *skip, assignment* and *write* statements do not change the I-component of their pre-configurations, thus it only remains to show that this property holds for *if* and *while* statements. This follows immediately because for any subprogram command c of P the pre-configuration and post-configuration are both elements of $\mathbf{\Phi}_{\mathit{init}}$ (see Theorem 5.7.10). In the information flow rules, the resulting post configuration $(E_{n+1}, I_{n+1}, O_{n+1})$ is computed from (E_n, I_n, O_n) as a join of $I_{n+1} = \mathit{flow}(e : \phi, (E_n, I_n, O_n))$ (conditional branching of *if* statements and termination analysis of *while* statements) - which means $\mathit{dom}(I_{n+1}) \subseteq \mathit{dom}(I_n)$, or by constructing a PER which represents the union of the domains of the I-component of the post-configuration of the branches of a conditional *if* statement, which by induction on the branches are both subsets of the domain of the I-component of their respective pre-configuration. □

We now prove properties of the static analysis which establish its information flow and semantic soundness. It is assumed that all variables of P are properly-initialised before use.

Theorem 5.7.10 (Semantic and Information Flow Correctness)**.** *Let* $\mathbf{\Sigma}$ *and* $\mathbf{Var}$ *respectively be the set of states and variables of a* While *program* P*, which does not modify the* $\mathbf{IVar}$ *projection of states. Furthermore, let* $(E, I, O) \in \mathbf{\Phi}_{\mathit{init}}$ *be an information configuration and let* $(E, I, O)\ P\ (E', I', O')$ *be the static analysis of* P*. Then*

(A) $\mathit{PER}((E,I,O)) \sqcup \lfloor T_P \rfloor \sqsubseteq \mathit{PER}((E,I,O)) \sqcup O'$ *and*

(B) $\{\sigma' \mid \sigma \in \mathit{dom}((E,I,O)), \langle P,\sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E',I',O'))$ *and*

(C) $(E',I',O') \in \mathbf{\Phi}_{\mathit{init}}$.

Proof. The proof proceeds by structural induction on the derivation tree of $(E,I,O)P(E',I',O')$. Suppose $P = P'_0; P'_1; \ldots; P'_m$, such that for all i, P'_i is a *skip, assignment, write,* or a *conditional if* or *while* statement. Furthermore, for any $n \leq m$, define $P_n \triangleq P'_0; \ldots; P'_n$ such that for all n, $(E,I,O)\ P_n\ (E_n,I_n,O_n)$. We will show that if the induction hypothesis holds for P_n then it holds also for $P_{n+1} \triangleq P_n; P'_{n+1}$.

- The proof when P'_{n+1} is a *skip* statement is straightforward.
- Let P'_{n+1} be $z := e$. Then we have $(E_{n+1},I_{n+1},O_{n+1}) = (E_n[z \mapsto \uparrow_X \overset{z:=e}{\rightsquigarrow} R\,], I_n, O_n)$ where $R = e : \mathit{id} \sqcup I_n \sqcup R_E$ and $\forall \sigma,\sigma' \in \mathbf{\Sigma}, \sigma R_E\ \sigma'$ iff $\sigma,\sigma' \in \mathit{dom}(\bigsqcup_{x \in FV(e)} E_n(x))$ and $X = \mathbf{TVar} \backslash \{z\}$.

  (A) Since $\lfloor T_{P_n} \rfloor = \lfloor T_{P_{n+1}} \rfloor$ and $O_{n+1} = O_n$ then by the induction hypothesis we have also that $\mathrm{PER}((E,I,O)) \sqcup \lfloor T_{P_{n+1}} \rfloor \sqsubseteq O_{n+1} \sqcup \mathrm{PER}((E,I,O))$.

  (B) Now let $\Sigma = \{\sigma' \mid \sigma \in \mathit{dom}((E,I,O)), \langle P_n,\sigma\rangle \Downarrow \sigma'\}$ and let $\Sigma' = \{\sigma' \mid \sigma \in \mathit{dom}((E,I,O)), \langle P_{n+1},\sigma\rangle \Downarrow \sigma'\}$, then $\Sigma' = \{\sigma[z \mapsto \sigma(e)] \mid \sigma \in \Sigma\}$. By the induction hypothesis $\Sigma \subseteq \mathit{dom}((E_n,I_n,O_n))$, thus $\Sigma \subseteq \mathit{dom}(\overset{z:=e}{\rightsquigarrow} R)$ since $e : \mathit{id}$ is an equivalence relation over $\mathbf{\Sigma}$. From the definition of $\overset{z:=e}{\rightsquigarrow} R$, we know that for any $\sigma \in \mathit{dom}(R)$, $\sigma[z \mapsto \sigma(e)] \in \mathit{dom}(\overset{z:=e}{\rightsquigarrow} R)$. Hence, $\Sigma' \subseteq \mathit{dom}(\overset{z:=e}{\rightsquigarrow} R)$ and since $\mathit{dom}(\overset{z:=e}{\rightsquigarrow} R) \subseteq \mathit{dom}(\uparrow_X \overset{z:=e}{\rightsquigarrow} R)$ then $\Sigma' \subseteq \mathit{dom}(\uparrow_X \overset{z:=e}{\rightsquigarrow} R)$. Now let $Z = \{z\}$, since $\Sigma \subseteq \mathit{dom}((E_n,I_n,O_n))$, then $\Sigma' \subseteq \mathit{havocZ}(\mathit{dom}((E_n,I_n,O_n)))$ because Σ'

is obtained from Σ by modifying the value of the variable z alone. Furthermore, $\mathit{dom}((E_{n+1}, I_{n+1}, O_{n+1})) = \mathit{havocZ}(\mathit{dom}((E_n, I_n, O_n))) \cap \mathit{dom}(\uparrow_X \overset{z:=e}{\rightsquigarrow} R)$ since $(E_n, I_n, O_n) \in \mathbf{\Phi}_{\mathit{init}}$ by the induction hypothesis. Hence, $\Sigma' \subseteq \mathit{dom}((E_{n+1}, I_{n+1}, O_{n+1}))$, since Σ' is a subset of both $\mathit{havocZ}(\mathit{dom}((E_n, I_n, O_n)))$ and $\mathit{dom}(\uparrow_X \overset{z:=e}{\rightsquigarrow} R)$.

(C) By the induction hypothesis $(E_n, I_n, O_n) \in \mathbf{\Phi}_{\mathit{init}}$, it is thus clear that $(E_{n+1}, I_{n+1}, O_{n+1}) \in \mathbf{\Phi}_{\mathit{init}}$.

- Let P'_{n+1} be `write` e. Then we have the post-configuration $(E_{n+1}, I_{n+1}, O_{n+1})$ $= (E_n, I_n, O_n \uplus \mathit{flow}(e : \mathit{id}, (E_n, I_n, O_n)))$. Let $\varphi_0 = (E_n, I_n, O_n)$.

  (A) Now $\lfloor T_{P_{n+1}} \rfloor$ is derived from $\lfloor T_{P_n} \rfloor$ as follows: for all $\sigma, \sigma' \in \mathbf{\Sigma}, \sigma \lfloor T_{P_{n+1}} \rfloor \sigma'$ iff $\sigma \lfloor T_{P_n} \rfloor \sigma'$ and if $\langle P_n, \sigma \rangle \Downarrow \sigma_n$ and $\langle P_n, \sigma' \rangle \Downarrow \sigma'_n$ then $\sigma_n(e) = \sigma'_n(e)$. But by the induction hypothesis $\mathrm{PER}((E, I, O)) \sqcup \lfloor T_{P_n} \rfloor \sqsubseteq \mathrm{PER}((E, I, O)) \sqcup O_n$ and also $\{\sigma' | \sigma \in \mathit{dom}((E, I, O)), \langle P_n, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_0)$. Hence, by applying lemma 5.7.7 and since $O_n \Subset O_{n+1}$, it only remains to show that $\mathrm{PER}(\varphi_0) \sqcup \left\lfloor T_{P'_n} \right\rfloor = \mathrm{PER}(\varphi_0) \sqcup e : \mathit{id} \sqsubseteq \mathrm{PER}(\varphi_0) \sqcup O_{n+1}$ in order to show that $\mathrm{PER}((E, I, O)) \sqcup \lfloor T_{P_{n+1}} \rfloor \sqsubseteq \mathrm{PER}((E, I, O)) \sqcup O_{n+1}$.

  Let $\mathit{flow}(e : \mathit{id}, \varphi_0) = \uparrow_{\mathbf{TVar}} R$, where $R = e : \mathit{id} \sqcup I_n \sqcup R_E$ and where $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_E \; \sigma'$ iff $\sigma, \sigma' \in \mathit{dom}(\bigsqcup_{x \in FV(e)} E_n(x))$. Furthermore, let $\Sigma' = \mathit{dom}(O_n) \cup \mathit{dom}(\uparrow_{\mathbf{TVar}} R)$ such that $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; \overline{R} \; \sigma' \iff \sigma, \sigma' \in \Sigma' \backslash \mathit{dom}(\uparrow_{\mathbf{TVar}} R)$ and $\sigma \; \overline{O_n} \; \sigma' \iff \sigma, \sigma' \in \Sigma' \backslash \mathit{dom}(O_n)$. Hence, we have $O_{n+1} = O_n \uplus \uparrow_{\mathbf{TVar}} R = (O_n \cup \overline{O_n}) \sqcup (\uparrow_{\mathbf{TVar}} R \cup \overline{R})$. Thus, $O_{n+1} \sqcup \mathrm{PER}(\varphi_0) = O_n \sqcup \uparrow_{\mathbf{TVar}} R \sqcup \mathrm{PER}(\varphi_0)$ since $\mathit{dom}(\varphi_0) \subseteq \mathit{dom}(O_n)$ and $\mathit{dom}(\varphi_0) \subseteq \mathit{dom}(\uparrow_{\mathbf{TVar}} R)$ and hence $\mathrm{PER}(\varphi_0)$ is disjoint with $\overline{O_n}$ and $\overline{R}$, and $\overline{O_n} \sqcup \overline{R} = \varnothing$ by definition. Therefore, to show that $\mathrm{PER}(\varphi_0) \sqcup e : \mathit{id} \sqsubseteq \mathrm{PER}(\varphi_0) \sqcup O_{n+1}$ we need to show that for any $\sigma, \sigma' \in$

$dom(\varphi_0), \sigma\ (O_n \sqcup \uparrow_{\mathbf{TVar}} R)\ \sigma' \implies \sigma(e) = \sigma'(e)$. Since $\uparrow_{\mathbf{TVar}} I_n = I_n$, then $I_n \in \mathcal{R}_{init}$ by observing that for all $\sigma \in dom(I_n), havoc\mathbf{TVar}([\sigma]_{I_n}) = [\sigma]_{I_n}$. Furthermore, since $\mathbf{TVar}$ variables are properly-initialised before use, then the $FV(e) \cap \mathbf{TVar}$ projection of state is a function of the $\mathbf{IVar}$ projection, which means that by applying (3) and (4) of lemma 5.7.3, $R_E \in \mathcal{R}_{init}$ and hence $R \in \mathcal{R}_{init}$. Hence, since $dom(\varphi_0) \subseteq dom(R)$, then by (2) of lemma 5.7.3, for all $\sigma, \sigma' \in dom(\varphi_0), \sigma \uparrow_{\mathbf{TVar}} R \sigma' \implies \sigma\ R\ \sigma' \implies \sigma(e) = \sigma'(e)$. Thus, $\text{PER}(\varphi_0) \sqcup e : id \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$.

(B) It is clear that $\Sigma = \{\sigma' \mid \sigma \in dom((E, I, O)), \langle P_n, \sigma\rangle \Downarrow \sigma'\} = \{\sigma' \mid \sigma \in dom((E, I, O)), \langle P_{n+1}, \sigma\rangle \Downarrow \sigma'\}$ is the set of terminating states of P_{n+1} starting from $dom((E, I, O))$. By the induction hypothesis $\Sigma \subseteq dom((E_n, I_n, O_n))$. Furthermore, by the domain-preserving property of $\uplus$, $dom(O_n) \subseteq dom(O_{n+1})$. Hence, $\Sigma \subseteq dom((E_n, I_n, O_n)) \subseteq dom((E_{n+1}, I_{n+1}, O_{n+1}))$.

(C) Since by the induction hypothesis $(E_n, I_n, O_n) \in \mathbf{\Phi}_{init}$, it is thus clear by applying (7) of Lemma 5.4.8 that $(E_{n+1}, I_{n+1}, O_{n+1}) \in \mathbf{\Phi}_{init}$.

- Let P'_{n+1} be $\texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2$. Let $\varphi_0 = (E_n, I_n, O_n)$ so that $I'_n = flow(b : \mathbf{T}, \varphi_0)$ and $I''_n = flow(b : \mathbf{F}, \varphi_0)$ and $(E_n, I'_n, O_n) c_1 (E'_{n+1}, I'_{n+1}, O'_{n+1})$ and $(E_n, I''_n, O_n)\ c_2\ (E''_{n+1}, I''_{n+1}, O''_{n+1})$. By applying the induction hypothesis to P_n, $\varphi_0 \in \mathbf{\Phi}_{init}$ and hence $(E_n, I'_n, O_n), (E_n, I''_n, O_n) \in \mathbf{\Phi}_{init}$ because $I'_n = \uparrow_{\mathbf{TVar}} I'_n$ and $I''_n = \uparrow_{\mathbf{TVar}} I''_n$ by definition. By further applying the induction hypothesis to c_1 we obtain $\text{PER}((E_n, I'_n, O_n)) \sqcup \lfloor T_{c_1} \rfloor \sqsubseteq O'_{n+1} \sqcup \text{PER}((E_n, I'_n, O_n))$, and also that $\{\sigma' | \sigma \in dom((E_n, I'_n, O_n)), \langle c_1, \sigma\rangle \Downarrow \sigma'\} \subseteq dom((E'_{n+1}, I'_{n+1}, O'_{n+1}))$, and also that $(E'_{n+1}, I'_{n+1}, O'_{n+1}) \in \mathbf{\Phi}_{init}$. Similarly, for the *else* branch we have that $\text{PER}((E_n, I''_n, O_n)) \sqcup \lfloor T_{c_2} \rfloor \sqsubseteq O''_{n+1} \sqcup$

$\text{PER}((E_n, I''_n, O_n))$ and that $\{\sigma' \mid \sigma \in \mathit{dom}((E_n, I''_n, O_n)), \langle c_2, \sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E''_{n+1}, I''_{n+1}, O''_{n+1}))$, and that $(E''_{n+1}, I''_{n+1}, O''_{n+1}) \in \mathbf{\Phi}_{\mathit{init}}$. Now define I_{n+1} such that $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ I_{n+1} \sigma' \iff \sigma, \sigma' \in \mathit{dom}(I'_{n+1}) \cup \mathit{dom}(I''_{n+1})$ and for all $x \in \mathbf{TVar}$, let $\overline{E'_{n+1}}(x) = E'_{n+1}(x) \sqcup I'_n$ if $E''_{n+1}(x) \neq E_n(x)$ and $\overline{E'_{n+1}}(x) = E'_{n+1}(x)$ otherwise. Similarly, for all $x \in \mathbf{TVar}$, let $\overline{E''_{n+1}}(x) = E''_{n+1}(x) \sqcup I''_n$ if $E'_{n+1}(x) \neq E_n(x)$ and $\overline{E''_{n+1}}(x) = E''_{n+1}(x)$ otherwise. Then the post-configuration of P'_{n+1} is defined as $(E_{n+1}, I_{n+1}, O_{n+1}) = (\overline{E'_{n+1}}, I_{n+1}, O'_{n+1}) \uplus (\overline{E''_{n+1}}, I_{n+1}, O''_{n+1})$.

(A) Let $\Sigma' = \mathit{dom}(O'_{n+1}) \cup \mathit{dom}(O''_{n+1})$ so that $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ \overline{O'_{n+1}}\ \sigma' \iff \sigma, \sigma' \in \Sigma' \backslash \mathit{dom}(O'_{n+1})$ and $\sigma\ \overline{O''_{n+1}}\ \sigma' \iff \sigma, \sigma' \in \Sigma' \backslash \mathit{dom}(O''_{n+1})$. Since $O_{n+1} = O'_{n+1} \uplus O''_{n+1}$, and $\overline{O'_{n+1}} \sqcup \overline{O''_{n+1}} = \varnothing$ by definition, then $O_{n+1} = (O'_{n+1} \sqcup O''_{n+1}) \cup ((O'_{n+1} \sqcup \overline{O''_{n+1}})) \cup ((\overline{O'_{n+1}} \sqcup O''_{n+1}))$. Furthermore, by the domain-preserving property of $\uplus$, we know that $\mathit{dom}(O_n) \subseteq \mathit{dom}(O'_{n+1})$ and $\mathit{dom}(O_n) \subseteq \mathit{dom}(O''_{n+1})$, and also by definition $\mathit{dom}(\varphi_0) \subseteq \mathit{dom}(O_n)$. Hence, by definition $\text{PER}(\varphi_0) \sqcup \overline{O'_{n+1}} = \text{PER}(\varphi_0) \sqcup \overline{O''_{n+1}} = \varnothing$. Therefore, $\text{PER}(\varphi_0) \sqcup O_{n+1} = \text{PER}(\varphi_0) \sqcup O'_{n+1} \sqcup O''_{n+1}$. We now show that $\text{PER}(\varphi_0) \sqcup \left\lfloor T_{P'_{n+1}} \right\rfloor \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$.

We consider three cases based on how the boolean guard b evaluates. Firstly, we note that I'_n and I''_n partition the domain of I_n such that the set of states in $\mathit{dom}((E_n, I'_n, O_n))$ and $\mathit{dom}((E_n, I''_n, O_n))$ evaluate b to $\mathbf{tt}$ and $\mathbf{ff}$ respectively. This is clear since (see Definition 5.4.10) we have that $I'_n = \uparrow_{\mathbf{TVar}}(b : \mathbf{T} \sqcup I_n \sqcup R_E)$ and $I''_n = \uparrow_{\mathbf{TVar}}(b : \mathbf{F} \sqcup I_n \sqcup R_E)$. Hence, since $I_n \sqcup R_E \in \mathcal{R}_{\mathit{init}}$ and because $\mathit{dom}(\varphi_0) \subseteq \mathit{dom}(I_n \sqcup R_E)$ by definition, then by (4) and (2) of lemma 5.7.3 we have that for all

$\sigma, \sigma' \in \mathit{dom}(\varphi_0), \sigma\ I'_n\ \sigma' \implies \sigma(b) = \sigma'(b) = \mathbf{tt}$ and $\sigma\ I''_n\ \sigma' \implies \sigma(b) = \sigma'(b) = \mathbf{ff}$.

Now take any $\sigma, \sigma' \in \Sigma$ such that $\sigma(\text{PER}(\varphi_0) \sqcup O_{n+1})\sigma'$, since $\text{PER}(\varphi_0) \sqcup O_{n+1} = \text{PER}(\varphi_0) \sqcup O'_{n+1} \sqcup O''_{n+1}$ then we know that $\sigma, \sigma' \in \mathit{dom}(\varphi_0)$ and $\sigma\ O'_{n+1}\ \sigma'$ and $\sigma\ O''_{n+1}\ \sigma'$ hold.

(a) Suppose $\sigma(b) = \sigma'(b) = \mathbf{tt}$, then $\sigma, \sigma' \in \mathit{dom}((E_n, I'_n, O_n))$. But by the induction hypothesis $\text{PER}((E_n, I'_n, O_n)) \sqcup \lfloor T_{c_1} \rfloor \sqsubseteq O'_{n+1} \sqcup \text{PER}((E_n, I'_n, O_n))$, hence $\sigma\, O'_{n+1}\, \sigma' \implies \sigma \lfloor T_{c_1} \rfloor \sigma'$. Since $\sigma(b) = \sigma'(b) = \mathbf{tt}$ and $\sigma \lfloor T_{c_1} \rfloor \sigma'$ holds, then $\sigma \left\lfloor T_{P'_{n+1}} \right\rfloor \sigma'$ holds.

(b) The proof in the case that $\sigma(b) = \sigma'(b) = \mathbf{ff}$ is similar to that of the case when $\sigma(b) = \sigma'(b) = \mathbf{tt}$.

(c) Now suppose $\sigma(b) = \mathbf{tt}$ and $\sigma'(b) = \mathbf{ff}$. Thus, we have that $\sigma \in \mathit{dom}((E_n, I'_n, O_n))$ and $\sigma' \in \mathit{dom}((E_n, I''_n, O_n))$. Since $\sigma\ O'_{n+1} \sqcup O''_{n+1}\ \sigma'$ holds, then there does not exist a *write* or *while* command along the execution paths of σ or σ' within P'_{n+1}. Suppose that there exist a *write* or *while* statement along the execution path of σ, then there exists in the information flow analysis an expression e and a configuration (E_e, I_e, O_e) such that $\sigma \in \mathit{dom}(\mathit{flow}(e : \mathit{id}, (E_e, I_e, O_e)))$ and $O'_{n+1} = O'_{n+1} \uplus \mathit{flow}(e : \mathit{id}, (E_e, I_e, O_e))$. By lemma 5.7.9 we have that $\mathit{dom}(\mathit{flow}(e : \mathit{id}, (E_e, I_e, O_e))) \subseteq \mathit{dom}(I_e) \subseteq \mathit{dom}(I'_n)$ and $\sigma \in \mathit{dom}(I'_n)$. Furthermore, since $\sigma'(b) = \mathbf{ff}$ then $\sigma' \notin \mathit{dom}(I'_n)$ and hence we have that $\sigma' \notin \mathit{dom}(\mathit{flow}(e : \mathit{id}, (E_e, I_e, O_e)))$. Therefore, by (3) of proposition 5.4.5 $(\sigma, \sigma') \notin O_{n+1}$ since $O'_{n+1} = O'_{n+1} \uplus \mathit{flow}(e : \mathit{id}, (E_e, I_e, O_e))$, which contradicts our assumption

that $\sigma O'_{n+1}\sigma'$ holds. Similarly, there does not exist a *write* or *while* statement along the execution path of σ' in P'_{n+1}. Thus, because P'_{n+1} neither produces an output (no *write* statement), nor diverges (no *while* statement) along the execution paths of the states σ and σ' in P'_{n+1} then $\sigma \left\lfloor T_{P'_{n+1}} \right\rfloor \sigma'$ holds.

Thus, $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ (\text{PER}(\varphi_0) \sqcup O_{n+1})\ \sigma' \implies \sigma \left\lfloor T_{P'_{n+1}} \right\rfloor \sigma'$, that is, $\text{PER}(\varphi_0) \sqcup \left\lfloor T_{P'_{n+1}} \right\rfloor \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$.

Now define $\left\lfloor T_{P_n \bullet P'_{n+1}} \right\rfloor$ as $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \left\lfloor T_{P_n \bullet P'_{n+1}} \right\rfloor \sigma'$ iff $\sigma \lfloor T_{P_n} \rfloor \sigma'$ and if $\langle P_n, \sigma \rangle \Downarrow \sigma_n$ and $\langle P_n, \sigma' \rangle \Downarrow \sigma'_n$ then also $\sigma_n \left\lfloor T_{P'_{n+1}} \right\rfloor \sigma'_n$. By proposition 5.7.6 we know that $\lfloor T_{P_{n+1}} \rfloor \sqsubseteq \left\lfloor T_{P_n \bullet P'_{n+1}} \right\rfloor$, and hence that $\lfloor T_{P_{n+1}} \rfloor \sqcup \text{PER}((E, I, O)) \sqsubseteq \left\lfloor T_{P_n \bullet P'_{n+1}} \right\rfloor \sqcup \text{PER}((E, I, O))$. Since by the induction hypothesis we have $\lfloor T_{P_n} \rfloor \sqcup \text{PER}((E, I, O)) \sqsubseteq O_n \sqcup \text{PER}((E, I, O))$ and also $\{\sigma' | \sigma \in \mathit{dom}((E, I, O)), \langle P_n, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_0)$, and $\text{PER}(\varphi_0) \sqcup \left\lfloor T_{P'_{n+1}} \right\rfloor \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$, hence by lemma 5.7.7 we have $\lfloor T_{P_{n+1}} \rfloor \sqcup \text{PER}((E, I, O)) \sqsubseteq O_{n+1} \sqcup \text{PER}((E, I, O))$.

(B) By the induction hypothesis we already know that for the *then* branch we have $\{\sigma' | \sigma \in \mathit{dom}((E_n, I'_n, O_n)), \langle c_1, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E'_{n+1}, I'_{n+1}, O'_{n+1})) \subseteq \mathit{dom}((\overline{E'_{n+1}}, I_{n+1}, O'_{n+1}))$ since by lemma 5.7.9 we have $\mathit{dom}(I'_{n+1}) \subseteq \mathit{dom}(I'_n)$ and also by definition $\mathit{dom}(I'_{n+1}) \subseteq \mathit{dom}(I_{n+1})$. Similarly, for the *else* branch $\{\sigma' | \sigma \in \mathit{dom}((E_n, I''_n, O_n)), \langle c_2, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((\overline{E''_{n+1}}, I_{n+1}, O''_{n+1}))$. Since $\mathit{dom}((E_n, I'_n, O_n)) = \{\sigma \in \mathit{dom}((E_n, I_n, O_n)) \mid \sigma(b) = \mathbf{tt}\}$ and $\mathit{dom}((E_n, I''_n, O_n)) = \{\sigma \in \mathit{dom}((E_n, I_n, O_n)) \mid \sigma(b) = \mathbf{ff}\}$, therefore the set of terminating states of P'_{n+1} starting from a state in $\mathit{dom}((E_n, I_n, O_n))$ satisfies the property $\{\sigma' \mid \sigma \in \mathit{dom}((E_n, I_n, O_n)), \langle P'_{n+1}, \sigma \rangle \Downarrow \sigma'\} \subseteq$

$(E_{n+1}, I_{n+1}, O_{n+1})$ since by the domain-preserving property of $\uplus$, we have that $\mathit{dom}((\overline{E'_{n+1}}, I_{n+1}, O'_{n+1})) \subseteq (E_{n+1}, I_{n+1}, O_{n+1})$ and also that $\mathit{dom}((\overline{E''_{n+1}}, I_{n+1}, O''_{n+1})) \subseteq (E_{n+1}, I_{n+1}, O_{n+1})$. Hence, by the induction hypothesis, $\{\sigma' | \sigma \in \mathit{dom}((E, I, O)), \langle P_n, \sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E_n, I_n, O_n))$ implies $\{\sigma' | \sigma \in \mathit{dom}((E, I, O)), \langle P_{n+1}, \sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E_{n+1}, I_{n+1}, O_{n+1}))$.

(C) We have already shown by the induction hypothesis at the beginning of the *if* proof that $(E'_{n+1}, I'_{n+1}, O'_{n+1}), (E''_{n+1}, I''_{n+1}, O''_{n+1}) \in \mathbf{\Phi}_{\mathit{init}}$. In the updated post-configuration $(\overline{E'_{n+1}}, I_{n+1}, O'_{n+1})$ of the *then* branch, and for a given variable $x \in \mathbf{TVar}$, $\overline{E'_{n+1}}(x)$ may be $E'_{n+1}(x) \sqcup I'_n$ or E'_{n+1} according to the *if* rule. Hence, by applying (5) and (6) of lemma 5.4.8 we know that $\overline{E'_{n+1}}(x) = \uparrow_{\mathbf{TVar}\setminus\{x\}}(E'_{n+1}(x) \sqcup I'_n)$ because $E'_{n+1}(x) = \uparrow_{\mathbf{TVar}\setminus\{x\}} E'_{n+1}(x)$ and $I'_n = \uparrow_{\mathbf{TVar}\setminus\{x\}} I'_n$ since $I'_n = \uparrow_{\mathbf{TVar}} I'_n$. Furthermore, since $\uparrow_{\mathbf{TVar}} I'_{n+1} = I'_{n+1}$ and $\uparrow_{\mathbf{TVar}} I''_{n+1} = I''_{n+1}$ by the induction hypothesis, then by (4) of lemma 5.4.8 $\mathit{havoc}\mathbf{TVar}(\mathit{dom}(I'_{n+1}) \cup \mathit{dom}(I''_{n+1})) = \mathit{dom}(I'_{n+1}) \cup \mathit{dom}(I''_{n+1})$, which implies that $\uparrow_{\mathbf{TVar}} I_{n+1} = I_{n+1}$. Hence, $(\overline{E'_{n+1}}, I_{n+1}, O'_{n+1}) \in \mathbf{\Phi}_{\mathit{init}}$. Similarly, for the post-configuration of the *else* branch, we have $(\overline{E''_{n+1}}, I_{n+1}, O''_{n+1}) \in \mathbf{\Phi}_{\mathit{init}}$. Finally, by applying (7) of lemma 5.4.8 we have that $(E_{n+1}, I_{n+1}, O_{n+1}) = (\overline{E'_{n+1}}, I_{n+1}, O'_{n+1}) \uplus (\overline{E''_{n+1}}, I_{n+1}, O''_{n+1}) \in \mathbf{\Phi}_{\mathit{init}}$.

- Let P'_{n+1} be `while` (b) `do` c, and let $C_0 \triangleq$ `if(`b`)then` c `else skip` and for all $i \geq 0$ define $C_{i+1} \triangleq C_i; C_0$. Furthermore, let $\varphi_0 = (E_n, I_n, O_n)$ and for all $i \geq 0$ let $(E'_i, I'_i, O'_i) = \varphi_i$ such that $\varphi_0\, C_i\, \varphi_{i+1}$, and define $P^i_n \triangleq P_n; C_i$.

  We know that $\{\sigma' \mid \sigma \in \mathit{dom}((E, I, O)), \langle P_n, \sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_0)$ and that for all $i \geq 0$, $\{\sigma' \mid \sigma \in \mathit{dom}(\varphi_0), \langle C_i, \sigma\rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_{i+1})$ by applying

the induction hypothesis. Now let $(E'', I'', O'') = \biguplus_{i \geq 0} \varphi_i$. Furthermore, define the E-component of the *while* post-configuration so that for any $x \in \mathbf{Var}, \bar{X} = \mathbf{TVar} \backslash \{x\}$ and for all $\sigma, \sigma' \in \mathbf{\Sigma}$, $\sigma \; E_W(x) \; \sigma' \iff \exists i \in \mathbb{N}, \sigma{\uparrow_{\bar{X}}}(E'_i(x) \sqcup I''_i \sqcup b : \mathbf{F})\sigma'$, where $I''_i = \mathit{flow}(b : \mathbf{F}, \varphi_i)$. Now define the I-component and the O-component of the *while* statement post-configuration to be $I_W = \mathit{flow}(b : \mathbf{F}, (E_W, I'', O''))$ and $O_W = \biguplus_{i \geq 0} \mathit{flow}(b : \mathit{id}, \varphi_i) \uplus O''$, so that $(E_{n+1}, I_{n+1}, O_{n+1}) = (E_W, I_W, O_W)$.

Define the set of states that may occur before or after an iteration of the *while* statement P'_{n+1}, starting from the set $\mathit{dom}(\varphi_0)$, as

$$\Sigma = \mathit{dom}(\varphi_0) \cup \{\sigma' \mid \sigma \in \mathit{dom}(\varphi_0), j \in \mathbb{N}, \langle C_j, \sigma \rangle \Downarrow \sigma'\} \tag{5.1}$$

Now let the subset of Σ under which the *while* statement terminates be given by

$$\Sigma_{\Downarrow} = \{\sigma \in \Sigma \mid \langle P'_{n+1}, \sigma \rangle \Downarrow \sigma'\}. \tag{5.2}$$

Thus, the subset of Σ under which the *while* statement diverges is given by $\Sigma_{\Uparrow} = \Sigma \backslash \Sigma_{\Downarrow}$.

We also identify the subset of Σ under which the *outer while* statement diverges, excluding those under which the subprogram c diverges

$$\Sigma^W_{\Uparrow} = \{\sigma \in \Sigma \mid \forall j \in \mathbb{N}. \langle C_j, \sigma \rangle \Downarrow \sigma'_j \wedge \sigma'_j(b) = \mathbf{tt}\}. \tag{5.3}$$

Finally, the set of states in Σ under which the subprogram c diverges is given

by

$$\Sigma^{c}_{\Uparrow} \triangleq \Sigma_{\Uparrow} \backslash \Sigma^{W}_{\Uparrow}. \tag{5.4}$$

(A) We start by showing that O_{n+1} contains termination information. More specifically, that O_{n+1} distinguishes any state in $\Sigma_{\Downarrow}$ from any state in $\Sigma_{\Uparrow}$. This is shown in two steps, firstly that O_{n+1} distinguishes states in $\Sigma_{\Downarrow}$ from all other states in $\Sigma^{W}_{\Uparrow}$, and secondly, by induction on c, that O_{n+1} distinguishes all states in $\Sigma_{\Downarrow}$ from those in $\Sigma^{c}_{\Uparrow}$.

We now observe that for every $\sigma \in \Sigma_{\Downarrow} \cup \Sigma^{W}_{\Uparrow}$ there exists $j \in \mathbb{N}$ such that $\langle C_j, \sigma\rangle \Downarrow \sigma'$ and $\sigma' \in \mathit{dom}(\varphi_{j+1})$ by the induction hypothesis. Therefore, since the program does not modify the $\mathbf{IVar}$-projection of states, it is clear that $\sigma \in \mathit{havoc}\mathbf{TVar}(\{\sigma'\})$, hence we have that $\Sigma_{\Downarrow} \cup \Sigma^{W}_{\Uparrow} \subseteq \bigcup_{j\geq 0} \mathit{havoc}\mathbf{TVar}(\mathit{dom}(\varphi_j))$. Now, since $b : \mathit{id}$ is an equivalence relation, then for any $j \in \mathbb{N}$ $\mathit{dom}(\varphi_j) \subseteq \mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_j))$ by definition. Furthermore, $\mathit{havoc}\mathbf{TVar}(\mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_j))) = \mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_j))$ and hence $\Sigma_{\Downarrow} \cup \Sigma^{W}_{\Uparrow} \subseteq \bigcup_{j\geq 0} \mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_j))$ by the monotonicity and idempotency of $\mathit{havoc}\mathbf{TVar}(\cdot)$. Hence, we have that $\Sigma_{\Downarrow} \cup \Sigma^{W}_{\Uparrow} \subseteq \mathit{dom}(\bigsqcup_{j\geq 0} \mathit{flow}(b : \mathit{id}, \varphi_j))$ by the domain-preserving property of $\uplus$.

Now suppose $\sigma, \sigma' \in \Sigma_{\Downarrow} \cup \Sigma^{W}_{\Uparrow}$, such that $\sigma\ O_{n+1}\ \sigma'$ holds. Since $\sigma, \sigma' \in \mathit{dom}(\bigsqcup_{j\geq 0} \mathit{flow}(b : \mathit{id}, \varphi_j))$ and $O_{n+1} = \bigsqcup_{j\geq 0} \mathit{flow}(b : \mathit{id}, \varphi_j) \uplus O''$ then $\sigma \bigsqcup_{j\geq 0} \mathit{flow}(b : \mathit{id}, \varphi_j) \sigma'$ holds by (2) of proposition 5.4.5. Now let $\Sigma'' = \mathit{dom}(\bigsqcup_{j\geq 0} \mathit{flow}(b : \mathit{id}, \varphi_j))$, then for all $j \in \mathbb{N}, \sigma \mathcal{C}_{\Sigma''}(\mathit{flow}(b : \mathit{id}, \varphi_j))\sigma'$. That is, for all $j \in \mathbb{N}$, we have that $\sigma, \sigma' \in \mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_j))$ and $\sigma\ \mathit{flow}(b : \mathit{id}, \varphi_j)\ \sigma'$ or $\sigma, \sigma' \in \Sigma'' \backslash \mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_j))$. Now since $\sigma, \sigma' \in \Sigma_{\Downarrow} \cup \Sigma^{W}_{\Uparrow}$ then there exists $k \in \mathbb{N}$ and $\sigma_A, \sigma_B \in \boldsymbol{\Sigma}$ such that

$\langle C_k, \sigma\rangle \Downarrow \sigma_A$ and $\langle C_k, \sigma'\rangle \Downarrow \sigma_B$ and $\sigma_A, \sigma_B \in \mathit{dom}(\varphi_k)$ by the induction hypothesis. Now since $\sigma_A, \sigma_B \in \mathit{dom}(\varphi_k)$ then $\sigma_A, \sigma_B \in \mathit{dom}(R)$, where $R = b : \mathit{id} \sqcup I'_k \sqcup R_E$ and where $\forall \sigma_1, \sigma_2 \in \mathbf{\Sigma}, \sigma_1 \; R_E \; \sigma_2 \iff \sigma_1, \sigma_2 \in \mathit{dom}(\bigsqcup_{y \in FV(b)} E'_k(y))$ since $b : \mathit{id}$ is an equivalence relation. But we know that the program does not modify the $\mathbf{IVar}$ projection of states and hence, $\sigma_A \in \mathit{havoc}\mathbf{TVar}(\{\sigma\})$ and $\sigma_B \in \mathit{havoc}\mathbf{TVar}(\{\sigma'\})$. We now observe that $\mathit{flow}(b : \mathit{id}, \varphi_k) = \uparrow_{\mathbf{TVar}} R$, which means that $\sigma, \sigma' \in \mathit{dom}(\mathit{flow}(b : \mathit{id}, \varphi_k))$ and hence that $\sigma \; \mathit{flow}(b : \mathit{id}, \varphi_k) \; \sigma'$ holds. Since $\sigma \uparrow_{\mathbf{TVar}} R \; \sigma'$ holds and $\sigma_A, \sigma_B \in \mathit{havoc}\mathbf{TVar}([\sigma]_{\uparrow_{\mathbf{TVar}} R})$, then by (4) of lemma 5.4.8, $\sigma_A \uparrow_{\mathbf{TVar}} R \; \sigma_B$ holds. Now we know that $R \in \mathcal{R}_{\mathit{init}}$ by applying (3) and (4) of lemma 5.7.3, since all variables are properly-assigned before use and hence for all $y \in FV(b), E'_k(y) \in \mathcal{R}_{\mathit{init}}$ and $I'_k \in \mathcal{R}_{\mathit{init}}$. Thus, $\sigma_A \uparrow_{\mathbf{TVar}} R \; \sigma_B$ means that $\sigma_A \; R \; \sigma_B$ holds by (2) of lemma 5.7.3, since $\sigma_A, \sigma_B \in \mathit{dom}(R)$, which implies that $\sigma_A(b) = \sigma_B(b)$ by the definition of R. Thus, there are only two cases to consider. Case 1, $\sigma_A(b) = \sigma_B(b) = \mathbf{ff}$, which means that the *while* statement terminates, and hence that $\sigma, \sigma' \in \Sigma_{\Downarrow}$ by the definition of $\Sigma_{\Downarrow}$. This leaves only the case 2, where for all $j \in \mathbb{N}$, $\langle C_j, \sigma\rangle \Downarrow \sigma_j$ and $\langle C_j, \sigma'\rangle \Downarrow \sigma'_j$ and $\sigma_j(b) = \sigma'_j(b) = \mathbf{tt}$, which means that $\sigma, \sigma' \in \Sigma^W_{\Uparrow}$. Hence, $\forall \sigma, \sigma' \in \Sigma_{\Downarrow} \cup \Sigma^W_{\Uparrow}$, $\sigma \; O_{n+1} \; \sigma'$ implies $\sigma, \sigma' \in \Sigma_{\Downarrow}$ or $\sigma, \sigma' \in \Sigma^W_{\Uparrow}$, which shows that O_{n+1} distinguishes the terminating states from the nonterminating ones in $\Sigma_{\Downarrow} \cup \Sigma^W_{\Uparrow}$.

We now show the second part that O_{n+1} distinguishes terminating states in $\Sigma_{\Downarrow}$, from those in $\Sigma^c_{\Uparrow}$, under which c diverges. This is achieved by induction on c, in particular, because we know by definition of O_{n+1}

that for all $i \geq 0$, $O'_i \Subset O_{n+1}$, and by the induction hypothesis we know that $\{\sigma' \mid \sigma \in \mathit{dom}(\varphi_i, \langle C_0, \sigma\rangle \Downarrow \sigma')\} \subseteq \mathit{dom}(\varphi_{i+1})$ and that $\text{PER}(\varphi_i) \sqcup \lfloor T_{C_0} \rfloor \sqsubseteq \text{PER}(\varphi_i) \sqcup O'_{i+1}$, and hence O'_{i+1} distinguishes terminating states in $\mathit{dom}(\varphi_i)$ from those that make C_0 diverge, because $\lfloor T_{C_0} \rfloor$ does also, as lemma 4.3.2 shows. When we combine this with the fact that for all i, $\mathit{dom}(\varphi_i) \subseteq \mathit{dom}(O'_i)$ and $O'_i \Subset O_{n+1}$, then by applying (2) of proposition 5.4.5 we know that for all $\sigma, \sigma' \in \mathit{dom}(\varphi_i), \sigma\, O_{n+1}\, \sigma' \implies \sigma\, O'_i\, \sigma'$, and hence by the contrapositive, O_{n+1} distinguishes the states that O'_i does. Thus, since $\Sigma^c_{\Uparrow} \subseteq \Sigma$ and $\Sigma \subseteq \bigcup_{i\geq 0} \mathit{dom}(\varphi_i)$ then $\sigma, \sigma' \in \Sigma_{\Downarrow} \cup \Sigma^c_{\Uparrow}$, $\sigma\, O_{n+1}\, \sigma'$ implies $\sigma, \sigma' \in \Sigma_{\Downarrow}$ or $\sigma, \sigma' \in \Sigma^c_{\Uparrow}$.

From the above we have that for all $\sigma, \sigma' \in \Sigma, \sigma\, O_{n+1}\, \sigma'$ implies $\sigma, \sigma' \in \Sigma_{\Downarrow}$ or $\sigma, \sigma' \in \Sigma^W_{\Uparrow} \cup \Sigma^c_{\Uparrow}$. Thus, O_{n+1} distinguishes the set of states, starting from $\mathit{dom}(\varphi_0)$, in which the *while* statement terminates, from those in which the statement diverges. Furthermore, we observe from the property of $\lfloor T_{C_j} \rfloor$ which, for all j, distinguishes the states under which C_j terminates from those under which it diverges (see lemma 4.3.2), that for any pair of states $\sigma, \sigma' \in \Sigma^W_{\Uparrow} \cup \Sigma^c_{\Uparrow}$, $\sigma\, O_{n+1}\, \sigma'$ implies $\sigma, \sigma' \in \Sigma^W_{\Uparrow}$ or $\sigma, \sigma' \in \Sigma^c_{\Uparrow}$. This is because by definition for all starting states in $\Sigma^W_{\Uparrow}$, C_i terminates for all i, whereas for all starting states in $\Sigma^c_{\Uparrow}$, there exists a j where C_j diverges. Hence, by applying the induction hypothesis and the definition of O_{n+1}, for any $\sigma_0, \sigma'_0 \in \Sigma^W_{\Uparrow} \cup \Sigma^c_{\Uparrow}$ such that $\sigma_0\, O_{n+1}\, \sigma'_0$ holds, $\sigma_0 \lfloor T_{C_0} \rfloor \sigma'_0$ holds and for all $i \geq 0$, $\langle C_0, \sigma_i\rangle \Downarrow \sigma_{i+1}$ and $\langle C_0, \sigma'_i\rangle \Downarrow \sigma'_{i+1}$ implies $\sigma_{i+1} \lfloor T_{C_0} \rfloor \sigma'_{i+1}$. But, by lemma 4.3.2, for any i, $\sigma_i \lfloor T_{C_0} \rfloor \sigma'_i$ implies that C_0 terminates under both σ_i and σ'_i or diverges

under both states. But since σ_j terminates and σ'_j diverges when C_0 is executed, $\lfloor T_{C_0} \rfloor$ does not relate them. Thus, for all $\sigma, \sigma' \in \Sigma^W_{\Uparrow} \cup \Sigma^c_{\Uparrow}$, we have that $\sigma\, O_{n+1}\, \sigma'$ implies $\sigma, \sigma' \in \Sigma^W_{\Uparrow}$ or $\sigma, \sigma' \in \Sigma^c_{\Uparrow}$. Combining this with the earlier results, we have that for all $\sigma, \sigma' \in \Sigma$, $\sigma\, O_{n+1}\, \sigma'$ implies $\sigma, \sigma' \in \Sigma_{\Downarrow}$ or $\sigma, \sigma' \in \Sigma^W_{\Uparrow}$ or $\sigma, \sigma' \in \Sigma^c_{\Uparrow}$.

Finally, we now show that the property $\text{PER}((E, I, O)) \sqcup \lfloor T_{P_{n+1}} \rfloor \sqsubseteq \text{PER}((E, I, O)) \sqcup O_{n+1}$ holds. By the induction hypothesis $\text{PER}((E, I, O)) \sqcup \lfloor T_{P_n} \rfloor \sqsubseteq \text{PER}((E, I, O)) \sqcup O_n$ and $\{\sigma' | \sigma \in \mathit{dom}((E, I, O)), \langle P_n, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_0)$, and we know from the flow rules that $O_n \Subset O_{n+1}$. By applying proposition 5.7.6, it thus remains to show that $\text{PER}(\varphi_0) \sqcup \left\lfloor T_{P'_{n+1}} \right\rfloor \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$. Let $\mathbf{\Sigma}^W_{\Uparrow} = \{\sigma_0 \in \mathbf{\Sigma} \mid \forall i \in \mathbb{N}, \langle C_0, \sigma_i \rangle \Downarrow \sigma_{i+1}, \sigma_{i+1}(b) = \mathtt{tt}\}$ be the set of *all* states under which the outer *while* loop diverges and define $\overline{W}$ such that for all $\sigma_0, \sigma'_0 \in \mathbf{\Sigma}$, $\sigma_0\, \overline{W}\, \sigma'_0$ iff $(\sigma_0, \sigma'_0 \in \mathbf{\Sigma} \backslash \mathbf{\Sigma}^W_{\Uparrow}$ or $\sigma_0, \sigma'_0 \in \mathbf{\Sigma}^W_{\Uparrow})$ and $\sigma_0 \lfloor T_{C_0} \rfloor \sigma'_0$ and $\forall i \geq 0$, $\langle C_0, \sigma_i \rangle \Downarrow \sigma_{i+1}$, $\langle C_0, \sigma'_i \rangle \Downarrow \sigma'_{i+1} \implies \sigma_{i+1} \lfloor T_{C_0} \rfloor \sigma'_{i+1}$. We know from lemma 5.7.8 that $\left\lfloor T_{P'_{n+1}} \right\rfloor \sqsubseteq \overline{W}$. We shall show that $\text{PER}(\varphi_0) \sqcup \overline{W} \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$. Suppose $\sigma_0, \sigma'_0 \in \mathit{dom}(\varphi_0)$ and that $\sigma_0\, O_{n+1}\, \sigma'_0$, then from the partitioning of states by O_{n+1} shown above, we know that $\sigma_0, \sigma'_0 \in \Sigma^W_{\Uparrow} \cap \mathit{dom}(\varphi_0)$ or $\sigma_0, \sigma'_0 \in \mathit{dom}(\varphi_0) \backslash \Sigma^W_{\Uparrow}$. By the induction hypothesis, for all $i \geq 0$ we have that $\{\sigma' \mid \sigma \in \mathit{dom}(\varphi_0), \langle C_i, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_{i+1})$ and $\text{PER}(\varphi_i) \sqcup \lfloor T_{C_0} \rfloor \sqsubseteq \text{PER}(\varphi_i) \sqcup O'_{i+1}$. Hence, by applying (2) of proposition 5.4.5, since for all $i \geq 0$ we have that $\mathit{dom}(\varphi_i) \subseteq \mathit{dom}(O'_i)$ and $O'_i \uplus O_{n+1} = O_{n+1}$, and $O'_i \Subset O'_{i+1}$, then $\sigma_0\, O_{n+1}\, \sigma'_0$ implies that for all $j \geq 0$, $\sigma_0\, O'_j\, \sigma'_0$. Thus, $\sigma_0\, O_{n+1}\, \sigma'_0$ implies $\sigma_0\, O'_1\, \sigma'_0 \implies \sigma_0 \lfloor T_{C_0} \rfloor \sigma'_0$ by the induction hypothesis, and furthermore we have that for all $i \geq 0, \langle C_0, \sigma_i \rangle \Downarrow \sigma_{i+1}$,

$\langle C_0, \sigma'_i \rangle \Downarrow \sigma'_{i+1} \implies \sigma_{i+1}\, O'_{i+1}\, \sigma'_{i+1} \implies \sigma_{i+1}\, \lfloor T_{C_0} \rfloor\, \sigma'_{i+1}$ by the induction hypothesis and because of (4) of lemma 5.4.8 since C_0 only modifies $\mathbf{TVar}$ variables and hence $\sigma_{i+1} \in \mathit{havoc}\mathbf{TVar}(\{\sigma_0\})$ and $\sigma'_{i+1} \in \mathit{havoc}\mathbf{TVar}(\{\sigma'_0\})$, and we already know that $\sigma_0\, O'_{i+1}\, \sigma'_0$ holds. Thus, $\sigma_0\, \overline{W}\, \sigma'_0$ holds, and hence $\text{PER}(\varphi_0) \sqcup \overline{W} \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$. Since by lemma 5.7.8 $\lfloor T_{P'_{n+1}} \rfloor \sqsubseteq \overline{W}$, it follows that $\text{PER}(\varphi_0) \sqcup \lfloor T_{P'_{n+1}} \rfloor \sqsubseteq \text{PER}(\varphi_0) \sqcup O_{n+1}$.

(B) We know by the induction hypothesis that for all $i \in \mathbb{N}$, and $\sigma \in \mathit{dom}(\varphi_0)$, $\langle C_i, \sigma \rangle \Downarrow \sigma' \implies \sigma' \in \mathit{dom}(\varphi_{i+1})$. But for any $\sigma \in \mathit{dom}(\varphi_0)$ such that $\langle P'_{n+1}, \sigma \rangle \Downarrow \sigma'$, then $\sigma'(b) = \mathbf{ff}$ since the *while* statement terminates and hence there exists a $j \in \mathbb{N}$ such that $\langle C_j, \sigma \rangle \Downarrow \sigma'$, which means that $\sigma' \in \mathit{dom}(\varphi_{j+1})$. Furthermore, since $\sigma'(b) = \mathbf{ff}$, it cannot be modified by further execution of C_0, and hence for all $k \geq j + 1$, we have also that $\sigma' \in \mathit{dom}(\varphi_k)$. Since there exists a $k \in \mathbb{N}$ such that $\sigma' \in \mathit{dom}(\varphi_k)$ and $\sigma'(b) = \mathbf{ff}$, then it is easy to see that for all $x \in \mathbf{Var}$, $\sigma' \in \mathit{dom}(E_{n+1}(x))$ by definition. Furthermore, since $\sigma' \in \mathit{dom}(\varphi_k)$, then $\sigma' \in \mathit{dom}(I'')$ defined as $I'' = \bigsqcup_{i \geq 0} I'_i$. Hence, $\sigma' \in \mathit{dom}(I_{n+1})$, which is defined as $I_{n+1} = \mathit{flow}(b : \mathbf{F}, (E_{n+1}, I'', O''))$. Finally, since $\sigma' \in \mathit{dom}(\varphi_k)$, by the domain preserving property of $\boxplus$, we know that $\sigma' \in O_{n+1}$, which is computed by taking the join $\boxplus$ of O'_k and some other PERs. Hence $\sigma' \in \mathit{dom}((E_{n+1}, I_{n+1}, O_{n+1}))$. That is, $\{\sigma' \mid \sigma \in \mathit{dom}(\varphi_0), \langle P'_{n+1}, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E_{n+1}, I_{n+1}, O_{n+1}))$. By combining this with the induction hypothesis on P_n, where we have that $\{\sigma' \mid \sigma \in \mathit{dom}((E, I, O)), \langle P_n, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}(\varphi_0)$, it then follows that $\{\sigma' \mid \sigma \in \mathit{dom}((E, I, O)), \langle P_{n+1}, \sigma \rangle \Downarrow \sigma'\} \subseteq \mathit{dom}((E_{n+1}, I_{n+1}, O_{n+1}))$.

(C) Since $\varphi_0 \in \mathbf{\Phi}_{init}$. We know from lemma 5.7.4 that $(E_{n+1}, I_{n+1}, O_{n+1}) \in \mathbf{\Phi}_{init}$.

□

A corollary to (A) of Theorem 5.7.10 shows the soundness property of the static analysis that links the information flow derived by the static analysis to the semantic information flow.

Corollary 5.7.11. *Let* $(E_\perp, I_\perp, O_\perp)$ P (E', I', O') *be the static analysis of the* While *program* P. *Then* $\lfloor T_P \rfloor \sqsubseteq O'$.

Proof. The proof follows easily from the (A) part of Theorem 5.7.10. Since PER$((E_\perp, I_\perp, O_\perp))$ = *all*, we have that $\lfloor T_P \rfloor \sqsubseteq O'$. □

5.7.1 Flow Sensitivity

The information flow analysis is *flow-sensitive* [NNH99], which means that the order of program execution matters to the analysis. For example, while the program $l := 0; l := h;$ `write` $l;$ is insecure, it is easy to show that the program $l := h; l := 0;$ `write` $l;$ is secure because it does not leak the secret h. Assuming that h contains a secret value and l is public, conventional type-based analyses, which are usually flow-insensitive, reject the latter program because of the initial assignment of h to l.

5.7.2 Termination Properties

Under the semantic attacker model, the attacker may be able to gain information when it determines that the program does not terminate. This information is

captured by the static analysis by taking a join of the O-component in the post-configuration of *while* statements with a PER which partitions states into those under which the *while* statement terminates and those under which it does not terminate. Furthermore, since subsequent statements after a diverging *while* statement cannot cause further information flow, the definition of the E- and I-components of the post-configuration of a *while* statement ensures that subsequent analysis is restricted to only those states under which the preceding *while* statement terminates. This is illustrated in the following program of Figure 5.17.

```
1 if (h ≤ 10) then
2     while (tt) do
3           skip
4     l := h;
5 else
6     l := h;
7 write l;
```

Figure 5.17: *Nontermination and unreachable code*

It is clear that this program will reveal the value of h when $h > 10$, but when $h \leq 10$ the program diverges preventing the assignment statement on line 4 and subsequent statements from being executed. Thus, the attacker only learns that $h \leq 10$ when the program diverges. Assume that the pre-configuration of the analysis is $(E_\perp, I_\perp, O_\perp)$, and that $\mathbf{TVar} = \{l\}$ and $\mathbf{IVar} = \{h\}$. The static analysis derives the information release as follows. The implicit context on entering the *then* branch is the PER $I_1 \equiv \{\{\sigma \in \mathbf{\Sigma} \mid \sigma(h) \leq 10\}\}$. Since $\mathbf{tt} : \mathbf{F} = \varnothing$ is the empty PER, the post-configuration of the *while* statement is (E_2, I_2, O_2) where $\forall x \in \mathbf{Var}, E_2(x) = I_2 = \varnothing$, and the attacker's knowledge is given by $O_2 \equiv \{\{\sigma \in \mathbf{\Sigma} \mid \sigma(h) \leq 10\}, \{\sigma \in \mathbf{\Sigma} \mid \sigma(h) > 10\}\}$. The meaning of E_2 and

I_2 is that the program points after the *while* statement are unreachable since the PER $\varnothing$ relates no state, and O_2 means that the attacker can distinguish between the terminating and non-terminating trace of the program. Applying the *assignment* rule on line 4 does not change the pre-configuration (E_2, I_2, O_2). In fact, for any statement whose pre-configuration is (E_2, I_2, O_2), the post-configuration is the same since for any expression e and PER ϕ on values and variable l we have that $\mathit{flow}(e : \phi, (E_2, I_2, O_2)) = \mathit{aflow}(l := e, (E_2, I_2, O_2)) = \varnothing$ such that $\mathit{flow}(e : \phi, (E_2, I_2, O_2)) \uplus O_2 = O_2$ (since for any PER R, $R \uplus \varnothing = R$).

Now for the *else* branch of the program, the implicit context is given by $I_2' \equiv \{\{\sigma \in \mathbf{\Sigma} \mid \sigma(h) > 10\}\}$ and after the assignment we have the post-configuration $(E_2', I_2', \bot)$, where $E_2' = E_\bot[l \mapsto \mathit{aflow}(l := h, (E_\bot, I_2', \mathit{all}))]$ and where $\mathit{aflow}(l := h, (E_\bot, I_2', \mathit{all})) \equiv \{\{\sigma' \in \mathbf{\Sigma} \mid \sigma'(l) = \sigma'(h) = \sigma(h)\} \mid \sigma \in \mathbf{\Sigma}, \sigma(h) > 10\}$. Thus, $E_2'(l)$ is the PER which requires that the value of h and l be the same (due to the assignment) and that $h > 10$ (due to branching). Applying the *if* statement rule, the post-configuration is thus $(E_2, I_2, O_2) \uplus_{(E_\bot, I_\bot, O_\bot)} (E_2', I_2', \mathit{all}) = (E_3, I_2', O_2)$ where $\forall x, \in \mathbf{Var}, E_3(x) = E_2'(x) \sqcup I_2'$. This is the expected result, since E_3 and I_2' both restrict the set of states to those where $h > 10$ (the terminating traces) and where $h = l$ (due to the assignment on line 6).

Finally, the *write* statements reveals the value of h whenever it is greater than 10. This is clear from the result (E_3, I_2', O_2) `write` l (E_3, I_2', O_3) where $O_3 = O_2 \uplus \mathit{flow}(l : \mathit{id}, (E_3, I_2', O_2)) \equiv \{\{\sigma' \in \mathbf{\Sigma} \mid \sigma'(h) = \sigma(h)\}, \{\sigma'' \in \mathbf{\Sigma} \mid \sigma''(h) \leq 10\} \mid \sigma \in \mathbf{\Sigma}, \sigma(h) > 10\}$. Thus, the final result demonstrates that the attacker either learns the value of h whenever $h > 10$ or learns that $h \leq 10$ since $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma O_3 \sigma'$ iff $\sigma(h) = \sigma'(h) \geq 10$ or $\sigma(h), \sigma'(h) \leq 10$, which agrees with the intuition about the information flow of the program.

5.7.3 Dead Code Analysis

In program analysis, *dead code* (also known as unreachable code) refers to a portion of program code that can never be executed [NNH99, Muc97]. Intuitively, from an information flow perspective dead code should never cause information flow. We encountered a dead code scenario in the previous analysis example because the code after the *while* statement is unreachable. Dead code may also arise due to program branching, where the conditional guard always evaluates to *false*. An example is shown in Figure 5.18. In this example, the implicit context in the *then* branch is the empty PER, and therefore all the commands in that branch are harmless because they cannot be executed. This is revealed by the analysis of the program, where for any pre-configuration φ of the *if* statement, the post-configuration of the analysis is also φ. This makes sense (from an information flow perspective, and also from a semantic point of view) because this program behaves exactly like a `skip` statement - revealing no information at all.

```
if (ff) then
  write h;
else
  skip;
```

Figure 5.18: *A dead code scenario.*

5.7.4 Implicit Flow Approximation

Under certain circumstances implicit information flows are approximated by our analysis. For example, the analysis of the program

$$(\texttt{if(}h = 10\texttt{)then}\, l := 1\, \texttt{else}\, l := 1);\ \texttt{write}\, l$$

says that the attacker either learns that $h = 10$ or that $h \neq 10$. However, since the attacker cannot determine which program path is taken by observing the output the result is only an approximation of the actual information flow.

Additionally, because the attacker's knowledge may only increase in the analysis it is possible that the analysis may be less precise under certain program compositions which, which does not increase the semantic attacker's knowledge about secrets, but where the individual programs themselves would reveal the information. For example, consider the programs

$$P_1 \triangleq \texttt{if(}h = 10\texttt{)then write}\, 1\, \texttt{else skip}$$

and

$$P_2 \triangleq \texttt{if(}h \neq 10\texttt{)then write}\, 1\, \texttt{else skip}.$$

In both cases the analysis precisely determines that the attacker learns whether $h = 10$ or not. However, if these programs are composed as $P_1; P_2$ then the attacker cannot gain any information about h. By combining the information released in both subprograms, the analysis overapproximates the information flow, and determines that the attacker learns whether $h = 10$ or not. This is related to the previous implicit flow problem since the program $P_1; P_2$ is observationally similar

to the program $\texttt{if(}h = 10\texttt{)then write}\,1\,\texttt{else write}\,1$, which does not reveal branching information. More generally, this problem is related to the proof of observational and semantic equivalence of program branches in a given context, for which techniques from [Ben04], presented in the next section, are useful.

5.8 Relational Correctness

Benton [Ben04] introduced proof techniques whereby the correctness of static analyses and the program transformations that they enable may be shown by using relations, rather than predicates, to express program properties. The key observation is that one may view the semantics of types as a special kind of relation, rather than as predicates. Thus, a typing judgement of the form:

$$\Gamma \vdash M = M' : A$$

which asserts that under the assumption Γ, the terms M and M' are equal at type A induces a relation over terms. In particular, types may now be interpreted as partial equivalence relations over some untyped universe. The idea now is that types extensionally specify properties that are preserved by program transformations, and typing judgements identify terms that can be rewritten while preserving the observable semantics specified by those extensional properties.

The language setting of [Ben04] is similar to *While*, with the exception of the *write* construct and the fact that all variables are of integer data type. However, boolean expressions are constructed from integer expressions and constants in the usual way. Expression types are thus taken from $\{\mathit{int}, \mathit{bool}\} \ni \tau$, whose denota-

tions are sets, and where $[\![\mathit{int}]\!] = \mathbb{Z}$ and $[\![\mathit{bool}]\!] = \mathbb{B}$. The program semantics is presented in the standard denotational style [Win93].

Some non-standard expression types were introduced in [Ben04] for τ-expressions:

$$\phi_\tau ::= \mathbb{F}_\tau \mid \{v\}_\tau \mid \Delta_\tau \mid \mathbb{T}_\tau.$$

Intuitively, $\mathbb{F}_\tau$ is the empty expression type, $\{v\}_\tau$ is the type of constant expressions whose value is $v \in [\![\tau]\!]$, Δ_τ is the type of an expression that we do not know its value, and $\mathbb{T}_\tau$ is the type of an expression that we do not care about its value. The metavariable e_τ ranges over τ-expressions. The denotation of ϕ_τ is a relation on $[\![\tau]\!]$ as follows:

$$[\![\mathbb{F}_\tau]\!] = \varnothing, \quad [\![\{v\}_\tau]\!] = \{(v, v)\}, \quad [\![\Delta_\tau]\!] = \{(v, v) \mid v \in [\![\tau]\!]\}, \quad [\![\mathbb{T}_\tau]\!] = [\![\tau]\!] \times [\![\tau]\!].$$

It is clear that $[\![\phi_\tau]\!]$ is a PER on $[\![\tau]\!]$, we shall thus use our existing notations (dropping the τ-subscripts when it is clear from the context) for these non-standard types:

$$\mathbb{F}_\tau \equiv \varnothing_\tau, \quad \{v\}_\tau \equiv \mathit{id}^v_\tau, \quad \Delta_\tau \equiv \mathit{id}_\tau, \quad \mathbb{T}_\tau \equiv \mathit{all}_\tau.$$

These are the PERs, where $\varnothing_\tau$ relates no τ-values, id^v_τ only relates v to itself, id_τ only relates any $v \in [\![\tau]\!]$ to itself, and all_τ relates all pairs of τ-values.

[Ben04] also introduced state types, which are finite maps from (*int*) variables to *int*-expression types, and whose denotations are PERs on the set of states Σ.

State types are written as lists[1]:

$$\Theta ::= - \mid \Theta, x : \phi_{int}.$$

The interpretation of state types is a PER on $\mathbf{\Sigma}$, where $[\![-]\!] = \mathbf{\Sigma} \times \mathbf{\Sigma}$ and $\Theta, x : \phi_{int} = [\![\Theta]\!] \cap \{(\sigma, \sigma') \mid (\sigma(x), \sigma'(x)) \in [\![\phi_{int}]\!]\}$. The full inference system for program analysis and transformation which tracks *Dependency*, *Dead Code* and *Constancy* information *DDCC* is shown in Figure 5.19.

Expression and state types are ordered by a subtyping relation $\leq$ as follows: $\varnothing_\tau \leq \phi_\tau, \quad id^v_\tau \leq id_\tau, \quad \phi_\tau \leq all_\tau, \quad \phi_\tau \leq \phi_\tau, \quad \dfrac{\phi_\tau \leq \phi'_\tau \quad \phi'_\tau \leq \phi''_\tau}{\phi_\tau \leq \phi''_\tau}$ for expression types, and for state types we have $\Theta \leq -, \quad \Theta, x : \varnothing_{int} \leq \Theta', \quad \dfrac{\Theta \leq \Theta'}{\Theta \leq \Theta', x : all_\tau}$, $\dfrac{\Theta \leq \Theta' \quad \phi_{int} \leq \phi'_{int}}{\Theta, x : \phi_{int} \leq \Theta', x : \phi'_{int}}$. Note that the ordering $\leq$ is the dual of our ordering $\sqsubseteq$ on relations.

In Figure 5.19 *op* stands for any applicable binary operation on τ expressions and $\hat{op}$ is an abstract interpretation of *op* over the domain of expression types. As usual, $\hat{op}$ is a sound abstraction of *op* if $\forall (x, x') \in [\![\phi_\tau]\!], (y, y') \in [\![\phi'_\tau]\!].(x \; op \; y, x' \; op \; y') \in [\![\phi_\tau \; \hat{op} \; \phi'_\tau]\!]$.

5.8.1 Judgements

There are two basic typing judgements in *DDCC*:

$$\vdash e_\tau \sim e'_\tau : \Theta \Rightarrow \phi_\tau$$

[1]The state type $\Theta, x : \phi_{int}$ means that the variable x does not appear in Θ, and that x has the expression type ϕ_{int}.

Subtyping and Structural

$$\vdash c \sim c' : \Theta, x : \varnothing_{int} \Rightarrow \Phi' \qquad \vdash e_\tau \sim e'_\tau : \Theta \Rightarrow all_\tau \qquad \vdash e_\tau \sim e'_\tau : \Theta, x : \varnothing_{int} \Rightarrow \phi_\tau$$

$$\frac{\vdash e_\tau \sim e'_\tau : \Theta \Rightarrow \phi_\tau}{\vdash e'_\tau \sim e_\tau : \Theta \Rightarrow \phi_\tau} \qquad \frac{\vdash e_\tau \sim e'_\tau : \Theta \Rightarrow \phi_\tau \quad \Theta' \leq \Theta \quad \phi_\tau \leq \phi'_\tau}{\vdash e_\tau \sim e'_\tau : \Theta' \Rightarrow \phi'_\tau} \qquad \frac{\vdash c \sim c' : \Theta \Rightarrow \Theta'}{\vdash c' \sim c : \Theta \Rightarrow \Theta'}$$

$$\frac{\vdash e_\tau \sim e'_\tau : \Theta \Rightarrow \phi_\tau \quad \vdash e'_\tau \sim e''_\tau : \Theta \Rightarrow \phi_\tau}{\vdash e_\tau \sim e''_\tau : \Theta \Rightarrow \phi_\tau} \qquad \frac{\vdash c \sim c' : \Theta_1 \Rightarrow \Theta_2 \quad \Theta'_1 \leq \Theta_1 \quad \Theta_2 \leq \Theta'_2}{\vdash c \sim c' : \Theta'_1 \Rightarrow \Theta'_2}$$

$$\frac{\vdash c \sim c' : \Theta \Rightarrow \Theta' \quad \vdash c' \sim c'' : \Theta \Rightarrow \Theta'}{\vdash c \sim c'' : \Theta \Rightarrow \Theta'}$$

Expressions

$$\vdash x \sim x : \Theta, x : \phi_{int} \Rightarrow \phi_{int} \qquad \vdash \mathbf{n} \sim \mathbf{n} : \Theta \Rightarrow id^n_{int} \qquad \vdash \mathbf{b} \sim \mathbf{b} : \Theta \Rightarrow id^b_{bool}$$

$$\frac{\vdash e_\tau \sim f_\tau : \Theta \Rightarrow \phi_\tau \quad \vdash e'_\tau \sim f'_\tau : \Theta \Rightarrow \phi'_\tau}{\vdash e_\tau \; op \; e'_\tau \sim f_\tau \; op \; f'_\tau : \Theta \Rightarrow (\phi_\tau \; \hat{op} \; \phi'_\tau)}$$

Commands

$$\vdash \texttt{skip} \sim \texttt{skip} : \Theta \Rightarrow \Theta \qquad \frac{\vdash c_1 \sim c'_1 : \Theta \Rightarrow \Theta' \quad \vdash c_2 \sim c'_2 : \Theta' \Rightarrow \Theta''}{\vdash (c_1; c_2) \sim (c_1; c'_2) : \Theta \Rightarrow \Theta''}$$

$$\frac{\vdash e \sim e' : \Theta, z : \phi_{int} \Rightarrow \phi'_{int}}{\vdash z := e \sim z := e' : \Theta, z : \phi_{int} \Rightarrow \Theta, z : \phi'_{int}} \qquad \frac{\vdash b \sim b' : \Theta \Rightarrow id_{bool} \quad \vdash c \sim c' : \Theta \Rightarrow \Theta}{\vdash \texttt{while}\,(b)\,\texttt{do}\,c \sim \texttt{while}\,(b')\,\texttt{do}\,c' : \Theta \Rightarrow \Theta}$$

$$\frac{\vdash b \sim b' : \Theta \Rightarrow id_{bool} \quad \vdash c_1 \sim c'_1 : \Theta \Rightarrow \Theta' \quad \vdash c_2 \sim c'_2 : \Theta \Rightarrow \Theta'}{\vdash \texttt{if}\,(b)\,\texttt{then}\,c_1\,\texttt{else}\,c_2 \sim \texttt{if}(b')\texttt{then}\,c'_1\,\texttt{else}\,c'_2 : \Theta \Rightarrow \Theta'}$$

Figure 5.19: *Core DDCC System [Ben04].*

which intuitively expresses that under the constraint Θ on states, the τ-expressions e_τ and e'_τ are interchangeable as they both produce indistinguishable "observations" under ϕ_τ; similarly, for commands,

$$\vdash c \sim c' : \Theta \Rightarrow \Theta'$$

$$\vdash \texttt{skip} \sim \texttt{skip} : \Theta \Rightarrow \Theta \qquad \frac{\vdash c_1 \sim c_1' : \Theta \Rightarrow \Theta' \quad \vdash c_2 \sim c_2' : \Theta' \Rightarrow \Theta''}{\vdash c_1; c_2 \sim c_1; c_2' : \Theta \Rightarrow \Theta''}$$

$$\vdash x := e \sim y := e' : \Theta[e\langle 1\rangle / x\langle 1\rangle, e'\langle 2\rangle / y\langle 2\rangle] \Rightarrow \Theta$$

$$\frac{\vdash c_1 \sim c_1' : \Theta \wedge (b\langle 1\rangle \wedge b'\langle 2\rangle) \Rightarrow \Theta' \quad \vdash c_2 \sim c_2' : \Theta \wedge \mathit{not}(b\langle 1\rangle \vee b'\langle 2\rangle) \Rightarrow \Theta'}{\vdash \texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \sim \texttt{if(}b'\texttt{)then}\ c_1'\ \texttt{else}\ c_2' : \Theta \wedge (b\langle 1\rangle = b'\langle 2\rangle) \Rightarrow \Theta'}$$

$$\frac{\vdash c \sim c' : \Theta \wedge (b\langle 1\rangle \wedge b'\langle 2\rangle) \Rightarrow \Theta \wedge (b\langle 1\rangle = b'\langle 2\rangle)}{\vdash \texttt{while}\ (b)\ \texttt{do}\ c \sim \texttt{while}\ (b')\ \texttt{do}\ c' : \Theta \wedge (b\langle 1\rangle = b'\langle 2\rangle) \Rightarrow \Theta \wedge \mathit{not}(b\langle 1\rangle \vee b'\langle 2\rangle)}$$

$$\frac{c \sim c' : \Theta_1 \Rightarrow \Theta_2 \quad \vDash \Theta_1' \leq \Theta_1 \quad \vDash \Theta_2 \leq \Theta_2'}{c \sim c' : \Theta_1' \Rightarrow \Theta_2'} \qquad \frac{c \sim c' : \Theta \Rightarrow \Theta' \quad \vDash \mathrm{PER}(\Theta \Rightarrow \Theta')}{c' \sim c : \Theta \Rightarrow \Theta'}$$

$$\frac{c \sim c' : \Theta \Rightarrow \Theta' \quad c' \sim c'' : \Theta \Rightarrow \Theta' \quad \vDash \mathrm{PER}(\Theta \Rightarrow \Theta')}{c \sim c'' : \Theta \Rightarrow \Theta'}$$

Figure 5.20: *Core Relational Hoare Logic [Ben04].*

intuitively means that under the context Θ as the precondition of execution state and the postconditional requirement Θ' on states, c and c' are can be interchanged. We have the following semantic definitions:

$$[\![\Theta \Rightarrow \phi_\tau]\!] \triangleq \{(e, e') \mid \forall (\sigma, \sigma') \in [\![\Theta]\!].(\sigma(e), \sigma'(e')) \in [\![\phi_\tau]\!]\}$$

and

$$[\![\Theta \Rightarrow \Theta']\!] \triangleq \{(c, c') \mid \forall (\sigma, \sigma') \in [\![\Theta]\!].([\![c]\!](\sigma), [\![c']\!](\sigma')) \in [\![\Theta']\!]_\perp\}.$$

5.8.2 Relational Hoare Logic

One of the properties that cannot be expressed under the *DDCC* type system is the knowledge of how the boolean guard evaluates when control is passed to a particular branch of a conditional statement. This is addressed by the system called *Relational Hoare Logic* (*RHL*) where one may specify such properties, and is intended to be a basis for developing various specific program analyses and transformations. This system is presented in Figure 5.20.

RHL defines generalised expressions and relational assertions as follows:

$$\textit{gexp} \ni \textit{GE} ::= \mathrm{n} \mid x\langle 1\rangle \mid x\langle 2\rangle \mid \textit{GE iop GE}$$

$$\textit{relexp} \ni ::= \mathrm{b} \mid \textit{GE bop GE} \mid \textit{not } \Theta \mid \Theta \textit{ lop } \Theta.$$

The semantics of generalised expressions and relational assertions are given by:

$$
\begin{array}{lcl}
[\![GE]\!] & \in & \mathbf{\Sigma} \times \mathbf{\Sigma} \to \mathbb{Z} \\
[\![\mathrm{n}]\!] & = & n \\
[\![x\langle 1\rangle]\!](\sigma_1, \sigma_2) & = & \sigma_1(x) \\
[\![x\langle 2\rangle]\!](\sigma_1, \sigma_2) & = & \sigma_2(x) \\
[\![e \text{ iop } e']\!](\sigma_1, \sigma_2) & = & ([\![e]\!](\sigma_1, \sigma_2)) \text{ iop } ([\![e']\!](\sigma_1, \sigma_2))
\end{array}
$$

$$
\begin{array}{lcl}
[\![\Theta]\!] & \subseteq & \mathbf{\Sigma} \times \mathbf{\Sigma} \\
 & = & \{(\sigma, \sigma') \mid \chi_\Theta(\sigma, \sigma') = \mathbf{tt}\} \\
\chi_{\mathbf{tt}}(\sigma, \sigma') & = & \mathbf{tt} \\
\chi_{\mathbf{ff}}(\sigma, \sigma') & = & \mathbf{ff} \\
\chi_{b \; bop \; b'}(\sigma, \sigma') & = & [\![b]\!](\sigma, \sigma') \text{ bop } [\![b']\!](\sigma, \sigma') \\
\chi_{\Theta \; lop \; \Theta'}(\sigma, \sigma') & = & \chi_\Theta(\sigma, \sigma') \text{ lop } \chi_{\Theta'}(\sigma, \sigma') \\
\chi_{not\Theta}(\sigma, \sigma') & = & \neg(\chi_\Theta(\sigma, \sigma'))
\end{array}
$$

The basic RHL judgement is of the form $c \sim c' : \Theta \Rightarrow \Theta'$, and the meaning of this judgement is given by

$$\vDash c \sim c' : \Theta \Rightarrow \Theta' \equiv \forall (\sigma, \sigma') \in [\![\Theta]\!].([\![c]\!](\sigma), [\![c']\!](\sigma')) \in [\![\Theta']\!]_\perp.$$

The lift of $[\![\Theta']\!]$ is denoted by the bottom reflecting relation $[\![\Theta']\!]_\perp = [\![\Theta']\!] \cup \{(\perp, \perp)\}$ and $[\![c]\!] \in \mathbf{\Sigma} \to \mathbf{\Sigma}_\perp$ is the denotational semantics of c, which is given in the category of predomains and continuous functions. Furthermore, we have the following

auxiliary judgements and their meanings:

$$\vDash \Theta \leq \Theta' \equiv [\![\Theta]\!] \subseteq [\![\Theta']\!]$$
$$\vDash \mathrm{PER}(\Theta) \equiv ([\![\Theta]\!] \circ [\![\Theta]\!]) \text{ and } ([\![\Theta]\!]^{-1} \subseteq [\![\Theta]\!]).$$

5.8.3 Static Analysis

The *DDCC* and the *RHL* system may be used to provide proofs of correctness of program transformation as well as static analysis. For static analysis, emphasis is laid on the semantics of (expression and command) types as a description of program properties. Thus, $\vdash e_\tau \sim e_\tau : \Theta \Rightarrow \phi_\tau$ or simply $\vdash e_\tau : \Theta \Rightarrow \phi_\tau$ describes a property of the evaluation of e_τ under the constraint Θ on states: that is, given the context Θ, the evaluation of e_τ is indistinguishable via the PER ϕ_τ. This interpretation is the same as the definition $e : \phi$ (dropping the τ subscripts) in the information flow analysis, which is the greatest PER on states (using the order $\leq$) for which the evaluation of e is indistinguishable under ϕ. In other words, for all $\Theta \leq e : \phi$ we have $\vdash e : \Theta \Rightarrow \phi$. As an interpretation of information released, the attacker that cannot distinguish evaluations of e that are related by ϕ, gains at most the information modelled by the PER $e : \phi$ on program states. Thus, for the semantic attacker in our analysis $\phi = id$, being able to observe precisely the evaluation of expressions as prescribed by the operational semantics. In the following, we shall omit the τ subscripts.

For program commands, the static analysis of c is specified under the *DDCC* and *RHL* system as $\vdash c \sim c : \Theta \Rightarrow \Theta'$, or simply as $\vdash c : \Theta \Rightarrow \Theta'$, which may be interpreted to mean that under the context prescribed by the pre-relation Θ, the execution of c satisfies the properties described by the post-relation Θ' on states.

Under *DDCC*, the relations Θ and Θ' are PERs over program states, although they may not necessarily be under the *RHL* system. We restrict our discussions to PERs only.

5.8.4 Improving the Precision of Information Flow Analysis

The proofs from the *DDCC* and *RHL* may be used to improve the precision of our information flow analysis. For example, the approximation of information flow that may result due to equivalent branches in a conditional statement can be made more precise. Suppose the subprograms c_1 and c_2 of a conditional *if* statement contain no *write* statements and that $\vdash c_1 \sim c_2 : \Theta \Rightarrow \Theta'$, then by this equivalent branches property, the analysis of $\texttt{if(}b\texttt{)then}\, c_1\, \texttt{else}\, c_2$ under the configuration φ such that $\mathit{dom}(\varphi) \subseteq \mathit{dom}([\![\Theta]\!])$ may be replaced with

$$\frac{\mathit{dom}(\varphi) \subseteq \mathit{dom}([\![\Theta]\!]) \quad \vdash c_1 \sim c_2 : \Theta \Rightarrow \Theta' \quad \varphi\, c_1\, \varphi'}{\varphi\, \texttt{if(}b\texttt{)then}\, c_1\, \texttt{else}\, c_2\, \varphi'}. \tag{5.5}$$

This takes advantage of the fact that the subprograms are semantically equivalent under the context provided by Θ. By applying this, the analysis of the earlier program, $\texttt{if(}h = 10\texttt{)then}\, l := 1\, \texttt{else}\, l := 1; \texttt{write}\, l$, becomes precise under any starting pre-configuration. However, (5.5) does not help with the program $\texttt{if(}h = 10\texttt{)then}\, l := h\, \texttt{else}\, l := 10; \texttt{write}\, l$, because $l := h$ and $l := 10$ are not equivalent under all contexts. In this case however, the program releases no information about h because the value of l after the conditional is independent of the execution path. We can improve the precision in such a case by taking advantage of the knowledge of how the boolean guard evaluates when conditional subprograms are executed. This can be specified under the *RHL* system by using

the *common branch* [Ben04] rule and the following:

$$\frac{\mathit{dom}(\varphi) \subseteq \mathit{dom}(\llbracket \Theta \rrbracket) \quad \vdash c_1 : \Theta \wedge b\langle 1 \rangle \Rightarrow \Theta' \quad \vdash c_2 : \Theta \wedge \mathit{not}\, b\langle 1 \rangle \Rightarrow \Theta' \quad \varphi\, c_1\, \varphi'}{\varphi\, \texttt{if(}b\texttt{)then}\, c_1\, \texttt{else}\, c_2\, \varphi'}. \tag{5.6}$$

Although the language of *RHL* does not have an explicit output construct, we can deal with *write* statements under the *RHL* system by extending state with fresh ("output") variables, which record the value of program output, by adding the rule

$$\frac{\vdash e \sim x : \Theta \Rightarrow \mathit{id}}{\vdash \texttt{write}\, e \sim \texttt{write}\, x : \Theta \Rightarrow \Theta, x : \mathit{id}} \tag{5.7}$$

This rule extends state with the new variable x, which is observationally indistinguishable from the expression e under the context Θ. With this extension we can specify when branches are observationally equivalent. For example, by combining (5.7) and (5.6) we are able to derive a more precise analysis for the program $\texttt{if(}h = 10\texttt{)then write}\, 1\, \texttt{else write}\, 1$.

Summary In this chapter we have presented a static information flow analysis technique for *While* programs using lattices of PERs over the program state. This analysis is developed relative to the semantic attacker model, which can observe the execution of programs in the usual way, with the additional ability to determine whether the program terminates or not. We proved the correctness of the analysis. In the next chapter, we shall show how abstract interpretation techniques can be employed to adapt the analysis to less precise lattices, which may be tailored towards a particular policy to make the static analysis more tractable.

Chapter 6

Abstract Information Flow Analysis

In the previous chapter we presented a static analysis technique based on PERs for the analysis of information flow in *While* programs. In practice, on one hand, we may not need the level of expressiveness, for example, "the attacker learns the secret h when the input $l = 10$" that may be expressed by using the PER lattice. Instead, for example, we may just be interested in knowing whether the secret h may ever be released. Thus, depending on the policy to be enforced, it may be possible to choose a less expressive lattice of information, which may lead to the simplification of the analysis. On the other hand, a large or complex lattice of information may be computationally prohibitive, necessitating the choice of a less computationally expensive lattice to make the analysis of information flow more tractable. This chapter demonstrates how abstract interpretation techniques may be used to address this problem by allowing us to perform correct analyses over simpler (but possibly less-precise) information lattices.

6.1 Abstract Interpretation

Abstract interpretation is a standard formal framework in which a provably *correct* static analysis may be performed over a non-standard *abstract* domain [CC77, CC79, CC92, NNH99]. The abstract domain is defined relative to a standard *concrete* domain, where elements of the abstract domain encode properties of elements in the concrete domain. The most important requirement for abstract interpretation is that of *correctness*, so that program properties which are derived under the non-standard abstract interpretation are satisfied by the standard *concrete* interpretation. The notion of the "abstract domain", however, is relative to a chosen "concrete domain", which may itself be abstract relative to another domain.

The process of selecting an abstract domain often involves approximations, which may make the analysis less precise. For example, we might be interested in studying the properties of a program, which computes with integer values, and how it transforms these values over an abstract domain of *sign* and *parity*. Thus, when the program output is 2, an abstract analysis which judges the output of the program as a positive even integer is correct, although less precise than the concrete analysis 2. We may choose an even more abstract domain relative to the parity-sign domain, which contains only sign information and which judges the output even less precisely as a positive integer. While some precision may be sacrificed, the choice of the space over which analysis is performed can sometimes determine whether an analysis will be tractable or not. Traditionally, the space over which an analysis is performed is often arranged as a complete lattice such that the lattice order relation specifies the relative degree of precision of the judgements of analysis [NNH99].

6.1.1 Design Space for Approximate Analyses

The more precise an analysis is, usually the larger and more complex its property space will be because fine-grained properties can be represented. In abstract interpretation, the passage from more complex concrete domains to simpler, but possibly less precise, abstract domains is usually formalised as a *Galois connection* [CC79]. A Galois connection between two complete lattices $\langle A_1, \sqsubseteq_1 \rangle$ and $\langle A_2, \sqsubseteq_2 \rangle$ is defined via a pair of adjoint functions (α, γ), where the *abstraction function* $\alpha : A_1 \to A_2$ maps elements in A_1 to their abstraction in A_2 and the *concretisation function* $\gamma : A_2 \to A_1$ expresses the meaning of elements of A_2 using the elements of A_1. The quadruple $(A_1, \alpha, \gamma, A_2)$ is said to be a Galois connection iff α and γ are total and all $a_1 \in A_1$ and $a_2 \in A_2$ satisfy the property

$$\alpha(a_1) \sqsubseteq_2 a_2 \iff a_1 \sqsubseteq_1 \gamma(a_2). \tag{6.1}$$

The meaning of this definition is that if a_2 safely approximates the abstraction of a_1 (that is, $\alpha(a_1) \sqsubseteq_2 a_2$), then the concretisation of a_2 must safely approximate a_1 (that is, $a_1 \sqsubseteq_1 \gamma(a_2)$) [NNH99]. Suppose id_A is the identity function on set A, an alternative, but equivalent, formulation of (6.1) is that $(A_1, \alpha, \gamma, A_2)$ is a Galois connection iff α and γ are monotone and also satisfy the property

$$id_{A_1} \sqsubseteq_1 \gamma \circ \alpha \text{ and } \alpha \circ \gamma \sqsubseteq_2 id_{A_2}. \tag{6.2}$$

It is sufficient to specify just one of the adjunction α, γ in a Galois connection because one is uniquely determined by the other: for all $a_1 \in A_1$ and $a_2 \in A_2$, $\alpha(a_1) = \bigsqcap\{a_1' \mid a_1 \sqsubseteq_1 \gamma(a_1')\}$ and $\gamma(a_2) = \bigsqcup\{a_2' \mid \alpha(a_2') \sqsubseteq_2 a_2\}$ [NNH99].

6.2 Dependency Analysis

By applying abstract interpretation techniques we shall show in this section how to transfer the information flow analysis of Chapter 5 to a suitable abstract lattice. The main steps are standard and fairly mechanical, which involves the choice of abstraction function from PERs to the chosen abstract lattice, and the definition of sound abstractions for the operations on PERs. This is illustrated by the development of a termination-sensitive dependency analysis for *While* programs.

6.2.1 Dependency Abstractions

PERs over the set of program states represent information about program variables. This information can be interpreted as variable dependencies, where the dependency that a PER encodes is the set of variables that the PER contains information about. We shall start by defining information abstractions mapping PERs to lattices of variable dependencies, by which we can extract dependency information from PERs.

Definition 6.2.1 (Dependency Abstraction)**.** *Let* $\mathbf{Var}$ *be the set of variables of a program whose set of states is* $\mathbf{\Sigma}$*. The set* $\mathbf{Var}$ *is assumed to be finite. Define* $\mathcal{L} \triangleq \langle \mathcal{P}(\mathbf{Var}), \subseteq \rangle$ *and* $\mathcal{L}_2 \triangleq \langle \mathcal{P}(\mathcal{P}(\mathbf{Var})), \subseteq \rangle$ *to be lattices ordered by the subset inclusion order. Hence, the natural join operation on the lattices* $\mathcal{L}$ *and* $\mathcal{L}_2$ *is the set union operation* $\cup$*.*

Define the dependency operation $\Delta : \mathcal{P}(\mathbf{\Sigma}) \to \mathcal{P}(\mathbf{Var})$ *such that for any* $\Sigma \subseteq \mathbf{\Sigma}$

$$\Delta(\Sigma) \triangleq \{x \in \mathbf{Var} \mid X = \{x\},\ \mathit{havoc}X(\Sigma) \neq \Sigma\}.$$

Define the dependency abstractions $\alpha_{\mathcal{L}} : \mathit{PER}(\mathbf{\Sigma}) \to \mathcal{L}$ *and* $\alpha_{\mathcal{L}_2} : \mathit{PER}(\mathbf{\Sigma}) \to \mathcal{L}_2$ *on PERs such that for any* $R \in \mathit{PER}(\mathbf{\Sigma})$

$$\alpha_{\mathcal{L}}(R) \triangleq \bigcup_{\sigma \in \mathit{dom}(R)} \Delta([\sigma]_R)$$

and

$$\alpha_{\mathcal{L}_2}(R) \triangleq \{\Delta([\sigma]_R) \mid \sigma \in \mathit{dom}(R)\}.$$

Define a compositional *join operation* $\sqcup$ *on the lattice* $\mathcal{L}_2$ *such that for any* $X, Y \in \mathcal{L}_2$, $X \sqcup Y = \{Z \cup Z' \mid Z \in X, Z' \in Y\}$ *and* $X \sqcup \varnothing = \varnothing \sqcup X = X$. *The natural extension of* $\sqcup$ *to subsets of* $\mathcal{L}_2$, *for any* $\mathcal{X} = \{X_j \mid j \in J\} \subseteq \mathcal{L}_2$, *is given by* $\bigsqcup \mathcal{X} = \{\bigcup_{j \in J} Z_j \mid Z_j \in X_j\}$.

An element $X \in \mathcal{L}_2$ *is said to represent* disjunctive dependency *about variables* $x, y \in \mathbf{Var}$ *iff for all* $Z \in X, x \in Z \implies y \notin Z$ *and* $y \in Z \implies x \notin Z$.

A non-disjunctive interpretation of elements of $\mathcal{L}_2$ *on the lattice* $\mathcal{L}$ *is given by the function* $\alpha_{\cup} : \mathcal{L}_2 \to \mathcal{L}$ *which is defined for any* $X \in \mathcal{L}_2$ *as the union* $\alpha_{\cup}(X) = \bigcup_{Z \in X} Z$.

The lattice $\mathcal{L}$ models variable dependency information. For example, in the program $l := h_1 + h_2 + h_3$, the set $\{h_1, h_2, h_3\}$ models the dependency of l on h_1 and h_2 and h_3 after the assignment. The motivation behind the lattice $\mathcal{L}_2$ is to differentiate this dependency from, for example, the dependency $\{\{h_1, h_2\}, \{h_1, h_3\}\}$ on the lattice $\mathcal{L}_2$, due to the program $\texttt{if}(h_1)\texttt{then}\, l := h_2\, \texttt{else}\, l := h_3$, where the dependency $\{\{h_1, h_2\}, \{h_1, h_3\}\}$ of l is interpreted to mean that l may be dependent on h_1 and h_2, or on h_1 and h_3; but never on h_2 and h_3 at the same time. This is a disjunctive dependency as defined above. Let us now show some properties

of the definitions.

Lemma 6.2.2. *Let $\Sigma_1, \Sigma_2 \subseteq \mathbf{\Sigma}$ be subsets of $\mathbf{\Sigma}$. Then we have that $\Delta(\Sigma_1 \cap \Sigma_2) \subseteq \Delta(\Sigma_1) \cup \Delta(\Sigma_2)$ and $\Delta(\Sigma_1 \cup \Sigma_2) \subseteq \Delta(\Sigma_1) \cup \Delta(\Sigma_2)$ and $\Delta(\Sigma_1 \backslash \Sigma_2) \subseteq \Delta(\Sigma_1) \cup \Delta(\Sigma_2)$.*

Proof. Define the complement of Δ as $\overline{\Delta}$, which for any $\Sigma' \subseteq \mathbf{\Sigma}$ is given by $\overline{\Delta}(\Sigma') = \{x \in \mathbf{Var} \mid X = \{x\},\ \mathit{havoc}X(\Sigma') = \Sigma'\}$. It is thus clear that $\overline{\Delta}(\Sigma') = \mathbf{Var} \backslash \Delta(\Sigma')$. But for any $X = \{x\} \subseteq \mathbf{Var}$, we have that $\mathit{havoc}X(\Sigma_1) \cap \mathit{havoc}X(\Sigma_2) = \mathit{havoc}X(\mathit{havoc}X(\Sigma_1) \cap \mathit{havoc}X(\Sigma_2))$ by (3b) lemma 5.4.8, and hence if $\mathit{havoc}X(\Sigma_1) = \Sigma_1$ and $\mathit{havoc}X(\Sigma_2) = \Sigma_2$ then $\Sigma_1 \cap \Sigma_2 = \mathit{havoc}X(\Sigma_1 \cap \Sigma_2)$. Therefore, $x \in \overline{\Delta}(\Sigma_1)$ and $x \in \overline{\Delta}(\Sigma_2)$ implies that $x \in \overline{\Delta}(\Sigma_1 \cap \Sigma_2)$ (since $\Sigma_1 \cap \Sigma_2 = \mathit{havoc}X(\Sigma_1 \cap \Sigma_2)$), that is, $\overline{\Delta}(\Sigma_1) \cap \overline{\Delta}(\Sigma_2) \subseteq \overline{\Delta}(\Sigma_1 \cap \Sigma_2)$. Hence, $\Delta(\Sigma_1 \cap \Sigma_2) \subseteq \Delta(\Sigma_1) \cup \Delta(\Sigma_2)$.

Similarly, since $\mathit{havoc}X(\Sigma_1) \cup \mathit{havoc}X(\Sigma_2) = \mathit{havoc}X(\mathit{havoc}X(\Sigma_1) \cup \mathit{havoc}X(\Sigma_2))$ by (3a) of lemma 5.4.8 , it follows that $\overline{\Delta}(\Sigma_1) \cap \overline{\Delta}(\Sigma_2) \subseteq \overline{\Delta}(\Sigma_1 \cup \Sigma_2)$, which means that $\Delta(\Sigma_1 \cup \Sigma_2) \subseteq \Delta(\Sigma_1) \cup \Delta(\Sigma_2)$.

Finally, by (3c) of lemma 5.4.8 we know that $\mathit{havoc}X(\Sigma_1) \backslash \mathit{havoc}X(\Sigma_2) = \mathit{havoc}X(\mathit{havoc}X(\Sigma_1) \backslash \mathit{havoc}X(\Sigma_2))$. By similar argumentation as above, we observe that $x \in \overline{\Delta}(\Sigma_1) \cap \overline{\Delta}(\Sigma_2)$, implies $x \in \overline{\Delta}(\Sigma_1 \backslash \Sigma_2)$ and thus $\overline{\Delta}(\Sigma_1) \cap \overline{\Delta}(\Sigma_2) \subseteq \overline{\Delta}(\Sigma_1 \backslash \Sigma_2)$, which shows that $\Delta(\Sigma_1 \backslash \Sigma_2) \subseteq \Delta(\Sigma_1) \cup \Delta(\Sigma_2)$. □

Proposition 6.2.3. *For any PER $R \in \mathit{PER}(\mathbf{\Sigma})$, $\Delta(\mathit{dom}(R)) \subseteq \alpha_{\mathcal{L}}(R)$.*

Proof. The proof follows from Lemma 6.2.2 since the domain of a PER is the union of its equivalence classes, and the fact that by the definition of $\alpha_{\mathcal{L}}(R)$, for every equivalence class $[\sigma]_R$ of R, we have $\Delta([\sigma]_R) \subseteq \alpha_{\mathcal{L}}(R)$. □

The operation $\cup$ on the lattice $\mathcal{L}$ soundly abstracts $\sqcup$ and $\uplus$ on $\mathrm{PER}(\mathbf{\Sigma})$.

Lemma 6.2.4. *For any* $R, R' \in \mathit{PER}(\mathbf{\Sigma})$, $\alpha_{\mathcal{L}}(R \sqcup R') \subseteq \alpha_{\mathcal{L}}(R) \cup \alpha_{\mathcal{L}}(R')$ *and* $\alpha_{\mathcal{L}}(R \uplus R') \subseteq \alpha_{\mathcal{L}}(R) \cup \alpha_{\mathcal{L}}(R')$.

Proof. Let $R'' = R \sqcup R'$. By definition, for any $\sigma \in \mathit{dom}(R'')$ the equivalence class of σ in R'' is given by $[\sigma]_{R''} = [\sigma]_R \cap [\sigma]_{R'}$. Hence by lemma 6.2.2 we have that $\Delta([\sigma]_{R''}) \subseteq \Delta([\sigma]_R) \cup \Delta([\sigma]_{R'})$. Hence, $\alpha_{\mathcal{L}}(R'') \subseteq \alpha_{\mathcal{L}}(R) \cup \alpha_{\mathcal{L}}(R')$.

For the second part of the proof, now let $R'' = R \uplus R'$ and let $\Sigma = \mathit{dom}(R) \backslash \mathit{dom}(R')$ and let $\Sigma' = \mathit{dom}(R') \backslash \mathit{dom}(R)$. Then, by definition, we have that for any $\sigma \in \mathit{dom}(R'')$, $[\sigma]_{R''} = [\sigma]_R \cap [\sigma]_{R'}$ if $\sigma \in \mathit{dom}(R) \cap \mathit{dom}(R')$, and $[\sigma]_{R''} = [\sigma]_R \cap \Sigma'$ if $\sigma \in \mathit{dom}(R)$ and $\sigma \notin \mathit{dom}(R')$, and $[\sigma]_{R''} = [\sigma]_{R'} \cap \Sigma$ if $\sigma \in \mathit{dom}(R')$ and $\sigma \notin \mathit{dom}(R')$. Now take any $x \in \mathbf{Var}$ such that $X = \{x\}$ and suppose $x \notin \alpha_{\mathcal{L}}(R) \cup \alpha_{\mathcal{L}}(R')$. Then we know that for all $\sigma \in \mathit{dom}(R)$, $\mathit{havoc}X([\sigma]_R) = [\sigma]_R$ and for all $\sigma \in \mathit{dom}(R')$, $\mathit{havoc}X([\sigma]_{R'}) = [\sigma]_{R'}$. Thus, by applying (3a) of lemma 5.4.8, $\mathit{havoc}X(\mathit{dom}(R)) = \mathit{dom}(R)$ and $\mathit{havoc}X(\mathit{dom}(R')) = \mathit{dom}(R')$. Therefore, by applying (3c) of lemma 5.4.8, this means that $\Sigma = \mathit{dom}(R) \backslash \mathit{dom}(R') = \mathit{havoc}X(\Sigma)$ and similarly, $\Sigma' = \mathit{havoc}X(\Sigma')$. Thus, we have that $[\sigma]_{R'} \cap \Sigma = \mathit{havoc}X([\sigma]_{R'} \cap \Sigma)$ and $[\sigma]_R \cap \Sigma' = \mathit{havoc}X([\sigma]_R \cap \Sigma')$ and $[\sigma]_R \cap [\sigma]_{R'} = \mathit{havoc}X([\sigma]_R \cap [\sigma]_{R'})$ by applying (3b) of lemma 5.4.8, and hence that for any $\sigma \in \mathit{dom}(R'')$, $[\sigma]_{R''} = \mathit{havoc}X([\sigma]_{R''})$. Thus, $x \notin \alpha_{\mathcal{L}}(R) \cup \alpha_{\mathcal{L}}(R') \implies x \notin \alpha_{\mathcal{L}}(R \uplus R')$, which by the contrapositive means that $\alpha_{\mathcal{L}}(R \uplus R') \subseteq \alpha_{\mathcal{L}}(R) \cup \alpha_{\mathcal{L}}(R')$. □

Lemma 6.2.5. *For any expression* e *and PER* ϕ *over the values of* e, $\alpha_{\mathcal{L}}(e : \phi) \subseteq FV(e)$.

Proof. Take any variable $x \in \mathbf{Var}$ such that $x \notin FV(e)$, it is clear that the value of e at any state $\sigma \in \mathbf{\Sigma}$ is independent of the value of x in that state. That is, for any

$\sigma' \in \mathit{havoc}X(\{\sigma\}), \sigma'(e) = \sigma(e)$, where $X = \{x\}$. Since a PER is reflexive on its domain, we have that for any possible values v, v' of e such that $v \,\phi\, v'$ holds, then $v \,\phi\, v$ holds and thus if $\sigma(e) = v$, then for all $\sigma' \in \mathit{havoc}X(\{\sigma\}), \sigma'(e) = v$. Hence, for any $\sigma \in \mathit{dom}(e : \phi)$, $[\sigma]_{e:\phi} = \mathit{havoc}X([\sigma]_{e:\phi})$, which means that $x \notin \alpha_{\mathcal{L}}(e : \phi)$. Thus, $x \notin FV(e) \implies x \notin \alpha_{\mathcal{L}}(e : \phi)$. Therefore, by the contrapositive, we have $\alpha_{\mathcal{L}}(e : \phi) \subseteq FV(e)$. □

Proposition 6.2.6. *For any $Z \subseteq \mathbf{Var}$, and $x \in \mathbf{Var}$, and $R \in \mathit{PER}(\mathbf{\Sigma})$, we have that $\alpha_{\mathcal{L}}(\uparrow_Z R) \subseteq \alpha_{\mathcal{L}}(R) \backslash Z$, and $\alpha_{\mathcal{L}}(\overset{x:=e}{\rightsquigarrow} R) \subseteq \alpha_{\mathcal{L}}(R) \cup \{x\}$.*

Proof. We know from the definition that for any $\sigma \in \mathit{dom}(\uparrow_Z R)$, there exist $\Sigma \subseteq \mathit{dom}(R)$ such that $[\sigma]_{\uparrow_Z R} = \bigcup_{\sigma' \in \Sigma} \mathit{havoc}Z([\sigma']_R)$. Furthermore, we know that, by definition, for any $\sigma' \in \mathbf{\Sigma}$, $\Delta(\mathit{havoc}Z([\sigma']_R)) \subseteq \Delta([\sigma']_R) \backslash Z$ and hence by applying lemma 6.2.2, $\Delta([\sigma]_{\uparrow_Z R}) \subseteq \bigcup_{\sigma' \in \Sigma} \Delta([\sigma']_R) \backslash Z$ and, therefore, $\alpha_{\mathcal{L}}(\uparrow_Z R) \subseteq \alpha_{\mathcal{L}}(R) \backslash Z$.

Now let $X = \{x\}$ and let $R' = \overset{x:=e}{\rightsquigarrow} R$. By definition, for any $\sigma \in \mathit{dom}(R')$, there exist $\Sigma \subseteq \mathit{dom}(R)$ such that $[\sigma]_{R'} = \bigcup_{\sigma' \in \Sigma} \{\sigma''[x \mapsto \sigma''(e)] \mid \sigma'' \in [\sigma']_R\}$. Since $\{\sigma''[x \mapsto \sigma''(e)] \mid \sigma'' \in [\sigma']_R\}$ is obtained from $[\sigma']_R$ by modifying x alone, we have that $\Delta(\{\sigma''[x \mapsto \sigma''(e)] \mid \sigma'' \in [\sigma']_R\}) \subseteq \Delta([\sigma']_R) \cup \{x\}$. Hence, by applying lemma 6.2.2, $\Delta([\sigma]_{R'}) \subseteq \bigcup_{\sigma' \in \Sigma} \Delta([\sigma']_R) \cup \{x\}$. Therefore, $\alpha_{\mathcal{L}}(\overset{x:=e}{\rightsquigarrow} R) \subseteq \alpha_{\mathcal{L}}(R) \cup \{x\}$. □

Lemma 6.2.7. *For any $(E, I, O) \in \mathbf{\Phi}$, we have that $\alpha_{\mathcal{L}}(\mathit{flow}(e : \phi, (E, I, O))) \subseteq (\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I)) \backslash \mathbf{TVar}$.*

Proof. The definition of $\mathit{flow}(e : \phi, (E, I, O))$ is the PER $\uparrow_{\mathbf{TVar}} R$, where $R = e : \phi \sqcup I \sqcup R_E$ and $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma R_E \sigma' \iff \sigma, \sigma' \in \mathit{dom}(\bigsqcup_{x \in FV(e)} E(x))$. By applying proposition 6.2.3 and lemma 6.2.4, we know that $\alpha_{\mathcal{L}}(R_E) = \Delta(\mathit{dom}(R_E)) \subseteq$

$\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x))$. Furthermore, by lemma 6.2.5, $\alpha_{\mathcal{L}}(e : \phi) \subseteq FV(e)$. Hence, by lemma 6.2.4 and proposition 6.2.6, we have $\alpha_{\mathcal{L}}(\uparrow_{\mathbf{TVar}} R) \subseteq \alpha_{\mathcal{L}}(R) \backslash \mathbf{TVar} \subseteq (\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I)) \backslash \mathbf{TVar}$. □

Lemma 6.2.8. *Let* $z \in \mathbf{TVar}$ *and let* $X = \mathbf{TVar} \backslash \{z\}$. *For any* $(E, I, O) \in \mathbf{\Phi}$, $\alpha_{\mathcal{L}}(\mathit{aflow}(z := e, (E, I, O))) \subseteq \left(\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I) \cup \{z\}\right) \backslash X$.

Proof. By definition $\mathit{aflow}(z := e, (E, I, O)) = \uparrow_X \overset{z:=e}{\rightsquigarrow} R$, where $R = e : \phi \sqcup I \sqcup R_E$ and $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma \; R_E \; \sigma' \iff \sigma, \sigma' \in \mathit{dom}(\bigsqcup_{x \in FV(e)} E(x))$. By applying proposition 6.2.3 and lemma 6.2.2, we know that $\alpha_{\mathcal{L}}(R_E) = \Delta(\mathit{dom}(R_E)) \subseteq \bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x))$. Furthermore, by lemma 6.2.5, $\alpha_{\mathcal{L}}(e : \phi) \subseteq FV(e)$. Hence, by lemma 6.2.4 we have $\alpha_{\mathcal{L}}(R) \subseteq \bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I)$. By proposition 6.2.6 we have $\alpha_{\mathcal{L}}(\uparrow_X \overset{z:=e}{\rightsquigarrow} R) \subseteq \alpha_{\mathcal{L}}(\overset{z:=e}{\rightsquigarrow} R) \backslash X \subseteq \alpha_{\mathcal{L}}(R) \backslash X \cup \{z\}$. However, since $z \notin X$, then we have that $\alpha_{\mathcal{L}}(\uparrow_X \overset{z:=e}{\rightsquigarrow} R) \subseteq (\alpha_{\mathcal{L}}(R) \cup \{z\}) \backslash X$. Hence, $\alpha_{\mathcal{L}}(\mathit{aflow}(z := e, (E, I, O))) \subseteq \left(\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I) \cup \{z\}\right) \backslash X$.

□

Corollary 6.2.9. *Let* $(E, I, O) \in \mathbf{\Phi}$ *and* $\langle E', I', O' \rangle \in [\mathbf{Var} \to \mathcal{L}_2] \times \mathcal{L}_2 \times \mathcal{L}_2$ *such that for all* $x \in \mathbf{Var}$, $\alpha_{\mathcal{L}}(E(x)) \subseteq \alpha_{\cup}(E'(x))$ *and* $\alpha_{\mathcal{L}}(I) \subseteq \alpha_{\cup}(I')$, *then for any expression over variables in* $\mathbf{Var}$ *we have that* $\alpha_{\mathcal{L}}(\mathit{aflow}(z := e, (E, I, O))) \subseteq \bigcup_{x \in FV(e)} \alpha_{\cup}(E'(x)) \cup \alpha_{\cup}(I') \cup FV(e) \cup \{z\}$.

Proof. The proof follows easily from lemma 6.2.8 since for all $x \in \mathbf{Var}$, $\alpha_{\mathcal{L}}(E(x)) \subseteq \alpha_{\cup}(E'(x))$ and $\alpha_{\mathcal{L}}(I) \subseteq \alpha_{\cup}(I')$. □

6.2.2 Semantics-Based Dependency Analysis

Let us now demonstrate how PERs in the static analysis of Chapter 5 semantically encode variable dependencies. Suppose $h_1, h_2 \in \{0,1\}$ are input variables to the program `write` e, where $e \triangleq h_1$ XOR h_2 is the *exclusive OR* of h_1 and h_2. The PER $e : id \equiv \{\{(0,0),(1,1)\},\{(0,1),(1,0)\}\}$ represents[1] the information released by `write`e, whose analysis is given by $(E_\perp, I_\perp, O_\perp)$`write`$e(E_\perp, I_\perp, e : id)$ (recall from Chapter 5 that for all $x \in \mathbf{Var}, E_\perp(x) = I_\perp = O_\perp = all$). It is thus clear that the output of this program depends on both h_1 and h_2. This is shown by the fact that $\alpha_{\mathcal{L}}(e : id) = \{h_1, h_2\}$, which is the dependency information released to the output as encoded by the O-component of post-configuration of the static analysis.

Now consider another expression $e \triangleq h_1 + h_2 - h_2$, where h_1 and h_2 are natural numbers, where state is a pair of the form (h_1, h_2). Then, we obtain $e : id \equiv \{\{(n,m) \mid m \in \mathbb{N}\} \mid n \in \mathbb{N}\}$, and thus $\alpha_{\mathcal{L}}(e : id) = \{h_1\}$, which shows that e's value is dependent on h_1 but independent of h_2. Thus, by using the dependency abstraction $\alpha_{\mathcal{L}}$ on the PER $e : id$ induced by the evaluation of an expression e, we can obtain a more precise (semantics-based) dependency abstraction of e on its free variable, namely $\alpha_{\mathcal{L}}(e : id) \subseteq FV(e)$. This is as opposed to the usual static interpretation $FV(e)$, which approximates the dependency of e as the set of free variables of e.

Similarly to the treatment of outputs as demonstrated by the examples above, we can derive a more precise dependency that is propagated during assignment to the assigned variable from the PERs in the E-components of information config-

[1] This notation for PER representation, using the set of equivalence classes was introduced in Chapter 5.

urations. Consider the expression $e \triangleq h_1 + h_2 - h_2$ again, and the assignment $l := e$, under the pre-configuration $(E_\perp, I_\perp, O_\perp)$, then we have the post-configuration $(E, I_\perp, O_\perp)$, where $E = E_\perp[l \mapsto R]$ and (now representing state as triples of the form (h_1, h_2, l)) $R \equiv \{\{(n, m, n) \mid m \in \mathbb{N}\} \mid n \in \mathbb{N}\}$. Hence $\alpha_{\mathcal{L}}(E(l)) = \{l, h_1\}$, which means again that l depends on h_1 but not on h_2. The meaning of l in $\alpha_{\mathcal{L}}(E(l))$ can be explained by the fact that R encodes the equality of h_1 and l and since R contains information about h_1 it therefore also contains information about l.

The abstract analysis of implicit information flow due to *control dependency* is the same by considering the abstraction of the I-component of information configurations. Consider the programs of Figure 6.1. Again in this example state is represented by triples of the form (h_1, h_2, l) and the variables are assumed to be natural numbers. First, consider the left-hand-side program. Starting the analysis at the pre-configuration $(E_\perp, I_\perp, O_\perp)$, we have the implicit context in the *then* branch as the PER $I_1 = ((h_1 = h_2) : \mathbf{T}) \equiv \{\{(n, n, m) \mid n, m \in \mathbb{N}\}\}$ and hence $\alpha_{\mathcal{L}}(I_1) = \{h_1, h_2\}$ showing the dependency on h_1 and h_2. Similarly, for the *else* branch, the implicit context is $I_2 = ((h_1 = h_2) : \mathbf{F}) \equiv \{\{(n, n', m) \mid n, n', m \in \mathbb{N}, n' \neq n\}\}$ and hence $\alpha_{\mathcal{L}}(I_2) = \{h_1, h_2\}$. This implicit dependency is propagated to l due to the assignments in the conditional branch. This is reflected by the fact that the post-configuration of the *if* statement is $(E, I_\perp, O_\perp)$, where $E = E_\perp[l \mapsto R]$ and $R \equiv \{\{(n, n, 1) \mid n \in \mathbb{N}\}, \{(n, n', 2) \mid n, n' \in \mathbb{N}, n' \neq n\}\}$. Thus, $\alpha_{\mathcal{L}}(E(l)) = \{h_1, h_2, l\}$. It is also clear that the output depends on h_1 and h_2 on line 5 since $(E, I_\perp, O_\perp)\,\texttt{write}\,l\,(E, I_\perp, O)$, where $O \equiv \{\{(n, n, m) \mid n, m \in \mathbb{N}\}, \{(n, n', m) \mid n, n', m \in \mathbb{N}, n' \neq n\}\}$ and hence $\alpha_{\mathcal{L}}(O) = \{h_1, h_2\}$.

The right-hand-side program of Figure 6.1 demonstrates flow sensitivity. In this program, on line 5, the post-configuration $(E, I_\perp, O_\perp)$ of the conditional state-

ment in the previous example now serves as the pre-configuration of the assignment as $(E, I_\perp, O_\perp)l := 3(E', I_\perp, O_\perp)$, where $E' = E[l \mapsto R']$ and $R' \equiv \{\{(n, n, 3) | n \in \mathbb{N}\}, \{(n, n', 3) \,|\, n, n' \in \mathbb{N}, n \neq n'\}\}$. Hence, the analysis of the following statement is $(E', I_\perp, O_\perp)$ `write` l $(E', I_\perp, O')$, where $O' \equiv \{\{(n, m, m') \mid n, m, m' \in \mathbb{N}\}\}$, which shows that no information is released since $\alpha_{\mathcal{L}}(O') = \varnothing$.

```
if ( h1 = h2) then          1 if ( h1 = h2) then
   l := 1;                  2    l := 1;
else                        3 else
   l := 2;                  4    l := 2;
write  l;                   5 l := 3;
                            6 write  l;
```

Figure 6.1: *Dependency Analysis and Flow Sensitivity*

6.2.3 Disjunctive Dependency, Nontermination, Dead Code

It was shown in Definition 3.5.9 how a PER over program states may describe disjunctive information. A restatement of this definition in terms of variable dependency is captured by the abstraction $\alpha_{\mathcal{L}_2}$, which extracts the corresponding disjunctive dependency. We say that a PER R contains disjunctive dependency about variables $x, y \in \mathbf{Var}$ if $\alpha_{\mathcal{L}_2}(R)$ represents a disjunctive dependency (see Definition 6.2.1) about these variables. The intended meaning is that a PER R, which contains disjunctive dependency about x and y does not reveal information about x and y simultaneously. Consider the program listing in Figure 6.2, and let us assume that $l, h_1, h_2 \in \mathbf{IVar}$, and that l has a boolean data type, whereas h_1 and h_2 are integers. The program either reveals the value of secret h_1 or the parity of secret h_2 but never both at the same time - even if the attacker has con-

trol over the choice of l. Thus, information is released about at most one of h_1 and h_2 during any given run. The analysis shows this, and consequently the induced PER demonstrates the disjunctive dependency of the observed output on h_1 or h_2. Let the program of Figure 6.2 be P, then its analysis is given by $(E_\perp, I_\perp, O_\perp)\ P\ (E_\perp, I_\perp, O)$, where $O \equiv \{\{(n, m, \mathbf{tt}) \mid m \in \mathbb{Z}\}, \{(m, n, \mathbf{ff}) \mid m \in \mathbb{Z}, n \ \mathit{mod}\ 2 = 0\}, \{(m, n, \mathbf{ff}) \mid m \in \mathbb{Z}, n \ \mathit{mod}\ 2 = 1\} \mid n \in \mathbb{Z}\}$. Hence, $\alpha_{\mathcal{L}_2}(O) = \{\{h_1, l\}, \{h_2, l\}\}$ showing that O contains disjunctive dependency about h_1 and h_2, that is, the output value of the program does not at any time depend on both h_1 and h_2.

```
if (l) then
  write h1;
else
  write h2 mod 2;
```

Figure 6.2: *Disjunctive Dependency*

The semantic nature of the information flow analysis means that the PER abstractions also capture when the value of a secret input affects termination behaviour. Furthermore, non-termination of a subprogram prevents further information from being released in the trailing subprogram, because the program that trails the diverging subprogram cannot be executed. These properties are illustrated in the dependency abstraction of the analysis of the program listing of Figure 6.3 (h_1 is boolean and h_2 is integer), where whenever $h_1 = \mathbf{tt}$ the program diverges, but the program terminates otherwise (that is, whenever $h_1 = \mathbf{ff}$). Applying the flow rules and starting the program analysis with the pre-configuration $(E_\perp, I_\perp, O_\perp)$, the post-configuration of the *while* statement is $(E, \varnothing, O)$, where according to the *while* rule the resulting implicit context is the empty PER, and also

$E(h_1) = E(h_2) = \varnothing$ and $O \equiv \{\{(\mathbf{tt}, n) \mid n \in \mathbb{Z}\}, \{(\mathbf{ff}, n) \mid n \in \mathbb{Z}\}\}$. Hence, we have $(E, \varnothing, O)$ `write` h_2 $(E, \varnothing, O)$ since $O \uplus \mathit{flow}(h_2 : \mathit{id}, (E, \varnothing, O)) = O \uplus \varnothing = O$. Finally, applying the *if* rule, we obtain the post-configuration of the *if* statement is $(E', h_1 : \mathbf{F}, O)$, where $E'(h_1) = E'(h_2) = h_1 : \mathbf{F}$. Now, the PER O shows that the attacker only gains information about h_1, but not h_2, since $\alpha_{\mathcal{L}_2}(O) = \{\{h_1\}\}$. This is because, semantically, the *write* statement is dead code.

```
if (h1) then
   while (tt) do
        skip;
   write h2;
else
   skip;
```

Figure 6.3: *Nontermination and Dependency*

Another dead code example is `if(`$h_1 \neq h_1$`)then write` h_2 `else skip`, where the dependency analysis shows that the attacker gains no information about h_1 or h_2. This is because $(h_1 \neq h_1) : \mathbf{T} = \varnothing$, which means the branch is never executed. In fact, suppose this program is P, then we have that $(E_\perp, I_\perp, O_\perp) P (E_\perp, I_\perp, O_\perp)$ showing that the attacker gains nothing since $\alpha_{\mathcal{L}}(O_\perp) = \varnothing$.

6.2.4 A Dependency Type System

We now present a type system that computes program dependency, and which improves on existing flow-sensitive dependency type systems such as [AB04, HS06]. The improvements are in the identification of some disjunctive dependencies, termination sensitivity, and interactive outputs. This type system is shown to be a sound abstraction of the information flow analysis of Chapter 5. Later on, in

section 6.4, we shall look at a technique that takes advantage of the dependency abstraction of PERs to improve the precision of the dependency analysis.

For the analysis we shall use *dependency configurations*, as we did with information configurations, to track dependencies. The analysis is performed over the lattice $\mathcal{L}_2$ to identify disjunctive dependencies. Thus, a dependency configuration is a typing environment which assigns dependencies to variables, the "program counter", and outputs; and is written in the form $\langle E, I, O\rangle$, where $E : \mathbf{Var} \to \mathcal{L}_2$, and $I, O \in \mathcal{L}_2$. The interpretation of the dependency of a variable x on the initial values of program input under the configuration $\langle E, I, O\rangle$ is given by $\alpha_\cup(E(x))$. Similarly, the interpretations of the dependency of the implicit context and the output under this configuration are $\alpha_\cup(I)$ and $\alpha_\cup(O)$ respectively.

Typing judgement are of the form $\langle E, I, O\rangle\ c\ \langle E', I', O'\rangle$, which represents how the *While* command c transforms dependencies. Under a dependency configuration $\langle E, I, O\rangle$, the dependency typing judgement for an expression e is given by

$$E \vdash e : t \iff t = \bigsqcup_{x \in FV(e)} E(x). \tag{6.3}$$

Definition (6.3) is fairly standard in dependency analyses, but more precise analyses can be performed as demonstrated later by the semantic typing judgement of (6.6), which uses PERs to compute expression types.

The full algorithmic dependency type system, which computes input dependencies is shown in Figure 6.4. In the typing rules, the operation $\boxplus$ on dependency configurations is defined for any pair of dependency configurations $\langle E_1, I_1, O_1\rangle$ and $\langle E_2, I_2, O_2\rangle$ as $\langle E_1, I_1, O_1\rangle \boxplus \langle E_2, I_2, O_2\rangle \triangleq \langle E', I_1 \cup I_2, O_1 \cup O_2\rangle$, where for all $x \in \mathbf{Var}, E'(x) = E_1(x) \cup E_2(x)$. The dependency configurations are partially or-

dered by $\sqsubseteq$, such that $\langle E_1, I_1, O_1\rangle \sqsubseteq \langle E_2, I_2, O_2\rangle$ iff for all $x \in \mathbf{Var}, E_1(x) \subseteq E_2(x)$ and $I_1 \subseteq I_2$ and $O_1 \subseteq O_2$. Furthermore, the predicate $W(c)$ on a command c holds if c contains a conditional *while* statement. Similarly to the concrete analysis presented in Chapter 5, the implicit context can encode information about branching and termination. In particular, after a *while* statement, the execution of subsequent programs is dependent on the termination or not of the preceding *while* statement. This information about termination dependency on program variables is encoded in the I-component of the dependency configuration. The termination dependency is calculated in the post-condition of the *while* statement by taking a join with the dependency of the boolean guard at the fixpoint. The possibility of information flow due to a non-terminating branch is also derived in the analysis of *if* statements by checking for the presence of *while* statements in the branches and retaining the dependency of the implicit context as necessary. Let us illustrate the dependency type system with some examples.

6.2.5 Sample Analyses

We define a starting configuration $\langle E^\alpha_\perp, I^\alpha_\perp, O^\alpha_\perp\rangle$ for dependency analyses, such that for all $x \in \mathbf{Var}$, $E^\alpha_\perp(x) = \{\{x\}\}$ and $I^\alpha_\perp = O^\alpha_\perp = \{\varnothing\}$. The interpretation of $\langle E^\alpha_\perp, I^\alpha_\perp, O^\alpha_\perp\rangle$ is that the execution of the program starts at a dependency configuration where each variable depends only on its own initial value, but the implicit context and output have no initial dependency.

Consider the program listing of Figure 6.5. By applying the *if* rule at the starting configuration $\langle E^\alpha_\perp, I^\alpha_\perp, O^\alpha_\perp\rangle$ we obtain the implicit dependency $I_1 = \{\{l\}\}$, since $E^\alpha_\perp \vdash l : \{\{l\}\}$. Thus, the *then* branch analysis is $\langle E^\alpha_\perp, \{\{l\}\}, O^\alpha_\perp\rangle z := h_1 \langle E_1, \{\{l\}\}, O^\alpha_\perp\rangle$,

$$\frac{}{\langle E,I,O\rangle\ \texttt{skip}\ \langle E,I,O\rangle} \qquad \frac{E \vdash e : t}{\langle E,I,O\rangle\ x := e\ \langle E[x \mapsto I \sqcup t], I, O\rangle}$$

$$\frac{E \vdash e : t}{\langle E,I,O\rangle\ \texttt{write}\ e\ \langle E,I,O \sqcup I \sqcup t\rangle}$$

$$\frac{\langle E,I,O\rangle\ c_1\ \langle E',I',O'\rangle \quad \langle E',I',O'\rangle\ c_2\ \langle E'',I'',O''\rangle}{\langle E,I,O\rangle\ c_1; c_2\ \langle E'',I'',O''\rangle}$$

$$\frac{E \vdash b : t \quad \langle E, I \sqcup t, O\rangle\ c_i\ \langle E_i, I_i, O_i\rangle \quad i = 1,2}{\langle E,I,O\rangle\ \texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \langle E_1, I_1', O_1\rangle \uplus \langle E_2, I_2', O_2\rangle}\ I_i' = \begin{cases} I_i & \textit{if} \quad W(c_i) \\ I & \textit{otherwise} \end{cases}$$

$$\frac{\begin{array}{c}\langle E_i, I_i, O_i\rangle\ \texttt{if(}b\texttt{)then}\ c\ \texttt{else skip}\ \langle E_{i+1}', I_{i+1}', O_{i+1}'\rangle \\ \langle E_{i+1}, I_{i+1}, O_{i+1}\rangle = \langle E_{i+1}', I_{i+1}', O_{i+1}'\rangle \uplus \langle E_i, I_i, O_i\rangle \\ \forall x \in \mathbf{Var}, E'(x) = E''(x) \sqcup I' \quad O' = O'' \sqcup I'\end{array}}{\langle E,I,O\rangle\ \texttt{while}\ (b)\ \texttt{do}\ c\ \langle E', I', O'\rangle}\ \begin{array}{c}\langle E_0, I_0, O_0\rangle = \langle E, I, O\rangle \\ \langle E'', I'', O''\rangle = \biguplus_{i\geq 0} \langle E_i, I_i, O_i\rangle \\ E'' \vdash b : t, I' = I'' \sqcup t\end{array}$$

Figure 6.4: *An Algorithmic Dependency Type System*

where $E_1 = E_\perp^\alpha[z \mapsto \{\{h_1, l\}\}]$. Similarly, for the *else* branch we have the analysis $\langle E_\perp^\alpha, \{\{l\}\}, O_\perp^\alpha\rangle z := h_2 \langle E_2, \{\{l\}\}, O_\perp^\alpha\rangle$, where $E_2 = E_\perp^\alpha[z \mapsto \{\{h_2, l\}\}]$. The post-condition of the *if* statement is thus $\langle E_1, I_\perp^\alpha, O_\perp^\alpha\rangle \uplus \langle E_2, I_\perp^\alpha, O_\perp^\alpha\rangle = \langle E_3, I_\perp^\alpha, O_\perp^\alpha\rangle$, where we have $E_3(z) = \{\{h_1, l\}, \{h_2, l\}\}$. The meaning of $E_3(z)$ is that after the *if* statement, z either depends on l and h_1 or it depends on l and h_2. However, z is disjunctively dependent on h_1 and h_2 since it is never at any one time dependent on both h_1 and h_2. By applying the *write* rule, for the next statement we now obtain $\langle E_3, I_\perp^\alpha, O_\perp^\alpha\rangle\ \texttt{write}\ z\ \langle E_3, I_\perp^\alpha, O_3\rangle$, where $O_3 = \{\{h_1, l\}, \{h_2, l\}\}$ - which means that the attacker gains information at most about l and h_1 or l and h_2 at any one time.

Now consider the program (P) shown in listing of Figure 6.6, which is similar

```
if (l) then
   z := h1;
else
   z := h2;
write z;
```

Figure 6.5: *Assignments and Disjunctive Dependency*

to the program of Figure 6.5 by replacing the assignments with *write* statements. We now obtain $\langle E, I, O\rangle\, P\, \langle E, I, O_3\rangle$. Thus, the program releases the same information to the attacker as in the previous example, where the attacker (O_3) either gains information about h_1 and l or h_2 and l but the attacker cannot learn about h_1 and h_2 in the same run of the program.

```
if (l) then
   write h1;
else
   write h2;
```

Figure 6.6: *Outputs and Disjunctive Dependency*

The static analysis is termination-sensitive. To demonstrate this, consider the program of Figure 6.7, where a choice of h_1 may lead to program divergence, revealing information about h_1 and also about l because nontermination reveals which branch of the conditional *if* statement has been executed.

```
if (l) then
       while (h1) do
           skip;
else
   write h2;
```

Figure 6.7: *Nontermination and Dependency*

Again, starting with the dependency configuration $\langle E^\alpha_\perp, I^\alpha_\perp, O^\alpha_\perp\rangle$, the analysis of the *while* statement is $\langle E^\alpha_\perp, I_1, O^\alpha_\perp\rangle$ `while` (h_1) `do skip` $\langle E_1, I'_1, O_1\rangle$, where $I_1 = \{\{l\}\}$ and $I'_1 = \{\{l, h_1\}\}$, and for all $x \in \mathbf{Var}$, $E_1(x) = E^\alpha_\perp(x) \sqcup \{\{l, h_1\}\}$ and $O_1 = \{\{l, h_1\}\}$. The interpretation of O_1 is that the attacker now gains information about h_1 (due to the possibility of nontermination) and l (due to the knowledge of the path taken when the program diverges). The possibility of nontermination along a path containing a *while* statement means that the value that a variable takes after the *while* statement now depends on whether the *while* terminates or not, which in turn depends on how the boolean guard of the *while* evaluates. Hence, for any $x \in \mathbf{Var}$, $E_1(x)$ reflects the possible dependency of the value of x on the *while* guard, that is h_1, and on l because of the fact that the execution of the *then* branch is dependent on l.

For the *else* branch we have the analysis $\langle E^\alpha_\perp, I_1, O^\alpha_\perp\rangle$ `write` h_2 $\langle E^\alpha_\perp, I_1, O_2\rangle$, where $O_2 = \{\{l, h_2\}\}$. The meaning of the dependency of O_2 is clear since an output from this program reveals the value of h_2 and also reveals how l evaluates. Hence, the post-condition of the *if* statement is $\langle E', I'_1, O'\rangle = \langle E_1, I'_1, O_1\rangle \uplus \langle E^\alpha_\perp, I^\alpha_\perp, O_2\rangle$. Since $O' = \{\{l, h_1\}, \{l, h_2\}\}$, the attacker may gain information about l and h_1 or about l and h_2. The implicit context I'_1 of the *if* post-condition also shows that the execution of commands after the *if* statement may depend on the termination of the *then* branch and hence on l and h_1.

The next example demonstrates the flow-sensitivity of the type system. Consider the program listing of Figure 6.8, which does not reveal the secret h. Starting at the configuration $\langle E^\alpha_\perp, I^\alpha_\perp, O^\alpha_\perp\rangle$, the dependency of z after the first assignment is $\{\{h\}\}$ (which means that z depends on h at that point) and after the second assignment the dependency of z is $\varnothing$ which means that z's value is independent

of any input value. Finally, we have $\langle E^{\alpha}_{\perp}[z \mapsto \varnothing], I^{\alpha}_{\perp}, O^{\alpha}_{\perp}\rangle$ `write` z $\langle E'', I^{\alpha}_{\perp}, O^{\alpha}_{\perp}\rangle$ showing that the attacker gains no information about the secret input h.

$z := h;$
$z := 0;$
write $z;$

Figure 6.8: *Flow-Sensitivity of Dependency Analysis*

6.2.6 Correctness of Dependency Analysis

We now show the correctness of the dependency analysis by proving that it is a sound abstraction of the semantics-based information flow analysis of Chapter 5, whose correctness has been shown. This is a standard technique. Firstly, we define an abstraction function from information configurations ($\mathbf{\Phi}$) introduced in Chapter 5 to the dependency configurations $\mathbf{\Phi}^{\mathcal{L}} \triangleq [\mathbf{Var} \to \mathcal{L}] \times \mathcal{L} \times \mathcal{L}$ over $\mathcal{L}$, and show that the dependency computation over $\mathbf{\Phi}^{\mathcal{L}}$ is a sound approximation of the semantic analysis over $\mathbf{\Phi}$. The dependency configurations in $\mathbf{\Phi}^{\mathcal{L}}$ are ordered by $\sqsubseteq$, which is the subset inclusion order applied in the usual way, such that for any $\langle E, I, O\rangle, \langle E', I', O'\rangle \in \mathbf{\Phi}^{\mathcal{L}}$, $\langle E, I, O\rangle \sqsubseteq \langle E', I', O'\rangle$ iff $\forall x \in \mathbf{Var}$, $E(x) \subseteq E'(x)$ and $I \subseteq I'$ and $O \subseteq O'$.

The dependency analysis of Figure 6.4 is carried out over the dependency configurations $\mathbf{\Phi}^{\mathcal{L}_2} \triangleq [\mathbf{Var} \to \mathcal{L}_2] \times \mathcal{L}_2 \times \mathcal{L}_2$, which is ordered by $\sqsubseteq\!\!\!\sqsubseteq$ such that for any $\langle E, I, O\rangle, \langle E', I', O'\rangle \in \mathbf{\Phi}^{\mathcal{L}_2}$, $\langle E, I, O\rangle \sqsubseteq\!\!\!\sqsubseteq \langle E', I', O'\rangle$ iff $\forall x \in \mathbf{Var}$, $E(x) \subseteq E'(x)$ and $I \subseteq I'$ and $O \subseteq O'$. However, we only want to show that the dependency computation is a correct abstraction of the concrete analysis of Chapter 5, but we do not want to model the disjunctive aspect. It appears to be the case that the

dependency type system also correctly abstracts the disjunctive information flow model of Chapter 5, but we have not proved this. For the dependency analysis correctness, we will extend $\alpha_{\mathcal{L}_2}$ and $\alpha_\cup$ respectively to information configurations and dependency configurations. This is defined for any $(E, I, O) \in \mathbf{\Phi}$ as $\alpha_{\mathcal{L}_2}((E, I, O)) = \langle E', I', O' \rangle$, where for all $x \in \mathbf{Var}$, $E'(x) = \alpha_{\mathcal{L}_2}(E(x)) \sqcup \{\{x\}\}$ and $I' = \alpha_{\mathcal{L}_2}(I)$ and $O' = \alpha_{\mathcal{L}_2}(O)$. The interpretation of the extension of $\alpha_{\mathcal{L}_2}$ to information configuration means that every variable x depends also on its own value. Similarly, the extension of $\alpha_\cup$ to $\mathbf{\Phi}^{\mathcal{L}_2}$ is defined such that for any $\langle E, I, O \rangle \in \mathbf{\Phi}^{\mathcal{L}_2}$, $\alpha_\cup(\langle E, I, O \rangle) = \langle E', I', O' \rangle$, where for all $x \in \mathbf{Var}$, $E'(x) = \alpha_\cup(E(x)) \cup \{x\}$ and $I' = \alpha_\cup(I)$ and $O' = \alpha_\cup(O)$. The extension of the abstraction function $\alpha_{\mathcal{L}}$ to information configurations is now given by the composition of the extended functions: $\alpha_{\mathcal{L}} = \alpha_\cup \circ \alpha_{\mathcal{L}_2}$. Hopefully, it will be clear from the context when we are referring to the abstraction functions in Definition 6.2.1 or their extensions to information or dependency configurations.

The statement of correctness is familiar from abstract interpretation.

Theorem 6.2.10. *Let P be a* While *program, which does not modify its* $\mathbf{IVar}$ *projection of states and which properly-initialises all its* $\mathbf{TVar}$ *variables before use as required by the concrete analysis of information flow $(E_\perp, I_\perp, O_\perp)\, P\, (E, I, O)$. Let $\langle E_0, I_0, O_0 \rangle \in \mathbf{\Phi}^{\mathcal{L}_2}$, such that $\alpha_{\mathcal{L}_2}((E_\perp, I_\perp, O_\perp)) \Subset \langle E_0, I_0, O_0 \rangle$, then the abstract dependency analysis $\langle E_0, I_0, O_0 \rangle\, P\, \langle E', I', O' \rangle$ of P satisfies the property $\alpha_{\mathcal{L}}((E, I, O)) \sqsubseteq \alpha_\cup(\langle E', I', O' \rangle)$.*

Proof. The proof proceeds by structural induction on the derivation trees. The inductive step of the proof is that if $P = P_0; \ldots; P_m;$ such that for any $n \leq m$, P_n is either a *skip* statement, or an *assignment*, or a *write* statement, or a conditional *if* or

while statement, and such that the inductive property holds for $P_0; \ldots; P_{n-1}$, then it holds also for $P_0; \ldots; P_n$. Now suppose $(E_\perp, I_\perp, O_\perp)\, P_0; \ldots; P_{n-1}\, (E_1, I_1, O_1)$ and $(E_1, I_1, O_1) P_n (E_2, I_2, O_2)$ hold as the concrete analyses of these programs and let $\alpha_{\mathcal{L}_2}((E_\perp, I_\perp, O_\perp)) P_0; \ldots; P_{n-1} \langle E_1^{\mathcal{L}_2}, I_1^{\mathcal{L}_2}, O_1^{\mathcal{L}_2}\rangle$ and $\langle E_1^{\mathcal{L}_2}, I_1^{\mathcal{L}_2}, O_1^{\mathcal{L}_2}\rangle\ P_n\ \langle E_2^{\mathcal{L}_2}, I_2^{\mathcal{L}_2}, O_2^{\mathcal{L}_2}\rangle$ be their respective dependency analyses. We know by the induction hypothesis that $\alpha_{\mathcal{L}}((E_1, I_1, O_1)) \sqsubseteq \alpha_\cup(\langle E_1^{\mathcal{L}_2}, I_1^{\mathcal{L}_2}, O_1^{\mathcal{L}_2}\rangle) = \langle E_1^{\mathcal{L}}, I_1^{\mathcal{L}}, O_1^{\mathcal{L}}\rangle$, but we need to show that $\alpha_{\mathcal{L}}((E_2, I_2, O_2)) \sqsubseteq \langle E_2^{\mathcal{L}}, I_2^{\mathcal{L}}, O_2^{\mathcal{L}}\rangle = \alpha_\cup(\langle E_2^{\mathcal{L}_2}, I_2^{\mathcal{L}_2}, O_2^{\mathcal{L}_2}\rangle)$.

- The proof when P_n is the `skip` statement is clear.
- Let P_n be the assignment statement $z := e$, where $z \in \mathbf{TVar}$, then we have the concrete analysis of P_n as $(E_1, I_1, O_1)\ z := e\ (E_2, I_1, O_1)$, where $E_2 = E_1[z \mapsto \mathit{aflow}(z := e, (E_1, I_1, O_1))]$. Hence, it remains to show that $\alpha_{\mathcal{L}}(E_2(z)) \cup \{z\} \subseteq E_2^{\mathcal{L}}(z) = \alpha_\cup(E_2^{\mathcal{L}_2}(z)) \cup \{z\}$. Let $X = \mathbf{TVar} \backslash \{z\}$. We know that $\alpha_{\mathcal{L}}(E_2(z)) \subseteq \left(\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E_1(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I_1) \cup \{z\}\right) \backslash X$ from lemma 6.2.8. Since P does not assign to $\mathbf{IVar}$ variables, we know that for all $x \in \mathbf{IVar}, \{\{x\}\} \subseteq E_1^{\mathcal{L}_2}(x)$ since the starting dependency configuration of P has the property that $\langle E_\perp^\alpha, I_\perp^\alpha, O_\perp^\alpha\rangle = \alpha_{\mathcal{L}_2}((E_\perp, I_\perp, O_\perp)) \sqsubseteq \langle E_0, I_0, O_0\rangle$. Hence, if $E_1^{\mathcal{L}_2} \vdash e : t$ and $Y = \mathbf{IVar} \cap FV(e)$, then we know that $\bigcup_{x \in FV(e)} \alpha_\cup(E_1^{\mathcal{L}_2}) \cup Y \subseteq \alpha_\cup(t)$. By the induction hypothesis we know that for all $x \in \mathbf{Var}, \alpha_{\mathcal{L}}(E_1(x)) \cup \{x\} \subseteq \alpha_\cup(E_1^{\mathcal{L}_2}(x)) \cup \{x\}$ and $\alpha_{\mathcal{L}}(I_1) \subseteq \alpha_\cup(I_1^{\mathcal{L}_2})$, hence since $z \notin X$, we know that, $\alpha_{\mathcal{L}}(E_2(z)) \subseteq \left(\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E_1(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I_1) \cup \{z\}\right) \backslash X \subseteq (\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E_1^{\mathcal{L}_2}(x)) \cup Y \cup \alpha_\cup(I_1^{\mathcal{L}_2}) \cup \{z\}) \backslash X \subseteq \alpha_\cup(t) \cup \alpha_\cup(I_1^{\mathcal{L}_2}) \cup \{z\} = \alpha_\cup(E_2^{\mathcal{L}_2}(z)) \cup \{z\}$. By this we obtain the required property that $\alpha_{\mathcal{L}}(E_2(z)) \cup \{z\} \subseteq E_2^{\mathcal{L}}(z) = \alpha_\cup(E_2^{\mathcal{L}_2}(z)) \cup \{z\}$.
- Let P_n be the statement `write` e, whose concrete information flow anal-

ysis is (E_1, I_1, O_1) `write` e (E_2, I_2, O_2). By lemma 6.2.7 we know that $\alpha_{\mathcal{L}}(\textit{flow}(e : \textit{id}, (E_1, I_1, O_1))) \subseteq (\bigcup_{x \in FV(e)} \alpha_{\mathcal{L}}(E_1(x)) \cup FV(e) \cup \alpha_{\mathcal{L}}(I_1)) \backslash \mathbf{TVar}$. By the induction hypothesis we have that for all $x \in \mathbf{Var}, \alpha_{\mathcal{L}}(E_1(x)) \cup \{x\} \subseteq \alpha_{\cup}(E_1^{\mathcal{L}_2}(x)) \cup \{x\}$ and $\alpha_{\mathcal{L}}(I_1) \subseteq \alpha_{\cup}(I_1^{\mathcal{L}_2})$. Furthermore, suppose $E_1^{\mathcal{L}_2} \vdash e : t$. Since P does not assign to **IVar** variables, we know that for all $x \in \mathbf{IVar}, \{\{x\}\} \subseteq E_1^{\mathcal{L}_2}(x)$, and hence $Y = \mathbf{IVar} \cap FV(e) \subseteq \bigcup_{x \in FV(e)} \alpha_{\cup}(E_1^{\mathcal{L}_2}(x)) \subseteq \alpha_{\cup}(t)$. Therefore, $\alpha_{\mathcal{L}}(\textit{flow}(e : \textit{id}, (E_1, I_1, O_1))) \subseteq (\bigcup_{x \in FV(e)} \alpha_{\cup}(E_1^{\mathcal{L}_2}(x)) \cup Y \cup \alpha_{\cup}(I_1^{\mathcal{L}_2})) \backslash \mathbf{TVar} \subseteq (\alpha_{\cup}(t) \cup \alpha_{\cup}(I_1^{\mathcal{L}_2})) \backslash \mathbf{TVar} \subseteq \alpha_{\cup}(t) \cup \alpha_{\cup}(I_1^{\mathcal{L}_2})$. Since $O_2 = O_1 \uplus \textit{flow}(e : \textit{id}, (E_1, I_1, O_1))$, and by the induction hypothesis $\alpha_{\mathcal{L}}(O_1) \subseteq \alpha_{\cup}(O_1^{\mathcal{L}_2})$, then by applying lemma 6.2.4 we know that $\alpha_{\mathcal{L}}(O_2) \subseteq \alpha_{\mathcal{L}}(O_1) \cup \alpha_{\mathcal{L}}(\textit{flow}(e : \textit{id}, (E_1, I_1, O_1))) \subseteq O_2^{\mathcal{L}} = \alpha_{\cup}(O_1^{\mathcal{L}_2}) \cup \alpha_{\cup}(t) \cup \alpha_{\cup}(I_1^{\mathcal{L}_2})$.

- Let P_n be `if` (b) `then` c_1 `else` c_2. Suppose $E_1^{\mathcal{L}_2} \vdash b : t$, we observe by induction on the preceding program that the pre-configuration $\langle E_1^{\mathcal{L}_2}, I_1^{\mathcal{L}_2} \sqcup t, O_1^{\mathcal{L}_2} \rangle$ of c_1 and c_2 have the property that $\alpha_{\mathcal{L}}((E_1, \textit{flow}(b : \mathbf{T}, (E_1, I_1, O_1)), O_1)) \sqsubseteq \alpha_{\cup}(\langle E_1^{\mathcal{L}_2}, I_1^{\mathcal{L}_2} \sqcup t, O_1^{\mathcal{L}_2} \rangle)$ and that $\alpha_{\mathcal{L}}((E_1, \textit{flow}(b : \mathbf{F}, (E_1, I_1, O_1)), O_1)) \sqsubseteq \alpha_{\cup}(\langle E_1^{\mathcal{L}_2}, I_1^{\mathcal{L}_2} \sqcup t, O_1^{\mathcal{L}_2} \rangle)$ since by applying lemma 6.2.7 we know that the property $\alpha_{\mathcal{L}}(\textit{flow}(b : \phi, (E_1, I_1, O_1))) \subseteq \alpha_{\cup}(t) \cup \alpha_{\cup}(I_1^{\mathcal{L}_2})$ holds for any PER ϕ over booleans. Hence, by applying the induction hypothesis to c_1 and c_2 we know that the post-configuration of the *if* statement has the required property, in particular, since the operation $\uplus$ over dependency configurations preserves set union on the lattice $\mathcal{L}$.

- Let P_n be `while` (b) `do` c. The proof of *while* rule is similar to the *if* rule by applying the induction hypothesis to the derivation tree of P_n. Further-

more, since by the definition of $\boxplus$ forms an increasing chain on the lattice of dependency configurations we know that the fixpoint of the *while* analysis exists and is reached in a finite number of steps because $\mathbf{\Phi}^{\mathcal{L}_2}$ is a finite, and therefore complete, lattice due to the finiteness of the set $\mathbf{Var}$.

The base case of the inductive proof, before any command is processed, holds vacuously since for any $\langle E_0, I_0, O_0\rangle$ such that $\alpha_{\mathcal{L}_2}((E_\perp, I_\perp, O_\perp)) \boxplus \langle E_0, I_0, O_0\rangle$, and we have also that $\alpha_{\mathcal{L}}((E_\perp, I_\perp, O_\perp)) \sqsubseteq \alpha_\cup(\langle E_0, I_0, O_0\rangle)$. □

6.3 Flow-Sensitive Type Systems

A flow-sensitive type system is presented in [HS06], which deems more programs secure than traditional flow-insensitive noninterference security type systems, such as [VSI96]. The family of type systems proposed in [HS06] is parametrised by an arbitrarily chosen finite flow lattice. When the flow lattice is chosen to be the powerset lattice of program variables, the type system is the De-Morgan dual of the independency type system of [AB04]. While the type system of [AB04] computes independencies between variables, the type system of [HS06] more directly computes variable dependencies under the powerset lattice of program variables.

A command typing judgement in [HS06] has the following form

$$p \vdash_{\mathcal{L}_{HS}} \Gamma \{c\} \Gamma'. \tag{6.4}$$

This describes how the command c transforms type environments (Γ to Γ') under a given context p. The inference system ($\vdash_{\mathcal{L}_{HS}}$) is parametric to a chosen finite lattice $\mathcal{L}_{HS}$, and the type environments $\Gamma, \Gamma', \ldots$ are maps from the set $\mathbf{Var}$ of

variables to the lattice $\mathcal{L}_{HS}$. The implicit context type $p \in \mathcal{L}_{HS}$ records the type of a program point, and is used to eliminate implicit flows. For an expression e over $\mathbf{Var}$, the type derivation under the environment Γ is given by:

$$\Gamma \vdash_{\mathcal{L}_{HS}} e : t \iff t = \bigsqcup_{x \in FV(e)} \Gamma(x). \tag{6.5}$$

Although we can equally choose any arbitrary finite flow lattice under our dependency approach, we shall choose the powerset lattice of $\mathbf{Var}$ to compare our type system with that of [HS06] such that $\mathcal{L}_{HS} = \mathcal{L} = \mathcal{P}(\mathbf{Var})$. The (algorithmic) type system[2] of [HS06] is presented in Figure 6.9.

$$\text{Skip}\frac{}{p \vdash \Gamma\ \{\texttt{skip}\}\ \Gamma} \qquad \text{Assign}\frac{\Gamma \vdash e : t}{p \vdash \Gamma\ \{x := e\}\ \Gamma[x \mapsto p \sqcup t]}$$

$$\text{Seq}\frac{p \vdash \Gamma\ \{c_1\}\ \Gamma' \quad p \vdash \Gamma'\ \{c_2\}\ \Gamma''}{p \vdash \Gamma\ \{c_1; c_2\}\ \Gamma''}$$

$$\text{If}\ \frac{\Gamma \vdash b : t \quad p \sqcup t \vdash \Gamma\ \{c_i\}\ \Gamma'_i \quad i = 1,2}{p \vdash \Gamma\ \{\texttt{if}\ (b)\ \texttt{then}\ c_1\ \texttt{else}\ c_2\}\ \Gamma'} \quad \Gamma' = \Gamma'_1 \sqcup \Gamma'_2$$

$$\text{While}\ \frac{\Gamma'_i \vdash b : t_i \quad p \sqcup t_i \vdash \Gamma'_i\ \{c\}\ \Gamma''_i \quad 0 \le i \le n}{p \vdash \Gamma\ \{\texttt{while}\ (b)\ \texttt{do}\ c\}\ \Gamma'_n} \quad \begin{array}{c}\Gamma'_0 = \Gamma,\ \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma,\\ \Gamma'_{n+1} = \Gamma'_n\end{array}$$

Figure 6.9: *Hunt-Sands Flow-Sensitive Type Rules (Algorithmic Version)*

6.3.1 Comparing the Type Systems

The typing environment Γ serves a similar purpose to the E-environment of our dependency configurations by considering the union $\alpha_\cup(E(x))$ as the type of vari-

[2]The inference system ($\vdash_{\mathcal{L}_{HS}}$) of Figure 6.9 has not been parametrised by the choice of flow lattice with the hope that the choice is clear from the context.

able x under Γ. Furthermore, the "program counter" type p achieves the same objective as our I-component of dependency configuration to rule out implicit information flow. With the exception of the *write* construct in our analyses, the main differences between our type system and that of [HS06] lie in the treatment of *while* statements and in the detection of some disjunctive dependencies. We improve on the type system of [HS06] by detecting some disjunctive dependencies and by accounting for possible information release due to termination-sensitivity.

To illustrate the similarity, it is easy to see that for any *while* program, which does not have a *write* statement or *while* statement, we have the property that under any type environment Γ, and flow lattice $\mathcal{L}$, and context p such that for all $x \in \mathbf{Var}, \Gamma(x) = \alpha_{\cup}(E(x))$ and $p = \alpha_{\cup}(I)$, then $p \vdash_{\mathcal{L}} \Gamma \{P\} \Gamma'$ holds iff $\langle E, I, O\rangle\, P\, \langle E', I, O\rangle$ holds and $\Gamma'(x) = \alpha_{\cup}(E'(x))$.

The treatment of *while* statement is different because the type system of [HS06] does not take into account the ways in which values of variables may affect a program's termination behaviour. Specifically, our analysis keeps track of the dependencies of the *while* guard and that of the implicit context in which *while* statement is executed as a potential source of information leakage. This dependency is not thrown away after the fixpoint of the *while* rule, but is retained in the I-component of the post-condition, which intuitively means that the execution of statements afterwards is dependent on the termination or not of the preceding *while* statement. This dependency is also passed on to the E- and O-components since the values of variables after a *while* loop depend on the termination behaviour of the *while* statement, and the observation of termination or nontermination may reveal the execution path to the attacker. To illustrate these observations, consider the program $P \triangleq (l := 0; (\texttt{if(}h = 10\texttt{)then}\,(\texttt{while}\,(\texttt{tt})\,\texttt{do skip})\,\texttt{else skip}); l := 1)$, which

under the environment Γ where $\forall x \in \mathbf{Var}, \Gamma(x) = \{x\}$ and $p = \varnothing$ has the analysis $p \vdash_{\mathcal{L}} \Gamma\ \{P\}\ \Gamma[l \mapsto \varnothing]$. However, the observation of the value of l as 1 on termination reveals information about h, namely that its value is not 10, which this analysis does not capture. Under the environment $\langle E, I, O\rangle$, where $\forall x \in \mathbf{Var}, E(x) = \{\{x\}\}$ and $I = O = \{\varnothing\}$, the analysis of P is $\langle E, I, O\rangle\, P\, \langle E', I', O'\rangle$, where $E'(l) = I' = O' = \{\{h\}\}$. The implicit context I' shows the dependency of P's termination on h, and O' reflects the fact that the attacker may obtain information about h by observing whether or not the program terminates, and since the termination of P affects what final value l can take, $E'(l)$ shows the possible dependency of l on h.

Another area of improvement is in the identification of disjunctive dependencies, where we may want to ensure that an attacker cannot gain information about two chosen secrets at any one time. For example, consider the program $P' \triangleq \texttt{if(}y\texttt{)then}\ l := h_1\ \texttt{else}\ l = h_2$. Using Γ, p and $\langle E, I, O\rangle$ as given above, we obtain $p \vdash_{\mathcal{L}} \Gamma\ \{P'\}\ \Gamma[l \mapsto \{y, h_1, h_2\}]$ suggesting the possible dependency of l on y, h_1 and h_2 on termination of P'. However, the analysis $\langle E, I, O\rangle\ P'\ \langle E[l \mapsto \{\{y, h_1\}, \{y, h_2\}\}], I, O\rangle$ makes explicit the fact that l does not depend on both h_1 and h_2 on termination of P'.

6.4 Improving the Precision of Expression Types

This section shows how to use the abstraction of PERs to improve the typing judgement for expressions. The typing judgement of (6.3) uses the dependencies of the free variables of an expression in a given context to compute the dependency of that expression in the context. This approach is traditionally used in

dependency analyses. However, some free variables in an expression may be irrelevant because their values do not affect the final value of the expression. For example, although h_2 is a free variable in the expression $h_1 + h_2 - h_2$, the value of the expression is independent of h_2. The idea is to take advantage of the semantic information, which identifies this kind of independency, in the typing judgement of expressions, thereby improving the accuracy of analysis.

The equivalence relation construct $e : \mathit{id}$ in the information flow analysis of Chapter 5 already provides us with a way to eliminate irrelevant free variables in the expression e. Specifically, the abstraction $\alpha_{\mathcal{L}}(e : \mathit{id}) \subseteq FV(e)$, which identifies the set of variables that e may depend on in any context, can be used in the typing judgement of e to provide a more precise analysis. This more precise typing judgement for the expression e, under a dependency configuration $\langle E, I, O\rangle$, is given by

$$E \vdash e : t \iff t = \bigsqcup_{x \in \alpha_{\mathcal{L}}(e:\mathit{id})} E(x). \tag{6.6}$$

The dependency $\alpha_{\mathcal{L}}(e : \mathit{id})$ induced by the equivalence relation $e : \mathit{id}$ in (6.6) is the smallest subset of $FV(e)$, elements of which the evaluation of e depends on under any evaluation context. Using the earlier example, the expression $h_1 + h_2 - h_2$ is dependent only on the variable h_1 and hence $\alpha_{\mathcal{L}}((h_1 + h_2 - h_2) : \mathit{id}) = \{h_1\}$. However, the expression $(h_1 - h_1) \times (h_1 + h_2)$ does not depend on any variable, as shown by the fact that $\alpha_{\mathcal{L}}(((h_1 - h_1) \times (h_1 + h_2)) : \mathit{id}) = \varnothing$. Also, for any two expressions e and e' that are semantically equal (that is, for all $\sigma \in \Sigma, \sigma(e) = \sigma(e')$), it is easy to show that the boolean expressions $e = e'$ and $e \neq e'$ are independent of any variable and this is confirmed by the fact that $\alpha_{\mathcal{L}}((e = e') : \mathit{id}) = \alpha_{\mathcal{L}}((e \neq e') : \mathit{id}) = \varnothing$. In fact, any constant expression e has the property that

$\alpha_{\mathcal{L}}(e : id) = \varnothing$, showing that its value is not dependent on any variable.

Proposition 6.4.1. *For any expression* e, $\alpha_{\mathcal{L}}(e : id) \subseteq FV(e)$.

Proof. Since $e : id$ is an equivalence relation, for any $X = \{x\} \subseteq \mathbf{Var}$, such that $x \in \alpha_{\mathcal{L}}(e : id)$, then by definition there exists $\sigma \in \mathbf{\Sigma}$ where $havocX([\sigma]_{e:id}) \neq [\sigma]_{e:id}$. Since $havocX(\cdot)$ is extensive, hence there exists $\sigma' \in havocX([\sigma]_{e:id})$ such that $\sigma' \notin [\sigma]_{e:id}$ and therefore $\sigma'(e) \neq \sigma(e)$. Since there exists a variation in the value of x which causes a variation in the value e, then e is dependent on x and therefore $x \in FV(e)$. □

The equivalence relation $e : id$ used to compute the dependency of e in (6.6) enjoys a special status because it is the most informative PER with respect to the dependency of e in any evaluation context. Formally, this means that for any PER ϕ on the set of values of e, $\alpha_{\mathcal{L}}(e : \phi) \subseteq \alpha_{\mathcal{L}}(e : id)$. Furthermore, the abstraction $\alpha_{\mathcal{L}}(e : id)$ is the smallest set of variables under which the value of e remains invariant when values of variables in this set are fixed. In other words, if a variable $x \notin \alpha_{\mathcal{L}}(e : id)$, then a variation in the value of x cannot cause a variation in the value of e.

Proposition 6.4.2. *For any expression* e *and PER* ϕ *over the set of possible values of* e *we have* $\alpha_{\mathcal{L}}(e : \phi) \subseteq \alpha_{\mathcal{L}}(e : id)$. *Furthermore, if* $X = \{x\} \subseteq \mathbf{Var}$ *and* $x \notin \alpha_{\mathcal{L}}(e : id)$, *then for all* $\sigma, \sigma' \in \mathbf{\Sigma}$ *such that* $\sigma' \in havocX(\{\sigma\})$ *we have* $\sigma(e) = \sigma'(e)$.

Proof. Take any $v \in dom(\phi)$, since id is an equivalence relation $v \in dom(id)$. Now let $\Sigma_v = \{\sigma \in \mathbf{\Sigma} \mid \sigma(e) = v\}$ be the equivalence class of $e : id$ where e evaluates to v. Since PERs are reflexive on their domains then we have $[\sigma]_{e:\phi} =$

$\bigcup_{v \in [\sigma(e)]_\phi} \Sigma_v$ and hence $\Delta([\sigma]_{e:\phi}) \subseteq \bigcup_{v \in [\sigma(e)]_\phi} \Delta(\Sigma_v)$ by applying lemma 6.2.2. Since the dependency of any equivalence class of $e : \phi$ is smaller than the union of the dependency of some equivalence classes of $e : \mathit{id}$ we have that $\alpha_{\mathcal{L}}(e : \phi) \subseteq \alpha_{\mathcal{L}}(e : \mathit{id})$.

For the second part of the proof, now take any equivalence class $[\sigma]_{e:\mathit{id}}$ of the equivalence relation $e : \mathit{id}$, for some $\sigma \in \mathbf{\Sigma}$, then by definition for all $\sigma_1 \in [\sigma]_{e:\mathit{id}}$, $\sigma_1(e) = \sigma(e)$. Since $x \notin \alpha_{\mathcal{L}}(e : \mathit{id})$, then $\mathit{havoc}X([\sigma]_{e:\mathit{id}}) = [\sigma]_{e:\mathit{id}}$ and since $\sigma \in [\sigma]_{e:\mathit{id}}$ then $\mathit{havoc}X(\{\sigma\}) \subseteq \mathit{havoc}X([\sigma]_{e:\mathit{id}})$ by the extensivity of $\mathit{havoc}X(\cdot)$, it thus follows that for any $\sigma' \in \mathit{havoc}X(\{\sigma\}), \sigma' \in [\sigma]_{e:\mathit{id}}$, and hence $\sigma'(e) = \sigma(e)$. □

By replacing the definition of $\vdash$ in the dependency type system of Figure 6.4 with the one given in (6.6) we can thus obtain a more precise analysis by eliminating irrelevant free variables of expressions in typing judgements.

Summary In this chapter we have studied how abstract interpretation techniques may be used to make the analysis of information flow more tractable by simplifying the analysis space. A dependency analysis, developed in this chapter, which is an abstract interpretation of the information flow analysis of Chapter 5, demonstrates the application of the theory of abstract interpretation to information flow analysis. The dependency analysis, which is termination-sensitive, can also detect some disjunctive dependencies. To the best of our knowledge, the dependency analysis is the first to account for information release due to nontermination. A technique presented in this chapter shows how one can improve the precision of expression dependency analysis by using PER abstractions induced by expression evaluations. The next chapter concludes the thesis with further examples and

lessons learnt, and identifies areas of future work.

Chapter 7

Analysis and Discussion

This chapter presents further examples, which illustrate the use of the modelling and analysis techniques presented in this thesis. Examples such as models of authentication, encryption, and statistical analysis are considered to highlight both the theoretical and practical aspects of policy development, and the security analyses of programs against such policies. The Chapter concludes with a review of the main contributions and achievements of the thesis and identifies possible areas of future work.

7.1 Policies for Authentication

Authentication is a fundamental security operation in many systems as the basis of access control. However, by definition, authentication reveals some information about secrets because, for example, a failed password authentication attempt reveals what the password is not. Due to the necessity to release some information about secrets, noninterference cannot be used as a policy for authentication.

We shall therefore study information release policies for authentication, which ensures that only the intended information release is possible in the implementation of the authentication program.

We start by considering an archetypal password authentication program, which demonstrates the release of information about the stored secret. Clearly a real implementation will be different and may perform additional steps, but the core step which is of concern to us is the part where the *user-supplied password* (u) is compared with a password (p), which has been previously stored[1] in the system and is supposed to be known only to the legitimate user. These secrets (or their images) are then compared for equality: if there is a match, the user is authenticated, otherwise the authentication fails. A program modelling the authentication step is shown in Figure 7.1, if the passwords match an output of 1 is produced and otherwise an output of 2 signals authentication failure.

```
1 if (u = p) then
2    write 1;        // authenticated
3 else
4    write 2;        // not authenticated
```

Figure 7.1: *A Model of Authentication*

Intuitively, the authentication program is only allowed to reveal whether the user-supplied password matches the stored password or not. The actual information gained by the attacker during the authentication process however depends on what the attacker knows. On one hand, if the attacker does not know the user-supplied password (say by observing someone being authenticated, but cannot see

[1] In many modern operating systems a password is not directly stored, but its image, which is usually a secure hash of the password itself. The authentication process involves checking the hash of the user-supplied password against the hash of the stored password. In Unix-based systems, salts are also used in order to make dictionary attacks less successful [MT79, Kle90, PS02].

the value that is being entered), the attacker should not learn anything about the password, regardless of the outcome of the authentication. On the other hand, if the attacker knows the supplied password (say, by looking over the shoulder of the user or by entering a guess himself or herself), the attacker can learn at most that the password is equal to the (known) value or not. We can represent this information flow with PERs.

The information released by the password test can be represented by the equivalence relation $(p = u) : \mathit{id}$, which relates only the states where the values of p and u both agree or both disagree. Thus, the desired information flow policy for authentication is $\mathscr{P}_{\mathit{auth}} = \{f_{\mathit{auth}} \mid R \in \mathrm{PER}(\Sigma), f_{\mathit{auth}}(R) \triangleq R \sqcup ((p = u) : \mathit{id})\}$, which allows the attacker to observe only whether the passwords match or not. Now let the program of Figure 7.1 be P, then the analysis, $(E_\perp, I_\perp, O_\perp)\, P\, (E, I, O)$, of this program shows that it satisfies the authentication policy, since $O = ((p = u) : \mathit{id})$.

To see that the policy $\mathscr{P}_{\mathit{auth}}$ captures the intuition about the authentication information release, consider an attacker which knows the user-supplied password. This knowledge can be represented by the equivalence relation id_u, which can distinguish different user supplied passwords:

$$\forall \sigma, \sigma' \in \Sigma, \sigma\ \mathit{id}_u\ \sigma' \iff \sigma(u) = \sigma'(u).$$

Thus, the information that the attacker gains on observing the result of authenti-

cation is represented by the PER $f_{auth}(id_u)$, where for any $\sigma, \sigma' \in \Sigma$

$$\begin{aligned}
\sigma\ f_{auth}(id_u)\ \sigma' &\iff \sigma\ ((p = u) : id \sqcup id_u)\ \sigma' \\
&\iff (\sigma(p = u) = \sigma'(p = u)) \wedge (\sigma(u) = \sigma'(u)) \\
&\iff (\sigma(p) = \sigma(u) \wedge \sigma'(p) = \sigma'(u)) \vee (\sigma(p) \neq \sigma(u) \wedge \sigma'(p) \neq \sigma'(u)) \\
&\quad \wedge (\sigma(u) = \sigma'(u)) \\
&\iff (\sigma(p) = \sigma(u) = \sigma'(p) = \sigma'(u)) \vee (\sigma(p) \neq \sigma(u) = \sigma'(u) \neq \sigma'(p)).
\end{aligned}$$

After observing the result of the execution of the authentication program, a pair of states cannot be distinguished by the attacker that knows the supplied password u, if it is related by the PER $f_{auth}(id_u)$. Since $\sigma\ f_{auth}(id_u)\ \sigma'$ means that either p has the same value as the known value of u under both states σ and σ' (the case for successful authentication), or (for the failed authentication attempt) p must have a value that is different from the known value of u in both states, the attacker therefore learns the value of p when the authentication is successful, otherwise the attacker learns that the value of p is *not* the chosen value of u (since in this case $f_{auth}(id_u)$ relates all states except those in which the value of p agrees with u). This agrees with the intuition.

Now consider the attacker which does not have any prior knowledge of p or u before the authentication, for example, the attacker that is observing another user's attempt to log in but the attacker cannot see the user-supplied password. This attacker's knowledge is the equivalence relation *all*, which cannot distinguish any pair of states. Thus the final knowledge of this attacker after observing the authentication output is $f_{auth}(all) = (p = u) : id$. Since $(p = u) : id$ relates a pair of states only if they both agree or both disagree on the values of p and u, the final

knowledge of this attacker is consistent with the intuition that the attacker only learns the fact that the supplied password and the stored password match in the case of a successful login, or that they do not match in the case of a failed attempt. This information has already been declassified by the policy $\mathscr{P}_{auth}$, and hence the information release is safe.

7.1.1 Authentication Attack

Let us now consider a rogue implementation of the authentication program, shown in Figure 7.2, which contains a trailing attack that reveals the user-supplied password. It is clear that this program contains an attack since the attacker needs not know the user-supplied password *a priori* in order to learn the stored password when there is a successful authentication, or, what the stored password is not otherwise.

```
if (u = p) then
   write 1;        // authenticated
else
   write 2;        // not authenticated
write u;           // attack
```

Figure 7.2: *A rogue authentication program*

Let us call the program of Figure 7.2 P_{Rogue}, the information flow analysis $(E_\perp, I_\perp, O_\perp)\ P_{Rogue}\ (E, I, O)$ shows that P_{Rogue} does not satisfy the policy $\mathscr{P}_{auth}$ since $O = ((u = p) : id) \sqcup id_u \not\sqsubseteq (u = p) : id$. Other variations of this program, for example, where the statement `write` u is moved to different places in the program such as within the conditional statement, or before it, also fail to satisfy the authentication policy.

7.1.2 Information-theoretic Characterisation

Now let us consider an information-theoretic policy for the authentication program above. The use of information theory to model information release in this scenario is reasonable because, for example, the security of password authentication systems is often based on the difficulty of obtaining the password by guesswork, which has a sound information-theoretic justification. In the ideal setting, the stored password should be selected with a uniform probability distribution over a large space of possibilities (although this is usually not the case in practice because of human limitations with respect to remembering long or cryptic passwords). A uniform distribution of the choice maximises the entropy of the password space, and since the maximum entropy over a space of possible choices increases with the size of the space, a large selection space further increases entropy [Sha48].

Now since password authentication does not satisfy noninterference with respect to the secret inputs, we expect some quantitative information to be released by the authentication system. Our objective is to characterise the quantity that may be legally released by the authentication system as a statement of its information flow policy. This can be achieved by using Definition 3.8.5 to derive the information flow of the genuine password authentication program, which gives us a policy characterising the maximum information that may be released legally by any implementation of the authentication program.

The input-output functional model[2] of the authentication program is given by

[2]It does not matter that we did not model the observation of termination, so that the model is $g' : \Sigma \to \{\langle 1, \downarrow\rangle, \langle 2, \downarrow\rangle\}$, because there exists an isomorphism between the range of the two functions so that $g' = \iota \circ g$, so that the kernels of g and g' coincide.

the function $g : \mathbf{\Sigma} \to \{\langle 1\rangle, \langle 2\rangle\}$ (for *genuine*) defined for any $\sigma \in \mathbf{\Sigma}$ as

$$g(\sigma) = \begin{cases} \langle 1\rangle & \text{if } \sigma(p) = \sigma(u) \\ \langle 2\rangle & \text{otherwise.} \end{cases}$$

We shall consider two attackers, A and B. Attacker A is trying to obtain the stored password by guesswork. Attacker B on the other hand is observing the result of A's authentication session, but cannot see the supplied password.

The view of B is defined by the function g because B can only observe the outcome of the authentication - which is public. However, unlike B, A is able to observe also supplied password since A is the one making the guess. So, A's view can be modelled by the function r (for *rogue*) defined for any $\sigma \in \mathbf{\Sigma}$ as

$$r(\sigma) = \begin{cases} \langle 1, i\rangle & \text{if } \sigma(p) = \sigma(u) = i \\ \langle 2, i\rangle & \text{if } \sigma(p) \neq \sigma(u) = i. \end{cases}$$

Thus, in addition to the ability to observe the outcome of the authentication process, attacker A also knows the supplied input $i = \sigma(u)$. Thus, the analysis of information flow to the attacker A is the same as the analysis of the *rogue* authentication program of Figure 7.2, which prints the user-supplied password in addition to revealing the authentication status.

The remainder of the analysis is based on Definition 3.8.5. Let us assume, for simplicity, that the password is chosen from the set $\{0, 1, 2, 3\}$ of possibilities, which is publicly known. Furthermore, we assume that the choice is uniformly distributed so that any of the four possibilities can be chosen with a probability of $\frac{1}{4}$. Since A does not know the stored password, we can assume that A makes

uniformly distributed random guesses on the remaining space of possibilities of the stored password values. So, u is initially chosen by A from the set $\{0,1,2,3\}$ with a probability of $\frac{1}{4}$, and if the authentication fails, A makes the next choice on the remaining set of possibilities with a probability of $\frac{1}{3}$, and so on.

For the attacker B, the initial probability measure $\mu^B \in \mathscr{M}(\mathbf{\Sigma})$ describing B's uncertainty about the password is $\mu^B(\sigma) = \frac{1}{16}$ for any $\sigma \in \mathbf{\Sigma}$, obtained as the joint probability of choosing any p and u. Thus by the definition of g, the (marginal) probability of observing output $\langle 1 \rangle$ for successful authentication is $\overline{\mu}^B(\langle 1 \rangle) = \frac{1}{4}$, and the probability of output $\langle 2 \rangle$ for failed authentication is $\overline{\mu}^B(\langle 2 \rangle) = \frac{3}{4}$. The conditional probabilities $\mu_i(\sigma)$ that input $\sigma \in \mathbf{\Sigma}$ was chosen, given the observed output $i \in \{1,2\}$ are given by

$$\forall \sigma \in \mathbf{\Sigma}, \mu^B_{\langle 1 \rangle}(\sigma) = \begin{cases} \frac{1}{4} & \text{if } \sigma(p) = \sigma(u) \\ 0 & \text{otherwise,} \end{cases}$$

and

$$\forall \sigma \in \mathbf{\Sigma}, \mu^B_{\langle 2 \rangle}(\sigma) = \begin{cases} 0 & \text{if } \sigma(p) = \sigma(u) \\ \frac{1}{12} & \text{otherwise.} \end{cases}$$

Furthermore, $\overline{\mu}^B(\langle 1 \rangle) = \frac{1}{4}$ and $\overline{\mu}^B(\langle 2 \rangle) = \frac{3}{4}$ are the marginal probabilities of producing the outputs under μ^B. Thus, the information released by the authentication program P (modelled by the function g) under the assumption μ^B of B's initial

uncertainty is the mutual information (see Definition 3.8.5)

$$\begin{aligned} I_{\langle P,\mu^B\rangle} &= \mathcal{H}(\mu^B) - \sum_{v\in\{1,2\}} \overline{\mu}^B(v)\mathcal{H}(\mu^B_v) \\ &= 4 - \left(\frac{1}{4}\log(4) + \frac{3}{4}\log(12)\right) \\ &\approx 0.8113 \end{aligned} \tag{7.1}$$

This value gives a measure of the information released by the genuine password program under the assumption μ^B about the attacker's initial uncertainty and it provides us with a statement of policy against which we can measure an implementation of the authentication program. The calculation of the measure of approximately 0.8113 bits of information in (7.1) quantifies the loss in uncertainty about the passwords by revealing their equality (which occurs $\frac{1}{4}$ of the time) or not (which occurs $\frac{3}{4}$ of the time). Thus, the policy allows about 0.8113 bits of information to be released by the implementation of the authentication program to an attacker whose initial uncertainty is modelled by μ^B over the input space.

To see how this policy rejects the implementation P_{Rogue}, now consider the information flow under the view of the attacker A. The view of A corresponds to the implementation P_{Rogue} that reveals also the user-supplied password in addition to the result of authentication. The initial measure of uncertainty of A is $\mu^A \in \mathscr{M}(\mathbf{\Sigma})$, where for any $\sigma \in \mathbf{\Sigma}, \mu^A(\sigma) = \frac{1}{16}$. Thus, $\mu^A = \mu^B$, because A is just making a purely random guess - having no initial knowledge which makes it prefer the selection of a particular password over another from the set of possible choices. Therefore, given the observation of the outputs $i \in \{1,2\}$ and $u' \in \{0,1,2,3\}$ of the possible outcome of the authentication and selected pass-

word respectively, A can compute the conditional probability $\mu_{\langle i,u'\rangle}$ that a given input state was selected as follows:

$$\forall \sigma \in \Sigma. \quad \mu^A_{\langle 1,u'\rangle}(\sigma) = \begin{cases} 1 & \text{if } \sigma(p) = \sigma(u) = u' \\ 0 & \text{otherwise,} \end{cases}$$

and

$$\forall \sigma \in \Sigma. \quad \mu^A_{\langle 2,u'\rangle}(\sigma) = \begin{cases} \frac{1}{3} & \text{if } \sigma(p) \neq \sigma(u) = u' \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, the marginal probability of observing the output sequence $\langle i, u'\rangle$ under μ^A is $\overline{\mu}^A(\langle i, u'\rangle)$, which for any $u' \in \{0, 1, 2, 3\}$ is given by

$$\overline{\mu}^A(\langle i, u'\rangle) = \begin{cases} \frac{1}{16} & \text{if } i = 1 \\ \frac{3}{16} & \text{if } i = 2. \end{cases}$$

Thus, the information released by the implementation P_{Rogue} is given by

$$I_{\langle P_{Rogue}, \mu^A\rangle} = 4 - \frac{3}{4}\log(3) \approx 2.8113. \tag{7.2}$$

Thus, P_{Rogue} is rejected because it releases more than the allowed information (that is, the 0.8113 bits specified by the genuine authentication model g) about the secret inputs. The result of (7.2) demonstrates the fact that the attacker gains complete knowledge of u (2 bits) in addition to the knowledge (about 0.8113 bits on average) about the equality or not of p and u.

Under the information-theoretic analysis, we have represented the security policy as the number of bits of information allow to be released. One argument

against information-theoretic characterisation of secure information flow is the fact that probability measures have to be assigned to each event in order to be able to perform the analysis, and that such probability measures may not be available. However, we note that although the approach requires the assignment of probability measures, the key idea is in the fact that we can *compare programs*, under the same measures to determine their relative security. The information policy actually specifies what information flow property the ideal (with respect to secure information flow) program should have, and measuring a given program against this policy is tantamount to comparing it to this ideal. This technique is what the analyses above suggest, where the rogue implementation is rejected based on the information-theoretic policy written for the genuine implementation (more precisely, its model g). In particular, in our view, the emphasis should not be on the probability measures themselves, or how a specific program fares under different assumptions about the attacker's uncertainty and the distribution of input data. These assumptions may be wrong, and the results, which are dependent on the choice of distributions may thus give us a false sense of (in)security. However, an insecure implementation of a particular system model cannot be made more secure by the choice of the distribution with which analysis is performed. When the precise probability distributions are known however, the argument for information-theoretic analysis is strong.

7.2 Policies For Encryption

Encryption is an important security primitive that is used widely as a security foundation in many systems. However, in order to protect the secrecy of sensitive

data we require policies that permit encryption to be used safely. Noninterference policies cannot be used for encryption since the resulting (public) cyphertext in an encryption scheme will depend on the supplied plaintext and key which are considered secret. Thus there remains the problem of the specification of policies that allow safe use of encryption in programs. In this section we shall study the development of information policies that allow the safe use of encryption and the security analysis of programs which use the encryption functions.

We start by considering an encryption function $\mathcal{E}_1 : K \times M \to C$, which accepts a key chosen from the set K of keys and message or plaintext chosen from the set M, and produces a cyphertext in the set C. Now suppose that $\mathcal{E}_1$ is considered secure and that its implementation, which we shall denote by the expression $\textbf{\textit{enc}}(k, m)$, is correct, so that under any state $\sigma(\textbf{\textit{enc}}(k, m)) = \mathcal{E}_1(\sigma(k), \sigma(m))$. Hence, we allow the attacker to observe the cyphertext $\textbf{\textit{enc}}(k, m)$ for any choice of key $k \in K$ and plaintext $m \in M$ values. Hence we can define an equivalence relation $\lfloor \mathcal{E}_1 \rfloor$ which captures the intended information release, where $\lfloor \mathcal{E}_1 \rfloor$ relates every pair of states σ and σ', where $\mathcal{E}_1(\sigma(k), \sigma(m)) = \mathcal{E}_1(\sigma'(k), \sigma'(m))$. Thus, the required information flow policy $\mathscr{P}_{\mathcal{E}_1} \triangleq \{f \mid R \in \text{PER}(\mathbf{\Sigma}), f(R) = R \sqcup \lfloor \mathcal{E}_1 \rfloor\}$ allows the attacker to observe the ciphertext generated by a correct implementation of the encryption function. Firstly, since the implementation ***enc*** is secure, it is easy to see that the information released by this implementation satisfies the policy $\mathscr{P}_{\mathcal{E}_1}$, because $\lfloor \mathcal{E}_1 \rfloor = \textbf{\textit{enc}}(k, m) : \textit{id}$.

Now consider a secure (as well as an insecure) data backup scenario (adapted from [AHS08]) as shown in the program listings of Figure 7.3. The LHS program securely releases the encrypted data (*ctxt*) to a public output channel after encrypting the data (*data*) with the key k. However, the RHS implementation

is insecure because the programmer releases the plaintext data instead of the ciphertext. The analysis detects that the RHS program violates the policy because $data : id \not\sharp \textbf{\textit{enc}}(k, data) : id$ - unless the encryption function by definition reveals the encrypted data, which violates our assumption that ***enc*** is *secure*. Thus, the analysis detects this flaw. This would be useful, for example, to the programmer who can avoid such programming error by checking his or her implementation against the desired policy.

ctxt : = ***enc***(k, *data*);	*ctxt* : = ***enc***(k, *data*);
write *ctxt*;	**write** *data*;

Figure 7.3: *Secure versus Insecure Data Backup*

The reason why the noninterference policy cannot be used for encryption lies in the fact that noninterference prohibits any sort of variation in the observed output from being induced by a variation in the secret input to the encryption function. However, one of the reasons why encryption is widely used as a security primitive is the fact that the security lies in the ability to protect secret data even when the encryption algorithm is known. Thus, a good encryption algorithm is already designed so that it is not easily invertible into its constituent arguments, although a variation in its input would cause a variation in its output for the algorithm to be useful. The safe input-to-output variation caused by the *definition* of the encryption function is captured by the $\textbf{\textit{enc}}(k, m) : id$ construct in the example above, which allows only the output variations due to the definition of the encryption algorithm to be observed by the attacker. This is safe since the encryption algorithm is already assumed public.

7.2.1 Nondeterministic Encryption

In nondeterministic encryption, such as *cipher-block chaining* encryption mode, an *initialisation vector* (iv) is used along with the key and plaintext such that if a different iv is used, a different ciphertext is generated under the same key and plaintext pair. The term "nondeterministic" refers to the fact that the implementation of such encryption algorithms generally have the property that encrypting the same plaintext several times using the same key would yield different ciphertexts. Let the function $\mathcal{E}_2 : IV \times K \times M \to C$ represent such an encryption scheme, where IV is the set of initialisation vectors, and let the expression $\textbf{\textit{enc}}_*(iv, k, m)$ be a correct implementation of $\mathcal{E}_2$. As with the encryption policy in the previous example above, the required PER modelling the safe release of the ciphertext is $\textbf{\textit{enc}}_*(iv, k, m) : \textit{id}$, which declassifies the ciphertext.

Now a known problem with declassification schemes is that of *occlusion* [SS05], where a legitimately declassified information masks the release of other secrets. Being a *what* policy model, our policy enforcement mechanism prevents such a flow by permitting only the information release that is explicitly allowed by the policy.

$l_1 :=$ ***enc***$_*(iv_1, k, m)$;
if (h) **then**
  $l_2 :=$ ***enc***$_*(iv_2, k, m)$;
else
  $l_2 := l_1$;
write l_1;
write l_2;

Figure 7.4: *The Occlusion Problem*

To illustrate the occlusion problem, consider the program listing of Figure 7.4,

which is adapted from [AHS08]. Suppose that we have declassified the encryption result by the policy $\mathscr{P}_{\mathcal{E}_2} \triangleq \{f \mid R \in \text{PER}(\Sigma), f(R) = R \sqcup \lfloor \mathcal{E}_2 \rfloor\}$, where, similarly to the previous example, for all $\sigma \in \Sigma, \sigma \lfloor \mathcal{E}_2 \rfloor \sigma'$ iff $\mathcal{E}_2(\sigma(iv), \sigma(k), \sigma(m)) = \mathcal{E}_2(\sigma'(iv), \sigma'(k), \sigma'(m))$ and $\lfloor \mathcal{E}_2 \rfloor = \textbf{\textit{enc}}_*(iv, k, m) : \textit{id}$. Thus, revealing the content of l_1 and l_2 is permitted, however the value of the boolean secret h will be released additionally by this program because the inequality of l_1 and l_2 will reveal the fact that the *then* branch was executed. Now let this program be P. Its analysis, $(E_\perp, I_\perp, O_\perp)\ P\ (E, I, O)$ shows that it does not satisfy the required policy because the equivalence relation O has the property that for any $\sigma, \sigma' \in \Sigma$, such that $\sigma(\textbf{\textit{enc}}_*(iv_1, k, m)) \neq \sigma'(\textbf{\textit{enc}}_*(iv_2, k, m))$ then $\sigma\ O\ \sigma' \implies \sigma(h) = \sigma'(h) = \mathbf{tt}$, which reveals the value of the secret h. Hence the analysis shows that P has insecure information flow, because $O \notin \lfloor \mathcal{E}_2 \rfloor$. Specifically, $\lfloor \mathcal{E}_2 \rfloor$ requires, for example, that for any choice of $\sigma \in \Sigma$, $\sigma[h \mapsto \mathbf{ff}]$ must be indistinguishable from σ, but O distinguishes any pair of states which disagree on the produced ciphertext and which also disagree on h, which means that the attacker gains illegal information about h through P which the policy does not allow. Thus, P is rejected.

7.2.2 Disjunctive Key-Ciphertext Release

This example demonstrates policies for the disjunctive release of information. For this we shall consider a symmetric-key encryption system, where on one hand we intend to distribute the key on a secure channel, but we do not want this channel to receive messages encrypted by the key to protect the message from being accessed on this channel. On the other hand, we want to distribute the ciphertext on another channel which may not have access to the key. Now suppose that the encryption

module is implemented as a single program which can be used both to encrypt data and to distribute the key. However, we wish to separate what can be observed on the output channel so that, depending on the usage scenario (indicated by a parameter to the program), the program serves exclusively the purpose of key distribution or exclusively the purpose of encryption. A key release parameter r is used to specify the intention, so that when the value of r is set to *true* only the key is allowed to be released, but when it is *false* only the ciphertext may be released. Such a program is shown in Figure 7.5.

```
if (r) then
   write k;
else
   write enc(k, m);
```

Figure 7.5: *Disjunctive Key-Ciphertext Release*

The required disjunctive release policy is captured by the equivalence relation $D \in \text{PER}(\mathbf{\Sigma})$, which is defined as the disjoint union of two PERs:

$$D = (k : id \sqcup r : \mathbf{T}) \cup (\textbf{\textit{enc}}(k, m) : id \sqcup r : \mathbf{F})$$

The PER $k : id \sqcup r : \mathbf{T}$ in the definition of D allows the key to be released when r has a value $\mathbf{tt}$ only, while the PER $\textbf{\textit{enc}}(k, m) : id \sqcup r : \mathbf{F}$ allows the ciphertext to be released only when r has the value $\mathbf{ff}$. This leads to an information flow policy $\mathscr{P}_D = \{f \mid R \in \text{PER}(\mathbf{\Sigma}), f(R) = R \sqcup D\}$, which is satisfied by the program of Figure 7.5. The programs of Figure 7.6, which can release both the key and ciphertext at the same time are however rejected by this policy.

Alternatively, we may want to force separate implementations of the key dis-

write k;
write ***enc***(k, m);
	write ***enc***(k, m);
	if (r) **then**
	  write k;
	else
	  skip;

Figure 7.6: *Non-Disjunctive Key-Ciphertext Release*

tribution and the encryption sub-modules via another disjunctive policy which is not predicated on the key release parameter r. Such a policy is given by $\mathscr{P}_{D_2} = \{f, f' \mid R \in \text{PER}(\Sigma), f(R) = R \sqcup k : \mathit{id}, f'(R) = R \sqcup \mathbf{enc}(k, m) : \mathit{id}\}$, which allows either the release of the key or the ciphertext but not both. This is possible because $k : \mathit{id}$ and $\mathbf{enc}(k, m) : \mathit{id}$ are incomparable for a secure encryption function ***enc*** - which means that $\mathbf{enc}(k, m)$ does not release the key ($k : \mathit{id} \not\in \mathbf{enc}(k, m) : \mathit{id}$), and variations will occur in the ciphertext due to a variation of m even under a fixed key ($\mathbf{enc}(k, m) : \mathit{id} \not\in k : \mathit{id}$). The programs of Figure 7.7 which implement key release module separately from the encryption module both satisfy the policy $\mathscr{P}_{D_2}$, whereas all the programs of Figure 7.5 and Figure 7.6 do not satisfy this disjunctive key-ciphertext release policy. The program of Figure 7.5 fails because it also releases information about r in addition, while the programs in Figure 7.6 fail because they have non-disjunctive flows.

write k;	**write** ***enc***(k, m);

Figure 7.7: *Separate Key-Ciphertext Release*

7.2.3 Perfect Secrecy

Shannon [Sha48], defined a notion of *perfect secrecy* which describes information flow during encryption where the attacker can observe encrypted messages directly but cannot gain any information about the plaintext or the key. A necessary and sufficient condition for an encryption scheme to satisfy perfect secrecy is that the probability of generating a particular ciphertext c given that a message m was encrypted (under some key) is the same as the probability of generating c given that some other message m' was encrypted (under a different key) [Den82].

The *one-time pad* is an encryption system with such a property, where the encryption key is completely random and is at least as long as the message. Let the relation $\textbf{\textit{enc}}_{otp} \subseteq M \times C$ represent such an encryption scheme, where the plaintext messages in M are of a fixed length n, and are encrypted with a completely random key of the same length to generate the ciphertext. In the following, $\textbf{\textit{enc}}_{otp}$ is defined for any message $m \in M$ as $(m, c) \in \textbf{\textit{enc}}_{otp}$ iff there exists a key $k \in K$ such that m XOR $k = c$.

In this encryption scheme, the required information flow policy on the message is that it reveals 0 bits of information when observing the ciphertext. Now let $\mu_c(m)$ be the conditional probability that m was encrypted given the observation of ciphertext c and let $\mu(m)$ be the (marginal) probability of selecting the message m. Then, the perfect secrecy requirement is that $\mu_c(m) = \mu(m)$ since the attacker does not gain any additional information about the message given any ciphertext. Thus, $\mu_c = \mu$. Furthermore, let the probability of generating the ciphertext c be given by $\overline{\mu}(c)$. By applying Definition 3.8.5 to $\textbf{\textit{enc}}_{otp}$, for any given initial

probability measure μ over M we obtain

$$\begin{aligned} I_{\langle \textbf{\textit{enc}}_{otp},\mu\rangle} &= \mathcal{H}(\mu) - \sum_{c\in C} \overline{\mu}(c)\mathcal{H}(\mu_c) \\ &= \mathcal{H}(\mu) - \mathcal{H}(\mu) \sum_{c\in C} \overline{\mu}(c) \\ &= 0 \end{aligned}$$

Thus our quantitative information analysis of the encryption function $\textbf{\textit{enc}}_{otp}$ shows that it does indeed have the required information flow property and satisfies flow policy which requires no information release.

We may also describe the information flow of $\textbf{\textit{enc}}_{otp}$ in possibilistic terms only using the lattice *FAM*(M) of possibilistic information flow over the set M. Since the length of the key is the same as the message length, then by definition for any ciphertext c all messages are possible. Thus the inverse image of $\textbf{\textit{enc}}_{otp}$ for any $c \in C$ is M and therefore, by using Definition 3.7.1, the information released by $\textbf{\textit{enc}}_{otp}$ is $\{M\}$, which is the least element of the lattice *FAM*(M) containing no information about the secret message. Thus, as expected, the nondeterministic model of $\textbf{\textit{enc}}_{otp}$ of the one-time pad encryption function releases no information about the message under the possibilistic definition of information flow.

7.3 Policies for Statistical Analysis

Issues of secure information release also arise in statistical analyses where we want to permit the safe use of statistical operation on confidential data. Again, noninterference policies cannot be used because the results of statistical computation on sensitive inputs, which we intend to make public, will depend on those

inputs. In this section we shall demonstrate the use of our policy framework for the enforcement of secure information flow in statistical analysis.

Suppose an organisation intends to publish the average salary (its arithmetic mean) of its employees in different sections. This information may be *sensitive* if it discloses too much information about particular individuals' salaries in a given section. Thus, suppose the organisation wishes to specify a policy which allows the average salary of a section to be published only if the section has at least n employees. Now let h_i be the salary of the i^{th} employee in a given section of m employees, the intention is to release the average salary $e_m = \frac{1}{m}\sum_{i=1}^{m} h_i$ only if $m \geq n$. Thus, if we define the flow function f_m such that for any $R \in \text{PER}(\mathbf{\Sigma})$, $f_m(R) = R \sqcup e_m : \mathit{id}$, which allows the release of the average salary of m employees, the intended information flow policy is $\mathscr{P}_{\mathit{avg\text{-}n}} \triangleq \{f_m \mid m \geq n\}$. The policy $\mathscr{P}_{\mathit{avg\text{-}n}}$ requires at least n employees to be considered in the computation of the average salary.

Now suppose $n = 10$, so that the intended policy is $\mathscr{P}_{\mathit{avg}\text{-}10}$. The analysis of the program listing of Figure 7.8 is accepted as safe by this policy, because it considers the salary of at least (actually, exactly) 10 different employees (we assume that h_i corresponds to the salary of a unique employee i). However, both programs of Figure 7.9 are rejected. The LHS program of Figure 7.9 is rejected because it reveals h_3 and the RHS program is rejected because there are inputs to this program (when $m < 10$), which violate the policy. Although, the RHS program of Figure 7.9 may be executed safely when $m \geq 10$, the policy however rejects it because it statistically contains insecure executions. An interesting approach would be a combination with a runtime enforcer, which checks the parameter m and allows the program to run if it will result in a safe execution (that is, whenever

$m \geq 10$). Such a technique is used in the runtime monitor of [GBJS06], which can admit secure execution of programs which may statistically contain insecure traces.

```
sum:=0;
i := 0;
while (i < 10) do
   sum:=sum+h_i;
   i := i + 1;
write  sum / i;
```

Figure 7.8: *Average Salary Calculation*

```
sum:=0;                          sum:=0;
i := 0;                          i := 0;
while (i < 10) do                while (i < m) do
   sum:=sum+h_3;                    sum:=sum+h_i;
   i := i + 1;                      i := i + 1;
write  sum / i;                  write  sum / i;
```

Figure 7.9: *Insecure Average Salary Calculation*

7.4 Electronic Wallet

This example demonstrates the prevention of information laundering by using the pattern of the declassified expression (in a *while* loop). Assume that the setting is that of a privacy-conscious customer engaging in an electronic transaction. In order to process the electronic purchase the vendor needs to verify that the customer has sufficient funds in customer's electronic wallet to proceed with the transaction. The customer is however not willing to divulge more than the fact that he or she has sufficient funds in the electronic wallet. So, the relevant policy is based on

the PER $(balance \leq cost) : id$ which declassifies the boolean test $(balance \leq cost)$ to check whether the customer has sufficient funds (*balance*) for the amount of the transaction (*cost*).

The program listings of Figure 7.10 are both accepted by the policy. The RHS program of Figure 7.10 in particular uses program divergence to signal the result of the electronic wallet check. This is detected by the analysis, but the information flow due to the divergence is safe and is accepted by the policy. However, the two programs of Figure 7.11 are both rejected. On one hand, the LHS program of Figure 7.11 is rejected because it releases the wallet balance in one branch after performing the legitimate check. On the other hand, by modifying the variable *mid* (originally containing the *cost*) in the RHS program of Figure 7.11, the attacker is able to essentially perform a binary search on the secret in the interval 0 and N without explicitly copying the secret. However, the analysis shows that the attacker indeed gains more than the policy allows and thus rejects the program.

```
if (balance ≤ cost) then
  write 1;
else
  write 2;
```

```
while (balance ≤ cost) do
  skip;
```

Figure 7.10: *Electronic Wallet Check*

```
if (balance ≤ cost) then
   write  balance;
else
   write  2;
```

```
bot:=0;
top:=N;
mid:=cost;
while (bot ≤ top) do
   if (balance ≤ mid) then
      if (balance = mid) then
           result:=mid;
      else top:=mid-1;
   else bot:=mid+1;
    mid:=(bot+top)/2;
   …
write  result;
```

Figure 7.11: *Electronic Wallet Attacks*

7.5 Conclusions

In conclusion, we shall summarise the main achievements of this thesis and suggest possible directions for future work.

7.5.1 Main Contributions and Achievements

We have presented a new semantic framework for the analysis and enforcement of secure information flow based on lattices of information. The lattice-theoretic model of information and information flow has the advantage that the approach to the enforcement of secure information flow can be applied independently of the particular representation of information chosen. Representations of information based on PERs, families of sets, and information-theoretic characterisations have been shown to fit into the lattice model of information, and various examples show that the security enforcement technique is the same - relying only on information levels as encoded by a given representation of the information lattice. The lattice-

based approach is simple to understand because it fits well with basic intuitions about information ordering. The view of the lattice structure of information as a general approach to the enforcement of secure information flow has not been systematically studied before. Although lattice-based techniques are commonly used in language-based information flow security, the lattices are usually of security classes in a multilevel system rather than lattices of information. An area of future work is to study how the lattice of information approach presented in this thesis can be integrated with a multilevel security system.

The development of the input-output relational model of systems in Chapter 3 as a foundation for the semantic analysis of information flow is a contribution to the theory of information flow analysis. The relational model was shown, by various definitions and examples, to be a quite general model for information flow analysis in both deterministic and nondeterministic systems. Various representations of information based on PERs, families of sets, and information-theoretic characterisations were developed by using the input-output relational model as the basic primitive. The relational model also shows how the semantics of a system, captured by how it transforms its inputs to outputs as observed by an attacker, can be linked directly to lattices of information under either a qualitative or a quantitative representation of information. Various examples in Chapter 4 demonstrated that reasoning about information flow in nonterminating systems does not pose additional difficulty to the relational model primitive. In particular, a semantic attacker model, defined in Chapter 4, provided a basis for studying information flow under nontermination. As shown by the results, the definitions of information flow under the relational model, induced by the semantic attacker model, account very well for information flow in diverging programs.

Chapter 4 introduced a notion of an attacker's observational power as a function of what the attacker can see during the traces of a program. The definition provided a framework for the study of information flow under various attacker models in relation to the operational semantics of the underlying system. The relationship of this definition to the input-output relational model was shown by relating each input state to what the attacker may observe in the ensuing program execution. The semantic attacker model of Chapter 4 shows how to define a concrete attacker model in a language-based setting, and demonstrates how to derive the relational model from the operational semantics of the language. The approach, applied to the deterministic *While* language, and the nondeterministic *While-ND* and *While-PND*, which feature possibilistic nondeterminism and probabilistic nondeterminism respectively, illustrated the application of the relational model definition from a language-based perspective. We demonstrated that our attacker observational model is more general than the attacker models of [GM04], showing how to obtain the Narrow Abstract Noninterference and the Abstract Noninterference definitions under our information flow definition by choosing suitable observational power functions.

Chapter 5 contributed a PER-based static information flow analysis for *While* programs with output. The analysis, which is flow-sensitive and termination-sensitive can also detect disjunctive information release as defined in Chapter 3. Although PERs were conjectured to be incapable of modelling disjunctive information release [SS05], we showed in Chapter 3 how PERs can represent disjunctive information. Various examples throughout the thesis were used to demonstrate the application of disjunctive information flow, where we want the assurance that a recipient can receive at most one of two secrets during the run of a

program. More specifically, the example of section 7.2.2 showed how disjunctive policies based on PERs can be used to model the disjunctive release of the secret key and ciphertext in a symmetric encryption module.

In Chapter 6 a dependency analysis of *While* programs with outputs was presented to demonstrate the application of abstract interpretation techniques to secure information flow. This dependency analysis was shown to be an abstract interpretation of the concrete analysis with PERs presented in Chapter 5. The dependency analysis is flow-sensitive, termination-sensitive, supports intermediate program outputs, and can detect some disjunctive dependencies. To the best of our knowledge, this is the first dependency type system applied to secure information flow that is termination-sensitive.

7.5.2 Future Work

A lattice-based approach to information and policy modelling has been presented in this thesis for the enforcement of *what* declassification policies. A useful extension to the policy model would be to incorporate it with other lattices, such as the lattice of security clearances in a multilevel security environment. This, for example, would provide a platform to express policies based on *what* and *who* properties [SS05], which is a useful combination of declassification dimensions. By incorporating the *who* dimension, we can express policies such as who is able to gain *what* information via a system or *whose* information a system is permitted to release, as well as *who* is capable of releasing *what* information in a system. This will involve, at the static analysis level, mechanism for annotating program inputs with the (security clearance of the) owner of the input data, and program

outputs with the associated (security clearance of the) observer.

The core *While* language used in this thesis and its extensions to *While-ND* and *While-PND* only have support for buffered input, where all the inputs are supplied at the beginning of program execution. An extension with construct for intermediate input will be a useful step towards the analysis of fully interactive systems. Other language extensions to model, for example, exceptions and object-orientation will also be appropriate steps towards the analysis of real programs.

A basic implementation of the static analysis of Chapter 5 is being developed. A potential application is to embed this analysis into a compiler, which can be used by application writers to certify their applications against policies, for example, under a proof-carrying code [NL97] framework. The full analysis of large programming language sources such as Java, C#, or C++ is still a very distant objective, because of the very rich set of constructs such as exceptions, object-orientation, threading, and so on, that our very basic language models did not study. However, intermediate languages such as the Java bytecode, or the .NET Framework Common Intermediate Language, or machine assembly language are possible initial targets because of the fewer number of language constructs under these intermediate representations. By targeting lower-level representations, there is also the potential to apply the analysis techniques to disassembled programs to check the conformance of a program to policies at the code consumer site, especially when the source of the program, which has access to sensitive information, is not available.

The dependency analysis of Chapter 6 is useful because it is less costly computationally than the PER analysis of Chapter 5, and can be used when one is interested in quickly making noninterference-style checks, or as a front-end to a

richer analysis, since if the dependency abstraction determines that a program has secure flow, then a more expensive analysis of information flow can be avoided. The abstraction functions for the analysis of Chapter 6 were defined with non-interference checks in mind. It will however be useful to study frameworks that can systematically derive efficient abstractions and information flow type systems, which are based on a given policy to be enforced. To give an example, a policy which checks whether at most the parity of a given secret may be released can model all information levels strictly greater than the parity of this secret with one abstraction, leading to far fewer elements in the abstract domain. Thus, a potential area of future work is the study of patterns for generating smaller abstractions, based on the policy to be enforced, which permit partial information flow. Such smaller information lattice abstractions, and the resulting security type systems, may be more suitable for the analysis of larger programs.

Appendix A

Proofs from Chapter 5

Lemma 5.4.8. *Let $\mathbf{\Sigma}$ be the set of all states, which are maps from $\mathbf{Var}$ to values. Suppose $\Sigma, \Sigma' \subseteq \mathbf{\Sigma}$ and that $Z \subseteq \mathbf{Var}$ and let $R, R' \in PER(\mathbf{\Sigma})$. Then we have the following properties:*

1. *Let $X, Y \subseteq \mathbf{Var}$ such that $X \cup Y = Z$, then $havoc Z(\Sigma) \cup havoc Z(\Sigma') = havoc Z(\Sigma \cup \Sigma')$, and $havoc X(havoc Y(\Sigma)) = havoc Z(\Sigma)$.*
2. *The operator $havoc Z(\cdot)$ is an upper closure operator on the powerset lattice $\langle \mathcal{P}(\mathbf{\Sigma}), \subseteq \rangle$ with respect to the subset inclusion order.*
3. *The following identities hold*
   (a) *$havoc Z(\Sigma) \cup havoc Z(\Sigma') = havoc Z(havoc Z(\Sigma) \cup havoc Z(\Sigma'))$.*
   (b) *$havoc Z(\Sigma) \cap havoc Z(\Sigma') = havoc Z(havoc Z(\Sigma) \cap havoc Z(\Sigma'))$.*
   (c) *$havoc Z(\Sigma) \backslash havoc Z(\Sigma') = havoc Z(havoc Z(\Sigma) \backslash havoc Z(\Sigma'))$.*
4. *For all $\sigma \in dom(\uparrow_Z R)$ we have $[\sigma]_{\uparrow_Z R} = havoc Z([\sigma]_{\uparrow_Z R})$.*
5. *For any $X, Y \subseteq \mathbf{Var}$ we have $\uparrow_X \uparrow_Y R = \uparrow_Y \uparrow_X R = \uparrow_{X \cup Y} R$.*
6. $\uparrow_Z R \sqcup \uparrow_Z R' = \uparrow_Z (\uparrow_Z R \sqcup \uparrow_Z R')$.
7. $\uparrow_Z R \uplus \uparrow_Z R' = \uparrow_Z (\uparrow_Z R \uplus \uparrow_Z R')$.
8. $R \sqsubseteq R' \implies \uparrow_Z R \sqsubseteq \uparrow_Z R'$.

Proof.

1. The proof is straightforward from the definition.

2. We show that $\mathit{havocZ}(\cdot)$ is extensive, monotone and idempotent on the powerset lattice $\mathcal{P}(\mathbf{\Sigma})$. Extensivity, $\Sigma \subseteq \mathit{havocZ}(\Sigma)$, is clear from the definition. Now suppose $\Sigma \subseteq \Sigma' \subseteq \mathbf{\Sigma}$ and let $\Sigma' = \Sigma \cup \Sigma''$. Applying (1), we have $\mathit{havocZ}(\Sigma') = \mathit{havocZ}(\Sigma) \cup \mathit{havocZ}(\Sigma'')$. Thus, $\mathit{havocZ}(\Sigma) \subseteq \mathit{havocZ}(\Sigma')$ showing monotonicity. For idempotency, we first observe that extensivity shows that $\mathit{havocZ}(\Sigma) \subseteq \mathit{havocZ}(\mathit{havocZ}(\Sigma))$. Now, suppose $\sigma \in \mathit{havocZ}(\mathit{havocZ}(\Sigma))$, then there exists $\sigma' \in \mathit{havocZ}(\Sigma)$ such that for all $y \in \mathbf{Var}\backslash Z, \sigma(y) = \sigma'(y)$. Hence, there exists $\sigma'' \in \Sigma$ such that for all $y \in \mathbf{Var}\backslash Z, \sigma'(y) = \sigma''(y)$. Since $\forall y \in \mathbf{Var}\backslash Z, \sigma(y) = \sigma'(y) = \sigma''(y)$, then it is clear that $\sigma \in \mathit{havocZ}(\Sigma)$, which implies $\mathit{havocZ}(\mathit{havocZ}(\Sigma)) \subseteq \mathit{havocZ}(\Sigma)$, showing the idempotency of $\mathit{havocZ}(\cdot)$.

3. (a) This property follows directly from the idempotency of $\mathit{havocZ}(\cdot)$ since by (1) $\mathit{havocZ}(\Sigma) \cup \mathit{havocZ}(\Sigma') = \mathit{havocZ}(\Sigma \cup \Sigma')$.

   (b) Let $\Sigma'' = \mathit{havocZ}(\Sigma) \cap \mathit{havocZ}(\Sigma')$. It is clear that $\Sigma'' \subseteq \mathit{havocZ}(\Sigma'')$ by the extensivity of $\mathit{havocZ}(\cdot)$. It now remains to be shown that $\mathit{havocZ}(\Sigma'') \subseteq \Sigma''$. Take any $\sigma \in \mathit{havocZ}(\Sigma'')$, then there exists $\sigma' \in \Sigma''$ such that for all $y \in \mathbf{Var}\backslash Z, \sigma(y) = \sigma'(y)$. Furthermore, there exist $\sigma_1 \in \Sigma$ and $\sigma_2 \in \Sigma'$ such that for all $y \in \mathbf{Var}\backslash Z$, $\sigma'(y) = \sigma_1(y) = \sigma_2(y)$ by the fact that $\sigma' \in \mathit{havocZ}(\Sigma) \cap \mathit{havocZ}(\Sigma')$. Hence for all $y \in \mathbf{Var}\backslash Z$, $\sigma(y) = \sigma'(y) = \sigma_1(y) = \sigma_2(y)$ and therefore $\sigma \in \mathit{havocZ}(\Sigma)$ and $\sigma \in \mathit{havocZ}(\Sigma')$, that is, $\sigma \in \Sigma''$. Hence, $\mathit{havocZ}(\Sigma'') \subseteq \Sigma''$.

(c) Let $\Sigma'' = \mathit{havoc}Z(\Sigma) \backslash \mathit{havoc}Z(\Sigma')$. We have $\Sigma'' \subseteq \mathit{havoc}Z(\Sigma'')$ by the extensivity of $\mathit{havoc}Z(\cdot)$. Thus, it now remains to show that $\mathit{havoc}Z(\Sigma'') \subseteq \Sigma''$. Take any $\sigma \in \mathit{havoc}Z(\Sigma'')$, then there exists $\sigma' \in \Sigma''$ such that for all $y \in \mathbf{Var} \backslash Z, \sigma(y) = \sigma'(y)$. Furthermore, by definition of $\mathit{havoc}Z(\cdot)$ and set difference, there exists $\sigma_1 \in \Sigma$, but there does not exist $\sigma_2 \in \Sigma'$, such that for all $y \in \mathbf{Var} \backslash Z$, $\sigma'(y) = \sigma_1(y) = \sigma_2(y)$. Hence, we have that $\sigma \in \mathit{havoc}Z(\Sigma)$, but $\sigma \notin \mathit{havoc}Z(\Sigma')$ by definition, and therefore, $\sigma \in \mathit{havoc}Z(\Sigma) \backslash \mathit{havoc}Z(\Sigma')$. Thus, $\mathit{havoc}Z(\Sigma'') \subseteq \Sigma''$.

4. Since $\mathit{havoc}Z(\cdot)$ is extensive, it is clear that $[\sigma]_{\uparrow_Z R} \subseteq \mathit{havoc}Z([\sigma]_{\uparrow_Z R})$. Now take any $\sigma' \in \mathit{havoc}Z([\sigma]_{\uparrow_Z R})$, then there exists $\sigma'' \in [\sigma]_{\uparrow_Z R}$ such that for all $y \in \mathbf{Var} \backslash Z, \sigma'(y) = \sigma''(y)$. Since $\sigma'' \in [\sigma]_{\uparrow_Z R}$ then $\sigma \uparrow_Z R\, \sigma''$ holds. It is thus clear from the definition of $\uparrow_Z R$ that $\sigma'' \uparrow_Z R\, \sigma'$ holds since σ' is be obtained from σ'' possibly by modifying values of variables in Z. Transitivity of $\uparrow_Z R$ means that $\sigma \uparrow_Z R\, \sigma'$ also holds and hence $\sigma' \in [\sigma]_{\uparrow_Z R}$, which means that $\mathit{havoc}Z([\sigma]_{\uparrow_Z R}) \subseteq [\sigma]_{\uparrow_Z R}$.

5. We shall start by showing that $\uparrow_X \uparrow_Y R = \uparrow_{X \cup Y} R$. Let $Z = X \cup Y$. Then from the definition of $\uparrow_X(\cdot)$ we have that for any $\sigma, \sigma' \in \mathbf{\Sigma}$, $\sigma \uparrow_X \uparrow_Y R\, \sigma'$ iff there exist sequences $\sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$ and $\tau_1, \ldots, \tau_{n-1} \in \mathit{dom}(\uparrow_Y R)$, such that $\sigma = \sigma_1$ and $\sigma' = \sigma_n$ and for all i, $1 \leq i \leq n-1$ implies $\sigma_i, \sigma_{i+1} \in \mathit{havoc}X([\tau_i]_{\uparrow_Y R}) = \mathit{havoc}Z([\tau_i]_{\uparrow_Y R})$ since by (4) $[\tau_i]_{\uparrow_Y R} = \mathit{havoc}Y([\tau_i]_{\uparrow_Y R})$. Hence, for all i, $1 \leq i \leq n-1$ there exist $\sigma'_i, \sigma'_{i+1} \in [\tau_i]_{\uparrow_Y R}$ such that $\sigma_i \in \mathit{havoc}Z(\{\sigma'_i\})$ and $\sigma_{i+1} \in \mathit{havoc}Z(\{\sigma'_{i+1}\})$ and since σ'_i and σ'_{i+1} are related by $\uparrow_Y R$ then by definition there exist sequences $\sigma^i_1, \ldots, \sigma^i_{m_i} \in \mathbf{\Sigma}$ and $\tau^i_1, \ldots, \tau^i_{m_i - 1} \in \mathit{dom}(R)$ such that $\sigma'_i = \sigma^i_1$ and $\sigma'_{i+1} = \sigma^i_{m_i}$ and $\forall j, 1 \leq j \leq m_{i-1} - 1 \implies \sigma^i_j, \sigma^i_{j+1} \in$

havocY$([\tau_j^i]_R)$. Since $\sigma_i' \in$ *havocY*$([\tau_1^i]_R)$ and $\sigma_i \in$ *havocZ*$(\{\sigma_i'\})$ hence $\sigma_i \in$ *havocZ*$([\tau_1^i]_R)$. Similarly, $\sigma_{i+1} \in$ *havocZ*$([\tau_{m_i-1}^i]_R)$. Hence for any i, $1 \leq i \leq n-1$ we obtain the sequences $\sigma_i, \sigma_1^i, \ldots, \sigma_{m_i-1}^i, \sigma_{i+1} \in \mathbf{\Sigma}$ and $\tau_1^i, \ldots, \tau_{m_i-1}^i \in$ *dom*(R) such that $\sigma_i, \sigma_1^i \in$ *havocZ*$([\tau_1^i]_R)$ and $\sigma_{i+1}, \sigma_{m_i-1}^i \in$ *havocZ*$([\tau_{m_1-1}^i]_R)$ and for all $j, 1 \leq j \leq m_{i-1}-1 \implies \sigma_j^i, \sigma_{j+1}^i \in$ *havocY*$([\tau_j^i]_R) \subseteq$ *havocZ*$([\tau_j^i]_R)$. Since $\sigma = \sigma_1$ and $\sigma' = \sigma_n$, hence by definition, $\sigma \uparrow_Z R\, \sigma'$.

The reverse implication is straightforward because by definition $\sigma \uparrow_Z R\, \sigma'$ holds iff $\exists \sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$ and $\tau_1, \ldots, \tau_{n-1} \in$ *dom*(R) and $\sigma = \sigma_1, \sigma' = \sigma_n$ such that for all $i, i \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in$ *havocZ*$([\tau_i]_R)$. Now, since for any $\tau \in$ *dom*$(R), [\tau]_R \subseteq [\tau]_{\uparrow_Y R}$ and *dom*$(R) \subseteq$ *dom*$(\uparrow_Y R)$ then $\tau_1, \ldots, \tau_{n-1} \in$ *dom*$(\uparrow_Y R)$ and hence by replacing $[\tau_i]_R$ above with $[\tau_i]_{\uparrow_Y R}$ for all i, we obtain $\sigma_i, \sigma_{i+1} \in$ *havocX*$($*havocY*$([\tau_i]_{\uparrow_Y R})) =$ *havocX*$([\tau_i]_{\uparrow_Y R})$ by applying (4). Hence, $\sigma \uparrow_X \uparrow_Y R\, \sigma'$ holds.

Since $\uparrow_X \uparrow_Y R = \uparrow_{X \cup Y} R$, then the fact that set union is commutative means that $\uparrow_X \uparrow_Y R = \uparrow_{X \cup Y} R = \uparrow_{Y \cup X} R = \uparrow_Y \uparrow_X R$.

6. From the definition we have that $\forall \sigma, \sigma' \in \mathbf{\Sigma}$, $\sigma \uparrow_Z (\uparrow_Z R \sqcup \uparrow_Z R')\ \sigma'$

$$\iff \exists \sigma_1, \ldots \sigma_n \in \mathbf{\Sigma}, \exists \sigma''_1, \ldots, \sigma''_{n-1} \in \mathit{dom}(\uparrow_Z R \sqcup \uparrow_Z R').\sigma = \sigma_1, \sigma' = \sigma_n.$$
$$\forall i, 1 \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in \mathit{havocZ}([\sigma''_i]_{\uparrow_Z R \sqcup \uparrow_Z R'})$$

$$\iff \exists \sigma_1, \ldots \sigma_n \in \mathbf{\Sigma}, \exists \sigma''_1, \ldots, \sigma''_{n-1} \in \mathit{dom}(\uparrow_Z R \sqcup \uparrow_Z R').\sigma = \sigma_1, \sigma' = \sigma_n.$$
$$\forall i, 1 \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in \mathit{havocZ}([\sigma''_i]_{\uparrow_Z R} \cap [\sigma''_i]_{\uparrow_Z R'})$$

$$\iff \exists \sigma_1, \ldots \sigma_n \in \mathbf{\Sigma}, \exists \sigma''_1, \ldots, \sigma''_{n-1} \in \mathit{dom}(\uparrow_Z R) \cap \mathit{dom}(\uparrow_Z R').\sigma = \sigma_1, \sigma' = \sigma_n.$$
$$\forall i, 1 \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in \mathit{havocZ}([\sigma''_i]_{\uparrow_Z R}) \cap \mathit{havocZ}([\sigma''_i]_{\uparrow_Z R'})$$
(*by (3b) since by (4)* $\mathit{havocZ}([\sigma''_i]_{\uparrow_Z R}) = [\sigma''_i]_{\uparrow_Z R}$ *and* $\mathit{havocZ}([\sigma''_i]_{\uparrow_Z R'}) = [\sigma''_i]_{\uparrow_Z R'}$)

$$\iff \sigma \uparrow_Z \uparrow_Z R\ \sigma' \text{ and } \sigma \uparrow_Z \uparrow_Z R'\ \sigma'$$

$$\iff \sigma \uparrow_Z R\ \sigma' \text{ and } \sigma \uparrow_Z R'\ \sigma'. \quad (\textit{by applying } (5))$$

7. Let $\Sigma = \mathit{dom}(\uparrow_Z R) \cup \mathit{dom}(\uparrow_Z R')$. Now define the PERs $\overline{\uparrow_Z R}$ and $\overline{\uparrow_Z R'}$ such that $\forall \sigma, \sigma' \in \mathbf{\Sigma}, \sigma\ \overline{\uparrow_Z R}\ \sigma' \iff \sigma, \sigma' \in \Sigma \backslash \mathit{dom}(\uparrow_Z R)$ and $\sigma\ \overline{\uparrow_Z R'}\ \sigma' \iff \sigma, \sigma' \in \Sigma \backslash \mathit{dom}(\uparrow_Z R')$. The PERs $\overline{\uparrow_Z R}$ and $\overline{\uparrow_Z R'}$ both have only one partition, which respectively are the sets $\Sigma_1 = \Sigma \backslash \mathit{dom}(\uparrow_Z R)$ and $\Sigma_2 = \Sigma \backslash \mathit{dom}(\uparrow_Z R')$. Therefore, by (3c) we have that $\mathit{havocZ}(\Sigma_1) = \Sigma_1$ and $\mathit{havocZ}(\Sigma_2) = \Sigma_2$ since by (4) we know that $\mathit{dom}(\uparrow_Z R) = \mathit{havocZ}(\mathit{dom}(\uparrow_Z R))$ and $\mathit{dom}(\uparrow_Z R') = \mathit{havocZ}(\mathit{dom}(\uparrow_Z R'))$ and hence by (1) that $\Sigma = \mathit{havocZ}(\Sigma)$, since $\mathit{havocZ}(\cdot)$ is idempotent. That is, $\uparrow_Z\big(\overline{\uparrow_Z R}\big) = \overline{\uparrow_Z R}$ and $\uparrow_Z\big(\overline{\uparrow_Z R'}\big) = \overline{\uparrow_Z R'}$. By definition $\mathcal{C}_\Sigma(\uparrow_Z R) = \uparrow_Z R \cup \overline{\uparrow_Z R}$, and hence by applying (4), we know that for

any $\sigma \in \mathit{dom}(\mathcal{C}_\Sigma(\uparrow_Z R))$, $[\sigma]_{\mathcal{C}_\Sigma(\uparrow_Z R)} = \mathit{havocZ}([\sigma]_{\mathcal{C}_\Sigma(\uparrow_Z R)})$, because $[\sigma]_{\uparrow_Z R} = \mathit{havocZ}([\sigma]_{\uparrow_Z R})$ and $[\sigma]_{\overline{\uparrow_Z R}} = \mathit{havocZ}([\sigma]_{\overline{\uparrow_Z R}})$. Therefore, for any $\sigma, \sigma' \in \mathbf{\Sigma}$, $\sigma \uparrow_Z \mathcal{C}_\Sigma(\uparrow_Z R)\, \sigma'$ iff $\exists \sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$, $\sigma'_1, \ldots, \sigma'_{n-1} \in \mathit{dom}(\mathcal{C}_\Sigma(\uparrow_Z R))$, such that $\sigma = \sigma_1, \sigma' = \sigma_n$ and $\forall i,\ i \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in \mathit{havocZ}([\sigma'_i]_{\mathcal{C}_\Sigma(\uparrow_Z R)}) = [\sigma'_i]_{\mathcal{C}_\Sigma(\uparrow_Z R)}$. Hence, we have that $\uparrow_Z(\mathcal{C}_\Sigma(\uparrow_Z R)) = \mathcal{C}_\Sigma(\uparrow_Z R)$. Similarly, we obtain $\uparrow_Z(\mathcal{C}_\Sigma(\uparrow_Z R')) = \mathcal{C}_\Sigma(\uparrow_Z R')$. Since by definition, $\uparrow_Z R \uplus \uparrow_Z R' = \mathcal{C}_\Sigma(\uparrow_Z R) \sqcup \mathcal{C}_\Sigma(\uparrow_Z R')$, hence we obtain $\uparrow_Z(\uparrow_Z R \uplus \uparrow_Z R') = \uparrow_Z R \uplus \uparrow_Z R'$ by applying (6).

8. It follows by definition that $\sigma \uparrow_Z R'\, \sigma'$ holds iff there exist $\sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$ and $\sigma'_1, \ldots, \sigma'_{n-1} \in \mathit{dom}(R')$ such that for all i, $1 \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in \mathit{havocZ}([\sigma'_i]_{R'})$ and $\sigma = \sigma_1, \sigma' = \sigma_n$. Since $R \sqsubseteq R'$, we know that $\mathit{dom}(R') \subseteq \mathit{dom}(R)$ and for all $\sigma'_i \in \mathit{dom}(R')$, we have that $[\sigma'_i]_{R'} \subseteq [\sigma'_i]_R$. Hence, $\sigma \uparrow_Z R'\, \sigma'$ implies by the monotonicity of $\mathit{havocZ}(\cdot)$ that there exist $\sigma_1, \ldots, \sigma_n \in \mathbf{\Sigma}$ and $\sigma'_1, \ldots, \sigma'_{n-1} \in \mathit{dom}(R)$ such that for all i, $1 \leq i \leq n-1 \implies \sigma_i, \sigma_{i+1} \in \mathit{havocZ}([\sigma'_i]_R)$ and $\sigma = \sigma_1, \sigma' = \sigma_n$, which implies that $\sigma \uparrow_Z R\, \sigma'$ holds. This shows the required property that $\uparrow_Z R \sqsubseteq \uparrow_Z R'$.

□

Lemma 5.7.3. *Let $Z \subseteq \mathbf{TVar}$, and let e be an expression such that $FV(e) \subseteq \mathbf{Var}$, and let $R, R' \in \mathcal{R}_{\text{init}} \subseteq PER(\mathbf{\Sigma})$.*

1. *For all $\sigma \in dom(R)$, $havocZ([\sigma]_R) = [\sigma]_{\uparrow_Z R}$.*
2. *For all $\sigma, \sigma' \in dom(R), \sigma \uparrow_Z R\, \sigma' \implies \sigma\, R\, \sigma'$.*
3. *$R \sqcup R' \in \mathcal{R}_{\text{init}}$.*
4. *Let $X \subseteq \mathbf{TVar}$ and $Y = \mathbf{TVar} \backslash X$ such that $\forall \sigma, \sigma' \in dom(R), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$ and $havocY([\sigma]_R) = [\sigma]_R$. Furthermore, suppose $FV(e) \cap \mathbf{TVar} \subseteq X$. Then for any PER ϕ over the values of e, we have that $e : \phi \sqcup R \in \mathcal{R}_{\text{init}}$.*

Proof.

1. Since $R \in \mathcal{R}_{init}$, then there exist $X \subseteq \mathbf{TVar}$ and $Y = \mathbf{TVar} \backslash X$ such that for all $\sigma, \sigma' \in dom(R)$, $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$ and $havocY([\sigma]_R) = [\sigma]_R$. Now take any $\sigma_1, \sigma_2 \in dom(R)$ and suppose that $(\sigma_1, \sigma_2) \notin R$, then, $[\sigma_1]_R \cap [\sigma_2]_R = \varnothing$. Hence, $havoc\mathbf{TVar}([\sigma_1]_R) \cap havoc\mathbf{TVar}([\sigma_2]_R) = \varnothing$. This is straightforward to show, since if there exist $\sigma \in [\sigma_1]_R$ and $\sigma' \in [\sigma_2]_R$ such that $havoc\mathbf{TVar}(\{\sigma\}) \cap havoc\mathbf{TVar}(\{\sigma'\}) \neq \varnothing$, then $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}}$ and therefore $\sigma_{\downarrow X} = \sigma'_{\downarrow X}$. However, $\sigma' \in havocY([\sigma]_R) = [\sigma]_R = [\sigma_1]_R$ violates our initial assumption that $[\sigma_1]_R$ and $[\sigma_2]_R$ are disjoint.

   Now since $Z \subseteq \mathbf{TVar}$, we have that for all $\sigma_1, \sigma_2 \in dom(R)$ then $(\sigma_1, \sigma_2) \notin R$ implies $havocZ([\sigma_1]_R) \cap havocZ([\sigma_2]_R) = \varnothing$ and hence, by definition, $\sigma \uparrow_Z R \sigma'$ iff there exists $\sigma'' \in dom(R)$ such that $\sigma, \sigma' \in havocZ([\sigma'']_R)$. That is, for any $\sigma \in dom(R), [\sigma]_{\uparrow_Z R} = havocZ([\sigma]_R)$.

2. We have shown from (1) that for any $\sigma_1, \sigma_2 \in dom(R)$ $(\sigma_1, \sigma_2) \notin R$ implies $havoc\mathbf{TVar}([\sigma_1]_R) \cap havoc\mathbf{TVar}([\sigma_2]_R) = \varnothing$. Furthermore, since

$Z \subseteq \mathbf{TVar}$, we know that $\mathit{havocZ}([\sigma_1]_R) = [\sigma_1]_{\uparrow_Z R}$ and $\mathit{havocZ}([\sigma_2]_R) = [\sigma_2]_{\uparrow_Z R}$. Therefore, $[\sigma_1]_{\uparrow_Z R} \cap [\sigma_2]_{\uparrow_Z R} = \varnothing$. That is, for any $\sigma_1, \sigma_2 \in \mathit{dom}(R)$, $(\sigma_1, \sigma_2) \notin R \implies (\sigma_1, \sigma_2) \notin \uparrow_Z R$. The contrapositive of this shows that for all $\sigma, \sigma' \in \mathit{dom}(R), \sigma \uparrow_Z R\, \sigma' \implies \sigma\, R\, \sigma'$.

3. Since $R, R' \in \mathcal{R}_{\mathit{init}}$, then there exist $X, X' \subseteq \mathbf{TVar}$ and $Y = \mathbf{TVar} \backslash X$ and $Y' = \mathbf{TVar} \backslash X'$ such that for any $\sigma, \sigma' \in \mathit{dom}(R), \mathit{havocY}([\sigma]_R) = [\sigma]_R$ and $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$, and such that for any $\sigma, \sigma' \in \mathit{dom}(R'), \mathit{havocY'}([\sigma]_{R'}) = [\sigma]_{R'}$ and $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$. Let $\hat{X} = X \cup X'$ and let $\hat{Y} = Y \cap Y'$, then it is clear that for all $\sigma, \sigma' \in \mathit{dom}(R) \cap \mathit{dom}(R') = \mathit{dom}(R \sqcup R')$, $\sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow \hat{X}} = \sigma'_{\downarrow \hat{X}}$. Furthermore, we also know that for any $\sigma, \sigma' \in \mathit{dom}(R \sqcup R')$, $\mathit{havoc}\hat{Y}([\sigma]_R) = [\sigma]_R$ and $\mathit{havoc}\hat{Y}([\sigma']_{R'}) = [\sigma']_{R'}$, hence by (3b) of lemma 5.4.8, we have that $\mathit{havoc}\hat{Y}(\mathit{havoc}\hat{Y}([\sigma]_R) \cap \mathit{havoc}\hat{Y}([\sigma']_{R'})) = \mathit{havoc}\hat{Y}([\sigma]_R \cap [\sigma']_{R'}) = \mathit{havoc}\hat{Y}([\sigma]_R) \cap \mathit{havoc}\hat{Y}([\sigma']_{R'}) = [\sigma]_R \cap [\sigma']_{R'}$. This means that for any $\sigma \in \mathit{dom}(R \sqcup R'), \mathit{havoc}\hat{Y}([\sigma]_{R \sqcup R'}) = [\sigma]_{R \sqcup R'}$. Since $\hat{X} \subseteq \mathbf{TVar}$ and $\hat{Y} = \mathbf{TVar} \backslash \hat{X}$, then $R \sqcup R' \in \mathcal{R}_{\mathit{init}}$.

4. Firstly, because $\mathit{dom}(e : \phi \sqcup R) \subseteq \mathit{dom}(R)$, then it is clear that $\forall \sigma, \sigma' \in \mathit{dom}(e : \phi \sqcup R), \sigma_{\downarrow \mathbf{IVar}} = \sigma'_{\downarrow \mathbf{IVar}} \implies \sigma_{\downarrow X} = \sigma'_{\downarrow X}$. Now let $FV(e) \cap \mathbf{TVar} = X'$ and let $Y' = \mathbf{TVar} \backslash X'$. Then, by definition, for any $\sigma \in \mathit{dom}(e : \phi)$, $\mathit{havocY'}([\sigma]_{e:\phi}) = [\sigma]_{e:\phi}$ since e has no free variable in Y' and thus its evaluation is not affected by the Y' projection of states. Since $X' \subseteq X$ and hence $Y \subseteq Y'$, by applying (3b) of lemma 5.4.8 then for any $\sigma \in \mathit{dom}(e : \phi \sqcup R)$, $\mathit{havocY}([\sigma]_{e:\phi \sqcup R}) = \mathit{havocY}([\sigma]_{e:\phi} \cap [\sigma]_R) = [\sigma]_{e:\phi} \cap [\sigma]_R = [\sigma]_{e:\phi \sqcup R}$. Thus, $e : \phi \sqcup R \in \mathcal{R}_{\mathit{init}}$. □

Bibliography

[AB04] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *11th Static Analysis Symposium (SAS), Verona, Italy*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2004.

[ABG04] A. Aldini, M. Bravetti, and R. Gorrieri. A process-algebraic approach for the analysis of probabilistic noninterference. *Journal of Computer Security*, 12(2):191–245, 2004.

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Conference Record of POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, January 20–22, 1999.

[AFV01] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, chapter 3, pages 197–291. Elsevier Science, 2001.

[Aga00] J. Agat. Transforming out timing leaks. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on*

Principles of Programming Languages, pages 40–53, Boston, Massachusetts, January 19–21, 2000.

[AGM92] S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors. *Handbook of Logic in Computer Science, Vol 1*. Oxford University Press, 1992.

[AHS08] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theoretical Computer Science*, 402(2-3):82–101, 2008.

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. 13th European Symposium on Research in Computer Security (ESORICS'08)*, volume 5283 of *Lecture Notes in Computer Science*, Malaga, Spain, October 2008. Springer-Verlag.

[AP08] A. Aldini and A. Di Pierro. Estimating the maximum information leakage. *International Journal of Information Security*, 7(3):219–242, 2008.

[AS07] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, pages 207–221. IEEE Computer Society, 2007.

[Bac05] M. Backes. Quantifying probabilistic information flow in computational reactive systems. In *Proceedings of 10th European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *Lecture Notes in Computer Science*, pages 336–354. Springer, September 2005.

[BD03] Y. Beres and C. I. Dalton. Dynamic label binding at run-time. In *Proceedings of the 2003 workshop on New security paradigms*, pages 39–46. ACM Press, 2003.

[Ben04] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 14–25, New York, NY, USA, 2004. ACM Press.

[BGM07] A. Banerjee, R. Giacobazzi, and I. Mastroeni. What you lose is what you leak: Information leakage in declassification policies. In *Mathematical Foundations of Programming Semantics (MFPS'07)*, volume 173, pages 47–66. Electronic Notes in Theoretical Computer Science, 2007.

[BL06] M. Backes and P. Laud. Computationally sound secrecy proofs by mechanized flow analysis. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 370–379, New York, NY, USA, 2006. ACM Press.

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353. IEEE Computer Society, 2008.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Sympo-*

sium on Principles of Programming Languages, pages 238–252, Los Angeles, California, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, January 1979.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CC08] S. Cavadini and D. Cheda. Run-time information flow monitoring based on dynamic dependence graphs. In *ARES '08: Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 586–591, Washington, DC, USA, 2008. IEEE Computer Society.

[CHM02] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2002.

[CHM05] D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science*, 112:149–166, January 2005. Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages (QAPL 2004).

[CHM07] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.

[CLN00] C. Colby, P. Lee, and G. Necula. A Proof-Carrying Code Architecture for Java. In *Tool section of the Proc. of the 12th International Conference on Computer Aided Verification (CAV00)*, 2000.

[CM04] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[CMS05] M. R. Clarkson, A. C. Myers, and F. B. Schneider. Belief in information flow. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 31–45, Washington, DC, USA, 2005. IEEE Computer Society.

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, pages 63–78, Berkeley, CA, USA, 1998. USENIX Association.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Den82] D. E. Denning. *Cryptography and Data Security*. Addison Wesley, 1982.

[DP03] B.A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2003.

[End77] H. B. Enderton. *Elements of Set Theory*. Academic Press, 1977.

[FG03] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[GBJS06] G. Le Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In Mitsu Okada and Ichiro Satoh, editors, *In Proceedings of 11th Annual Asian Computing Science Conference (ASIAN 2006)*, volume 4435 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2006.

[GHK+03] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Cambridge University Press, Cambridge, 2003.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, CA, April 1982. IEEE Computer Society Press.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 186–197. ACM Press, 2004.

[GM05] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *European Symposium on Programming (ESOP'05).*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, 2005.

[Gon99] L. Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Hal03] J. Y. Halpern. *Reasoning about Uncertainty*. The MIT Press, Cambridge, Massachusetts, 2003.

[HR98] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) , San Diego, California*, pages 365–377, 1998.

[HS91] S. Hunt and D. Sands. Binding time analysis: A New PERspective. *ACM SIGPLAN Notices*, 26(9):154–165, September 1991.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. Principles of Programming Languages, 33rd Annual ACM SIGPLAN - SIGACT Symposium (POPL'06)*, Charleston, South Carolina, USA, January 2006. ACM Press.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[Hun91a] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. Ph.D. thesis, Department of Computing, Imperial College, London, UK, 1991.

[Hun91b] S. Hunt. Pers generalise projections for strictness analysis (extended abstract). In *Proc. 1990 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ullapool, 1991. Springer-Verlag.

[JL00] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

[Kah87] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.

[Kle90] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In USENIX, editor, *UNIX Security II: USENIX workshop proceedings, August 27–28, 1990, Portland, Oregon*, pages 5–14, pub-USENIX:adr, 1990. USENIX.

[Koh03] J. Kohlas. *Information Algebras: Generic Structures for Inference*. Springer-Verlag, 2003.

[Lau01] P. Laud. Semantics and program analysis of computationally secure information flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2001.

[Lau03] P. Laud. Handling encryption in an analysis for secure information flow. In Pierpaolo Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2618 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2003.

[Ler03] X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[Low02] G. Lowe. Quantifying information flow. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 18–31. IEEE Computer Society, 2002.

[LR93] J. Landauer and T. Redmond. A lattice of information. In *Proceedings of the Computer Security Foundations Workshop VI (CSFW '93)*, pages 65–70, Washington - Brussels - Tokyo, June 1993. IEEE.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2 edition, 1999.

[Mac03] D. MacKay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, September 2003.

[Mal07] P. Malacaria. Assessing security threats of looping constructs. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 225–235. ACM, 2007.

[Mas05] I. Mastroeni. On the Rôle of abstract non-interference in language-based security. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2005.

[McL94] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[MF97] G. McGraw and E. Felten. *Java Security*. Wiley, 1997.

[Mil99] R. Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, 1999.

[MSZ06] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[MT79] R. Morris and K. Thompson. Password security: a case history. *Communications of the ACM*, 22(11):594–597, 1979.

[Muc97] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[MZZ+08] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001–2008.

[NCH+05] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005.

[NL97] G. Necula and P. Lee. Research on proof-carrying code for untrusted-code security. In *Proceedings of the 1997 Conference on Security and Privacy (S&P-97)*, pages 204–204, Los Alamitos, May 4–7 1997. IEEE Press.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[OCC06] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 190–201, Washington, DC, USA, 2006. IEEE Computer Society.

[PAK02] S. Prasad and S. Arun-Kumar. Introduction to operational semantics. In *The Compiler Design Handbook*, pages 841–890. CRC Press, 2002.

[PHW02] A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *15th IEEE Computer Security Foundations Work-*

shop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada, pages 3–17. IEEE Computer Society, 2002.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Report DAIMI FN–19, Aarhus University, September 1981.

[PS02] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 161–170, New York, NY, USA, 2002. ACM Press.

[RMMG01] P. Ryan, J. McLean, J. Millen, and V. Gligor. Non-interference, who needs it? In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 237–240, Washington - Brussels - Tokyo, June 2001. IEEE.

[Ros06] S. Ross. *A first course in probability*. Prentice Hall, New Jersey, 7 edition, 2006.

[Sab01] A. Sabelfeld. *Semantic Models for the Security of Sequential and Concurrent Programs*. PhD thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, May 2001.

[Sch00] F. B. Schneider. Enforceable security policies. *ACM Transaction of Information and System Security*, 3(1):30–50, 2000.

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[SM03a] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SM03b] A. Sabelfeld and A. C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *ISSS*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2003.

[Smi01] G. Smith. A new type system for secure information flow. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 115–125, Washington - Brussels - Tokyo, June 2001. IEEE.

[Smi03] G. Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *CSFW*, pages 3–13. IEEE Computer Society, 2003.

[Smi06] G. Smith. Improved typings for probabilistic noninterference in a multi-threaded language. *Journal of Computer Security*, 14(6):591–623, 2006.

[Smi07] G. Smith. Adversaries and Information Leaks (Tutorial). In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 383–400. Springer, 2007.

[SS01] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[SS07] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 2007.

[SST07] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, pages 203–217, 2007.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, San Diego, California, 19–21 January 1998.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, June 1955.

[Vol99a] D. Volpano. Formalization and proof of secrecy properties. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW '99)*, pages 92–97, Washington - Brussels - Tokyo, June 1999. IEEE.

[Vol99b] D. M. Volpano. Safety versus secrecy. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 303–311, London, UK, 1999. Springer-Verlag.

[VS97] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

[VS00] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–276, New York, NY, USA, 2000. ACM Press.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 1993.

[Zda04a] S. Zdancewic. Challenges for information-flow security. In *Proceedings of Programming Language Interference and Dependence (PLID)*, August 2004.

[Zda04b] S. Zdancewic. A type system for robust declassification. In Stephen Brookes and Prakash Panangaden, editors, *Electronic Notes in Theoretical Computer Science*, volume 83. Elsevier, 2004.

[ZM01] S. Zdancewic and A. C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, June 2001.