A Policy-Based Vulnerability Analysis Framework

By

SOPHIE JEAN ENGLE
B.S. (University of Nebraska at Omaha) 2002

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Matt Bishop (Chair)

Professor S. Felix Wu

Professor Karl Levitt

Professor Sean Peisert

Committee in Charge

2010

Dedicated to my dad, D. H. Engle.

Abstract

Repeatability is essential to any science—computer science is no exception. However, the area of vulnerability analysis suffers from ambiguous definitions that hinder the repeatability of analysis results. Many researchers have turned to policy-based definitions of a vulnerability in an attempt to alleviate this ambiguity. However, it is rare that security policies are explicitly and precisely defined. As a result, these policy-based approaches merely shift the ambiguity from defining vulnerabilities to defining policies. Other researchers turn to strictly formal models and methods to provide repeatable results, but the practicality of such analysis is limited by the complexity of the environment and the availability of resources. This creates a conflict between repeatability and practicality that is often left unresolved in existing vulnerability analysis methods; an analysis framework either focuses on formal models to provide repeatability, or uses an ad hoc approach to provide practicality.

This dissertation addresses this conflict by balancing specific formal and practical objectives to create a vulnerability analysis framework capable of producing repeatable results in realistic environments. This analysis framework relies on three major components: a hierarchy of security policies, a formal model of implementation vulnerabilities, and an implementation vulnerability classification scheme. We address the ambiguity surrounding security policies with a hierarchy that precisely defines security policies at four levels of abstraction. We use this policy hierarchy to provide a formal model of an implementation vulnerability. This model provides the formal foundation for our characteristic-based vulnerability classification scheme, which allows us to examine implementation vulnerabilities at a more practical level of abstraction. We combine these components into a cohesive implementation vulnerability analysis framework that provides insight into both when a system is non-secure, and how to mitigate that non-security.

Acknowledgments

I am grateful for the support of all those that helped me realize this degree. Foremost, I want to thank my adviser Professor Matt Bishop for his constant support and guidance. He provided invaluable advice throughout my entire graduate career. I am positive that I would not have achieved this goal without his help and encouragement. I also want to thank Professor Sean Peisert for his excellent feedback and advice, and everyone else on my dissertation and qualifying exam committees for guiding my progress over the years.

I want to also thank all of those that helped me before I became a graduate student. I especially want to thank Professor Blaine Burnham for his insight and advice. He not only introduced me to the area of computer security, but also encouraged me to explore graduate school. I want to thank Professor Ken Dick, whose early support allowed me to gain both valuable industry and teaching experience as an undergraduate. I am also grateful for the encouragement provided by Dr. Winnie Callahan. Her drive inspired me to push my own boundaries.

I want to thank my partner in research and in life, Sean Whalen, for the decade of support he has given me. His support gives me strength even in the toughest of times, and the courage to tackle the challenges each day brings. I would be lost without him.

I am also grateful for the constant support of my family. I want to thank my sisters Dannielle and Charlene for always being there for me, even as they work on their own graduate degrees. I want to thank my mom for her unwavering support and patience, which has helped me more than she knows.

Finally, I want to thank my dad. He taught me the importance of education, and was the best teacher (with or without a Ph.D.) that I have ever known. Without him, none of this would be possible. His legacy continues to inspire me today.

Table of Contents

CHAPTER 3
Security Policy

CHAPTER 4
Vulnerability Hierarchy

CHAPTER 5
Vulnerability Classification

CHAPTER 6
Vulnerability Analysis

APPENDIX B
Discussion

List of Figures

List of Numbered Definitions

CHAPTER 1
Introduction

This dissertation introduces the Policy-Based Vulnerability Analysis Framework, which balances formal and practical objectives to achieve a framework capable of producing repeatable results in realistic environments. This chapter discusses the motivation, scope, and objectives of this framework, and provides an overview of our approach.

§1.1 Motivation

The area of vulnerability analysis is plagued by ambiguity. The term "vulnerability" itself is not precisely defined, despite its "fundamental nature" in computer security [FIT04]. Authors often draw an ambiguous line between vulnerabilities, bugs, errors, exploits, exposures, flaws, incidents, threats, and weaknesses. As a result, what is considered a vulnerability varies between websites, books, papers, and online vulnerability databases. For example, a vulnerability is defined in one book as "an error or weakness in the design, implementation, or operation of the system" [SCH99, *p*316], but as only those weaknesses that "cause harm to the stakeholders of an application" by another [OWA09].

Krsul addresses this by introducing a "unifying definition" of a software vulnerability that is based on security policy [KRS98]. A security policy specifies what is considered misuse in a specific environment, allowing us to distinguish among software bugs, errors, or flaws from vulnerabilities. A policy-based notion of a vulnerability has been adopted by many in the security community [BIS99, FIT04, CVE09].

However, a security policy itself is often ambiguous and rarely explicitly or precisely defined. For example, the Common Vulnerabilities and Exposures (CVE) website defines a vulnerability as a mistake in software that violates a "reasonable security policy" for that system, but does

not specify what is considered reasonable in this context [CVE09]. As a result, policy-based definitions shift the ambiguity from the definition of a vulnerability to the definition of a security policy. The traditional pillars of confidentiality, integrity, and availability are often cited to make a distinction between security policies and system requirements. As pointed out by Sterne, this lacks the necessary precision and generality to define a specific security policy [STE91].

Further complicating matters, security policies exist at multiple levels of abstraction. For example, a security policy at the system level specifies the actions that are authorized for each user account. Vulnerability analysis usually places emphasis on this level of the security policy. However, security policies at an organizational level predate computer systems. Carlson addresses this with the "Unifying Policy Hierarchy," which provides a hierarchy of security policies and vulnerabilities at different levels of abstraction [CAR06].

One option is to combat this ambiguity with formal models and methods, which provide precise definitions and repeatable results [HUL94]. For example, Klein et al. formally verified 7,500 lines of C code for seL4, a general purpose operating system microkernel for embedded systems [KLE09]. They prove the implementation of the seL4 microkernel matches the specification, eliminating an entire class of vulnerabilities. However, it took over 20 person-years and 200,000 lines of proof script to formally verify the seL4 microkernel [KLE09]. The widely used Linux kernel (version 2.6.25) is estimated to be over 6,700,000 lines of source code [MCP08], several orders of magnitude larger than the seL4 microkernel. This type of formal verification is repeatable, but may be impractical for environments dependent on more complex systems.

Significant progress has been made in applying formal model checking to larger code bases such as the Linux kernel. For example, Chen et al. demonstrated that model checking using MOPS, a static analysis tool, is practical by analyzing over one million lines of code [CHE04]. Schwarz et al. later used MOPS to perform model checking on an entire Linux distribution in approximately 150 person-hours [SCH05]. However, to do so, they focused on a small number of temporal safety properties. While formal methods are repeatable, the scope at which they may be used in practice is limited by the complexity of the systems and the resources available.

As analysis becomes less formal, it becomes more difficult to replicate. Consider the recent recommendations to include "Open Ended Vulnerability Testing," a term for penetration testing, in the latest Voluntary Voting System Guidelines [VVS07]. Penetration testing is an "informal,

non-rigorous technique for checking the security of a system" and an important component of vulnerability analysis in practice [BIS03A, *p*660]. However, frameworks for penetration testing such as the "Flaw Hypothesis Methodology" [LIN75, WEI95] lack a "systematic examination" of the system [BIS03A, *p*660]. As a consequence, two different teams performing penetration tests on the same system may produce widely varying results.

Herein lies the problem. Repeatability is an integral part of a scientific experiment [BOY00, FEY74]. However, the repeatability of the results from vulnerability analysis is hindered by imprecise notions of a vulnerability and a security policy. Formal models and methods are able to provide this precision and repeatability, but at a cost that may not always be practical. This leads us to the following problem statement:

> PROBLEM STATEMENT: Vulnerability analysis requires a framework such that analysis is both repeatable and practical.

Unfortunately, repeatability and practicality are often conflicting objectives. Formal results are repeatable in theoretical environments, but are often too complex or time consuming to directly apply in practice. Methods with no formal considerations are often ad hoc and difficult if not impossible to replicate independently. This tension creates a wide gap between theory and practice. The focus of research is often on either theoretical or practical objectives, but rarely both. We examine how to bridge the gap between theory and practice for certain small-scale environments—and as a result, provide a vulnerability analysis framework that is capable of both repeatable and practical analysis.

§1.2 Analysis Scope

A flexible framework may be tailored based on the amount of resources available and the specific environment in question. However, the scope of vulnerability analysis can range from analyzing a single specific system to examining a large and diverse set of systems in a dynamic environment. We focus on vulnerability analysis for a stable, small-scale environment with a handful of systems. We informally define an *environment* to include a set of resources, the users of those resources, and the security policy specifying how those users are authorized to interact with those resources. We consider an environment *stable* when the resources, users, and security policy of that environment exhibit no significant change over moderate periods of time.

Finally, reproducibility is essential for scientific computer security experiments [PEI07A]. This dissertation focuses on repeatability, which is only one aspect of reproducibility.

§1.3 Objectives

We address the problem statement introduced in section 1.1 with the Policy-Based Vulnerability Analysis Framework. Our primary objective is as follows:

> PRIMARY OBJECTIVE: Provide a framework for policy-based vulnerability analysis such that analysis is both repeatable and practical.

A framework that achieves a balance between theory and practice provides a principled and repeatable approach to vulnerability analysis. However, we must avoid compromising our objectives to the degree that our framework is neither formal nor practical. To this end, our framework must have several other qualities.

We seek a strong theoretical foundation for the sake of repeatability. Along these lines, the framework should utilize well-established formal concepts in computer science and computer security, and replicate foundational theoretical results. We also consider how practice can influence our theoretical framework. For example, to capture a realistic environment, the framework should take into account both security procedures and security mechanisms—regardless of whether they are technical or non-technical in nature. The framework should ultimately reflect how the concepts of a vulnerability and a security policy are discussed and used by the security community. Specifically, the framework should allow analysts to gain *insight*, but not break well-established *intuition*.

We also take into consideration how ambiguity may be introduced into the framework when moving between theoretical and practical settings. We accept certain levels of ambiguity for the sake of practicality, but try to encapsulate where ambiguity is introduced and explore how ambiguity affects the results provided by our analysis framework. In essence, we should be unambiguous about where and how ambiguity affects the framework.

Finally, while practicality and repeatability are properties that are difficult to prove, these properties can be demonstrated. As such, the framework should be demonstrated on a realistic environment. We seek to balance all of these objectives throughout our entire Policy-Based

Vulnerability Analysis Framework. However, different components of our framework achieve different levels of balance between theory and practice. We discuss the specific focus of each component in the following chapters, and revisit how well we achieve our framework objectives after the entire framework has been introduced.

§1.4 Approach

We build the Policy-Based Vulnerability Analysis Framework from three primary components: the Policy-Based Vulnerability Hierarchy, Formal Implementation Vulnerability Model, and Characteristic-Based Vulnerability Classification Scheme. Each of these components provide a layer of formal foundation for our framework. We look to these components for insight, which influences our approach to vulnerability analysis.

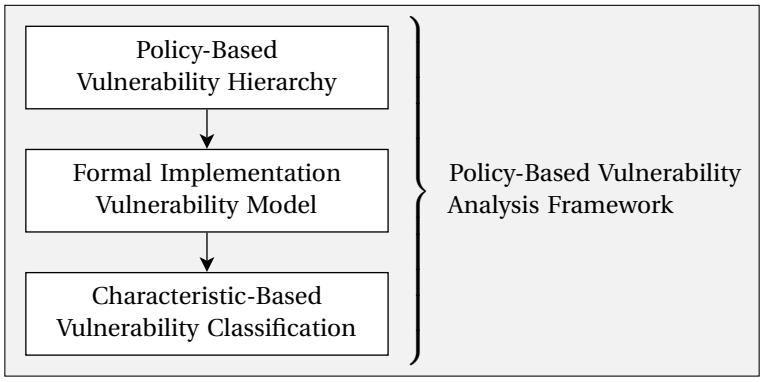
The Policy-Based Vulnerability Hierarchy defines a security policy at four levels of abstraction: ideal, feasible, configured, and instantiated. We use this policy hierarchy to create a hierarchy of vulnerabilities based on conflicts between these levels of policy. This results in three types of vulnerabilities: inherent, configuration, and implementation. At the lowest level of this hierarchy are implementation vulnerabilities, which occur when there is a policy conflict between the configured and instantiated levels of security policy.

We focus on implementation vulnerabilities with the Formal Implementation Vulnerability Model. This model formalizes the notions of a precondition and policy violation based on the computation trace of a deterministic universal Turing machine. We use this model to demonstrate that real-time security is decidable but likely intractable. These results suggest a shift in thinking, from detecting *if* a system is secure to detecting *when* a system is non-secure.

We pick up this line of reasoning with the Characteristic-Based Vulnerability Classification Scheme. We use the Formal Implementation Vulnerability Model as the formal foundation for characteristics and symptoms, which provide a more practical level of abstraction for implementation vulnerabilities. The characteristic-based approach allows us to determine which characteristics tend to lead to vulnerabilities, and detect when a system may be non-secure.

We use results from the Characteristic-Based Vulnerability Classification Scheme to inform how we perform implementation vulnerability analysis with the Policy-Based Vulnerability Analysis Framework. The framework is divided into three phases: preparation, analysis, and mitiga-

FIGURE 1.1: FRAMEWORK OVERVIEW



An overview of the Policy-Based Vulnerability Analysis Framework and its major components. The Policy-Based Vulnerability Hierarchy is introduced in chapter 4. The Formal Implementation Vulnerability Model and Characteristic-Based Vulnerability Classification Scheme are introduced in chapter 5, with additional details available in the appendices. The Policy-Based Vulnerability Analysis Framework itself is introduced in chapter 6.

tion. We define the scope and approximate the configured oracle in the preparation phase, and identify characteristics and vulnerabilities in the analysis phase. Finally, we determine if we are able to mitigate the discovered vulnerabilities by mitigating characteristics in the analysis phase. As a result, we are able to provide a framework for implementation vulnerability analysis that is capable of producing repeatable results in practice.

§1.5 Organization

This chapter discusses the motivation, scope, and objectives of this framework, as well as an overview of our approach. The remainder of this dissertation is organized as follows:

- Chapter 2: Background
- Chapter 3: Security Policy
- Chapter 4: Vulnerability Hierarchy
- Chapter 5: Vulnerability Classification
- Chapter 6: Vulnerability Analysis
- Chapter 7: Conclusion

We review security policy and Turing machines in chapter 2 to make our notation clear, as well as discuss prior and related work. We discuss our policy-based approach in chapter 3, and formally define the policy terminology used throughout this dissertation. Chapter 4 provides a policy-based hierarchy of vulnerabilities, and chapter 5 provides a characteristic-based classification scheme for the lowest level of vulnerabilities defined in this hierarchy. We combine

these components in chapter 6 to form the Policy-Based Vulnerability Analysis Framework. We summarize the framework and discuss how well we meet our objectives in chapter 7.

We also provide several appendices to discuss certain topics in more detail. This includes a detailed introduction of the Formal Implementation Vulnerability Model, which provides the theoretical foundation for the Characteristic-Based Vulnerability Classification Scheme. The appendices are organized as follows:

- Appendix A: Vulnerability Model
- Appendix B: Discussion

This dissertation introduces several new terms and notation. We provide a terminology and notation table at the end of most chapters, and a terminology index following the bibliography.

CHAPTER 2

Background

We examine related work in this chapter, including several foundational models and early iterations of the Policy-Based Vulnerability Analysis Framework. We then provide a general background on security policy and Turing machines to introduce the terminology and notation used throughout this dissertation.

§2.1 Foundations

Mathematical models of systems, security, and policy have existed since the early 1970s. While the term security policy does not appear in many of the earliest works, the concepts being discussed provide a foundation for how security policy is used today. We examine several of these foundational models† in this section.

§2.1.1 Access Control Matrix (ACM) Model

We start with the Access Control Matrix (ACM) model [LAM71], which is one of the earliest and most familiar models for expressing policy at the system level. The access control matrix model has evolved since its introduction in the early 1970s. We focus our discussion in this section on the variant by Graham and Denning [GRA71].

The model defines a protection system with three major components: a set of objects, a set of subjects, and a set of rules. The set of objects include any entity "to which access must be controlled," like files or memory. The set of objects includes the set of subjects, which are "active entit[ies]" capable of manipulating other objects. Subjects may be represented as a process and the domain or context that process operates. The rules govern how subjects may access objects based on an access control matrix and how the access control matrix changes.

† See [LAN81] or [BIS03A] for a more in depth overview of these models.

FIGURE 2.1: EXAMPLE ACCESS CONTROL MATRIX

	objects				
subjects	root	yasmin	zane	file1	file2
root	c	o	o	-	-
yasmin	-	c	-	o, r	r*, w
zane	-	-	c	r	o, r, w

c: control o: owner r: read w: write *: copy flag

An example access control matrix. The owner attribute allows subject root to remove subjects yasmin and zane from the matrix. The control attribute allows yasmin to read her attributes for file2. The copy flag, denoted by the * symbol, allows yasmin to transfer the read attribute for file2 to other subjects. The owner attribute allows zane to remove the write attribute for file2 from yasmin.

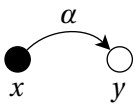
In our framework, we use a hierarchical approach rather than an access control matrix to capture a security policy. Our approach is still able to capture a security policy represented by an access control matrix, but also allows us to capture both system and non-system policy events at different levels of abstraction.

§2.1.2 Harrison-Ruzzo-Ullman (HRU) Model

The Harrison-Ruzzo-Ullman (HRU) Model [HAR76] is a formal protection system model based on the access control matrix. The HRU model defines a protection system to be a set of rights R and commands C, and the configuration of such a system to be the tuple (S, O, P) where S is the set of subjects, O is the set of objects such that $S \subseteq O$, and P is the access control matrix such that $P[s, o] \subseteq R$ for a subject $s \in S$ and object $o \in O$. The access control matrix P may only be manipulated by a fixed set of six primitive operations: enter/delete right r from $P[s, o]$, create/destroy subject s from P, and create/destroy object o from P. The set of commands C are built from these operations and may allow or deny operations based on the existing rights in the access control matrix.

The safety problem determines whether a subject may acquire new privileges for an object from a series of commands. The authors show that safety in the HRU Model is undecidable except in some special cases. Specifically, they show that no algorithm can decide whether a command $c \in C$ from an arbitrary protection system with configuration (S, O, P) adds a right $r \in R$ to an entry in P that did not already have that right. However, by restricting commands to a single operation, the safety problem becomes decidable—albeit intractable to actually solve in practice. These results provide insight into the undecidability of security itself. We know a system

FIGURE 2.2: TAKE-GRANT GRAPHICAL REPRESENTATION

Node Notation	Edge Notation
● subject ○ object ⊗ subject or object	

Graphical representation of subjects, objects, and access attributes in a Take-Grant protection system. The edge labeled α indicates subject x has set of access attributes α for object y. This is equivalent to stating $P[\, x, y \,] = \alpha$ where P is an access control matrix.

must be safe with respect to all rights in order to be secure† [BIS03A, *p*48]. Therefore, knowing when the safety problem is decidable informs whether it is possible to solve the broader problem of security in general.

Our formal model of implementation vulnerabilities has some similarities with the HRU model, but the models are fundamentally different. The HRU model provides a model of a protection system and proves whether safety in that system is decidable. We provide a formal model of policy, which we use to show whether security given that policy is decidable. This reflects the overall policy-based approach of our vulnerability analysis framework.

§2.1.3 Take-Grant Protection Model

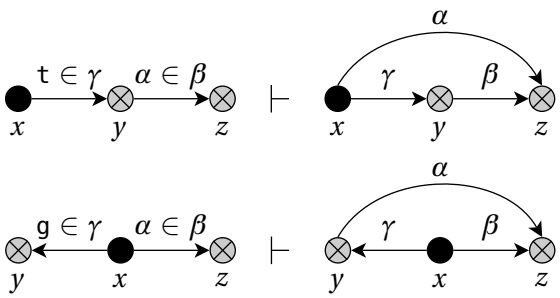
Instead of an access control matrix, the Take-Grant Protection Model uses a directed graph to model a protection system. The original take-grant system was introduced by Jones, Lipton, and Snyder [JON76] and has since been heavily used, studied, and extended [LAN81, BIS84]. For the purpose of this discussion, we focus on the model as presented by Bishop [BIS95, BIS03A].

The protection system is captured as a protection graph with nodes representing subjects and objects, and labeled edges representing access attributes (as illustrated in Figure 2.2). Subjects may modify the protection graph through a set of graph rewriting rules. The model's namesake comes from the two key rules `take` and `grant` (illustrated in Figure 2.3), which govern how rights may flow between nodes. Using techniques from graph theory, we are able to analyze the protection graph to determine if and when rights may be shared or stolen in the system.

However, these rules alone do not illustrate how information may flow in the system. Bishop and Snyder define two types of rules: *de jure* rules which govern the transfer of rights and *de facto*

† However, a safe system is not necessarily secure.

FIGURE 2.3: TAKE-GRANT RULES



Illustrates the `take` rule (top) and `grant` rule (bottom) [BIS95, *p*5]. A subject x may `take` attribute $\alpha \in \beta$ for node z from node y when x has the take attribute `t`. The `grant` rule operates similarly. The $\vdash$ symbol indicates the left graph is transformed into the right graph when the rule is applied.

rules which govern the transfer of information. These *de facto* rules illustrate how a subject may share or acquire information via the access attributes already present in the protection graph.

The Take-Grant protection system provides a powerful theoretical model for studying the safety problem. Recall from section 2.1.2 that "no algorithm can decide the safety of an arbitrary configuration of an arbitrary protection system" [HAR76]. However, the original Take-Grant results show that given a specific system with specific rules, the safety problem is decidable in linear time [SNY77]. This result illustrates that certain special cases of security may be decidable, which we explore further with the Formal Implementation Vulnerability Model. We discuss additional decidability results in section 2.3.2.

§2.2 Prior Work

This dissertation integrates and extends several pre-existing components to form the Policy-Based Vulnerability Analysis Framework. We discuss some of this prior work in this section.

§2.2.1 Unifying Policy Hierarchy

The Policy-Based Vulnerability Hierarchy is based on the "Unifying Policy Hierarchy" by Carlson [CAR06]. The Unifying Policy Hierarchy Model addresses ambiguous notions of security policy by defining security policy at four different levels of abstraction: the Oracle Policy, Feasible Oracle Policy, Configured Policy, and Actual Policy. The Oracle Policy (OP) is described as an "omnipresent policy" corresponding to the "desires of the policy makers." The Feasible Oracle Policy (FOP) is defined as representing "the will of policy makers" while taking into account

FIGURE 2.4: COMPARISON OF HIERARCHY TERMINOLOGY

Original	Current
Oracle Policy	Ideal Oracle
Feasible Oracle Policy	Feasible Oracle
Configured Policy	Configured Oracle
Actual Policy	Instantiated Oracle
Inherent Vulnerability	Inherent Vulnerability
Configuration Vulnerability	Configuration Vulnerability
Runtime Vulnerability	Implementation Vulnerability

Comparison of the original Unifying Policy Hierarchy terminology with that of our own. For example, we refer to the "Actual Policy" in the original hierarchy as the "Instantiated Oracle" in our framework.

"the mechanics and access controls of the particular system implementation." The Configured Policy (CP) represents "the access control settings of a system actually instantiated by an administrator." Finally, the Actual Policy (AP) represents "the policy in effect on a system at runtime." Carlson uses this policy hierarchy to define a hierarchy of vulnerabilities.

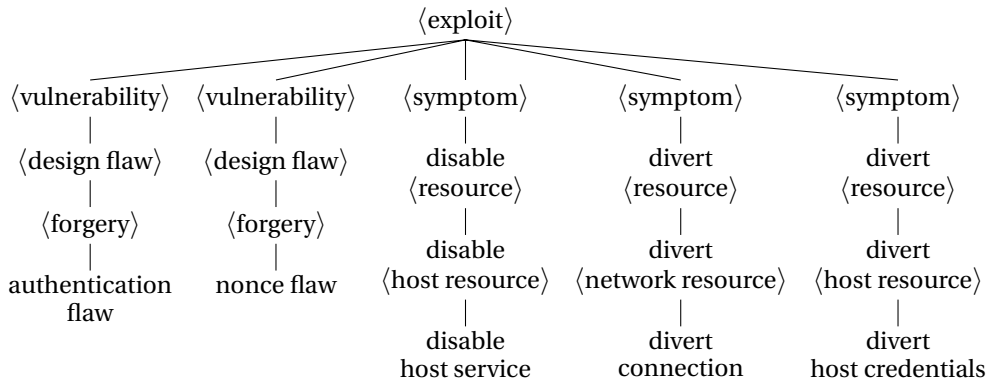
The hierarchy presented in this dissertation keeps in the same spirit as the original, but makes substantial alterations to the model. We summarize the terminology changes in Figure 2.4 and detail specific modifications in chapter 4.

§2.2.2 Vulnerability Analysis

The paper titled "Vulnerabilities Analysis" by Bishop [BIS99] provides the foundation for the Characteristic-Based Vulnerability Classification Scheme, and the motivation behind the entire Policy-Based Vulnerability Analysis Framework. This work compares the Research Into Secure Operating Systems (RISOS) [ABB76] and Neumann's organization [NEU78] of the Protection Analysis (PA) [BIS78] classification schemes, as well as Aslam's Taxonomy of Security Faults [ASL95]. This work illustrates how classification using these approaches varies depending on the point of view and level of abstraction, and proposes a characteristic-based vulnerability classification scheme with five specific properties to help alleviate these issues. We base our own classification objectives on these properties.

The classification scheme presented in chapter 5 is based on this work. A characteristic is originally defined in this work as "a condition that must hold for the vulnerability to exist." We

FIGURE 2.5: EXPLOIT CLASSIFICATION TREE



Classification of a VLAN trunking protocol exploit by Whalen et al. [WHA05]. The symptom tree captures a possible denial of service and man-in-the-middle attack.

formalize this notion with the Formal Implementation Vulnerability Model, and integrate this work with the hierarchical approach discussed in section 2.2.4.

§2.2.3 Buffer Overflow Characteristics

An initial set of characteristics for buffer overflow vulnerabilities is presented in the paper, "A Taxonomy of Buffer Overflow Preconditions" [BIS10]. Specifically, we identified four classes of buffer overflow vulnerabilities, each which result from a different set of characteristics. Additionally, we illustrate the connection between these characteristics and existing defenses against buffer overflow vulnerabilities. For example, StackGuard [COW98] focuses on detecting when the stored return address is countered, which is a characteristic of direct executable buffer overflow [BIS10, $p12$].

This work demonstrates how a characteristic-based approach illuminates possible defense vectors against vulnerabilities, as well as differentiates between a buffer overflow bug versus a buffer overflow vulnerability. We refine these original characteristics in using pseudo-code chapter 5, and develop a classification grammar for these characteristics.

§2.2.4 Protocol Exploit Classification

The paper "Protocol Vulnerability Analysis" by Whalen et al. extends the characteristic-based approach to protocol exploits using an Extended Backus-Naur Form (EBNF) grammar for classification, and introduces the notion of a symptoms to describe the policy violations that result from exploiting a vulnerability [WHA05]. The classification grammar provides a hierarchical approach to classification. The classification of individual exploits may then be visualized as trees,

illustrated in Figure 2.5, consisting of multiple vulnerability and symptom branches. Whalen et al. applied this approach to 24 different protocol exploits, most of which took advantage of protocol design flaws, misconfiguration, or authentication flaws. We generalize this hierarchical approach for our implementation vulnerability classification scheme.

§2.2.5 Insider Threat Analysis

We previously expanded the Unifying Policy Hierarchy to capture the insider problem [BIS08, BIS09B]. Specifically, the insider threat is defined as existing "whenever someone has more authorized privileges at a lower policy level than at a higher policy level." This mismatch captures a potential for misuse that can be leveraged by the insider. To do this, we expanded the ideal policy oracle to capture the intent of the subject. For example, at the ideal level, Yasmin is only authorized to access medical records to treat patients. However, at the feasible level, we are unable to capture Yasmin's intent. Therefore, she is able to access those medical records with the intent to sell that information to drug companies. We define the notion of conditional policy events in chapter 3 to capture this expansion of the policy hierarchy.

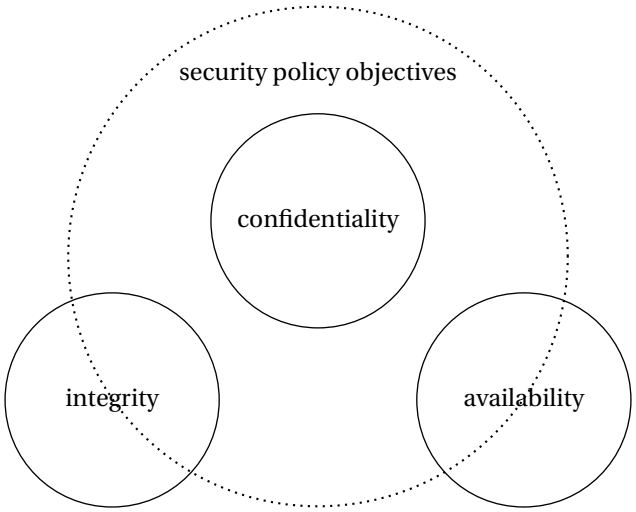
§2.2.6 Early Iterations

Early iterations of this work appears in numerous technical reports. The technical report "Tree Approach to Vulnerability Classification" introduces our first attempt at formalizing classification trees for characteristics and symptoms [ENG06A]. We followed up on this work with the technical report "A Practical Formalism for Vulnerability Comparison," in which we attempt to formalize the notion of a characteristic and vulnerability equivalence class, and introduce the perfect knowledge assumption for the first time [ENG06B]. We refined these notions further in the technical report "A Model for Vulnerability Analysis and Classification" [ENG08A]. We formalize the underlying notions of a precondition and policy violation in the technical report "Modeling Computer Insecurity," which had been left as primitives in previous technical reports.

§2.3 Related Work

This dissertation integrates security policy, vulnerability classification, and vulnerability analysis into a single framework. We discuss some relevant work in each of these areas next.

FIGURE 2.6: SECURITY POLICY OBJECTIVES



Sterne's depiction of the scope of *security policy objectives* as a combination of confidentiality, integrity, and availability (or assured service) concerns [STE91, Figure 2].

§2.3.1 Security Policy

The Policy-Based Vulnerability Hierarchy defines a security policy at multiple levels of abstraction. There are several other models that take a multi-level approach to defining security policy. We highlight some of these security policy models in this section.

Goguen and Meseguer define a security policy as "a set of noninterference assertions" and define two types of policy: static and dynamic [GOG82]. A static security policy includes normal noninterference assertions, whereas a dynamic security policy includes conditional noninterference assertions. Both types of security policies may be captured by the configured level of security policy in the Policy-Based Vulnerability Hierarchy. The security verification problem they present is equivalent to determining if there are any implementation vulnerabilities caused by the gap between the configured and instantiated levels of the hierarchy. They also use a formalization similar to the Formal Implementation Vulnerability Model introduced in Appendix A.

Sterne observes that although security policy is "fundamental to computer security," the difference between a security requirement versus any other critical system requirement is unclear [STE91]. Instead of relying on confidentiality, integrity, and availability to distinguish security policy, he defines the notion of a security policy objective as "a statement of intent to protect an identified resource from unauthorized use." This is closely related to our notion of a policy statement, which states what is authorized or unauthorized for a particular resource. Sterne

then defines two levels of security policy, organizational and automated, and illustrates the gap between them. The organizational security policy (OSP) is the set of "laws, rules, and practices" which specifies how a security policy objective "is to be manifested in the routine activities of the organization." In contrast, the automated security policy (ASP) "specifies what a trusted system is trusted to do." The OSP is roughly equivalent to the ideal and feasible policy levels, and the ASP corresponds to the configured policy level.

Bishop and Peisert examine the difference between a site's stated security policy, and the security policy enforced by the security mechanisms on the system and network [BIS06]. They show how to detect configuration errors between the stated and enforced security policies using an intermediate policy representation and policy discovery. This work differentiates between intended and implemented security policy, but only at the middle levels of the hierarchy. We can equate this work to configuration vulnerabilities in the Policy-Based Vulnerability Hierarchy, which occur in the gap between the feasible and configured policy levels.

Jajodia et al. define an Authorization Specification Language (ASL) that is computable in polynomial time [JAJ97A]. In this work, they define an authorization policy as "a mapping that maps 4-tuples (o, u, R, a) consisting of an object, user, role set, and action, respectively to the set $\{$ `authorized`, `denied` $\}$." An authorization policy is equivalent to the configured level of security policy in the hierarchy. The formalization of an authorization policy as a tuple is also nearly identical to our notion of a policy event and policy decision.

Schneider provides a predicate-based formalization of security policy, and illustrates that not all polices are enforceable [SCH00]. Specifically, he defines a class of Execution Monitoring (EM) enforceable security policies, and shows that if a policy is not limited to safety properties, then the policy is not EM-enforceable. Hamlen et al. expand on this initial work by examining the class of policies that may be enforced by program rewriting (RW), and provide a taxonomy of enforceable policies [HAM06]. They demonstrate that the class of RW-enforceable policies are a superset of EM-enforceable policies, and may contain some policies with non-safety properties. Ligatti et al. examine a more powerful class of program monitors, called edit automata, that may insert or suppress actions [LIG09]. They use a different set of assumptions than that of Hamel et al. regarding what information their program monitors may access, but demonstrate that these monitors are able to enforce "infinite renewal properties" that include certain non-

FIGURE 2.7: SECURITY POLICY MODELS

Policy Hierarchy	Related Policy Models
Ideal	OSP [STE91]
Feasible	Stated [BIS06], EM-Enforceable [SCH00], RW-Enforceable [HAM06]
Configured	Static/Dynamic [GOG82], ASP [STE91], Enforced [BIS06], ASL [JAJ97A]
Instantiated	N/A

Comparison of the Policy-Based Vulnerability Hierarchy with related security policy models. The Policy-Based Vulnerability Hierarchy captures a broader range of abstraction than most models of security policy.

safety properties. These results on enforceable security policies illustrate the complex gap between the ideal and feasible policy oracles in the Policy-Based Vulnerability Hierarchy. There are also similarities between the formalizations used by Hamlen et al. to the Formal Implementation Vulnerability Model, which we discuss in section 2.3.2.

The attack surface work by Howard et al. also observes that vulnerabilities occur at different levels of a system, and similarly defines a vulnerability as "the difference in behaviors" between these levels [HOW05]. The focus of the attack surface work is not on security policy or identifying system level vulnerabilities, but determining the attackability of a system.

§2.3.2 Theoretical Results

The Formal Implementation Vulnerability Model provides a new approach to formally defining security policy at the state machine level, and allows us to formally define implementation vulnerabilities. We use Turing machines as a foundation in this model, which allows us to also explore the decidability of various security problems. We highlight some of related decidability results and theoretical models in this section.

The foundational models introduced in section 2.1 provide a specific protection system with a predefined security policy, and determine whether security in that context is decidable. The Formal Implementation Vulnerability Model introduced in Appendix A instead attempts to define security and a security policy for a pre-existing theoretical model—a deterministic universal Turing machine. Our work also differs from these models in purpose. We primarily seek to provide a formalization of implementation vulnerabilities.

Goguen and Meseguer introduce the notion of noninterference assertions to describe security policy [GOG82]. Rushby summarizes the concept of noninterference as follows: "a security domain u is noninterfering with a domain v if no action performed by u can influence subsequent outputs as seen by v" [RUS92]. The work by Goguen and Meseguer shares a goal similar to our own. They use a state machine-based system model, examine how to express a security policy for that model, and discuss how to verify a machine satisfies that security policy. However, like other foundational work, they define a specific type of capability system comprising of a state of users, states, state commands, outputs, capability tables, an output function, a state transition function, and a capability transition function. Noninterference has been widely studied since its introduction by Goguen and Meseguer.† For example, D'Souza et al. demonstrate that model-checking information flow properties, such as noninterference, is undecidable for pushdown system models [D'S08].

Formal models are "not a panacea" for security [DEN99]. Our focus is on providing a formal model, not applying formal methods. However, there may be several areas where formal methods can help improve security [WIN98]. We focus on two of these: verification and model checking. Formal verification is the process of proving a system satisfies its formal specification [MEE05], and is able to eliminate errors in implementation. For example, Klein et al. formally verified the implementation of seL4, a general purpose operating system microkernel for embedded systems [KLE09]. However, it is possible for the formal specification itself to be incorrect. Formal verification is also resource intensive; the verification of the seL4 microkernel took over 20 person years.

Model checking is generally less complete than formal verification. Instead of testing entire implementations, model checking is used to test whether specific properties hold in a system. Wing describes several older model checking results [WIN98]. More recent results include the model checking results by Chen et al. and later Schwarz et al. on the Linux distribution for temporal safety properties [CHE04, SCH05]. While the scope of model checking is often smaller, it is usually less resource intensive than formal specification and verification. For example, model checking the entire Linux distribution took just over 150 person hours.

† For an introduction to some of this work, see [RUS92].

Fithen et al. observe that the concept of a vulnerability is ill-defined, and provide a formal model of a vulnerability that is able to uncover relationships between vulnerabilities [FIT04]. At a high level, their goal and approach is similar to ours. For example, they define a vulnerability as, "an unplanned system feature that an intruder may exploit, if he/she can establish certain preconditions, to achieve particular impacts on that system that violate its security policy." Our definition is similar, also relying on preconditions and security policy violations. At a lower level, our approaches differ significantly. They use an open source rule-based production system for implementing their model using high-level propositional logic. For example:

```
(software (name internet-explorer) (version 4.01))
```

While this is valuable for the administration of systems in practice, this approach captures *what* is vulnerable—not *why* it is vulnerable. Our notion of preconditions captures the underlying causes associated with a vulnerability, and operates at an entirely different level of abstraction.

Zhang introduces a general policy-based security model for information flow in nondeterministic systems [ZHA97]. Our notions of security are nearly identical. Zhang defines security based on the possible executions of the system, whereas we define security based on the possible configurations of a system. Our underlying system models and approaches are different. For example, Zhang models a generic nondeterministic system, whereas we model deterministic universal Turing machines.

Perhaps most similar to our formalization is the work by Hamlen et al. [HAM06]. They use program machines, which are multi-tape deterministic Turing machines, to model untrusted programs and specify a fixed set of observable events based on a "trace" of the Turing machine. They also prove a complex relationship between the classes of decidable, co-recursively enumerable, EM-enforceable, and RW-enforceable security policies. There are subtle differences between our models. For example, our formalizations of a policy event differ. The largest difference lies in the focus and application of these models. The focus of the Formal Implementation Vulnerability Model is to provide a formal foundation for the classification of implementation vulnerabilities, whereas this work provides a taxonomy of security policies.

Most decidability results focus on the safety problem, which is an important component of security. Li and Tripunitara examine the safety problem in the Graham-Denning discretionary access control (DAC) scheme, instead of the Harrison-Ruzzo-Ullman (HRU) scheme [LI05]. They

find that safety in this context is decidable in cubic running time. Kleiner and Newcomb revisit the safety problem in a protection matrix model [KLE07]. They formulate the safety problem by introducing Safety Access Temporal Logic (SATL) to express safety properties, and show that model checking for this logic is undecidable. Like other work on the safety problem, they are able to show a special-case of the problem is decidable when limiting the logic to a special subset called Universal SATL.

§2.3.3 Vulnerability Classification

The classification scheme in chapter 5 is based on the prior work by Bishop [BIS99] and Whalen et al. [WHA05]. Numerous other classification schemes exist, differing from ours in perspective or approach. We highlight some of these classification schemes in this section.

Early classification work tends to classify bugs, errors, faults, or flaws instead of actual vulnerabilities. For example, the Research Into Secure Operating Systems (RISOS) classification system provides seven classes of security flaws [ABB76], and Aslam provides a detailed UNIX security fault taxonomy [ASL95]. These classification schemes capture a wide array of flaws and faults observed in practice, but classification using these approaches may be inconsistent due to ambiguity or varying points of view [BIS99].

Our characteristic-based classification approach is similar to the Protection Analysis (PA) error classification system, as the notion of a "raw error pattern" is similar to our notion of a characteristic [BIS78, NEU78]. This work also uses a layered approach, starting with errors to derive raw error patterns, generalized patterns, and finally major error types. However, the original categories of this work exist across multiple levels of abstraction, and belong to multiple global categories. The attribute categorization work of Ostrand and Weyuker allows for development of new characteristics similar to our classification system as well [OST84], but requires further development to be widely applicable.

We classify vulnerabilities based on characteristics, which capture the conditions necessary for the vulnerability to exist. There are numerous classification schemes that use different criteria than ours. In his thesis, Krsul presents a formal vulnerability classification scheme based on assumptions made by programmers [KRS98]. Landwehr provides a classification scheme that focuses the genesis, time of introduction, and location of vulnerabilities [LAN94]. Howard provides

a taxonomy that focuses on computer incidents, and uses events, actions, targets, and attacks for classification [HOW97]. Cohen also provides a classification scheme based on attack and defense [COH97A, COH97B]. In his classification schemes, he presents an extensive list of 94 attack categories, 140 defense categories, and a list of 6 properties of attack and defense.

Some classification schemes have specific goals or a narrow scope. Endres presented a classification scheme from analyzing errors from a specific subset of programs [END75]. The focus of the work was to determine what meaningful conclusions may be drawn from an analysis of errors, and is not immediately extensible as a general vulnerability classification scheme. DeMillo and Mathur present a grammar-based classification scheme of faults [DEM95]. They provide algorithms for automated fault classification specific to problems found in TEX. Weber presents a taxonomy of computer intrusions that focuses on creating good classes for evaluating intrusion detection systems [WEB98].

§2.3.4 Vulnerability Analysis

The vulnerability analysis framework in chapter 6 builds off the characteristic-based analysis approach by Bishop [BIS99]. We discuss some related vulnerability analysis frameworks in this section, many of which differ from ours in purpose or approach.

An important component of vulnerability analysis in practice is penetration testing, which is a "non-rigorous technique for checking the security of a system" [BIS03A, *p*660]. For example, consider the Flaw Hypothesis Methodology (FHM) for penetration testing introduced by Linde [LIN75]. The process is divided into four stages: knowledge generation, flaw hypothesis generation, flaw hypothesis confirmation, and flaw generalization. Later iterations add a flaw elimination stage to this process [WEI95]. During the first phase, analysis attempt to understand how users interact with the system from both an abstract and implementation level perspective. The second phase uses this information to form "flaw hypothesis" of suspected weaknesses in the system. These hypothesis are tested in the third phase using thought experiments or live tests. In the fourth phase, analysts generalize the findings and categorize the discovered weaknesses. The last phase attempts to eliminate the discovered flaws. At an abstract level, our three stage analysis framework is similar to the Flaw Hypothesis Methodology, but how we perform hypothesis generation and mitigation differs significantly.

There are numerous other approaches for penetration testing, including the Open Source Security Testing Methodology Manual (OSSTMM) [HER06] and the Information Systems Security Assessment Framework (ISSAF) [OPE06]. Penetration testing is effective at discovering flaws or weaknesses, but the process often relies on the background experiences of the analysts. As a result, penetration tests performed by different teams may produce very different results. We attempt to add repeatability to the analysis process by adopting a characteristic-based approach.

The early analysis work by Ramakrishnan and Sekar uses a high-level specification language to model system components for vulnerability analysis [RAM98]. This approach allows for the use of formal methods to detect vulnerabilities. The analysis work by Jha et al. also takes a formal approach by automatically creating attack graphs and performing model checking on these graphs [JHA02]. While we begin with a formal model, we do not focus on developing a specification language or using formal methods in our analysis framework. Instead, we attempt to abstract our formal models for use in practical settings.

The assessment work by Jøsang et al. introduces a different approach to vulnerability analysis, focusing on the tradeoffs between theoretical security and the usability of systems [JØ07]. While the focus of our implementation vulnerability analysis framework is not on usability, several of the principles introduced in this work may be integrated into a high-level analysis using the Policy-Based Vulnerability Hierarchy.

The analysis approach by Abedin et al. focuses on the evaluation and comparison of security policies by analyzing a policy against known vulnerabilities. Whereas we use security policies for vulnerability analysis, their work focuses on the use of vulnerability analysis to improve security policies. Under our framework, their approach is similar to comparing the number of implementation vulnerabilities between different configured policies.

The work by Aven introduces a framework for quantitative risk analysis for both safety and security [AVE07]. The process is broken into an eight step process, including the identification of observable quantities, threats, hazards, and opportunities, an uncertainty and consequence analysis, and the description and evaluation of risks and vulnerabilities. This analysis relies on observable quantities and the ability to associate probabilities with events. In this dissertation, we focus on a scenario for vulnerability analysis where these types of measures are largely unknown and difficult to measure.

§2.4 Terminology

When possible, we use standard terminology and notation. However, there are multiple accepted definitions for some terms. For example, there are numerous (yet equivalent) formal definitions of a Turing machine used throughout computer science literature. As a result, we introduce the basic terminology and notation used in this dissertation to avoid confusion.

§2.4.1 Security Policy

Our terminology is consistent with that of Bishop [BIS03A, *p*95–111]. We informally define a **security policy** (or **policy** for short) as "a statement of what is, and what is not, allowed" and a **security mechanism** as a "method, tool, or procedure for enforcing a security policy." In general, a **policy language** provides a consistent way to represent a security policy. A **high-level policy language** is used to represent security policies independent of any underlying security mechanisms, whereas a **low-level policy language** is specific to the underlying mechanisms. However, for our framework, we are less concerned with *how* the security policy is represented than with *what* the policy represents.

We adapt Carlson's concepts of a policy event and policy decision for our work [CAR06]. Informally, a **policy event** is any event "of concern" to the policy makers. This may include any event which affects the confidentiality, integrity, or availability of a resource. For example, whether Yasmin is able to read a password file is an event that policy makers may want to control. However, whether Yasmin eats breakfast may not be a security concern. Informally, a **policy decision** indicates if a policy event is allowed or disallowed. Together, a policy event and policy decision represent a **policy statement**. For example, an informal policy statement may be: "Yasmin is not allowed to read password files." This captures both the policy event, Yasmin reading a password file, and the policy decision to disallow the event. A **policy violation** occurs whenever a policy statement is violated. This happens whenever a security mechanism enables a disallowed policy event or prevents an allowed policy event. In our example, Yasmin being allowed to read the password file represents a policy violation.

We also occasionally refer to open versus closed security policies [JAJ97B]. An **open policy** assumes all policy events are allowed except for a specific set of disallowed policy events. Al-

ternatively, a **closed policy** assumes all events are disallowed except for a specific set of allowed policy events. A security policy which specifies both allowed and disallowed events is a **hybrid** or **mixed policy**. Hybrid policies are more prone to conflicts, which occurs when an event is both allowed and disallowed by the security policy.

§2.4.2 Policy Properties

Ambiguity and implicit assumptions are often an issue when discussing security policy. Even if the policy itself is explicit, there is often an implicit assumption that the policy being discussed is appropriate for the environment. While this may be a fair assumption to make, doing so ambiguously or implicitly may lead to complications. For example, it may be unclear what makes a policy desirable in one context and not in another. It may also be unclear how this affects the discussion if the security policy does not satisfy this requirement.

We do not attempt to define what makes a "desirable" policy, but we do make some explicit assumptions about different properties that security policies have. Specifically, we rely on assumptions of preciseness, completeness, and correctness of a security policy when introducing our policy hierarchy in chapter 4.

We consider a security policy to be **precise** when it has the level of granularity necessary to express all policy events of interest. For example, the default security levels of high, medium-high, and medium in Internet Explorer may provide a security policy which is too coarse for certain users. While a security policy may be more precise than necessary, the unnecessary precision generally increases the complexity of enforcing and managing the policy in practice.

We say a security policy is **complete** when it includes a policy decision for every possible policy event. Security policies which include a default response, such as open or closed policies, may be considered complete. However, some high-level hybrid security policies may unintentionally omit whether uncommon policy events are authorized. It may be unclear whether to allow or disallow the event, leading to an inconsistent policy response.

Finally, we consider a security policy to be **correct** when it accurately captures the intent of the policy maker. For example, if a security mechanism is misconfigured, the resulting security policy as configured is incorrect. A security policy must be both precise and complete to be considered correct. Recall that a policy statement is a policy event and the associated policy

decision. A precise policy provides all of the policy events of interest to the policy makers, and a complete policy provides policy decisions for all of those policy events. Both are necessary to form correct policy statements. Otherwise, there will exist undefined policy events or unknown policy responses, making it impossible to form a correct security policy.

However, it is possible for a policy to be complete, but imprecise and incorrect. For example, the statement "everyone is allowed everything" is a trivial security policy. This security policy captures all policy events, and hence is complete. However, it does not allow for fine-grained control, making it imprecise. And, for most environments, this security policy is incorrect.

While ideally a security policy is precise, complete, and correct, in practice security policies are often imprecise, incomplete, and incorrect. This disconnect between the ideal and the practical is only one of many when it comes to security policy. We explore these gaps between different levels of policy in chapter 4.

§2.4.3 Turing Machines

Turing machines provide an important theoretical foundation for our framework. Turing machines and their variants are considered the most powerful theoretical models of computation, and are sometimes used to model the computational capability of modern systems. The definitions and notation presented in this section are consistent with those of Sipser [SIP97].

A Turing machine consists of three components: a control, an infinite tape, and a tape head. Based on an input string, the control moves the tape head left or right, reading and writing to the tape as necessary. Formally, a **Turing machine** is defined as a 7-tuple:

$$M = (\, Q,\, \Sigma,\, \Gamma,\, \delta,\, q_0,\, q_\mathrm{a},\, q_\mathrm{r}\,)$$

The set Q is the set of states, including the start state $q_0 \in Q$, the accept state $q_\mathrm{a} \in Q$, and the reject state $q_\mathrm{r} \in Q$ such that $q_\mathrm{a} \neq q_\mathrm{r}$. The Turing machine M uses an input alphabet Σ, which is a subset of the tape (or output) alphabet Γ. When M is given an input string $w \in \Sigma^*$, the start of the tape is initialized to the input string and completely blank everywhere else. The blank symbol $\sqcup \in \Gamma$ indicates the unused spaces on the tape, and may not appear in the input alphabet. The transition function δ specifies how the machine operates, and is defined as:

$$\delta \,:\, Q \times \Gamma \to Q \times \Gamma \times \{\, \mathrm{L},\, \mathrm{R}\,\}$$

The set $\{ \text{L}, \text{R} \}$ indicate whether the tape head moves left or right respectively. For example, consider the transition:

$$\delta(q_i, x) = (q_j, y, \text{L})$$

This indicates that when the machine is in state $q_i \in Q$ and reaches symbol $x \in \Gamma$, it writes symbol $y \in \Gamma$ on the current location of the tape, moves to state $q_j \in Q$, and moves the tape head left. The transition function is often too complex to fully specify. However, the **Church-Turing thesis** equates the notion of algorithms with Turing machines [Sip97, $p143$]. This allows us to use algorithms instead of the full formal specification to describe Turing machines.

A configuration of a Turing machine captures the current state, the contents of the tape, and the current location of the head. Formally, a **configuration**† is the string uqv where $q \in Q$ is the current state, the string $uv \in \Gamma^*$ gives the current contents of the tape, and the first position of v gives the current location of the tape head. The **start configuration** of M on input string w is the configuration $q_0 w$. Any configuration of the form $uq_{\text{a}}v$ is considered an **accepting configuration**, and configurations of the form $uq_{\text{r}}v$ are **rejecting configurations**. Since the Turing machine halts upon entering an accept or reject state, accepting and rejecting configurations are considered **halting configurations**. A Turing machine M *accepts* input w when it enters an accepting configuration, and the **language of M**, denoted as $L(M)$, is the set of all strings that M accepts.

There are several Turing machine variants. For example, an **enumerator** is a Turing machine variant that outputs accepted strings, and may or may not halt. A **computable function** f is another variant of a Turing machine that halts with just $f(w)$ on the tape. If the Turing machine *always* halts on any input w, it is considered a **decider** or **total** Turing machine. Finally, a **universal Turing machine** is a Turing machine which is capable of simulating any other Turing machine. We use the notation $\langle M \rangle$ to denote the string encoding of M such that it can be given as input to another Turing machine.

§2.4.4 Computability

We are able to determine whether a problem is solvable by studying whether the underlying language is "computable" with a Turing machine. We are specifically interested in two classes

† This is sometimes referred to as the **instantaneous description** (or ID) of the Turing machine.

of languages defined by computability theory: recursively enumerable and decidable. A **recursively enumerable** language includes any language that may be recognized by a Turing machine. We are able to determine if a string *does* belong to a recursively enumerable language, but may not be able to determine if a string *does not* belong. If the complement of the language is recursively enumerable, we call it **co-recursively enumerable**. For example, stating a language A is co-recursively enumerable indicates that the complement of A, denoted $\overline{A}$, is recursively enumerable.

A **decidable** language includes any language for which a decider Turing machine exists. We are always able to determine whether or not a string belongs to a decidable language. A language is decidable if and only if it is both recursively enumerable and co-recursively enumerable [SIP97, $p167$]. The complement to decidable languages is the class of **undecidable** languages. We may be unable to determine whether a string does or does not belong to an undecidable language. Any problem associated with an undecidable language is considered unsolvable.

An important tool to determining the decidability of a language is mapping reducibility. A language A is considered **mapping reducible** to language B, written $A \leq_{\mathrm{m}} B$, if there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$, where for every w:

$$w \in A \iff f(w) \in B$$

We use mapping reductions and their associated theorems to demonstrate the decidability of a languages defined by the Formal Implementation Vulnerability Model.

§2.4.5 Complexity

Determining whether languages are undecidable gives us important insight to which problems are unsolvable. However, even if a problem is decidable, it may not be practical to solve on a modern system. As such, we use techniques from both computability theory and complexity theory when examining our theoretical models. Specifically, we also consider whether a language belongs to the class P or NP.

Languages which belong to the **class *P*** may be *decided* in polynomial-time, and are associated with problems that are "realistically solvable" on modern systems [SIP97, 236]. Languages which belong to the **class *NP*** may be *verified* in polynomial-time but may not be realistic to solve. We are specifically concerned with languages which are considered ***NP*-complete**. The

complexity of such languages is tied to the entire class *NP* and whether $P = NP$, which is an open question. Since there are no known methods for solving an *NP*-complete problem in polynomial time, we consider such problems **impractical** or **intractable**. However, even if an optimal solution is intractable, there may exist an **approximation algorithm** for the problem which is able to provide a near-optimal solution in polynomial-time [COR03, $p1022$].

§2.4.6 Computation Traces

The **computation history** of a Turing machine M is defined as a finite sequence of configurations $h_0,\ h_1,\ \ldots,\ h_k$ that M enters when computing w, starting with the configuration $h_0 = q_0 w$ and ending with a halting configuration h_k. A deterministic decider Turing machine M has exactly one computation history for every input w [SIP97, $p177$].

There is no computation history if a Turing machine never halts. However, we are interested in all of the configurations a Turing machine enters during computation, whether or not it eventually halts. We introduce the notion of a Turing machine **computation trace** to track all of the configurations entered by a Turing machine on an input. If the Turing machine halts on the input, the trace is equivalent to the computation history. However, if the machine does not halt, the trace tracks the potentially infinite set of configurations entered by that machine. We demonstrate in Appendix A that the language of all configurations in the computation trace, denoted ***TRACE*(*M*, *w*)**, is recursively enumerable and undecidable.

In situations where we need a decidable language of configurations, we define the notion of a **partial trace**. A partial trace only tracks the computation of a Turing machine for a fixed number of n steps, providing a snapshot of the full trace. If M halts in input w within n steps, the partial trace ends with the halting configuration and is equivalent to the computation history. The language ***PARTIAL*(*M*, *w*, *n*)** of configurations in a partial trace is finite, and all finite languages are decidable.

The trace and partial trace capture the configurations a Turing machine may enter on a single input. We now turn our attention to the language of all configurations a Turing machine may enter when computing any input. Specifically, we define the notion of a **valid configuration** as any configuration entered by a Turing machine M during computation. The language ***VALID*(*M*)** of all valid configurations is recursively enumerable and undecidable.

§2.4.7 Set Notation

We use standard set notation and terminology, which we assume the reader is familiar with. However, some set operations have multiple valid notations. We specify the notation we use in this section, and provide some basic definitions for less common set operations.

We use the notation $A \setminus B$ instead of the notation $A - B$ to denote set difference. We use the notation $A \ominus B$ to denote the **symmetric difference** of two sets, similar to the exclusive-or operation $A \oplus B$ in logic. The symmetric difference is defined as:

$$\begin{aligned} A \ominus B &= (A \setminus B) \cup (B \setminus A) \\ &= (A \cup B) \setminus (A \cap B) \end{aligned}$$

For a finite set of sets $A = \{ A_1, A_2, \ldots, A_n \}$ where each $A_i \in A$ is a finite set of elements, we use the following notation for union and intersection [Ros07, $p127$]:

$$\bigcup A = \bigcup_{i=1}^{n} A_i = A_1 \cup A_2 \cup \cdots \cup A_n$$

$$\bigcap A = \bigcap_{i=1}^{n} A_i = A_1 \cap A_2 \cap \cdots \cap A_n$$

We frequently use **set-builder notation** to define the elements of an infinite set. Specifically, the notation $\{ x : \mathcal{P}(x) \}$ refers to the unique set of all elements x such that the property $\mathcal{P}(x)$ is true [Vau01, $p7$]. For example, the set of positive integers in set-builder notation is:

$$\{ x \in \mathbb{Z} : x > 0 \}$$

We also reference the minimal set cover problem [Cor03, $p1033$]. For a finite set A, we say the set of subsets B **covers** A when:

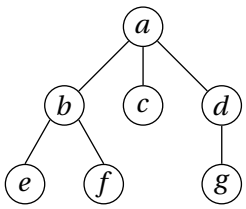
$$\bigcup B = A$$

For example, let $A = \{ a, b, c \}$ and $B = \{ \{ a \}, \{ b, c \}, \{ c \} \}$. By taking the union of all the subsets in B, we get the following:

$$\bigcup B = \{ a \} \cup \{ b, c \} \cup \{ c \} = \{ a, b, c \} = A$$

Notice, however, that B is not a minimal set cover of A. For a finite set A, we say that the set of subsets B is a **minimal set cover** of A if removing any element of B breaks the set cover. In our above example, the set $C = \{ \{ a \}, \{ b, c \} \}$ is a minimal set cover of A. The problem of finding a minimal set cover, known as the **set covering problem**, is *NP*-complete [Kar72].

FIGURE 2.8: TREE TERMINOLOGY AND NOTATION



$$G = (N, E) \textit{ where}$$
$$N = \{ a, b, c, d, e, f, g \}$$
$$E = \{ (a,b), (a,c), (a,d) \\ (b,e), (b,f), (d,g) \}$$

Example tree $G = (N, E)$. Some observations: Each node has a unique label. Node a is the root node. Nodes a, b, and d are internal nodes. Nodes c, e, f, and g are leaf nodes. Node d is a parent node to node g, and node g is a child node to d. Nodes e and f share the same parent node b and are considered siblings.

§2.4.8 Graph Notation

We use standard graph notation throughout this dissertation. Informally, a **graph** is a set of nodes connected together by edges. More formally, a graph G is defined as the pair (N, E) where N is the set of nodes and E is the set of edges in G such that a pair $(a, b) \in E$ indicates node $a \in N$ is connected to node $b \in N$. The union of two graphs G_1 and G_2 is defined as:

$$G_1 \cup G_2 = (N_1 \cup N_2, E_1 \cup E_2)$$

We rely on trees, a special type of graph, to provide a hierarchical classification structure for vulnerabilities. A **tree** is a connected, undirected, acyclic graph. A tree, unless otherwise specified, contains a special **root node** which has no parents. A connected node which is closer to the root is a **parent node**, and a connected node which is further from the root is a **child node**. Nodes which have no children are **leaf nodes**, and all other nodes are **internal nodes**. Nodes which share the same parent are **siblings**. See Figure 2.8 for a visual representation of these terms. We use **unordered trees** in our work, which indicates the order of child nodes is not significant. Furthermore, we use **labeled nodes**, which indicates each node has a unique label. Specifically, if some node a has the same label as node b, then a and b refer to the same node.

We also refer to the **graph of a function**, which for a function f, is defined as the set of all pairs $(x, f(x))$ [ROS07, $p142$]. This is not to be confused with the concept of a graph defined above, which is a set of nodes and the edges connecting them.

CHAPTER 3

Security Policy

Security policy plays a fundamental role in this dissertation. In this chapter, we motivate our policy-based approach and provide formal definitions for policy-related terminology.

§3.1 Introduction

Information and computer security focuses on protecting information and computer resources from misuse. We generally assume the misuse of these resources results in a violation of their confidentiality, integrity, or availability. However, exactly what constitutes misuse depends on the resource and environment. For these specifics, we turn to the security policy.

A security policy specifies what is considered misuse by providing "a statement of what is, and what is not, allowed" for a specific resource and environment [BIS03A, *p*9]. The method of expression and level of abstraction at which security policies are expressed varies greatly, and may not be explicitly defined at all. This inconsistency in expression is further aggravated by the separation of a security policy and the security mechanisms that enforce that policy.

When a security policy is entirely unspecified, most general discussions of security assume an implicit policy that focuses on protecting the confidentiality, integrity, and availability of a resource. However, the explicit expression of a security policy makes clear what is considered "misuse" and exposes gaps between what we want in an ideal sense, and what exists in a practical sense—providing powerful insight into where and why vulnerabilities occur. After all, it is a security policy (implicit or explicit) that distinguishes vulnerabilities from software bugs. Both cause a computer system to exhibit unintended behavior, but only vulnerabilities cause the system to exhibit unintended behavior with security-related consequences.

For this reason, security policies provide the foundation for how we define, understand, classify, and analyze vulnerabilities. We explain our policy-based approach in this chapter, which

provides a foundation for the entire Policy-Based Vulnerability Analysis Framework. We also provide formal definitions for the policy terminology introduced in chapter 2, which we use throughout the remainder of this dissertation.

§3.2 Policy-Based Approach

We take a hierarchical policy-based approach that separates intention from implementation in this dissertation. We also take a different approach for formalizing policy at the state-machine level, opting to define policy as a language of configurations instead of as a partition of states. We motivate each of these choices next, before introducing our framework in the following chapters.

§3.2.1 Levels of Security Policy

As pointed out by Andrews and Whittaker, "[s]ecurity has guiding tenets, but context is everything" [AND04]. Depending on the context, the scope and nature of a security policy varies. Various laws and practices that regulate the actions of people influence organizational level security policy [STE91]. This policy, in turn, influences the system level security policy, which regulates the actions of user accounts and processes. However, a security policy at this level is limited by the specific security mechanisms in place. Two different systems may fall under the same organizational security policy, but have different system security policies.

As a result, the focus is usually on a security policy at a system-level. However, this obscures the overall security of the resource. Consider a system placed in an unlocked room. The user accounts on the system may be unable to delete a backup file, but anyone may enter the room and destroy the physical hard drive. This system may be secure with respect to the system security policy, but vulnerable with respect to the organizational security policy. The end result is a vulnerable system, which is obscured if only the system security policy is considered.

Alternatively, consider a vulnerable system placed in a locked room with no Internet connection. For example, the election software for counting votes is often installed on a stand-alone commercial off-the-shelf (COTS) system located at the election headquarters. The underlying COTS system may have vulnerabilities, but vulnerabilities in the election software cannot be exploited remotely due to the physical isolation of the system. This illustrates how security procedures implemented at higher levels of abstraction may affect the overall security.

We capture access at different levels of abstraction with Attribute-Based Group Access Control (ABGAC), and illustrate how considering a security policy at different levels of abstraction is able to model the insider problem [BIS08]. We gain a better understanding of the overall security by considering these different levels of abstraction. Given these factors, we take a hierarchical approach to defining security policy and vulnerabilities in chapter 4.

§3.2.2 Intention versus Implementation

Understanding the distinction between the intended security policy and the implemented security policy is critical to understanding where and why certain types of vulnerabilities exist. For example, consider the gap between a security policy and the security mechanisms that enforce it. As stated by Whitten and Tygar, "[s]ecurity mechanisms are only effective when used correctly" [WHI99]. However, security mechanisms are notoriously difficult to configure correctly, especially as security policies become more complex and difficult to manage. For example, a 2004 study of 37 actual firewall rule sets for 12 specific configuration errors found at least one configuration error in every rule set [WOO04]. This is just one example of how the security policy as configured on the security mechanisms may not accurately reflect the security policy as intended by the administrators.

Organizations often have a dynamic set of security mechanisms protecting a dynamic set of systems with a dynamic set of users. Even if a security mechanism properly enforces a security policy, it is difficult to maintain the correctness of that security policy over time. In practice, we cannot assume that the security polices in place are correct [SIN08]. The result is a disconnect between the stated security policy and the security policy as configured on the mechanisms [BIS06].

The security mechanisms themselves may contain bugs or vulnerabilities. As a result, the security policy enforced may differ from the security policy as configured on the mechanism. For example, consider the buffer overflow reported in the `login` command of UNIX version 6 [BIS09A]. Exploiting this vulnerability allowed users to authenticate as `root` and bypass the configured access controls.

Furthermore, it may not be possible to fully implement the desired security policy. While most systems can control the transfer of rights, they are not able to fully control the transfer of information. For example, many systems are able to prevent the user account `yasmin` from

transferring the `read` privilege of a confidential report to the user account `xander`, but unable to stop Yasmin from taking a snapshot of that report and giving a copy to Xander. This highlights just one of the many technological constraints on system-based security mechanisms. Whereas an ideal policy deals with information and people, this must be translated to system objects and user accounts at a system-level in order to be implemented.

Each of these factors contribute to the gap between the intended security policy and the implemented security policy. Capturing this distinction is essential to vulnerability analysis. An action allowed by implementation but not intention represents a potential vulnerability. We capture this distinction with the Policy-Based Vulnerability Hierarchy in chapter 4.

§3.2.3 Policy as a Partition

At the state machine level, security policy is formally defined as a partition of states. For example, Bishop defines a security policy as, "a statement that partitions the states of the system into a set of *authorized*, or *secure*, states and a set of *unauthorized*, or *nonsecure*, states" [Bis03a, *p*95]. Given a state machine with states Q, a **security policy** is the partition $(A, Q \setminus A)$ where $A \subseteq Q$ is the set of authorized states and the set difference $Q \setminus A$ is the set of unauthorized states.

Using this model, a state machine is considered **secure** if it "starts in an authorized state and is unable to enter an unauthorized state" [Bis03a, *p*95]. Thus, if the unauthorized states are not reachable from any authorized start state, then the state machine is secure. We can determine if any unauthorized states are reachable from the start state by treating the state machine as a directed graph. This is equivalent to the s-t connectivity (STCON) problem for directed graphs, which can be solved using a sublinear-space, polynomial-time algorithm [Bar92].

However, the inverse is not true. A path from the start state to an unauthorized state does not indicate the system is non-secure. It is possible that the unauthorized state is never entered on any input, making it a **useless state**. A useless state may exist for a variety of reasons, including issues with the system design, assumptions made about the operating environment, changes made during maintenance, or due to code complexity. The decidability of determining if a state is a useless state depends on the underlying state machine. While this problem is decidable for a deterministic finite automaton (DFA) or push-down automaton (PDA), it is undecidable for

Turing machines [SIP97]. As a result, security based on a partition of states in a Turing machine is undecidable.

This formalization of a security policy and system security is appropriate in a theoretical environment. However, this approach is difficult to abstract to a practical level. For example, assume we have a pre-existing Turing machine $M = (\, Q,\, \Sigma,\, \Gamma,\, \delta,\, q_0,\, q_\mathrm{a},\, q_\mathrm{r}\,)$. Let the state $q_i \in Q$ delete the remainder of the tape. Specifically, let the transition function δ include:

$$\delta(\, q_i,\, x\,) = (\, q_i,\, \sqcup,\, \mathrm{R}\,) \quad \text{where } x \neq \sqcup$$

When state q_i is entered on M, the Turing machine replaces any symbol x with the blank symbol $\sqcup$ until another blank symbol is encountered. Ideally, this delete action should only be authorized if the correct password is provided at the start of the tape. Let $w \in \Sigma^*$ be the secret password. How do we capture this security policy with a partition of states, and determine if M satisfies this policy?

While awkward, we may be able to achieve this security policy by redesigning our system. For example, we can split the state q_i into an authorized state q_i' and unauthorized state q_i'', and update the transition function δ such that state q_i' is only reachable when w is on the tape. This results in a new machine:

$$M' = (\, Q',\, \Sigma,\, \Gamma,\, \delta',\, q_0,\, q_\mathrm{a},\, q_\mathrm{r}\,) \quad \text{where} \quad Q' = (\, Q \setminus \{\, q_i \,\}\,) \cup \{\, q_i',\, q_i'' \,\}$$

The security policy for M' may then be captured by a partition of states $(\, A, Q \setminus A\,)$ where A is the set of authorized states such that $q_i' \in A$ and $q_i'' \notin A$.

This approach allows us to express a security policy for the new machine M', but not for our pre-existing machine M. However, in practice, we often must express a security policy for a pre-existing general purpose system, and determine whether that system is capable of meeting our security requirements.

The redesign process itself is also troublesome. It is unreasonable to expect policy makers to have the expertise required to redesign these systems, and equally unreasonable to expect anyone to redesign a system each time a security policy changes. In practice, we may not even have access to the source code and system designs necessary to perform these changes.

This formalization also assumes the implementation of the machine itself is bug-free. Consider the earlier example, where we split q_i into an authorized state q_i' and unauthorized state q_i''. In practice, the state q_i may be reachable when the password w is not on the tape due to an im-

plementation bug. We are only able to say that state q_i' is authorized if we are able to *prove* it is reachable only when w is on the tape. This is equivalent to the "Instantaneous Description Problem" introduced in Appendix A, which is the *unsolvable* problem of determining if one configuration is reachable from another [DEN96].

Instead, we seek a formal model of security policy that is influenced by practice. Specifically, we focus on the scenario where we wish to express a system-specific security policy for a pre-existing general purpose system, and evaluate the security of that system with respect to that security policy. We expect the model to allow for the expression of a security policy without redesigning the system, and to capture both *what* is authorized or unauthorized and *when* it is authorized or unauthorized.

§3.2.4 Policy as a Language

Instead of defining a security policy for a Turing machine as a set of authorized *states*, we define security policy as a language of authorized *configurations*. This approach allows us to consider the contents of the tape when determining when a state should be authorized. Consider the earlier example, where state q_i is authorized only when the password string w is at the start of the tape. We can express this as the regular language:

$$w\,\Gamma^*\,q_i\,\Gamma^*$$

We are still able to capture any security policy expressed as a partition of states. For example, suppose we have a partition $(A, Q \setminus A)$ where $A \subseteq Q$ is the set of authorized states. We can capture this partition with the regular language:

$$\Gamma^*\,A\,\Gamma^*$$

We only capture those configurations that are authorized so far. In practice, it is important to understand both what is authorized and what is unauthorized. We are able to derive the language of unauthorized configurations using complementation. For example, the language of unauthorized configurations for M may be expressed as:

$$\overline{w\,\Gamma^*\,q_i\,\Gamma^*}$$

However, not all languages are closed under the complement operation. As a result, we restrict the class of languages used for expressing authorized configurations to the decidable languages, which are closed under the complement operation [SIP97, $p149$].

It is important to note that defining *policy as a language* is not the same as the notion of a *policy language*. A **policy language** is defined as "a language for representing a security policy" that may be high-level or low-level in nature [BIS03A, *p*104]. Policy languages in this sense attempt to capture an abstract notion of a security policy in a way that is more precise than natural languages but more understandable than machine languages. Consequently, a security policy and a policy language are separate notions.

This formalization allows us to capture a security policy more naturally than a partition of states. It does not require a system redesign to express a security policy for a pre-existing system, and captures the concept of conditionally allowing states and actions based on the user input and contents of the tape. We reflect this approach to formally defining a security policy in section 3.3.5 when defining policy conditions.

§3.3 Terminology

We informally introduced the concept of a security policy and related terminology in chapter 2. We provide formal definitions for these terms in this section, which serves as a foundation for the Policy-Based Vulnerability Hierarchy in chapter 4.

§3.3.1 Policy Events

Recall from section 2.4.1 that a policy event is any event "of concern" to the policy makers. Formally, we define this as:

> DEFINITION 3.1: A **policy event** is any subject, object, action triple $E = (\, s, o, a \,)$ such that policy maker(s) want to specifically enable or prevent subject s from performing action a on object o.

For example, suppose the policy makers want to control whether the user `yasmin` is able to perform the `read` action on the `passwd` file. The resulting policy event is:

$E = (\,$ `yasmin`, `passwd`, `read` $)$

The response to any policy event must be consistent within a security policy to ensure the repeatability of our analysis results later. However, there has been considerable work looking at conditional or adaptive security policies, which dynamically change the security policy based on certain conditions. Whether access may depend on the performance load of the system, rate of

requests by the user, or other environmental factors. The work on threat-adaptive security policy is one example of this type of security policy [VEN97].

The response to a policy event as defined above may not always be consistent in a conditional security policy. Instead, we capture these policies with *conditional policy events*, which embed the conditions that must be met for a policy event to be authorized or unauthorized:

> DEFINITION 3.2: A **conditional policy event** is the quadruple $E = (s, o, a, b)$ such that policy maker(s) want to specifically enable or prevent subject s from performing action a on object o when the Boolean condition b is true.

For example, consider a security policy which authorizes Xander's access to a detailed terrain map based on a need-to-know condition. When Xander's unit is under fire, an administrator updates the unit status in the system. The resulting policy event may look like:

$$E = (\text{Xander, map data, } \texttt{read}, (\text{UnitStatus(Xander)} \equiv \texttt{under\_fire}))$$

Any policy event may be also represented as an equivalent condition policy event by setting the condition to `true`. For a policy event $E = (s, o, a)$, the associated conditional policy event is $E' = (s, o, a, \texttt{true})$. Therefore, we assume all policy events from this point forward are conditional policy events. This allows us to maintain a consistent response while still capturing these conditional policies.

We capture all of the policy events under consideration by the policy makers in the *global policy event space*. Informally, the global policy event space is the *universal set* of policy events. The most straightforward definition of the global policy event space is the set of all the policy events under consideration by the policy makers. However, we must consider how this set is formed or approximated in a practical setting. This approach forces the policy makers to act as enumerators, listing a prohibitively large or possibly infinite number of policy events.

Instead, we break the problem into more manageable pieces by first defining the universal sets of subjects, objects, actions, and Boolean conditions that are under consideration by the policy makers. In some situations, we may be able to partially automate this task. For example, many organizations maintain a list of current employees. Combined with the set of user accounts present on the systems, we may be able to infer the universal set of subjects.

Using this approach, the global policy event space is the Cartesian product of these sets. This approach may capture more policy events than necessary, but in most cases the number of

extra policy events captured will be negligible. More formally, we define the global policy event space as follows:

> DEFINITION 3.3: Let the sets $\mathbb{S}$, $\mathbb{O}$, $\mathbb{A}$, and $\mathbb{B}$ be the universal sets of subjects, objects, actions, and Boolean conditions under consideration by the policy maker(s). The **global policy event space** is the Cartesian product:
>
> $$\mathbb{E} = \mathbb{S} \times \mathbb{O} \times \mathbb{A} \times \mathbb{B}$$

We do not limit the universal sets to system-defined subjects, objects, actions, or Boolean conditions. For example, the universal set of subjects may contain both the user Yasmin and the user account `yasmin`. Also, the universal sets do not need to be mutually exclusive. For example, a process may be considered both a subject and an object. If the security policy is not conditional or adaptive, the universal set of Boolean conditions is simply $\mathbb{B} = \{ \texttt{true} \}$.

§3.3.2 Policy Responses

The next step is to identify whether a policy event is authorized or unauthorized. This is indicated by the **policy decision** for that policy event:

> DEFINITION 3.4: A **policy decision** is any decision $d \in D$ such that:
>
> $$D = \{ \texttt{yes}, \texttt{no} \}$$

Ideally, a security policy should be unambiguous and conflict-free. However, this is rarely the case in practice. A straightforward yes or no decision is not always possible for a policy event. For example, we may be unable to determine at the polling place whether a voter is eligible to vote in a particular precinct. As a result, whether the voter may cast a ballot is unknown and a provisional ballot must be provided. We define the notion of a *policy response* to handle these situations. Unlike a policy decision, a policy response may be `unknown`, indicating that we are unable to determine whether the policy event should be authorized. Formally:

> DEFINITION 3.5: A **policy response** is any response $r \in R$ such that:
>
> $$R = \{ \texttt{yes}, \texttt{no}, \texttt{unknown} \}$$

Since $D \subseteq R$, every policy decision $d \in D$ is also a policy response $d \in R$. The exact meaning of a policy response depends on the level of abstraction. For example, there is a subtle difference between a policy event that should be *authorized* versus one that is *allowed*.

§3.3.3 Policy Statements and Sets

Recall from section 2.4.1 that a security policy is informally defined as "a statement of what is, and what is not, allowed" [BIS03A, *p*9]. A policy event alone does not provide enough information. We must associate policy responses with a policy events to capture a security policy. We use a *policy statement* to associate a policy response with an individual policy event:

> DEFINITION 3.6: A **policy statement** is the pair $S = (E, r)$ where $r \in R$ is the policy response for the policy event $E \in \mathbb{E}$.

Conceptually, a security policy is a set of policy statements. However, it is possible to have two conflicting policy statements within a set. For example, consider the set:

$\{(E, \texttt{yes}), (E, \texttt{no})\}$

This set indicates that the policy event E should be both authorized and unauthorized, which is non-determinate. We capture this with the notion of a *policy conflict*, which occurs whenever two statements have different policy responses for the same policy event. Formally:

> DEFINITION 3.7: Two policy statements $S_i = (E_i, r_i)$ and $S_j = (E_j, r_j)$ are in **conflict** if and only if $E_i = E_j$ and $r_i \neq r_j$. A set of policy statements is **conflict-free** if and only if every pair of statements in that set is conflict-free.

By definition, a conflict-free set of policy statements has only one policy response for each policy event. We call such sets *policy sets*. Formally:

> DEFINITION 3.8: A **policy set** is the conflict-free set of policy statements:
>
> $P = \{(E, r) : (E, r) \text{ is a policy statement and } E \text{ is unique}\}$

A policy set captures a security policy at a specific level of abstraction. While policy statements *within* a policy set must be conflict-free, statements *across* policy sets at different levels of abstraction may conflict. We explore this further in chapter 4.

§3.3.4 Policy Properties

In addition to being conflict-free, a policy set should be precise, complete, and unambiguous. We informally introduced these properties in section 2.4.2. We use the global policy event space to formally define these properties for a policy set in this section.

Recall that a policy is considered *precise* when it has the level of granularity necessary to capture all policy events. We formally define this as:

> DEFINITION 3.9: A policy set P is **precise** with respect to the global policy event space $\mathbb{E}$ if and only if for all policy events $E \in \mathbb{E}$, there exists a policy statement $S \in P$ such that $S = (E, r)$ for some policy response r.

This formally defines the precision of the policy set with respect to the global policy event space. However, it is possible to have an imprecise global policy event space. For the purpose of this dissertation, we assume the global policy event space itself is precise.

Policy statements may include `unknown` policy responses. We say a policy set is *complete* when each policy event in the set is associated with a policy decision. Formally:

> DEFINITION 3.10: A policy set P is **complete** if and only if for all statements $S \in P$, we have $S = (E, d)$ for some policy decision d.

Notice that completeness as defined above does not depend on the global policy event space. Ideally, we want a policy set that is both precise and complete. When both properties hold, we call the policy set *unambiguous*:

> DEFINITION 3.11: A policy set P is **unambiguous** with respect to the global policy event space $\mathbb{E}$ if and only if for all policy events $E \in \mathbb{E}$, there exists a policy statement $S \in P$ such that $S = (E, d)$ for some policy decision d.

It is possible to have a policy set which is unambiguous but incorrect. Unfortunately, defining correctness for a policy set is more complicated. We must be able to capture the intent of the policy makers to determine if a policy set is correct. For this, we turn to the Policy-Based Vulnerability Hierarchy in chapter 4.

§3.3.5 Policy Conditions

The terminology discussed so far is non-specific to any underlying system or security mechanism. We use Turing machines as a system model when discussing policy in a theoretical setting. At this level of abstraction, we represent security policy using *policy conditions*, which captures security policies as a language of authorized configurations. Formally:

DEFINITION 3.12: Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, a **policy condition** is a decidable language of configurations:

$\{ uqv : uqv \text{ is an authorized configuration on } M \}$

Conceptually, a policy condition captures a single policy statement. A security policy generally consists of more than one policy statement. We capture this with the notion of a *policy condition set*:

DEFINITION 3.13: Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, a **policy condition set** is a union of a nonempty finite set of policy conditions.

A policy condition and policy condition set are both languages of authorized configurations and may be viewed as being theoretically equivalent. However, in practice, this conceptual separation captures how security policies are created. Using this formalization, we capture a *security policy* as follows:

DEFINITION 3.14: Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, a **security policy** for machine M is a Turing machine P such that $L(P)$ is a policy condition set.

Since decidable languages are closed under the union and complement operations, the language $L(P)$ is decidable and the Turing machine P will always halt [SIP97, $p149$]. However, we demonstrate in Appendix A that the language of valid configurations for a Turing machine is undecidable. As a result, this approach only captures a subset of possible security policies.

Recall from section 3.2.3 that some security policies at a state machine level are captured as a partition of states. Given a partition $(A, Q \setminus A)$, we capture this type of policy as follows:

$L(P) = \Gamma^* A \Gamma^*$

In this example, any state $q \in A$ is considered authorized, no matter what is on the tape. This brings us to a special type of policy condition:

DEFINITION 3.15: Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, a **state condition** is any policy condition that does not depend on the current contents of the tape.

This allows us to capture unrestricted portions of the Turing machine. For example, we may intend to always authorize the start and accept states. Let $\circ$ denote the concatenation operation.

We capture this as the state condition:

$$\{\, uqv \,:\, u \circ v \in \Gamma^* \text{ and } q \in \{\, q_0, q_a \,\} \,\}$$

We also define a type of policy condition which is *not* dependent on a state. Specifically, we define *tape conditions*, which are based on the content stored on the tape during computation:

> DEFINITION 3.16: Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, a **tape condition** is a policy condition that does not depend on the current state or position of the read head.

This types of condition may be useful when dealing with sensitive information that should not appear on the tape. For example, suppose we never want a secret string w to appear on the output tape of an enumerator. We can capture this with the tape condition:

$$\{\, uqv \,:\, u \circ v \notin \Gamma^* w \Gamma^* \,\}$$

We can use mixed conditions to make security policy decisions based on the language of the Turing machine. For example, suppose we never want our machine to accept with substring w on the tape. We can capture this by creating a security policy concerned only with the accepting configurations of M:

$$\{\, uq_a v \,:\, u \circ v \notin \Gamma^* w \Gamma^* \,\}$$

This captures security policy at a theoretical system-level, but we know from section 3.2.1 that other levels of security policy exist. We discuss these levels next in chapter 4.

CHAPTER 4

Policy-Based Vulnerability Hierarchy

This chapter introduces the Policy-Based Vulnerability Hierarchy, which defines a hierarchy of vulnerabilities based on the gaps between security policies at different levels of abstraction. We use this policy-based notion of a vulnerability throughout the remainder of this dissertation. This work extends the "Unifying Policy Hierarchy" by Carlson [CAR06] introduced in chapter 2.

§4.1 Introduction

Before we are able to provide a policy-based vulnerability analysis framework, we must provide a policy-based notion of a vulnerability. However, as we discuss in chapter 3, we need a model of security policy that captures the difference between intention versus implementation at multiple levels of abstraction. We address this with the Policy-Based Vulnerability Hierarchy, which is the first component of our vulnerability analysis framework.

We begin by defining security policies at four levels of abstraction: ideal, feasible, configured, and instantiated. This approach separates intention, which is captured at the higher levels of the hierarchy, from implementation, which is captured at the lowest level of the hierarchy. We use this hierarchy of security policies to define three types of vulnerabilities: inherent, configuration, and implementation.

The focus of this dissertation is on implementation vulnerability analysis. However, we demonstrate how we may use the Policy-Based Vulnerability Hierarchy to perform high-level vulnerability analysis with an insider threat case study.

§4.2 Policy Hierarchy

We introduce the policy hierarchy in this section, which provides the foundation for the vulnerability hierarchy in section 4.3.

§4.2.1 Policy Oracle

We first introduce the notion of a policy oracle. We use the term *oracle* here similar to how the term *random oracle* is used in cryptography or the term *oracle Turing machine* is used in computability theory [SIP97, *p*211]. A *policy oracle* is an all-knowing entity represented as a function that maps queries to responses. We initially treat a policy oracle as a black box and make no assumptions as to *how* the oracle arrives at the policy response. Formally:

> DEFINITION 4.1: A **policy oracle** is the function $\mathcal{P} : \mathbb{E} \rightarrow R$ that is always able to provide a consistent response to any query.

A policy oracle, being a function, associates exactly one policy response with every policy event and never changes its response to a query. The graph† of an oracle $\mathcal{P}$ is defined as the set of all statements $(E, \mathcal{P}(E))$. These statements will be conflict-free due to the consistent response returned by the oracle. As a result, the graph of $\mathcal{P}$ is a policy set that is precise with respect to the global policy event space, but may be incomplete.

Whereas precision and completeness are defined with respect to the global policy event space, we define the correctness of a policy set with respect to a policy oracle. Informally, we say a policy set is *correct* when the policy responses in all of the policy statements equal those of the oracle. Formally, we define this as:

> DEFINITION 4.2: A policy set P is **correct** with respect to a policy oracle $\mathcal{P}$ if and only if for all statements $S \in P, S = (E, \mathcal{P}(E))$.

We consider the policy set associated with a policy oracle to be correct by definition, but in practice this depends on our approximation of the oracle function.

We determine whether a security policy is correct by comparing it to policy oracles at different levels of abstraction. These levels of abstraction separate events allowed by *intention* from those events allowed by *implementation*. We define these specific policy oracles next.

§4.2.2 Ideal Oracle

The *ideal oracle* captures the "ideal" or "perfect" policy as envisioned by the policy makers. The ideal oracle provides a policy decision for every policy event in the global policy event space:

† Recall from section 2.4.8 that the graph of a function f is the set of all pairs $(x, f(x))$.

DEFINITION 4.3: The **ideal oracle** is a policy oracle $\mathcal{P}_{\text{id}} : \mathbb{E} \rightarrow D$ where $\mathbb{E}$ is the global policy event space and D is the set of all policy decisions such that:

$$\mathcal{P}_{\text{id}}(E) = \begin{cases} \texttt{yes} & \text{if } E \text{ should be authorized} \\ \texttt{no} & \text{if } E \text{ should be unauthorized} \end{cases}$$

The **ideal policy** is the policy set associated with the graph of $\mathcal{P}_{\text{id}}$.

The ideal oracle is able to reason about policy events involving people and information, in addition to policy events involving user accounts and system files. For example, ideally Yasmin is the only user authorized to authenticate to the user account `yasmin` on the system. The ideal oracle reflects this as follows:

$\mathcal{P}_{\text{id}}(\text{Yasmin}, \texttt{yasmin}, \text{authenticate}, \texttt{true}) = \texttt{yes}$

For this to be true with password-based authentication systems, Yasmin must also be the only person authorized to acquire that password. Specifically:

$\mathcal{P}_{\text{id}}(\text{Yasmin}, \text{Yasmin's password}, \text{acquire}, \texttt{true}) = \texttt{yes}$

$\mathcal{P}_{\text{id}}(\text{Xander}, \text{Yasmin's password}, \text{acquire}, \texttt{true}) = \texttt{no}$

The organization's security policy may include additional requirements to ensure this holds, such as requiring users to use strong passwords, to never write down passwords, and to never share their passwords with others.

The ideal oracle *always* provides a `yes` or `no` policy decision for every policy event, making it unambiguous (both precise and complete). The ideal oracle accurately captures the intent of the policy makers. Therefore, a policy set or statement that is correct with respect to the ideal policy oracle represents exactly the intent of the policy makers.

The unambiguity of the ideal oracle is important to our analysis framework. However, the intent of the policy makers is not always explicitly or precisely expressed in practice. We must approximate the $\mathcal{P}_{\text{id}}$ function in practice by querying actual policy makers. Whether the ideal oracle remains unambiguous in practice depends on the correctness and consistency of this approximation.

For these reasons, we limit the scope of the global policy event space to stable environments with well-defined security requirements. If we are unable to resolve `unknown` policy events in our approximation of the ideal oracle, we drop those policy events from the global policy event space.

FIGURE 4.1: POLICY HIERARCHY

Policy Oracle	Subdomain	Decision Type
$\mathcal{P}_{\text{id}}$: Ideal	All Events	Authorized (Ideally)
$\mathcal{P}_{\text{fe}}$: Feasible	System-Definable	Authorized (Realistically)
$\mathcal{P}_{\text{co}}$: Configured	System-Defined	Allowed (By Configuration)
$\mathcal{P}_{\text{in}}$: Instantiated	All Events	Possible (By Implementation)

The policy hierarchy, consisting of four policy oracles defined at different levels of abstraction. Each oracle provides a different type of decision for a specific subdomain. For example, the configured oracle determines what is allowed by configuration for all system-defined events.

§4.2.3 Feasible Oracle

The ideal policy oracle represents an ideal which cannot be implemented precisely due to technological, procedural, environmental, or practical constraints. For example, an embedded system with limited performance capability and memory capacity may be unable to provide the ideal level of security. The *feasible oracle* captures the compromises made to achieve a practical and realistic security policy, and may be considered a subset of what is actually *enforceable* given the security mechanisms in place. We formally define this oracle as follows:

DEFINITION 4.4: The **feasible oracle** is a policy oracle $\mathcal{P}_{\text{fe}} : \mathbb{E} \to R$ where $\mathbb{E}$ is the global policy event space and R is the set of all policy responses such that:

$$\mathcal{P}_{\text{fe}}(E) = \begin{cases} \texttt{yes} & \text{if } E \text{ is authorized (realistically)} \\ \texttt{no} & \text{if } E \text{ is unauthorized} \\ \texttt{unknown} & \text{if } E \text{ is infeasible to determine} \end{cases}$$

The **feasible policy** is the policy set associated with the graph of $\mathcal{P}_{\text{fe}}$.

The feasible oracle must provide a policy set for each mechanism being considered in our environment. For example, if we have two mechanisms being considered, there will be two policy subsets in the feasible policy. This allows us to address specific security mechanisms while still considering multiple systems in our environment.

Unlike the ideal oracle, the feasible oracle may be unable to provide a decision for every event. The feasible oracle must return an `unknown` response for any event that is undefined by the security mechanism. For example, we want to ideally prevent Xander from acquiring Yasmin's password. However, systems only understand user accounts, not the actual users themselves. We also have no way to determine whether Xander is able to acquire this knowledge. Therefore:

$\mathcal{P}_{\text{fe}}(\text{Xander}, \text{Yasmin's password}, \text{acquire}, \texttt{true}) = \texttt{unknown}$

Alternatively, consider a physical security system that controls access to a control room via special ID badges. Suppose Xander is assigned badge ID #14, and should not be authorized to enter the control room. At an ideal level:

$$\mathcal{P}_{\text{id}}(\text{Xander},\ \text{control room},\ \text{enter},\ \texttt{true}) = \texttt{no}$$

$$\mathcal{P}_{\text{id}}(\texttt{bid:14},\ \text{control room},\ \text{enter},\ \texttt{true}) = \texttt{no}$$

While this security mechanism is able to recognize the badge ID #14 assigned to Xander, the person Xander is undefined by this security mechanism. Therefore, at a feasible level:

$$\mathcal{P}_{\text{fe}}(\text{Xander},\ \text{control room},\ \text{enter},\ \texttt{true}) = \texttt{unknown}$$

$$\mathcal{P}_{\text{fe}}(\texttt{bid:14},\ \text{control room},\ \text{enter},\ \texttt{true}) = \texttt{no}$$

Certain environments may even force the policy administrators to violate the ideal oracle policy with the feasible oracle policy. Consider security policies for electronic medical records at a hospital [AND96]. Ideally, we may only want doctors to have access to medical records for the patients at the hospital that they are currently treating. However, due to the time-critical nature of treating patients and the complexity of tracking interacting providers, the policy makers may decide to compromise and allow access to any provider at the hospital, but log each access for later review [PEI07C, *p*133] [BISAR, §5].

Theoretically, the feasible policy is precise but not complete. The feasible oracle still captures the *intent* of the policy makers; a policy set or statement that is correct with respect to $\mathcal{P}_{\text{fe}}$ reflects the intentional compromises made by the policy makers from the ideal policy. An approximation of the feasible oracle should still be precise and complete in practice, but it is possible to misrepresent the intent of the policy makers in our approximation. We may query the policy makers directly to ensure an accurate approximation, or infer the feasible oracle in cases where the organization uses a high-level policy language.

The feasible oracle captures intent at a practical level rather than at an ideal level. However, it is possible that the intended security policy may not match the policy as configured on the security mechanisms. We capture this with the configured policy oracle in the next section.

§4.2.4 Configured Oracle

Whereas the feasible oracle provides what is authorized in practice, the *configured oracle* provides what is allowed by the configuration of the security mechanism. Formally:

DEFINITION 4.5: The **configured oracle** is a policy oracle $\mathcal{P}_{\text{co}} : \mathbb{E} \rightarrow R$ where $\mathbb{E}$ is the global policy event space and R is the set of policy responses such that:

$$\mathcal{P}_{\text{co}}(E) = \begin{cases} \texttt{yes} & \text{if } E \text{ is allowed (by configuration)} \\ \texttt{no} & \text{if } E \text{ is disallowed} \\ \texttt{unknown} & \text{if unable to determine} \end{cases}$$

The **configured policy** is the policy set associated with the graph of $\mathcal{P}_{\text{co}}$.

Consider the physical security example from before. Suppose a typographical error occurs when configuring the system, and badge ID #14 is allowed access to the control room. As a result:

$\mathcal{P}_{\text{fe}}(\texttt{bid:14}, \text{control room}, \text{enter}, \texttt{true}) = \texttt{no}$

$\mathcal{P}_{\text{co}}(\texttt{bid:14}, \text{control room}, \text{enter}, \texttt{true}) = \texttt{yes}$

In both theoretical and practical settings, the configured oracle is precise and incomplete. We may approximate this oracle by inspecting the actual configurations on each security mechanism, which may be automated in some cases. As a result, it is possible to achieve an accurate approximation of the configured oracle. We illustrate how to perform this approximation for an access control matrix in Appendix B. However, depending on the number of interoperating security mechanisms present in the environment, this process may still be complex. There has been work on how to capture the configured policy in more complex environments [BIS06, CHU01].

In some sense, a policy set or statement that is correct with respect to the configured oracle captures the intent of the security mechanism and its designers. Unfortunately, these mechanisms are difficult to configure and manage as the size and complexity of the security policy increases. The result is a configured policy which is not correct with respect to the feasible policy. This is compounded by issues with the implementation of the security mechanism itself, which is captured by the instantiated oracle in the next section.

§4.2.5 Instantiated Oracle

A security mechanism may not accurately enforce the security policy as configured on the system due to bugs, errors, flaws, faults, or incorrect assumptions in the implementation or design of the mechanism. We capture what is actually enforced by the mechanism with the *instantiated policy oracle*. We formally define this policy oracle as:

DEFINITION 4.6: The **instantiated oracle** is a policy oracle $\mathcal{P}_{\text{in}} : \mathbb{E} \rightarrow R$ where $\mathbb{E}$ is the global policy event space and R is the set of all policy responses such that:

$$\mathcal{P}_{\text{in}}(E) = \begin{cases} \texttt{yes} & \text{if } E \text{ is possible (by implementation)} \\ \texttt{no} & \text{if } E \text{ is impossible} \end{cases}$$

The **instantiated policy**† is the policy set associated with the graph of $\mathcal{P}_{\text{in}}$.

Due to the misconfiguration captured by the configured oracle, Xander's badge is allowed to open the control room. Although Xander should not be allowed to enter the control room, the instantiated oracle captures that it is possible for him to enter the room as a result:

$\mathcal{P}_{\text{id}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{no}$

$\mathcal{P}_{\text{in}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{yes}$

The instantiated oracle captures the difference between policy events allowed by *intention* at the higher levels of the hierarchy, and those allowed by *implementation* of the security mechanism. Specifically, a `yes` response from the ideal oracle indicates what we want to be authorized, whereas a `yes` response from the instantiated oracle indicates what is possible. In theory, the instantiated oracle is unambiguous like the ideal policy oracle and never returns an `unknown` response for any policy event.

However, the instantiated oracle is one of the most difficult oracles to accurately approximate for an actual system. If we were able to easily determine when the implementation does not properly enforce its configuration, the area of vulnerability analysis would not be so daunting. As we know from previous decidability results, it is easier to verify what is possible than to prove what is impossible. Therefore, in practice, this oracle is largely incomplete until a specific event is demonstrated.

There is no notion of correctness for the instantiated oracle, since the instantiated oracle does not reflects intent. Instead, we consider policy statements at this level of abstraction and determine whether they are correct with respect to other policy oracles. We capture this with the notion of policy violations in the next section.

† The instantiated policy is not a security policy strictly speaking. It is a description of an implementation that accounts for any bugs or vulnerabilities present in the security mechanism.

§4.3 Vulnerability Hierarchy

In an ideal world, each level of the policy hierarchy would be identical. Specifically:

$$\forall E \in \mathbb{E} \quad \mathcal{P}_{\text{id}}(E) \equiv \mathcal{P}_{\text{fe}}(E) \equiv \mathcal{P}_{\text{co}}(E) \equiv \mathcal{P}_{\text{in}}(E)$$

However, reality does not reflect this. The separation of policies into different levels of abstraction highlights where and why certain vulnerabilities occur. We use the policy hierarchy to define several types of vulnerabilities in this section, after we formalize the notion of a policy violation.

§4.3.1 Policy Violations

Recall from Definition 3.7 that two policy statements *conflict* if they have different policy responses for the same policy event. We can use this notion to detect when different levels of the policy hierarchy conflict. For example, suppose $\mathcal{P}_{\text{id}}(E) = \texttt{no}$ and $\mathcal{P}_{\text{in}}(E) = \texttt{yes}$. The two statements $(E, \mathcal{P}_{\text{id}}(E))$ and $(E, \mathcal{P}_{\text{in}}(E))$ conflict, resulting in a policy violation between the two oracles. Specifically, we define a policy violation as follows:

> DEFINITION 4.7: A **policy violation** occurs whenever $\mathcal{P}_i(E) \neq \mathcal{P}_j(E)$ for two different policy oracles $\mathcal{P}_i, \mathcal{P}_j$ in the policy hierarchy.

There are several different types of policy violations, depending on the oracles and the type of policy responses involved in the conflict. Informally, an *unequivocal violation* is a policy violation where the conflict is between two policy decisions. Formally:

> DEFINITION 4.8: An **unequivocal violation** is a policy violation between two different policy oracles $\mathcal{P}_i, \mathcal{P}_j$ such that $\mathcal{P}_i(E) \in D$ and $\mathcal{P}_j(E) \in D$ for a policy event E and the set of policy decisions D.

With an unequivocal violation, one policy oracle returns `yes` whereas the other returns `no` (or visa versa). These policy violations represent a clear and unequivocal conflict between the two levels of policy. For example:

$$\mathcal{P}_{\text{fe}}(\texttt{bid:14}, \text{control room}, \text{enter}, \texttt{true}) = \texttt{no}$$

$$\mathcal{P}_{\text{co}}(\texttt{bid:14}, \text{control room}, \text{enter}, \texttt{true}) = \texttt{yes}$$

Sometimes the policy violation is more vague. Whereas unequivocal violations only involve policy decisions, an *equivocal violation* occurs when exactly one of the responses is `unknown`:

DEFINITION 4.9: An **equivocal violation** is a policy violation between two different policy oracles $\mathcal{P}_i, \mathcal{P}_j$ such that $\mathcal{P}_i(E) \neq \texttt{unknown}$ and $\mathcal{P}_j(E) = \texttt{unknown}$ for a policy event E.

An equivocal violation indicates that there *may* be an unequivocal violation elsewhere in the hierarchy, but this is not always the case. For example, the following is an equivocal violation between the ideal and feasible oracles:

$$\mathcal{P}_{\text{id}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{no}$$

$$\mathcal{P}_{\text{fe}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{unknown}$$

We already know that the system was misconfigured such that Xander's badge is able to open the control room door. Therefore, at an instantiated level, Xander is able to enter the room. The result is an unequivocal violation between the ideal and instantiated oracles:

$$\mathcal{P}_{\text{id}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{no}$$

$$\mathcal{P}_{\text{in}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{yes}$$

However, if the physical security system is properly implemented and configured, then the configured and instantiated oracles will return `no`. Despite the fact that an equivocal policy violation exists elsewhere in the hierarchy, there is no unequivocal policy violation in this case.

When two oracles both return `unknown` policy responses, there is no direct policy violation. However, this does indicate that a policy violation exists elsewhere in the hierarchy. For example, since the physical security system is only able to respond to badges and not people, both the feasible and configured oracles return `unknown` below:

$$\mathcal{P}_{\text{fe}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{unknown}$$

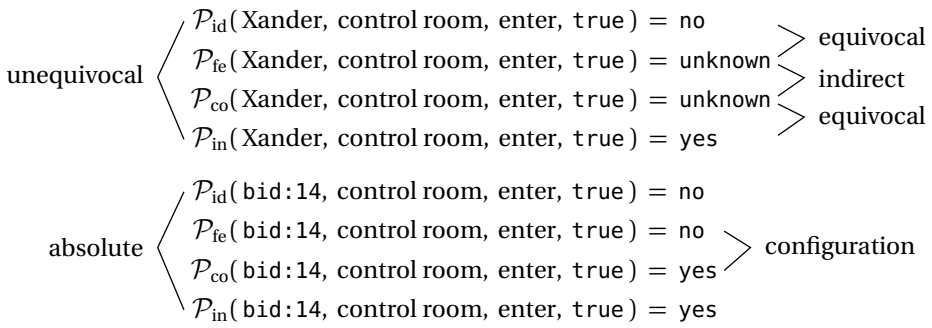
$$\mathcal{P}_{\text{co}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{unknown}$$

As we saw in the earlier examples, this event causes both an unequivocal and equivocal violation elsewhere in the hierarchy. We define *indirect policy violations* to capture this:

DEFINITION 4.10: An **indirect violation** occurs between two oracles $\mathcal{P}_i, \mathcal{P}_j$ such that $\mathcal{P}_i(E) = \texttt{unknown}$ and $\mathcal{P}_j(E) = \texttt{unknown}$ for a policy event E.

The only two oracles that return `unknown` responses are the feasible and configured oracles. Since the ideal and instantiated oracles never return an `unknown` response, we know there exists exactly two equivocal violations whenever we find an indirect violation. As we discussed earlier, there may or may not be an associated unequivocal violation elsewhere in the hierarchy.

FIGURE 4.2: EXAMPLE POLICY VIOLATIONS AND VULNERABILITIES



Policy violations (top) and vulnerabilities (bottom) for two example policy events.

Whether or not a vulnerability exists elsewhere in the hierarchy, equivocal and indirect violations provide valuable insight. These violations indicate that the security mechanisms in place are unable to restrict a particular policy event. Either such a security mechanism does not exist at this level of abstraction, or the mechanism required has not been installed or purchased for various reasons. Elucidating why these types of policy violations exist may help determine when additional procedures are necessary to mitigate the risk associated with those violations.

Both equivocal and indirect violations indicate there is a potential problem elsewhere in a hierarchy, but only unequivocal violations represent a direct threat. We assume that if an equivocal or indirect violation is encountered, additional analysis will be done to determine whether the policy event results in an unequivocal violation elsewhere in the hierarchy. Therefore, we only consider unequivocal violations in our policy-based definition of a vulnerability. Specifically, we define a *vulnerability* as follows:

> DEFINITION 4.11: A **vulnerability** is the set of conditions that enable an unequivocal policy violation.

Informally, a vulnerability *causes* or *enables* a policy violation. For example, consider the `login` vulnerability mentioned earlier [BIS09A]. That buffer overflow enables the user to bypass the configured access controls, causing several policy violations. We leave the concept of a "condition" as a primitive for now, and expand on this notion in the next chapter.

We form the vulnerability hierarchy by examining unequivocal violations between consecutive levels of the policy hierarchy. This results in three levels of vulnerabilities, as illustrated in Figure 4.3. We discuss each of these vulnerability types further in the following sections.

§4.3.2 Inherent Vulnerabilities

An *inherent violation* is associated with policy violations that occur at the most abstract levels of the policy hierarchy. These policy violations come from the *intentional* compromises made considering the security mechanisms and non-security requirements in place. Formally:

> DEFINITION 4.12: An **inherent violation** occurs for a policy event E whenever $\mathcal{P}_{\text{id}}(E) \neq \mathcal{P}_{\text{fe}}(E)$.

Therefore, *inherent vulnerabilities* are at the top of the Policy-Based Vulnerability Hierarchy:

> DEFINITION 4.13: A **inherent vulnerability** is a vulnerability that enables an unequivocal inherent policy violation.

For example, consider the use of Voter Verified Paper Audit Trails (VVPATs) in electronic voting. These paper audit trails allow voters to verify his or her vote, and provide a means for a hand recount should problems occur with the electronic record. However, these paper trails are often recorded sequentially on a single roll of paper. The use of VVPATs illustrates an intentional trade-off between vote verifiability and voter privacy [HAL08].

We can identify these trade-offs with inherent vulnerability analysis. These vulnerabilities often indicates where the functionality, configuration, manageability, or usability of a security mechanism may be improved. As a result, this type of analysis provides a "big picture" of how capable the systems and security mechanisms are at addressing the ideal set of security requirements. However, some inherent policy violations may not be avoided. For example, consider the subset of security policies called *enforceable security policies* introduced in section 2.3.1 [SCH00]. This work captures those policies that may be enforced by certain security mechanisms. For some security policies, no enforcement mechanism exists—guaranteeing a gap between the ideal and feasible policy oracles.

§4.3.3 Configuration Vulnerabilities

Vulnerabilities that enable policy violations between the feasible and configured oracles are called *configuration vulnerabilities*. These vulnerabilities indicate that the security policy as configured on the security mechanisms is incorrect, i.e., it does not match the intention of the policy makers. Formally:

DEFINITION 4.14: A **configuration violation** occurs for policy event E whenever $\mathcal{P}_{\text{fe}}(E) \neq \mathcal{P}_{\text{co}}(E)$.

DEFINITION 4.15: A **configuration vulnerability** is a vulnerability that enables an unequivocal configuration policy violation.

These vulnerabilities are often the result of misconfiguration. A general purpose system or security mechanism must be configured to match the environment and feasible security policy by an individual, which introduces the possibility of human error. However, simple human error does not explain the prevalence of these vulnerabilities. As we point out in section 3.2.2, correctly configuring a security policy on these security mechanisms is often complex and difficult to manage. Even if we start with a correct configuration, the rate of change at many organizations impedes the ability of the administrators to correctly manage that configuration. In some cases, this situation may be improved by increasing the usability of access controls [BEZ09].

Misconfiguration is not the only underlying cause of configuration policy violations in practice. It is possible that the feasible policy is not well-defined or properly articulated, making it difficult to accurately approximate this oracle. There are numerous high-level policy languages capable of articulating a security policy at the feasible level, many of which are highlighted by Alm and Drouineaud [ALM06].

We may choose to focus on configuration vulnerability analysis to identify misconfiguration, improve the configured policy, identify security mechanisms that are difficult to configure or maintain, or to determine where the expression of the feasible policy must be better articulated. Significant research has been done in this area. For example, see the research done by Wool [WOO04], Bishop and Peisert [BIS06], Sakaki et al. [SAK06], Yuan et al. [YUA06], and by Ramakrishnan and Sekar [RAM02].

§4.3.4 Implementation Vulnerabilities

Finally, *implementation vulnerabilities* enable policy violations between the configured and instantiated oracles. This captures the traditional notion of a vulnerability at a system level, and indicates that the implementation of a security mechanism is not properly enforcing the desired security policy. Formally:

DEFINITION 4.16: An **implementation violation** occurs for a policy event E whenever $\mathcal{P}_{\text{co}}(E) \neq \mathcal{P}_{\text{in}}(E)$.

DEFINITION 4.17: A **implementation vulnerability** is a vulnerability that enables an unequivocal inherent policy violation.

These vulnerabilities are often caused by software bugs that enable a user to bypass the installed security mechanisms. These are often unintentional mistakes made by developers, such as failing to sanitize user data or check the bounds of a buffer. When these occur in conjunction with other conditions, these vulnerabilities may be exploited to either change how the security mechanism enforces the configured policy, or the actual configuration itself. The buffer overflow vulnerability in the Unix `login` process is a perfect example of an implementation vulnerability.

However, not all of these vulnerabilities are caused by unintentional errors in implementation. For example, with a little guesswork, users of the Sequoia WinEDS client version 3.1.012 for administrating an election are able to bypass all of the access restrictions in place by WinEDS and directly manipulate the underlying Microsoft SQL database [BLA07, §4.1.1]. This is not a result of an unintentional software bug; it is the (possibly unintentional) consequence of intentional choices made in the software design.

Implementation vulnerability analysis identifies where the implementation of a security mechanism fails to enforce the configured policy. This is important to understanding where our assumptions of security are incorrect, such as assuming that an underlying database may not be improperly manipulated. Even in situations where a fix to the implementation is not immediately available, we are often able to mitigate the threat represented by these vulnerabilities. Consider the above example, which illustrates the importance of physically securing and restricting access to the systems involved in an election. We focus on this type of vulnerability analysis with the Policy-Based Vulnerability Analysis Framework.

§4.3.5 Absolute Vulnerabilities

The focus thus far has been on unequivocal violations between consecutive levels of the policy hierarchy. However, we may gain some insight from policy violations between non-consecutive levels of the policy hierarchy. Specifically, we may be interested when the ideal oracle does not

FIGURE 4.3: VIOLATION AND VULNERABILITY HIERARCHY

Vulnerability	Violation	Causes
Inherent	$\mathcal{P}_{\text{id}}(E) \neq \mathcal{P}_{\text{fe}}(E)$	Practical Constraints
Configuration	$\mathcal{P}_{\text{fe}}(E) \neq \mathcal{P}_{\text{co}}(E)$	Configuration Errors
Implementation	$\mathcal{P}_{\text{co}}(E) \neq \mathcal{P}_{\text{in}}(E)$	Implementation Flaws

The policy violation and vulnerability hierarchy. Illustrates the policy violation associated with each vulnerability, and the most likely causes for the violation.

match the instantiated oracle. An unequivocal policy violation between these two levels represents an *absolute vulnerability*, indicating that the policy we want does not match the policy we have. Formally:

> DEFINITION 4.18: An **absolute vulnerability** is a vulnerability that enables an unequivocal policy violation such that $\mathcal{P}_{\text{id}}(E) \neq \mathcal{P}_{\text{in}}(E)$.

Consider our earlier example. The badge system is misconfigured, allowing Xander's badge to open the control room door. Even though Xander is not authorized at an ideal level, he is allowed to enter the room at a instantiated level. As a result, an absolute vulnerability exists:

$\mathcal{P}_{\text{id}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{no}$

$\mathcal{P}_{\text{in}}(\text{Xander, control room, enter, } \texttt{true}) = \texttt{yes}$

The ideal and instantiated oracles are difficult to accurately approximate in practice. As a result, a comparison between these oracles should only happen when absolutely necessary.

§4.4 Case Study: Insider Threat

Our focus in this dissertation is on developing a framework for implementation vulnerability analysis, but we are able to use the Policy-Based Vulnerability Hierarchy directly for high-level vulnerability analysis. We demonstrate this with an insider threat analysis case study.

We generalize the access-based approach of our prior work on the insider threat as a basis for this case study [BIS08, BIS09B, BISAR]. This work states that the insider threat "exists whenever someone has more authorized privileges at a lower policy level than at a higher policy level." The gap between different levels of policy exposes where there is a potential for abuse by an insider, and the size of this gap illustrates the "insiderness" of the individual.

We demonstrate our approach for insider threat analysis with a hypothetical election process case study. We begin with a naïve set of security policies and procedures, and use the Policy-

Based Vulnerability Hierarchy to explain how these policies and procedures may be improved to reduce the insider threat.

Suppose a small county acquires new voting systems to improve accessibility and reduce the number of ballots invalidated due to overvotes or stray marks. Under the new system, a voter makes his or her selections on an electronic touch screen device. This device prints those selections onto a paper ballot, which is deposited by the voter into a ballot box. At the end of the day, the poll workers deliver the ballot boxes to the election headquarters for counting.

The county has several rural polling places, and must deliver all of the necessary equipment and supplies to the polling places several days prior to the election. One of these polling places is a high school, which keeps this equipment in a locked storage room. Some bystanders, such as administrators and staff members, have the key to this storage room to access school supplies.

We add additional details to this process as necessary to illustrate our analysis approach. Instead of performing an analysis on the entire election process, we focus on the demonstrating the "insiderness" of any bystanders that have access to the election equipment storage room.

§4.4.1 Approach

We begin with a preparation phase, which identifies the scope of the global policy event space. This includes identifying the systems, security mechanisms, and users in the environment. However, enumerating the universal subject, object, and action sets may be impractical in a realistic environment. We approximate the global policy event space in practice with special classes and define policy events based on those classes, instead of individual subjects or objects.

The next step is to identify the set of privileges that may be misused by an insider. Two types of primitive actions may be performed by an insider: the violation of a policy using authorized access, and the violation of a policy by obtaining unauthorized access [Bis08]. We focus on the first type of violation, where an insider misuses access authorized at a lower level of policy to violate a higher level of policy. There are four types of vulnerabilities to consider:

1. Implementation vulnerabilities such that $\mathcal{P}_{\text{in}}(E) = \texttt{yes}$ and $\mathcal{P}_{\text{co}}(E) = \texttt{no}$
2. Configuration vulnerabilities such that $\mathcal{P}_{\text{co}}(E) = \texttt{yes}$ and $\mathcal{P}_{\text{fe}}(E) = \texttt{no}$
3. Inherent vulnerabilities such that $\mathcal{P}_{\text{fe}}(E) = \texttt{yes}$ and $\mathcal{P}_{\text{id}}(E) = \texttt{no}$
4. Absolute vulnerabilities such that $\mathcal{P}_{\text{in}}(E) = \texttt{yes}$ and $\mathcal{P}_{\text{id}}(E) = \texttt{no}$

We omit any implementation vulnerabilities from this example insider threat analysis, as they are addressed by the Policy-Based Vulnerability Analysis Framework. We also assume that any configuration vulnerabilities have already been detected and remedied. As a result, we perform this analysis in two phases: inherent vulnerability analysis and absolute vulnerability analysis.

We only focus on identifying the potential for misuse of privileges, and not the potential any individual has to abuse those privileges. In this sense, we are identifying the possible insider threat—not actual insiders. We may use psychological indicators to identify possible insiders [BISAR, §4], but that is outside the scope of this case study.

§4.4.2 Phase 1: Preparation

We must first determine the scope of the global policy event space. Consider the universal set of subjects for an electronic voting environment. Enumerating every possible voter may be impractical for larger districts. Instead, we create a class of subjects and define the criteria required to belong to that subset. For example, we may define the class:

$$\textit{voter} = \{\, s \,:\, s \text{ is a registered voter} \,\}$$

The class *voter* captures a subset of actual subjects. However, managing the relationships between these subsets may get complicated quickly, especially if they are not independent sets. For example, election officials, poll workers, and bystanders may also be registered voters. We must keep these relationships in mind when performing our analysis. We may do this by defining hypothetical subjects that belong to multiple classes.

We define the following subject classes for the election process scenario: *voter* for registered voters, *worker* for poll workers, *official* for election officials, and *bystander* for bystanders with access to the election equipment storage rooms. We also define a subject, Yasmin, who belongs to both the *bystander* and *voter* classes.

We define the following object classes: *ballot* for official paper ballots, *printer* for the electronic ballot printers, *box* for the ballot boxes, and *tally* for the tally machines at the election headquarters. Each *printer* object has two user accounts defined: an administrator account named `root`, and a guest user account named `anon`. We add these user accounts to both the universal set of subjects and universal set of objects. We define additional objects, actions, and conditions as needed.

§4.4.3 Phase 2: Inherent Vulnerability Analysis

In this phase, we identify those policy events that are authorized by the feasible oracle, but unauthorized by the ideal oracle. This captures the potential for an insider to take advantage of the intentional compromises made by the policy makers to violate the ideal policy.

However, approximating the configured oracle for all events may be intractable depending on the environment. We address this by carefully selecting the policy events we consider in this phase. We assume at this stage in our analysis that there are no configuration vulnerabilities present for our global policy event space. As a result, the feasible and configured oracles should be identical. Since the configured oracle is often easier to approximate in practice, we begin by identifying those events allowed by the configured oracle. These events should also be authorized by the feasible oracle.

For example, suppose only election officials are authorized to configure the ballot printers, but anyone may print out ballots. At a configured policy level, this is captured by only allowing the `root` user account to perform the `config()` action. The events allowed by the configured oracle include:

$$\mathcal{P}_{\text{co}}(\texttt{root}, \textit{printer}, \texttt{config()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\texttt{root}, \textit{printer}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\texttt{anon}, \textit{printer}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

Since the configured and feasible oracles are identical, we know these policy events are also authorized by the feasible oracle:

$$\mathcal{P}_{\text{fe}}(\texttt{root}, \textit{printer}, \texttt{config()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{fe}}(\texttt{root}, \textit{printer}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{fe}}(\texttt{anon}, \textit{printer}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

The next step in this process is to approximate the ideal oracle for these events. This must be done by querying the actual policy makers, which in this case will be the election officials. Suppose we find:

$$\mathcal{P}_{\text{id}}(\texttt{root}, \textit{printer}, \texttt{config()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{id}}(\texttt{root}, \textit{printer}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{id}}(\texttt{anon}, \textit{printer}, \texttt{print()}, \texttt{true}) = \texttt{no}$$

The ideal oracle also specifies which users are authorized to authenticate to user accounts on the ballot prints. Specifically:

$\mathcal{P}_{\text{id}}($ *official*, `root`, `authenticate()`, `true` $) =$ `yes`

$\mathcal{P}_{\text{id}}($ *anyone*, `anon`, `authenticate()`, `true` $) =$ `yes`

There is only one resulting inherent vulnerability in this case. The user account `anon` is able to perform `print()` action under any condition:

$\mathcal{P}_{\text{co}}($ `anon`, *printer*, `print()`, `true` $) =$ `yes`

$\mathcal{P}_{\text{id}}($ `anon`, *printer*, `print()`, `true` $) =$ `no`

The next step in our analysis process is to look for other vulnerabilities involving the `anon` account. Upon further investigation, we learn that ballots should ideally be printed by registered voters only on election day. The ideal policy captures this by only authorizing voters to authenticate as the `anon` account, and restricting the access of the `anon` account by the date:

$\mathcal{P}_{\text{id}}($ *voter*, `anon`, `authenticate()`, `true` $) =$ `yes`

$\mathcal{P}_{\text{id}}(\mathbb{S} \setminus$ *voter*, `anon`, `authenticate()`, `true` $) =$ `no`

$\mathcal{P}_{\text{id}}($ `anon`, *printer*, `print()`, `getdate()` $=$ `2012-11-04` $) =$ `yes`

$\mathcal{P}_{\text{id}}($ `anon`, *printer*, `print()`, `getdate()` $\neq$ `2012-11-04` $) =$ `no`

This captures an insider threat. Anyone with access to the *printer* has access to the `anon` user account, and may print out official ballots. Consider the subject Yasmin, who belongs to both the bystander and registered voter classes. While it may be less than ideal, Yasmin must have access to the storage room at the high school where the election equipment is stored prior to election day:

$\mathcal{P}_{\text{fe}}($ Yasmin's key, storage room, unlock, `true` $) =$ `yes`

As a result, Yasmin is able to authenticate as the `anon` user account, and access the ballot printers prior to the election. Therefore, for the subject Yasmin we find:

$\mathcal{P}_{\text{in}}($ Yasmin, `anon`, `authenticate()`, `true` $) =$ `yes`

$\mathcal{P}_{\text{in}}($ Yasmin, *printer*, `print()`, `getdate()` $\neq$ `2012-11-04` $) =$ `yes`

$\mathcal{P}_{\text{id}}($ Yasmin, *printer*, `print()`, `getdate()` $\neq$ `2012-11-04` $) =$ `no`

Yasmin is able to unlock the storage room, access the election equipment prior to the election, print several ballots using the ballot printer, and finally place these ballots in the ballot box. Under our naïve set of security procedures, this insider attack would go undetected.

The final step in this phase is to mitigate the discovered insider threats. The election officials decide that adding more sophisticated access control mechanisms to the ballot printers is impractical. Between the pre-election testing and early voting, restricting the `print()` function by date is too difficult to maintain. In light of this, the election officials instead decide to mitigate Yasmin's insider threat by placing ballot printers in tamper resistant locked boxes that may only be unlocked by poll workers or election officials. As a result, under our new policy:

$\mathcal{P}_{\text{in}}(\text{ Yasmin's key, } \textit{printer}\text{, unlock, } \texttt{true}\text{ }) = \texttt{no}$

$\mathcal{P}_{\text{in}}(\text{ Yasmin's key, } \textit{box}\text{, unlock, } \texttt{true}\text{ }) = \texttt{no}$

This reduces Yasmin's ability to stuff the ballot boxes prior to the election. To be safe, we may also decide to add a security procedure requiring poll workers to verify the ballot boxes are empty prior to the election.

We only consider policy events that are authorized by the feasible oracle in this phase. However, several policy events may be `unknown` by the feasible oracle. For example, the ballot printers are unable to recognize specific individuals:

$\mathcal{P}_{\text{fe}}(\text{ Yasmin, } \textit{printer}\text{, } \texttt{config()}\text{, } \texttt{true}\text{ }) = \texttt{unknown}$

Yasmin may still have privileges at a lower level of policy that she may misuse to violate the ideal policy. For example, Yasmin may be able to guess the `root` password on the ballot printers. We resolve these `unknown` policy events in the next phase.

§4.4.4 Phase 3: Absolute Vulnerability Analysis

Our insider threat analysis has focused so far on unequivocal policy violations between consecutive levels of the policy hierarchy. However, equivocal policy violations indicate that there may be a potential problem elsewhere in the hierarchy. We determine whether the policy event associated with an equivocal violation represents an insider threat by determining if an absolute vulnerability exists for that event. Specifically, we search for any policy events E such that:

$\mathcal{P}_{\text{id}}(\, E\,) = \texttt{no}$

$\mathcal{P}_{\text{fe}}(\, E\,) = \texttt{unknown}$

$\mathcal{P}_{\text{co}}(\, E\,) = \texttt{unknown}$

$\mathcal{P}_{\text{in}}(\, E\,) = \texttt{yes}$

The first step in this phase is to identify those policy events that the security mechanisms are unable to restrict. Since the configured and feasible oracles are equivalent, we focus on the policy events that have a `unknown` response by the feasible oracle. Once these policy events have been identified, we must determine if an absolute vulnerability exists for those events. For example, Yasmin is allowed access to the storage room for the intent of accessing school supplies. She is not authorized to access the storage room to tamper with the election equipment. However, realistically, we are unable to measure or restrict intent. Therefore:

$$\mathcal{P}_{\text{id}}(\text{ Yasmin, storage room, unlock, Intent(}\text{ Yasmin }) \equiv \texttt{tamper\_equip}\text{ }) = \texttt{no}$$

$$\mathcal{P}_{\text{fe}}(\text{ Yasmin, storage room, unlock, Intent(}\text{ Yasmin }) \equiv \texttt{tamper\_equip}\text{ }) = \texttt{unknown}$$

$$\mathcal{P}_{\text{in}}(\text{ Yasmin, storage room, unlock, Intent(}\text{ Yasmin }) \equiv \texttt{tamper\_equip}\text{ }) = \texttt{yes}$$

The equivocal policy violation between the ideal and feasible oracles is associated with an unequivocal violation between the ideal and instantiated oracles, and hence an absolute vulnerability exists. Yasmin is able to unlock the storage room for any purpose, including tampering with the election equipment. There is no technological solution that is capable of restricting the free will of an individual, which limits our options for mitigating this insider threat. We may be able to implement additional security procedures to minimize the potential threat, but at some level we must trust Yasmin not to misuse her privileges.

For example, Yasmin should only need access to the storage room during the day. Evening activities may be postponed for the few days that the election equipment is located at the high school, and the entire school locked down after classes end. As a result, Yasmin is no longer able to access the storage room in the evenings:

$$\mathcal{P}_{\text{fe}}(\text{ Yasmin, high school, access, AfterHours}() \equiv \texttt{true}\text{ }) = \texttt{no}$$

$$\mathcal{P}_{\text{in}}(\text{ Yasmin, storage room, access, AfterHours}() \equiv \texttt{true}\text{ }) = \texttt{no}$$

This restricts the number of bystanders that may access the equipment during hours at which this activity would go unnoticed. We may additionally require that all access during the day to the storage room is monitored. However, we must still trust that Yasmin does not misuse her privileges during the day.

This analysis of absolute vulnerabilities not only illustrates the potential insider threat in this case study, but also emphasizes where trust is a necessary part of this hypothetical election process. While it is clear that the election officials must be trusted entities in this process, we

may discover that the process affords more trust to bystanders than previously understood. We may not be able to limit the free will of these subjects, but we may be able to improve our security procedures to mitigate the insider threat.

§4.5 Prior Work

The Policy-Based Vulnerability Hierarchy is based on the "Unifying Policy Hierarchy," which we introduced in chapter 2. The hierarchy presented in this chapter keeps in the same spirit as the original. The specific modifications we have made to the hierarchy include:

- We introduce the notion of a *policy oracle* and use this to define the different levels of a security policy.
- We slightly alter the terminology for each level of the policy hierarchy to more accurately describe what each level represents. For example, instead of "Actual Policy," we name the lowest level of the hierarchy the *instantiated oracle*.
- We define the notions of precision, completeness, and correctness, and discuss these properties with respect to the policy oracles at different levels of abstraction.
- We use the same notion of a policy event, except we introduce the term *global policy event space* to refer to the "set of all policy events." Specifically, the set PE in the original hierarchy is the set $\mathbb{E}$ in this dissertation.
- We set the domain of every policy event to be the global policy event space $\mathbb{E}$, and introduce `unknown` policy responses for events that are outside the original domains defined for each level. This allows us to compare events across all levels of the policy hierarchy.
- We clarify exactly what a positive response indicates at each level of the hierarchy. For example, a `yes` response by the ideal oracle determines what should or should not be authorized. Contrast this with a `yes` by the configured oracle which indicates what is or is not allowed.
- The instantiated oracle is broader than the original Actual Policy level, capturing access outside the system as well. For example, if we are concerned

with whether a file may be deleted, we also want to capture whether someone has physical access to destroy the storage medium.

- In addition to general policy violations, we introduce the notions of an unequivocal, equivocal, and indirect violation.
- We expand the notion of a policy event and the global policy event space to accommodate conditional and adaptive policies.

Since we restrict vulnerabilities to unequivocal policy violations, our notions of an implementation vulnerability is equivalent to a runtime vulnerability in the original hierarchy. We also capture the previous modifications made to the hierarchy for cases of insider threat.

§4.6 Summary

This chapter introduced the Policy-Based Vulnerability Hierarchy, which provides a hierarchical policy-based approach capable of separating policy events allowed by intention from those allowed by implementation. The policy hierarchy defines four policy oracles at different levels of abstraction: ideal, feasible, configured, and instantiated. We use unequivocal policy violations between consecutive policy oracles to form the vulnerability hierarchy, resulting in three types of vulnerabilities: inherent, configuration, and implementation.

The Policy-Based Vulnerability Hierarchy addresses our overall objectives in multiple ways. We use this hierarchy to provide a policy-based notion of a implementation vulnerability, which becomes the basis for the Policy-Based Vulnerability Analysis Framework. The hierarchy is also influenced by practice, and capable of capturing both security procedures and security mechanisms. We take into consideration how approximation of the policy oracles affects properties such as precision, completeness, and correctness. Finally, we demonstrate how to use this framework for high-level insider threat analysis.

FIGURE 4.4: TERMINOLOGY AND NOTATION

Name	Notation	Reference
Policy Event	$E = (s, o, a)$	3.1 (p37)
Conditional Policy Event	$E = (s, o, a, b)$	3.2 (p38)
Subject	s	
Object	o	
Action	a	
Boolean Condition	b	
Global Policy Event Space	$\mathbb{E} = \mathbb{S} \times \mathbb{O} \times \mathbb{A} \times \mathbb{B}$	3.3 (p39)
Universal Subject Set	$\mathbb{S}$	
Universal Action Set	$\mathbb{O}$	
Universal Object Set	$\mathbb{A}$	
Universal Condition Set	$\mathbb{B}$	
Policy Decision	$d \in D = \{\,\texttt{yes}, \texttt{no}\,\}$	3.4 (p39)
Policy Response	$r \in R = \{\,\texttt{yes}, \texttt{no}, \texttt{unknown}\,\}$	3.5 (p39)
Policy Statement	$S = (E, r)$	3.6 (p40)
Policy Set	$P = \{\,S_1, S_2, \ldots\,\}$	3.8 (p40)
Precise	iff $\exists (E, r) \in P \quad \forall E \in \mathbb{E}$	3.9 (p41)
Complete	iff $S = (E, d) \quad \forall S \in P$	3.10 (p41)
Unambiguous	iff $\exists (E, d) \in P \quad \forall E \in \mathbb{E}$	3.11 (p41)
Correct	iff $\mathcal{P}(E) = r \quad \forall (E, r) \in P$	4.2 (p45)
Policy Oracle	$\mathcal{P} = \mathbb{E} \to R$	4.1 (p45)
Ideal Oracle	$\mathcal{P}_{\text{id}} = \mathbb{E} \to D$	4.3 (p45)
Feasible Oracle	$\mathcal{P}_{\text{fe}} = \mathbb{E} \to R$	4.4 (p47)
Configured Oracle	$\mathcal{P}_{\text{co}} = \mathbb{E} \to R$	4.5 (p48)
Instantiated Oracle	$\mathcal{P}_{\text{in}} = \mathbb{E} \to D$	4.6 (p49)
✻ Let $\mathcal{P}_i(E) = r_i$ and $\mathcal{P}_j(E) = r_j$ where $i \neq j$ for the same event E.		
Policy Violation	$r_i \neq r_j$ (✻ see above)	4.7 (p51)
Unequivocal Violation	and $r_i, r_j \in D$	4.8 (p51)
Equivocal Violation	and $r_i \in D$ and $r_j = \texttt{unknown}$	4.9 (p51)
Indirect Violation	$r_i = \texttt{unknown}$ and $r_j = \texttt{unknown}$	4.10 (p52)
Vulnerability Hierarchy		4.11 (p53)
Inherent	$\mathcal{P}_{\text{id}}(E) \neq \mathcal{P}_{\text{fe}}(E)$	4.13 (p54)
Configuration	$\mathcal{P}_{\text{fe}}(E) \neq \mathcal{P}_{\text{co}}(E)$	4.15 (p55)
Implementation	$\mathcal{P}_{\text{co}}(E) \neq \mathcal{P}_{\text{in}}(E)$	4.17 (p56)

Summary of the terminology and notation used throughout chapter 4. In general, universal sets are typeset in blackboard bold (such as $\mathbb{E}$) and oracle functions are typeset in script (such as $\mathcal{P}$).

CHAPTER 5
Characteristic-Based Classification

We introduce the Characteristic-Based Vulnerability Classification Scheme in this chapter, which classifies implementation vulnerabilities at a practical level of abstraction. This work extends the hierarchical and characteristic-based approaches of previous work [WHA05, BIS99].

§5.1 Introduction

Given the complexity of approximating the instantiated oracle in practice, an undirected search for implementation policy violations is not a practical use of finite resources such as time, money, or expertise. This highlights the need for *targeted* implementation vulnerability analysis in practice. For this, we turn to vulnerability classification.

Vulnerability classification schemes play an important and informative role in vulnerability analysis. Classification results allow analysts to "extract common features" between similar vulnerabilities [BIS03A, *p*660], and provide a distribution of prevalent vulnerability types. Consider the Common Weakness Enumeration (CWE) list and classification scheme [CWE09]. The latest CWE report illustrates that while buffer overflows were traditionally the most common type of software weakness, they have recently been overcome by web-based weaknesses such as cross-site scripting (XSS) and SQL injection [CHR07]. This information is useful for both finding and defending against vulnerabilities. For example, the prevalence of buffer overflow vulnerabilities may direct the type of hypotheses formed during analysis, while the rise in web-based weaknesses may influence the type of defenses developed and deployed by the community.

However, many existing classification schemes are ad hoc in nature and suffer from ambiguity. For example, the classification of the 1988 Internet worm [ROC89] in some classification schemes is inconsistent depending on point of view and level of abstraction [BIS99, BIS96]. This

inconsistency hinders the *repeatability* of vulnerability analysis based on these classification results. Repeatability is a fundamental objective of our Policy-Based Vulnerability Analysis Framework, making it vital that we use an approach to classification that provides repeatable results.

Some classification schemes also suffer from ambiguity in *what* is classified in addition to *how* it is classified. Consider the ambiguity surrounding how the term "vulnerability" is defined. Instead of classifying actual vulnerabilities, many classification schemes avoid this ambiguity by classifying errors, bugs, faults, or weaknesses that *tend* to lead to a vulnerability based on the observations of security analysts. For example, many vulnerabilities are classified as a buffer overflow, based on a bug caused by insufficient bounds checking before placing data into a buffer. Based on these classification results, we may turn to dynamic or static analysis to detect and eliminate these type of bugs from the software, or encourage better coding practices to prevent these types of bugs in the first place. From a software engineering standpoint, this focus on the software bug may help improve the reliability and quality of the software. However, not all buffer overflow bugs lead to vulnerabilities—resulting in many false positives. While this improves the quality of the software, it may not be the best use of resources from a security standpoint.

This also illustrates the difference between a bug versus a vulnerability. A vulnerability is *more* than a software bug. The focus on bugs or weaknesses may hide other important characteristics of that vulnerability. For example, we previously illustrated that many buffer overflow vulnerabilities also require stack modifications or the ability to upload executable code [Bis10]. Many defenses against buffer overflow vulnerabilities, such as address-space randomization or stack canaries, focus on these other characteristics. A characteristic-based approach not only illustrates additional defense vectors, but also enables a targeted search of unknown vulnerabilities in a system [Bis99].

We expand this characteristic-based approach with the Characteristic-Based Vulnerability Classification Scheme in this chapter, and illustrate how this enables a targeted analysis of implementation vulnerabilities in chapter 6. We begin by providing the foundation necessary to formally define implementation vulnerabilities for theoretical environments. We then illustrate the impracticality of the assumptions made by this theoretical model, and provide an abstraction of implementation vulnerabilities appropriate for practical environments. We use this abstraction in our hierarchical characteristic-based approach to classification.

§5.2 Terminology

This section formally defines implementation vulnerabilities using the Formal Implementation Vulnerability Model, which defines a security policy for a Turing machine as a decidable language of authorized configurations. This provides the foundation necessary for our characteristic-based approach to vulnerability classification.

§5.2.1 Implementation Vulnerabilities

We introduce the notion of an implementation vulnerability in chapter 4 as the set of conditions that enable an unequivocal policy violation between the configured and instantiated oracles. We refine this notion by formally defining conditions and policy violations in a theoretical setting, using deterministic universal Turing machines as the system model.

We start by formally defining the actual policy violation itself. An *implementation policy violation* for a Turing machine M is defined as a configuration u that is either valid but unauthorized, or authorized but invalid. Formally:

> DEFINITION 5.1: An **implementation violation** on a Turing machine M for a security policy P is a configuration u in the symmetric difference:
>
> $$u \in (\mathit{VALID}(M) \ominus L(P))$$

We use the notion of *real-time security* to capture when these implementation policy violations occur. Informally, a Turing machine M is **real-time secure** for an input string w when there are no implementation violations in its current partial computation trace. The earliest computation step i such that the machine M is no longer real-time secure captures the exact moment an implementation violation occurs. We describe the configurations just prior to time step i using *preconditions*:

> DEFINITION 5.2: A **precondition** for an implementation violation occurring at computation step i of a Turing machine M on input w with respect to a security policy P is a configuration t such that:
>
> $$t \in \mathit{PARTIAL}(M, P, w, i - 1)$$

We use both the implementation violation and these preconditions to formally define an implementation vulnerability for a Turing machine:

DEFINITION 5.3: An **implementation vulnerability** is the pair $V = (T, U)$ such that each $t \in T$ is a precondition for a implementation violation $u \in U$.

While Definition 5.3 provides a precise policy-based definition of an implementation vulnerability for a Turing machine, this low-level of abstraction is impractical for most environments. We discuss this impracticality next.

§5.2.2 Perfect Knowledge Assumption

The Formal Implementation Vulnerability Model summarized in section 5.2.1 operates in a theoretical setting where we have *perfect knowledge* of our systems and environment. However, a major objective of this dissertation is to provide a practical framework for implementation vulnerability analysis. As such, we make explicit the underlying assumptions in our theoretical model, and examine the practicality of these assumptions. We begin by explicitly defining the perfect knowledge assumption:

DEFINITION 5.4: The **perfect knowledge assumption** assumes that we have a well-defined non-empty set of systems $\mathbb{M}$, and for each system $M \in \mathbb{M}$ we know the Turing machine specification $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ and the computation trace for every input w computed by M.

These assumptions are not unreasonable or unusual in a theoretical setting. However, when moving from a theoretical to a practical setting, every part of the perfect knowledge assumption falls apart. In contrast to universal Turing machines, modern computers are not pre-defined or built using a formal Turing machine specification. The ad hoc and evolving mix of software and hardware make determining an equivalent Turing machine specification for most computers prohibitively complex at best.

The assumption that we know the computation trace is also problematic. While many modern systems are capable of recording an audit or system log, these logs may not capture every input or enough detail to comprise a full computation trace. There are some approaches that use virtualization to capture enough information for attack recovery or replay, but this does not address the insufficiency of log information for the underlying system itself [DUN02, OLI06].

Even if the processing and memory requirements for recording the trace for every input were practical, parsing these logs to locate policy violations may be prohibitively time consuming. For

example, the area of computer forensics must grapple with issues surrounding the amount of data collected, and the practical utility of that data [PEI07B, PEI07C].

Furthermore, the observer and observed are often the same in practice—the systems themselves are responsible for correctly recording and storing system or audit logs. If we are investigating the system for vulnerabilities, what confidence do we have that the recording mechanisms themselves are not vulnerable? Attacks on these recording mechanisms are standard, and the long-standing dilemma of "trusting trust" persists even at the hardware level [THO84, IRV07].

Even if we were able to provide the specification and the computation trace, we run into yet another practical issue. Classification is often done for a generic set of machines and environments. Without a specific set of machines, we are unable to provide the specification and computation history even in a theoretical setting. For example, Aslam's Taxonomy of Security Faults is designed to apply to any Unix-based system [ASL95]. The number of possible operating system and hardware combinations makes the resulting set of machines prohibitively large to enumerate. However, this broad applicability is vital to making a practical classification scheme.

It is possible to develop systems capable of addressing many of these concerns. However, in practice, those systems are unlikely to supplant the widely-used existing systems. As such, we focus our analysis on these existing systems and conclude:

ASSERTION: The perfect knowledge assumption is impractical.

The perfect knowledge assumption may be practical for some situations. There may exist a small-scale high-assurance environment with well-defined devices and security policies for which using the Formal Implementation Vulnerability Model and formal methods may be practical and desirable. However, the perfect knowledge assumption is impractical for most environments, especially those that we wish to target for vulnerability analysis.

This motivates our need to provide levels of abstraction that move away from the requirements of the perfect knowledge assumption. Consider, for example, Turing machine algorithms. Recall from section 2.4.3 that our intuitive notion of an algorithm is roughly equivalent to what may be computed by a Turing machine. Introducing some amount of ambiguity allows us to discuss the computation of a Turing machine at a more practical level of abstraction. If necessary, this ambiguity may be resolved by reversing this abstraction to provide a full Turing machine specification for any given algorithm.

We follow a similar approach in this chapter. We use abstraction to hide the specification and computation trace requirements. However, too much ambiguity hinders repeatability. We attempt to introduce as little ambiguity as possible to make this abstraction process reversible and repeatable, similar to how the abstraction of Turing machine algorithm may be reversed to the equivalent Turing machine specification. We introduce the mechanisms necessary for this abstraction next. While this adds a considerable amount of formal machinery to this approach, much of this will be hidden in practical settings.

§5.2.3 Representative System Set

Embedded in the perfect knowledge assumption is the need for an explicitly defined set of systems $\mathbb{M}$. This *representative set* of systems captures those systems that we are interested in analyzing for vulnerabilities. We treat this set as a universal set, restricting our discussion of preconditions and policy violations to this set of systems.

Our ability to abstract preconditions and violations for classification depends on the similarity of systems in this set. If the systems are too disparate, we will be unable to find common preconditions and apply our analysis across multiple systems. If the systems are too similar, our abstractions will not generalize to other systems or environments, limiting the applicability of our results in practice.

We attempt to avoid these issues by requiring each system in the representative system set to be similar enough in specification and functionality such that they are capable of enforcing the same configured security policy for a nonempty subset of the global policy event space. For example, despite differences in hardware and operating environment, a laptop and desktop running the same operating system share some aspects of the configured security policy for a subset of the global policy event space. Specifically:

> DEFINITION 5.5: The **representative system set** is the universal set of systems $\mathbb{M}$ such that the configured security policy for each $M \in \mathbb{M}$ is equivalent for some nonempty subset of the global policy event space $\mathbb{E}$.

As we discussed in section 5.2.2, the representative system set $\mathbb{M}$ is often ambiguously defined in practice to cover a wide-range of actual machines. For example, the representative system set may include any Linux-based operating system. While we may be unable to enumerate

all of the systems in the representative system set, we are often able to unambiguously determine if a specific system is part of the representative set. For example, if the representative set includes any Linux-based operating system, we know that a specific machine M running an Ubuntu Linux distribution belongs to our representative system set.

We capture this ability with a special oracle, similar to an *indicator function*,† to determine if a given machine M is part of our representative system set. Specifically:

> DEFINITION 5.6: The **system oracle** is a function $\mathcal{I}_{\mathbb{M}}$ for the representative system set $\mathbb{M}$ such that:
>
> $$\mathcal{I}_{\mathbb{M}}(M) = \begin{cases} 1 & \text{if } M \in \mathbb{M} \\ 0 & \text{if } M \notin \mathbb{M} \end{cases}$$

In practice, we approximate the operation of the system oracle to avoid enumeration of the representative system set. Specifically, we define the *properties* of systems that belongs to the representative system set instead of enumerating all of the systems that satisfy that property. As a result, we may approximate the oracle using set-builder notation:

$$\mathbb{M} = \{ M : M \text{ is a high-performance machine} \}$$

However, if we use this notation, the properties must be testable by the system oracle. In this example, the "high-performance" property is too ambiguous. Depending on who, how, and when the system oracle is approximated, what constitutes a high-performance machine may change drastically. We recommend using properties based on the formal Turing machine specification. In practice, this equates to properties based on source code, machine state, architecture, and memory contents. For example, consider:

$$\mathbb{M} = \{ M : M \text{ is running the FreeBSD operating system} \}$$

In this example, the system oracle should be able to determine if the system is running the FreeBSD operating system source code.

§5.2.4 System Sets

Under the perfect knowledge assumption, we are able to use the Turing machine specification and computation trace to determine the preconditions that lead to known policy violations dis-

† Indicator functions are also called "characteristic functions" in mathematics or "membership functions" when dealing with fuzzy sets.

covered on each machine $M \in \mathbb{M}$. We capture the known preconditions and policy violations for each system with the following sets:

DEFINITION 5.7: The **system precondition set** for a system $M \in \mathbb{M}$ is the set:

$$\mathbb{T}_M = \{\, t \,:\, t \text{ is a known precondition on } M \,\}$$

DEFINITION 5.8: The **system violation set** for a system $M \in \mathbb{M}$ is the set:

$$\mathbb{U}_M = \{\, u \,:\, u \text{ is a known policy violation on } M \,\}$$

Recall that both preconditions and policy violations describe the configurations of a Turing machine. Each configuration is a string that describes the state of the Turing machine and the current tape contents. As a result, the system precondition set and system violation set are both sets of strings. However, what each represent conceptually are very different. A policy violation captures *what* went wrong, and a precondition captures *how* that violation occurred. As a result, which configurations are described by a precondition or policy violation will differ.

At the start of our analysis, these sets may be empty depending on the pre-existing information available for our systems. As new policy violations are discovered, they are added to the system violation set along with the preconditions that led to that violation.

§5.2.5 Universal Sets

We must be especially careful of the system-specific nature of preconditions and policy violations. Both preconditions and policy violations are strings describing one or more configurations of a Turing machine. Equivalent strings refer to equivalent preconditions *within* a single system precondition set. The same holds with the system violation set.

However, this is not the case *across* multiple different system sets. Equivalent strings across system precondition or violation sets do not necessarily refer to equivalent configurations. For example, the configuration $01q_2110$ may exist on more than one machine, but q_2 refers to a specific state on a specific machine. The strings, while they may appear equivalent, have different meanings depending on the system. Given this, we do not simply union together the system precondition and violation sets to form our universal sets. We must keep the association between the preconditions (or policy violations) and the system for which they are defined. Specifically, we define these sets as follows:

DEFINITION 5.9: The **universal precondition set** for the representative system set $\mathbb{M}$ is the set of pairs:

$$\mathbb{T} = \{ (M, \mathbb{T}_M) : \mathbb{T}_M \text{ is the system precondition set for } M \in \mathbb{M} \}$$

DEFINITION 5.10: The **universal violation set** for the representative system set $\mathbb{M}$ is the set of pairs:

$$\mathbb{U} = \{ (M, \mathbb{U}_M) : \mathbb{U}_M \text{ is the system violation set for } M \in \mathbb{M} \}$$

Despite the system-specific nature of preconditions and policy violations, there may be similarities across machines. For example, state q_i in M_i may clear the tape and state q_j in M_j may do the same. Preconditions or policy violations based on these states may share some similarities. We use characteristics and symptoms to capture these similarities across systems.

§5.3 Characteristic-Based Abstraction

We established in section 5.2.2 that the perfect knowledge assumption is impractical for most environments. We define characteristics and symptoms in this section to capture implementation vulnerabilities at a more practical level of abstraction. Whereas preconditions and policy violations describe machine-specific configurations, characteristics and symptoms describe configurations across multiple systems.

§5.3.1 Characteristics

Bishop introduces the notion of a *characteristic* as "a condition that must hold for the vulnerability to exist" [BIS99]. We refine this notion to provide an abstraction of preconditions, and move away from the perfect knowledge assumption. Informally, we define a characteristic as a set of similar known preconditions across multiple systems. We capture this behavior as follows:

DEFINITION 5.11: A **characteristic** for the representative system set $\mathbb{M}$ is the set of system-specific preconditions:

$$X = \{ (M, T) : T \subseteq \mathbb{T}_M \text{ for a system } M \in \mathbb{M} \}$$

For example, suppose a null character string, denoted in C as $\backslash 0$, triggers a violation in multiple systems. Upon analysis, we discover the preconditions $t_1, t_2 \in \mathbb{T}_{M_1}$ and $t_3 \in \mathbb{T}_{M_2}$ such that:

$$t_1 = \backslash 0 \backslash 0\, q_2\, \backslash 0 \qquad t_3 = \backslash 0 \backslash 0\, q_6$$

$$t_2 = \backslash 0 \backslash 0 \backslash 0\, q_3$$

We can group these system-specific preconditions together with a "null character" characteristic that spans across systems M_1, M_2, and M_3 as follows:

$$X_{\texttt{null}} = \{ (M_1, \{ t_1, t_2 \}), (M_2, \{ t_3 \}) \}$$

Conceptually, the characteristic $X_{\texttt{null}}$ *maps* the preconditions t_1 and t_2 on system M_1, and precondition t_3 on system M_2. We use a special oracle, called the *characteristic oracle* to capture the preconditions a characteristic maps onto a specific system without having to enumerate the characteristic itself. Specifically:

DEFINITION 5.12: The **characteristic oracle** is a function $\mathcal{X}$ such that:

$$\mathcal{X}(X, M) = \begin{cases} T & \text{if and only if } (M, T) \in X \\ \emptyset & \text{otherwise} \end{cases}$$

We say characteristic X **maps** the preconditions T onto system M.

For example, we may query the characteristic oracle to identify the set of preconditions that characteristic $X_{\texttt{null}}$ maps onto system M_1:

$$\mathcal{X}(X_{\texttt{null}}, M_1) = \{ t_1, t_2 \}$$

In a theoretical sense, the characteristic oracle is unnecessary. However, in practice, we approximate the characteristic oracle to avoid enumerating the preconditions associated with each system in the representative system set, and mitigate our reliance on the perfect knowledge assumption. For example, we may wish to define the null character characteristic more abstractly:

$$X_{\texttt{null}} = \{ t \mid t \text{ contains the null character } \backslash 0 \}$$

We rely on the oracle to translate this description into specific preconditions when necessary.

§5.3.2 Symptoms

Characteristics are abstractions of preconditions. Symptoms, on the other hand, are abstractions of policy violations. We define symptoms similarly to characteristics:

DEFINITION 5.13: A **symptom** for the representative system set $\mathbb{M}$ is the set of system-specific policy violations:

$$Y = \{ (M, U) : U \subseteq \mathbb{U}_M \text{ for a system } M \in \mathbb{M} \}$$

We define the symptom oracle similar to that of a characteristic oracle. Informally, the symptom oracle maps to different sets of policy violations depending on the system. Formally:

DEFINITION 5.14: The **symptom oracle** is a function $\mathcal{Y}$ such that:

$$\mathcal{Y}(Y, M) = \begin{cases} U & \text{if and only if } (M, U) \in Y \\ \emptyset & \text{otherwise} \end{cases}$$

We define two different types of symptoms to capture an increase in privileges versus a decrease in privileges. Specifically:

$$Y_{\texttt{incr}} = \{ u : VALID(M) \setminus L(P) \}$$

$$Y_{\texttt{decr}} = \{ u : L(P) \setminus VALID(M) \}$$

The symptom $Y_{\texttt{incr}}$ captures policy violations that result from configurations that are valid, but not authorized by the security policy. Since there are more configurations possible than are allowed, this symptom captures an increase in privileges. The symptom $Y_{\texttt{decr}}$ captures policy violations that result from configurations that are allowed, but not possible. Hence, this symptom captures a decrease in privileges.

We focus primarily on characteristics in this dissertation, but symptoms provide an important mechanism for measuring the impact of a implementation vulnerability at a higher level of abstraction. For example, a distributed denial of service will generate a large number of $Y_{\texttt{decr}}$ symptoms and gaining root privileges will generate a large number of $Y_{\texttt{incr}}$ symptoms.

§5.3.3 Properties

It is difficult to quantify what makes a good characteristic or symptom. For example, a characteristic made of randomly selected known preconditions will not be useful when it comes to vulnerability classification and analysis. However, we can ensure certain properties hold when it comes to sets of characteristics or symptoms.

First, we require characteristics and symptoms to be *sound*. Informally, two characteristics or symptoms are sound if they do not include overlapping subsets. More specifically:

DEFINITION 5.15: A set of characteristics is **sound** if and only if the characteristics are *pairwise disjoint*; i.e. for every distinct pair of characteristics X_i and X_j in the set where $i \neq j$, we have:

$$\mathcal{X}(X_i, M) \cap \mathcal{X}(X_j, M) = \emptyset \qquad \forall M \in \mathbb{M}$$

Similarly, a set of symptoms is **sound** if and only if:

$$\mathcal{Y}(Y_i, M) \cap \mathcal{Y}(Y_j, M) = \emptyset \qquad \forall M \in \mathbb{M}$$

For example, suppose we have the characteristics:

$$X_1 = \{ (M_1, \{ t_1, t_2 \}), (M_2, \{ t_3 \}) \} \qquad \mathcal{X}(X_1, M_1) \cap \mathcal{X}(X_2, M_1) = \emptyset$$
$$X_2 = \{ (M_1, \{ t_4 \}), (M_2, \{ t_5 \}) \} \qquad \mathcal{X}(X_2, M_1) \cap \mathcal{X}(X_3, M_1) = \emptyset$$
$$X_3 = \{ (M_1, \{ t_2, t_6 \}) \} \qquad \mathcal{X}(X_1, M_1) \cap \mathcal{X}(X_3, M_1) = \{ t_2 \}$$

The set of characteristics $\{ X_1, X_2, X_3 \}$ is not sound since both X_1 and X_3 both map to the characteristic t_2. However, the set of characteristics $\{ X_1, X_2 \}$ is sound.

Ideally, we want there to exist a characteristic for every known precondition and a symptom for every known policy violation. We define the notion of *completeness* for sets of characteristics and symptoms to capture this:

> DEFINITION 5.16: A set of characteristics X is **complete** if and only if the set X *covers* the universal precondition set $\mathbb{T}$; i.e. for every system precondition set, we have the following:
>
> $$\bigcup_{X_i \in X} \mathcal{X}(X_i, M) = \mathbb{T}_M \qquad \forall \mathbb{T}_M \in \mathbb{T}$$
>
> Similarly, a set of symptoms Y is **complete** if and only if:
>
> $$\bigcup_{Y_i \in Y} \mathcal{Y}(Y_i, M) = \mathbb{U}_M \qquad \forall \mathbb{U}_M \in \mathbb{U}$$

We enforce the properties of soundness and completeness in our universal sets of characteristics and symptoms. These properties are critical to our ability to form sound, unique, and minimal sets of characteristics or symptoms.

§5.3.4 Universal Sets

Before we are able to abstract and classify implementation vulnerabilities, we must develop *universal sets* of characteristics or symptoms that are both sound and complete. Specifically:

> DEFINITION 5.17: The **universal characteristic set**, denoted $\mathbb{X}$, is a sound and complete set of characteristics such that $|\mathbb{X}| < |\mathbb{T}|$ for the universal precondition set $\mathbb{T}$.
>
> DEFINITION 5.18: The **universal symptom set**, denoted $\mathbb{Y}$, is a sound and complete set of symptoms such that $|\mathbb{Y}| < |\mathbb{U}|$ for the universal violation set $\mathbb{U}$.

We require that the number of characteristics in our universal characteristic set be strictly less than the number of preconditions in our universal precondition set. Without this requirement,

we may form a sound and complete set of characteristics such that there exists one characteristic for every precondition in our universal sets. We end up with just as many characteristics as preconditions, which does not improve the practicality of our model.

The development of a universal characteristic set or universal symptom set that meets these requirements is non-trivial in practice. Consider the soundness requirement for the universal characteristic set. We approximate the characteristic oracle without explicitly defining preconditions in practice, making it difficult to determine when two characteristics overlap. We may be tempted to remove the soundness restriction, but this affects our ability to use characteristic-based analysis to mitigate vulnerabilities later. As a result, it is more beneficial to limit the scope of analysis over removing the requirement for sound universal sets.

In theory, the universal characteristic set is also complete with respect to the universal precondition set. Since we do not explicitly define the universal precondition set in practice, this property becomes difficult to measure. However, we assume that if a precondition is not captured at a theoretical level, it is not a "known precondition" at a practical level.

§5.3.5 Basic Sets

Suppose we have a set of preconditions, and wish to abstract those preconditions to a set of characteristics. We capture this with the *basic characteristic set*. Bishop introduces the notion of a basic characteristic set as a "unique, sound characteristic set of minimal size" [Bis99]. There are different possible formalizations for a basic characteristic set based on our Formal Implementation Vulnerability Model. We discuss the advantages and disadvantages of these formalizations in Appendix B. We use the following formalization:

> Definition 5.19: Given a set of preconditions $T \subseteq \mathbb{T}_M$ for system $M \in \mathbb{M}$, the **basic characteristic set** is the set $X_T \subseteq \mathbb{X}$ such that:
>
> $$X_T = \{ X_i \in \mathbb{X} : \mathcal{X}(M, X_i) \cap T \neq \emptyset \}$$

> Definition 5.20: Given a set of policy violations $U \subseteq \mathbb{U}_M$ for system $M \in \mathbb{M}$, the **basic symptom set** is the set $Y_U \subseteq \mathbb{Y}$ such that:
>
> $$Y_U = \{ Y_i \in \mathbb{Y} : \mathcal{Y}(M, Y_i) \cap U \neq \emptyset \}$$

We may capture more preconditions or policy violations in the basic sets than necessary in practice. However, since the universal characteristic and symptom sets are sound, this formal-

ization still provides a basic characteristic set that is minimal, sound, and unique. As a result, if we disable any characteristic in the basic set of a vulnerability, we disable the vulnerability itself. This helps us detect when a vulnerability may exist in the system, and allows us to target specific types of defenses against these discovered vulnerabilities.

§5.3.6 Vulnerabilities

We now have all of the necessary components to provide an abstraction of implementation vulnerabilities using characteristics and symptoms. Given a implementation vulnerability, we define the abstraction of this vulnerability as follows:

> DEFINITION 5.21: Given an implementation vulnerability $V = (T, U)$, the **implementation vulnerability abstraction (IVAB)** is the pair $Z = (X_T, Y_T)$ such that the set X_T is the basic characteristic set for the set of preconditions T and the set Y_T is the basic symptom set for the set of policy violations U.

Recall that different preconditions or policy violations may map to the same characteristic or symptom. As a result, distinct system-specific implementation vulnerabilities may have the same system-independent implementation vulnerability abstraction. We capture this with the notion of a *implementation vulnerability equivalence class* as follows:

> DEFINITION 5.22: An **implementation vulnerability equivalence class (IVEC)** is a implementation vulnerability abstraction $Z = (X, Y)$ associated with one or more implementation vulnerabilities.

In a sense, an implementation vulnerability equivalence class represents a common "signature" of an implementation vulnerability. When we classify and analyze vulnerabilities, we work with unique IVECs instead of low-level implementation vulnerabilities.

§5.3.7 Buffer Overflow Example

We avoid enumerating preconditions and policy violations in practice by approximating the characteristic oracle or symptom oracle. This approximation may be informal—however, an analyst must be able to consistently infer whether a particular characteristic belongs to the basic characteristic set for a vulnerability, or whether a characteristic is present on a system.

Consider how the characteristic oracle is used. Technically, the characteristic oracle provides the set of preconditions captured by the characteristic on a system. We use this information to determine whether the characteristic is present on the system. In essence, we use the characteristic oracle as an indicator function as follows:

$$\mathcal{I}_{\mathbb{X}}(X, M) = \begin{cases} 1 & \text{if } \mathcal{X}(X, M) \neq \emptyset \\ 0 & \text{if } \mathcal{X}(X, M) = \emptyset \end{cases}$$

In practice, we are more interested in whether the set of preconditions returned by the characteristic oracle is non-empty than what the specific preconditions are. Considering this, the approximation of the characteristic oracle must be able to answer the following questions:

- Given a system M, does the characteristic X map to a non-empty set of preconditions on system M?
- Give a vulnerability abstraction Z, does the characteristic X map to a non-empty set of preconditions for Z?

We demonstrate this oracle approximation by revisiting our previous work on buffer overflow characteristics [Bis10]. The original buffer overflow characteristics, detailed in Appendix B, are defined using natural language. For example, the characteristic P1 is defined as "[t]he length of the uploaded string is longer than that of the buffer." We refine this characteristic using pseudo-code:

```
x:buff:  len(input) > len(buffer)
```

Notice that we make no mention of preconditions. We are actually defining the *characteristic oracle* above—not the characteristic itself. Those familiar with the pseudo-code will be able to answer the two questions necessary of the characteristic oracle: does the `x:buff` characteristic exist in a particular block of code, and does `x:buff` belong to the basic characteristic set of an implementation vulnerability abstraction?

Not all buffer overflows result in a vulnerability. The original buffer overflow work defines several other characteristics associated with buffer overflow vulnerabilities. For example, many buffer overflow vulnerabilities must be able to jump into memory and begin executing instructions. The following characteristics capture this:

```
x:jmps:  can_jump(stack) ≡ true

x:jmph:  can_jump(heap ) ≡ true
```

x:exes: can_exec(stack) ≡ true

x:exeh: can_exec(heap) ≡ true

Several other characteristics involve modifying a variable or pointer that affects the control flow of the process. For example, an exploit of a buffer overflow vulnerability may attempt to modify the return address of a subroutine or a function pointer. We capture this by defining variables retnptr and funcptr respectively. Other exploits may attempt to modify a control flow variable or pointer to a control flow variable, like a Boolean flag that causes the process to branch. We define variables flowvar and flowptr to capture the control flow variable and pointer. We use these variables to define the following characteristics:

x:rval: may_modify(retnptr) ≡ true

x:fptr: may_modify(funcptr) ≡ true

x:vval: may_modify(flowvar) ≡ true

x:vptr: may_modify(flowptr) ≡ true

x:path: affects_flow(flowvar) ≡ true

Finally, many of these vulnerabilities require the ability to upload a specific data type in the input string. We capture this with the following:

x:addr: may_contain(input, addr) ≡ true

x:inst: may_contain(input, inst) ≡ true

x:type: may_contain(input, type(flowvar)) ≡ true

We developed these characteristics by examining the exploits that take advantage of these buffer overflows, and observing aspects of the system architectures where these types of problems occur. For example, consider the buffer overflow vulnerability in the IPv6 code of OpenBSD versions 3.1 through 4.1 [COR07]. An attacker is able to send fragmented ICMPv6 packets, and overflow the mbuf kernel memory structure (x:buff) that stores network packets. Additionally, the mbuf structure stores a function pointer named ext_free for the m_freem() function, which is called when mbuf is freed. An attacker may upload a memory address (x:addr) and modify this function pointer (x:fptr). The attacker then triggers the m_freem() function to execute the arbitrary code stored at the uploaded memory address in kernel mode (x:jmph and x:exeh). Each step of this attack may be mapped to a characteristic, allowing us to capture the underlying buffer overflow vulnerability being exploited.

The above `mbuf` example captures an *indirect executable* buffer overflow vulnerability. These vulnerabilities always follow the same pattern, including the `x:buff`, `x:addr`, `x:fptr`, `x:jmph`, and `x:exeh` characteristics. The original buffer overflow work introduces four types of buffer overflow vulnerabilities based on these patterns [BIS10]. We capture these by defining the basic characteristic sets for the following implementation vulnerability equivalence classes:

> DEFINITION 5.23: A **direct executable buffer overflow** is an implementation vulnerability equivalence class with the basic characteristic set:
>
> { `x:buff`, `x:addr`, `x:inst`, `x:rval`, `x:jmps`, `x:exes` }

> DEFINITION 5.24: An **indirect executable buffer overflow** is an implementation vulnerability equivalence class with the basic characteristic set:
>
> { `x:buff`, `x:addr`, `x:fptr`, `x:jmph`, `x:exeh` }

> DEFINITION 5.25: A **direct data buffer overflow** is an implementation vulnerability equivalence class with the basic characteristic set:
>
> { `x:buff`, `x:type`, `x:vval`, `x:path` }

> DEFINITION 5.26: An **indirect data buffer overflow** is an implementation vulnerability equivalence class with the basic characteristic set:
>
> { `x:buff`, `x:addr`, `x:vptr`, `x:path` }

We map various defenses against these buffer overflow vulnerabilities to the specific characteristics they attempt to disable in the original paper [BIS10]. For example, canaries often attempt to detect and react to the `x:rval` characteristic. We refer readers to this work for more details on these characteristics, vulnerabilities, and available defenses.

§5.4 Hierarchical Classification

Using characteristics and symptoms, we now have a level of abstraction for implementation vulnerabilities that we may work with for vulnerability classification. We take a hierarchical approach to classification by defining classes of characteristics or symptoms at different levels of abstraction. We detail our approach here, and illustrate a classification tree for buffer overflows.

§5.4.1 Classification Components

Informally, a *class* is a collection of sets that share some property [JEC03, *p*5]. This notion fits well with our model, since characteristics themselves are sets of preconditions. Therefore, we define a *characteristic class* as a collection of similar characteristics:

> DEFINITION 5.27: A **characteristic class** is a collection of characteristics that share a common property.

> DEFINITION 5.28: A **symptom class** is a collection of symptoms that share a common property.

The properties we use to define classes depends on how the characteristics or symptoms are expressed. For example, consider the buffer overflow characteristics introduced in section 5.3.7. We take advantage of the pseudo-code used to define these characteristics, and define classes based on the underlying functions. For example, three characteristics use the `may_contain()` function, both `x:jmps` and `x:jmph` use the function `can_jump()`, and both `x:exes` and `x:exeh` use the `can_exec()` function. Along these lines, we define the following characteristic classes using a simple grammar:

`X:CONTAINS` ::= `x:addr` | `x:inst` | `x:type`

`X:JUMP` ::= `x:jmps` | `x:jmph`

`X:EXEC` ::= `x:exes` | `x:exeh`

We make an additional observation about the characteristics that use the `may_modify()` function. The direct buffer overflows alter the control flow by modifying values directly, whereas indirect overflows modify pointer variables to alter the program flow. We form a class of classes `X:MODIFY` to capture this:

`X:MODIFY` ::= `X:DIRECT` | `X:INDIRECT`

`X:DIRECT` ::= `x:rval` | `x:vval`

`X:INDIRECT` ::= `x:fptr` | `x:vptr`

There are several other classes we could define. For example, both `x:jmps` and `x:exes` are dependent on stack-based memory. This highlights the importance the characteristic classes and master classification tree have on consistent vulnerability classification. We leave the development of a master symptom tree as future work.

§5.4.2 Classification Trees

We use trees to provide a hierarchical classification framework. This approach organizes classes of characteristics or symptoms into different levels of abstraction. We start by creating classification trees for characteristics and symptoms separately:

> DEFINITION 5.29: The **master characteristic tree** is a tree, rooted at a node labeled $C_{\mathbb{X}}$, such that each leaf node is a characteristic and each internal node is either a characteristic class or a class of characteristic classes.

> DEFINITION 5.30: The **master symptom tree** is a tree, rooted at a node labeled $C_{\mathbb{Y}}$, such that each leaf node is a symptom and each internal node is either a symptom class or a class of symptom classes.

We allow for characteristics and symptoms to appear multiple times in the hierarchy. The *master classification tree* is formed by combining the master characteristic and symptom trees:

> DEFINITION 5.31: The **master classification tree** is a supergraph of the master characteristic tree $C_{\mathbb{X}}$ and master symptom tree $C_{\mathbb{Y}}$ such that:
>
> $C_{\mathbb{X}} \cup C_{\mathbb{Y}} \cup \{ \mathbb{C}, \{ (\mathbb{C}, C_{\mathbb{X}}), (\mathbb{C}, C_{\mathbb{Y}}) \} \}$
>
> We denote this tree by the root node $\mathbb{C}$.

To determine the classification of a vulnerability, we take the appropriate subgraph of the master classification tree. A vulnerability belongs to every class captured by this subgraph. With this approach, the classification of an implementation vulnerability equivalence class (IVEC) depends on the classification of its characteristics:

> DEFINITION 5.32: The **vulnerability classification tree** for an implementation vulnerability equivalence class Z is the subgraph $C_Z \subset \mathbb{C}$ such that each internal node of C_Z is a valid a classification of Z.

The question then becomes, how do we determine if an internal class node is a valid classification of a vulnerability? The simplest way of determining the appropriate subgraph of Z is to include any branch of the master classification tree that ends in a characteristic or symptom in the basic sets of Z. For example, if X_i is in the basic characteristic set of Z, we include every branch of $\mathbb{C}$ that has X_i as a leaf node.

However, this approach may be too simplistic. For example, consider the buffer overflow characteristics introduced in section 5.3.7. A buffer overflow IVEC may require the `x:jmps` characteristic, but not every IVEC that requires the `x:jmps` characteristic should be considered a buffer overflow vulnerability.

Instead, we may wish to use a classification grammar to develop a more complex hierarchical structure, similar to approach taken in the prior protocol vulnerability work [WHA05]. This allows us to define simple rules of when an internal node may be considered a classification of a vulnerability. For example, an IVEC is only classified as a buffer overflow vulnerability in Figure 5.1 when all three characteristic classes `X:INPUT`, `X:MODIFY`, and `X:CONTROL` apply.

We form this classification grammar and master characteristic tree observing the similarities between the characteristic classes of the four buffer overflow IVECs. For example, observe the similarity between the characteristic classes of direct and indirect executable buffer overflow vulnerabilities:

$$\{\underbrace{\texttt{x:buff}}, \underbrace{\texttt{x:addr, x:inst}}, \underbrace{\texttt{x:rval}}, \underbrace{\texttt{x:jmps}}, \underbrace{\texttt{x:exes}}\} \quad \leftarrow \text{direct executable}$$
$$\quad \overbrace{\texttt{X:BUFF}} \quad \overbrace{\texttt{X:CONTAINS}} \; \overbrace{\texttt{X:MODIFY}} \; \overbrace{\texttt{X:JUMP}} \; \overbrace{\texttt{X:EXEC}} \quad \leftarrow \text{classes}$$
$$\{\overbrace{\texttt{x:buff}}, \quad \overbrace{\texttt{x:addr}}, \quad \overbrace{\texttt{x:fptr}}, \overbrace{\texttt{x:jmph}}, \overbrace{\texttt{x:exeh}}\} \quad \leftarrow \text{indirect executable}$$

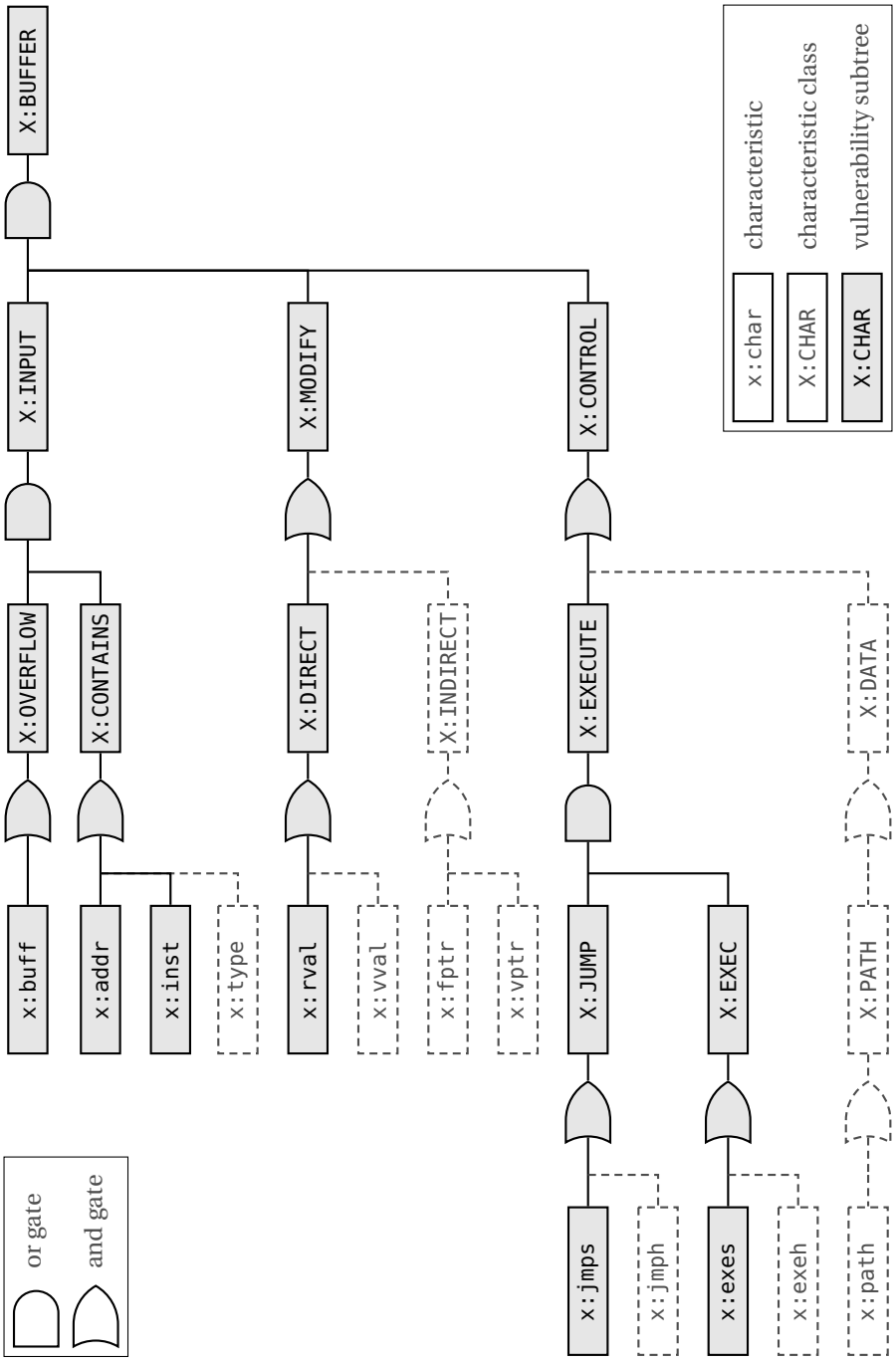
This suggests that these IVECs should fall under the same node at a higher level of abstraction. This matches our intuition, since both are executable buffer overflow vulnerabilities. In fact, all four of these IVECs share a common pattern. We illustrate this with the master characteristic tree in Figure 5.1, which is developed from the classification grammar in Figure 5.2.

Notice that the resulting vulnerability classification tree for a direct executable buffer overflow vulnerability, highlighted in Figure 5.1, gets classified as a direct (`X:DIRECT`) executable (`X:EXECUTE`) buffer overflow (`X:BUFFER`) vulnerability—matching our intuition.

§5.5 Prior Work

The Characteristic-Based Vulnerability Classification Scheme presented in this chapter is based on the characteristic-based approach introduced by Bishop [BIS99], and keeps in the same spirit as the original. We discuss this original work with respect to the Characteristic-Based Vulnerability Classification Scheme in this section.

FIGURE 5.1: BUFFER OVERFLOW CLASSIFICATION TREE



Master characteristic tree for buffer overflow vulnerabilities, developed from our earlier work [BIS10]. Characteristics, such as `x:buff`, are in lower case. Characteristic classes, such as `X:DIRECT`, are in uppercase. The highlighted nodes represent the vulnerability classification tree for a direct executable buffer overflow. Notice that both `X:JUMP` and `X:EXEC` must be present in the basic characteristic set for a vulnerability to be classified as an executable buffer overflow, labeled `X:EXECUTE` in the tree.

FIGURE 5.2: BUFFER OVERFLOW CLASSIFICATION GRAMMAR

```
 X:BUFFER  ::=  X:INPUT, X:MODIFY, X:CONTROL

  X:INPUT  ::=  X:OVERFLOW, X:CONTAINS+
 X:MODIFY  ::=  X:DIRECT | X:INDIRECT
X:CONTROL  ::=  X:EXECUTE | X:DATA

X:OVERFLOW ::=  x:buff
X:CONTAINS ::=  x:addr | x:inst | x:type
 X:DIRECT  ::=  x:rval | x:vval
X:INDIRECT ::=  x:fptr | x:vptr
 X:EXECUTE ::=  X:JUMP, X:EXEC
    X:DATA ::=  X:PATH

    X:JUMP ::=  x:jmps | x:jmph
    X:EXEC ::=  x:exes | x:exes
    X:PATH ::=  x:path
```

Buffer overflow characteristic classification grammar, developed from Bishop et al. [BIS10]. For example, every buffer overflow IVEC, captured by `X:BUFFER`, contains an `X:INPUT`, `X:MODIFY`, and `X:CONTROL` component. See Figure 5.1 for the resulting characteristic classification tree.

We add a hierarchy to the original characteristic-based approach, without compromising the five properties proposed in the original work. For example, the original work stated that similar vulnerabilities should be classified similarly. We classify vulnerability equivalence classes based on characteristics and symptoms. If these vulnerabilities share characteristics or symptoms, they will be classified similarly.

Another property in the original work was that vulnerabilities should be able to fall into multiple classes. Our hierarchical approach to classification allows vulnerabilities to fall into multiple classes. For example, the characteristic classification tree in Figure 5.1 illustrates the classification of a direct executable buffer overflow, which calls under the `X:BUFFER`, `X:DIRECT`, and `X:EXECUTE` classes.

The original work stated that classification should be primitive, well-defined, and based on technical details. Given a basic characteristic set and basic symptom set in our classification scheme, the classification of a vulnerability equivalence class meets this requirement. Our classification scheme is also well-defined in theory, although this depends on the quality of the characteristic and symptom oracles in practice.

The Characteristic-Based Vulnerability Classification Scheme is also based on technical details due to our formalization of implementation vulnerabilities. Preconditions describe the

state, transition matrix, and tape of a Turing machine. In theory, characteristics are abstractions of these technical details. In practice, this translates to properties of the system environment, code, and memory.

§5.6 Summary

We introduced the Characteristic-Based Vulnerability Classification Scheme in this chapter. We began by introducing the formal foundation for implementation vulnerabilities in a theoretical environment. We focused initially on vulnerabilities in a single Turing machine, and expanded this to cover a representative set of systems. We then discussed the impracticality of applying our theoretical model in a realistic environment, and provided an abstraction of implementation vulnerabilities for use in practical environments. We based the Characteristic-Based Vulnerability Classification Scheme on these abstractions, adding layers of hierarchy to our classification approach through classification grammars and trees. We demonstrate our approach by expanding our previous work on buffer overflow vulnerabilities, and discuss how moving between theoretical and practical environments may affect the completeness and soundness of our approach.

Given well-formed approximations of the characteristic oracles, symptom oracles, and master classification tree, the classification of individual IVECs will be consistent. This moves any ambiguity in the classification process to the initial development process. While this process is time consuming and potentially ad hoc, it only needs to be performed once and the results are broadly applicable. Therefore, we accept this ambiguity for the sake of practicality.

The Characteristic-Based Vulnerability Classification Scheme allows us to address our primary objective by providing a repeatable and practical approach to classification. We use the classification results from this scheme to inform our implementation vulnerability analysis in the next chapter.

FIGURE 5.3: TERMINOLOGY AND NOTATION

Name	Notation	Reference
Representative System Set	$\mathbb{M} = \{ M : \ldots \}$	5.5 (p72)
System Precondition Set	$\mathbb{T}_M$ where $M \in \mathbb{M}$	5.7 (p74)
System Violation Set	$\mathbb{U}_M$ where $M \in \mathbb{M}$	5.8 (p74)
Precondition	$t \in PARTIAL(\ldots)$	5.2 (p69)
Violation	$u \in (VALID(M) \ominus L(P))$	5.1 (p69)
Universal Precondition Set	$\mathbb{T} = \bigcup (M, \mathbb{T}_M) \quad \forall M \in \mathbb{M}$	5.9 (p74)
Universal Violation Set	$\mathbb{U} = \bigcup (M, \mathbb{U}_M) \quad \forall M \in \mathbb{M}$	5.10 (p75)
Characteristic	$X = \{ (M, T) : M \in \mathbb{M} \text{ and } T \subseteq \mathbb{T}_M \}$	5.11 (p75)
Symptom	$Y = \{ (M, U) : M \in \mathbb{M} \text{ and } U \subseteq \mathbb{U}_M \}$	5.13 (p76)
System Oracle	$\mathcal{I}_{\mathbb{M}} = 1$ iff $M \in \mathbb{M}$	5.6 (p73)
Characteristic Oracle	$\mathcal{X}(X, M) = T$ iff $(M, T) \in X$	5.12 (p76)
Symptom Oracle	$\mathcal{Y}(Y, M) = U$ iff $(M, U) \in Y$	5.14 (p76)
Implementation Vulnerability	$V = (T, U)$	5.3 (p69)
Vulnerability Abstraction	$Z = (X_T, Y_T)$	5.21 (p80)
Basic Characteristic Set	$X_T = \{ X_i \in \mathbb{X} : \mathcal{X}(M, X_i) \cap T \neq \emptyset \}$	5.19 (p79)
Basic Symptom Set	$Y_T = \{ Y_i \in \mathbb{Y} : \mathcal{Y}(M, Y_i) \cap U \neq \emptyset \}$	5.20 (p79)
Universal Characteristic Set	$\mathbb{X}$ *s.t.* $\forall X \in \mathbb{X} \ldots$	5.17 (p78)
Sound Characteristic Set	$\bigcap \mathcal{X}(X, M) = \emptyset \quad \forall M \in \mathbb{M}$	5.15 (p77)
Complete Characteristic Set	$\bigcup \mathcal{X}(X, M) = \mathbb{T}_M \quad \forall \mathbb{T}_M \in \mathbb{T}$	5.16 (p78)
Universal Symptom Set	$\mathbb{Y}$ *s.t.* $\forall Y \in \mathbb{Y} \ldots$	5.18 (p78)
Sound Symptom Set	$\bigcap \mathcal{Y}(Y, M) = \emptyset \quad \forall M \in \mathbb{M}$	5.15 (p77)
Complete Symptom Set	$\bigcup \mathcal{Y}(Y, M) = \mathbb{U}_M \quad \forall \mathbb{U}_M \in \mathbb{U}$	5.16 (p78)
Master Classification Tree	$\mathbb{C} = C_{\mathbb{X}} \cup C_{\mathbb{Y}}$	5.31 (p85)
Master Characteristic Tree	$C_{\mathbb{X}}$	5.29 (p85)
Master Symptom Tree	$C_{\mathbb{Y}}$	5.30 (p85)
Vulnerability Classification Tree	$C_Z \subseteq \mathbb{C}$	5.32 (p85)

Summary of the terminology and notation used throughout chapter 5. NOTE: In general, a single element is typeset in lowercase, whereas a set or tuple of elements is typeset in uppercase. For example, t is a single precondition and T is a set of preconditions.

CHAPTER 6
Policy-Based Vulnerability Analysis

The Policy-Based Vulnerability Hierarchy, the Formal Implementation Vulnerability Model, and the Characteristic-Based Vulnerability Classification Scheme provide the components necessary for the Policy-Based Vulnerability Analysis Framework. We highlight how to use this framework for implementation vulnerability analysis in this chapter.

§6.1 Introduction

The Policy-Based Vulnerability Hierarchy, the Formal Implementation Vulnerability Model, and the Characteristic-Based Vulnerability Classification Scheme have standalone benefits, but are designed to work together collectively. We take advantage of this in the Policy-Based Vulnerability Analysis Framework to locate, analyze, and mitigate implementation vulnerabilities.

Consider the Policy-Based Vulnerability Hierarchy, which provides a policy-based definition of an implementation vulnerability in chapter 4. An implementation vulnerability enables an unequivocal policy violation between the configured and instantiated policy oracles. This suggests that if we are able to approximate these policy oracles in practice, we could detect implementation vulnerabilities by searching for these policy violations.

However, we know from the Formal Implementation Vulnerability Model in Appendix A that security is undecidable, and non-security is recognizable. In other words, we are unable to detect the absence of a policy violation, but we are able to detect the presence of a policy violation. The decidable problem of real-time security suggests a different way forward—by changing the question from *if* a policy violation occurs to *when* a policy violation occurs.

This approach still runs into problems. Examining all of the possible partial traces to determine if a system is real-time secure is computationally expensive, and working with Turing

machines is impractical. As a result, we turn to the Characteristic-Based Vulnerability Classification Scheme in chapter 5 to provide an abstraction of implementation vulnerabilities at a more practical level. The characteristic-based approach also suggests a way to locate these vulnerabilities—not by the policy violations they cause, but by their characteristics.

The Policy-Based Vulnerability Analysis Framework builds on the insights gained by each of these components. Rather than attempting to locate and disable individual implementation vulnerabilities, we concentrate on locating and disabling characteristics. We start by classifying a large number of IVECs, and use those results in both the analysis and mitigation phases of the framework.

§6.2 Analysis Framework

The Policy-Based Vulnerability Analysis Framework uses a policy-based notion of a vulnerability and characteristic-based approach to classification for implementation vulnerability analysis in practical settings. The objective of this framework is to locate, analyze, and mitigate implementation vulnerabilities in a given environment.

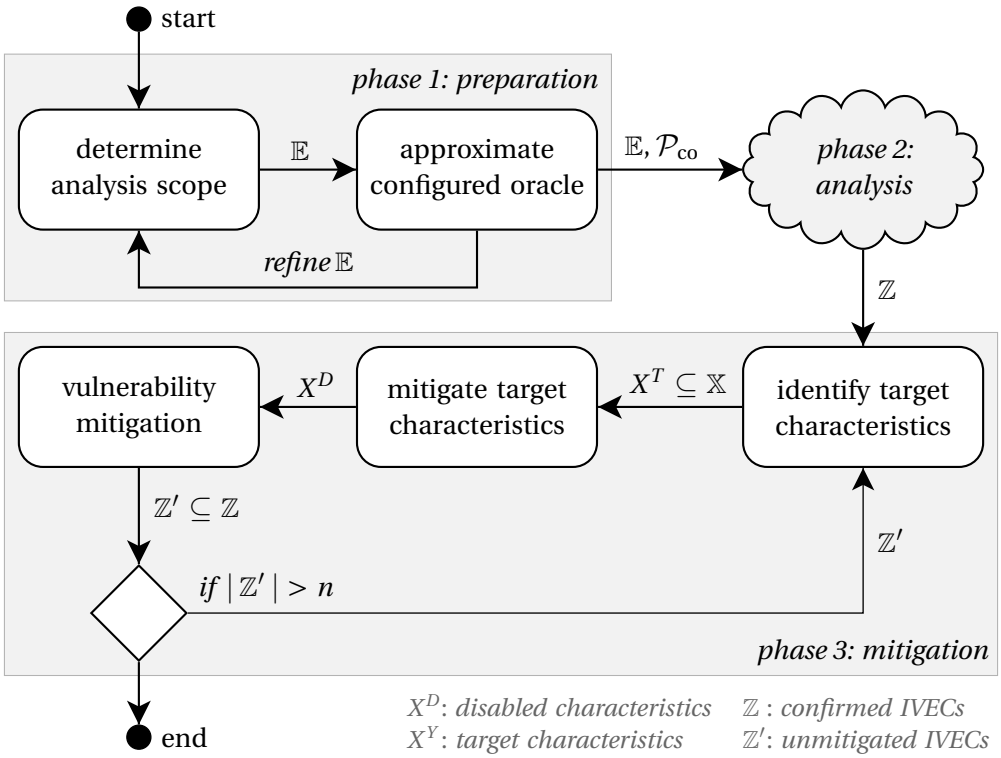
We break this process into three phases: preparation, analysis, and mitigation. The preparation phase determines the scope of the global policy event space $\mathbb{E}$ and approximates the configured oracle for $\mathbb{E}$. The analysis phase identifies a set of confirmed characteristics to discover implementation vulnerability equivalence classes (IVECs) and approximate the instantiated oracle. Finally, the mitigation phase attempts to mitigate the discovered IVECs by disabling their associated characteristics.

We summarize the entire process in Figure 6.1. Each phase consists of several complex steps, and must overcome several practical issues. We dive deeper into each of these phases in the next sections.

§6.2.1 Phase 1: Preparation

The preparation phase involves two primary steps: defining the scope of the global policy event space, and approximating the configured oracle for the policy events in the global policy event space. Whether we are able to approximate the configured and instantiated oracles depends on the global policy event space defined in this phase.

FIGURE 6.1: ANALYSIS FRAMEWORK OVERVIEW



A high-level diagram of the Policy-Based Vulnerability Analysis Framework. The analysis phase is expanded further in Figure 6.2. A more detailed key is available in Figure 6.3.

There are many factors that must be considered when defining the global policy event space, and several obstacles to approximating the configured oracle. We discuss these issues in detail in section 6.3. This phase does not take any input, and produces the global policy event space $\mathbb{E}$ and the configured policy oracle $\mathcal{P}_{co}$ as output.

§6.2.2 Phase 2: Analysis

The analysis phase is the most time consuming and complex. The primary purpose of this phase is to approximate the instantiated oracle and identify a set of implementation vulnerability equivalence classes present in the environment. We do this in four analysis steps, including instantiated oracle analysis, characteristic analysis, environment analysis, vulnerability analysis. Each of these steps may be iterated multiple times until the instantiated oracle has been fully approximated or until some threshold has been reached.

This phase takes as input the global policy event space $\mathbb{E}$ and configured policy oracle $\mathcal{P}_{co}$, generates an approximation of the instantiated oracle $\mathcal{P}_{in}$, and outputs a set $\mathbb{Z}$ of known IVECs complete with characteristics and symptoms. We provide more detail in section 6.4.

§6.2.3 Phase 3: Mitigation

In the mitigation phase, we take the set of IVECs from phase 2 and attempt to mitigate the vulnerabilities by mitigating their associated characteristics. These characteristics should ideally be disabled in the environment, but this may not always be practical. This phase ends when the number of unmitigated vulnerabilities drops below some threshold n, which is ideally set to zero. There is no specific output of this phase, except possibly the set of recommended mitigations. We discuss this phase in more detail in section 6.5.

In an abstract sense, this mitigation process addresses the real-time security of our environment. Recall from section A.3.3 that real-time security focuses on determining *when* a system is non-secure. By identifying when our environment is not secure and mitigating the risks associated with those vulnerabilities, we are reducing the instances that our environment is known to be non-secure.

§6.3 Phase 1: Preparation

We summarize the preparation phase in section 6.2.1, which identifies the scope of the implementation vulnerability analysis. All together, this phase consists of the following steps:

1. *Analysis Scope.* Identify the system(s) and security mechanism(s) to analyze and the associated global policy event space.
2. *Configured Oracle Approximation.* Approximate the configured policy oracle for each of the policy events in the global policy event space.

There is some iteration possible in this phase, but in most cases this iteration is not necessary. Once we have identified the analysis scope, we move on to phase 2.

§6.3.1 Step 1: Define Global Policy Event Space

An important factor in whether implementation vulnerability analysis is practical will be the scope of the global policy event space. Before we are able to specify this scope, however, we must define the target environment. This includes any systems and security mechanisms that will be included in the analysis process. This set should be kept to a small† number of representative systems, or the configured oracle may be too large and complex to feasibly approximate in step 2.

† We may even wish to limit our analysis to a single system.

We do not need to include policy events that result in an `unknown` response, since our analysis is focused on unequivocal policy violations. The configured oracle is only able to return a policy decision for system-defined subjects, objects, and actions, allowing us to restrict the global policy event space accordingly. However, due to the complexity of approximating the instantiated and configured oracles, we may have to further restrict the global policy event space. For example, we may decide to focus on the subset of objects protected by the operating system that may affect program flow and the set of actions capable of manipulating those objects.

When this step is complete, we should have a global policy event space $\mathbb{E}$ that is a subset of the system-defined subjects, objects, and actions in the environment. We output this set $\mathbb{E}$ to the next step.

§6.3.2 Step 2: Approximate Configured Policy Oracle

The next step is to approximate the configured policy oracle for the policy events in the global policy event space. We demonstrate in Appendix B.1 how to approximate the configured oracle for ACM-based security mechanisms. In general, we should be able to approximate the configured oracle by examining the current configurations of each system and security mechanism in the environment.

However, security mechanisms often interact in a complex environment with a porous security perimeter. We will need to limit our scope of analysis for this approximation to be practical, limiting the number of security mechanisms we may consider in the global policy event space. If the oracle approximation is too time consuming, we may return to step 1 and further reduce the global policy event space. Once the configured oracle approximation is complete, we output both the global policy event space $\mathbb{E}$ and configured oracle $\mathcal{P}_{\text{co}}$ to the next phase of the Policy-Based Vulnerability Analysis Framework.

§6.3.3 Phase 1 Summary

Phase 1 focuses on providing the scope of analysis, by restricting the global policy event space and providing an approximation of the instantiated policy oracle. The oracle approximation may be a complex process depending on the environment.

FIGURE 6.2: PHASE 2: ANALYSIS

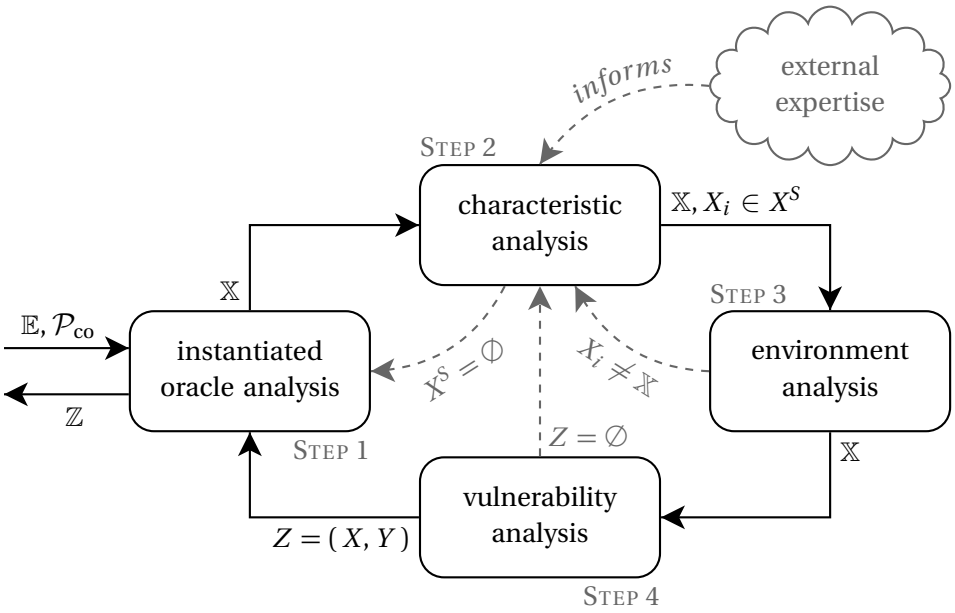


Diagram for the analysis phase. The process iterates until no more suspected characteristics are added, or we are confident in the $\mathcal{P}_{\text{in}}$ oracle approximation.

§6.4 Phase 2: Analysis

Once the preparation phase is complete, we are able to start our iterative analysis. The analysis phase attempts to approximate the configured oracle and determine a set of confirmed IVECs. We break this process into four steps:

1. *Instantiated Oracle Analysis.* Updates approximation of the instantiated oracle given the set of confirmed vulnerabilities.
2. *Characteristic Analysis.* Updates set of suspected characteristics based on the set of confirmed characteristics.
3. *Environment Analysis.* Determines whether a suspected characteristic exists in the system(s) and security mechanism(s) in the environment.
4. *Vulnerability Analysis.* Determines whether an IVEC exists based on the updated set of confirmed characteristics.

We continue to iterate between these steps until either we are satisfied with the approximation of the configured oracle or we no longer have an suspected characteristics to examine. We summarize these steps in Figure 6.2.

§6.4.1 Step 1: Instantiated Oracle Analysis

The first time we enter this step, we receive the global policy event space $\mathbb{E}$ and the approximated configured oracle as input from the preparation phase. We initialize the set $\mathbb{Z}$ of confirmed IVECs and the set $\mathbb{X}$ of confirmed characteristics to the empty set. Until there is a non-empty set of IVECs, we are unable to approximate the instantiated oracle. After initialization, we output the set $\mathbb{X}$ to the next step.

The next iteration of this step will receive an new IVEC $Z = (X, Y)$ from step 4. To approximate the instantiated oracle, we examine the symptoms of these IVECs and determine how they affect the operation of the security mechanism(s). The IVEC may or may not represent an actual risk, depending on the configured oracle. For example, suppose we are able to infer that $\mathcal{P}_{\text{in}}(E) = \texttt{no}$ from the basic set Y of symptoms. It is possible that the configured oracle does not allow this event, and thus no implementation vulnerability exists for this environment and security policy. However, if the configured oracle response is `yes`, then there exists an implementation vulnerability in the environment. We add this IVEC to our set $\mathbb{Z}$ and continue to the next step. Even if Z does not represent an implementation vulnerability in the environment, we do not remove the set X from the set $\mathbb{X}$ of confirmed characteristics.†

Once we are satisfied with the approximation of the implementation oracle, we may discontinue the analysis phase and output the final set $\mathbb{Z}$ of confirmed vulnerabilities. Otherwise, we output the set $\mathbb{X}$ to the next step. Ideally, we continue this process until all policy events have been approximated.

§6.4.2 Step 2: Characteristic Analysis

This step receives as input from step 1 the set $\mathbb{X}$ of confirmed characteristics. Assume for the moment that this set is not empty. We compare the set of confirmed characteristics with pre-existing classification results, captured in Figure 6.2 as the "external expertise" cloud. We are able to determine which characteristics tend to occur together, and build a set X^S of *suspected* characteristics to investigate. There are characteristics that we believe have a high likelihood of

† These characteristics may lead to another IVECs in the system, and therefore should never be removed from the set of confirmed characteristics.

occurring in the environment. For example, the `x:jmps` and `x:exes` characteristics usually occur together. Confirming one of these two characteristics is a good indicator that the other exists.

However, on the first iteration, the set of confirmed characteristics is empty. We run into a "chicken or the egg" problem here. We need the set $\mathbb{X}$ to form our set X^S of suspected characteristics. These suspected characteristics let build the set $\mathbb{Z}$ of IVECs, and hence form the set $\mathbb{X}$. We avoid this never ending cycle by relying on external expertise to inform our initial set of suspected characteristics. This may include previously confirmed characteristics in similar environments, or in a worst case scenario, all known characteristics.

From the set of suspected characteristics, we choose one to investigate. This may be a characteristic chosen at random from the set, or one we believe to be particularly likely to exist. We output this characteristic X_i to the next step and remove it from the set X^S of suspects. If the set of suspected characteristics reaches zero and we are unable to repopulate it using external knowledge, then we must return to step 1 and end the analysis phase prematurely.

§6.4.3 Step 3: Environment Analysis

Step 3 focuses on confirming whether X_i exists in the environment. To confirm the existence of certain characteristics, we may need access to the software, hardware, and source code for all of the systems and security mechanisms under consideration. This may also require tools such as source code analyzers. Finally, analysts must have an in-depth understanding of both the environment and the analysis tools to determine whether a characteristic exists.

Once we have confirmed the presence of the characteristic X_i, we add X_i to the set $\mathbb{X}$ of confirmed characteristics. We output the new set $\mathbb{X}$ to the next step. If we are unable to confirm the characteristic X_i exists in the environment, we return to step 2. This shortcut, depicted by a dashed line in Figure 6.2, allows for a new suspected characteristic to be chosen without having to iterate through steps 4 and 1 again.

§6.4.4 Step 4: Vulnerability Analysis

The vulnerability analysis step examines the set $\mathbb{X}$ of confirmed characteristics from step 3, and determines if there is a new IVEC possible in the environment. This is done by comparing the

set $\mathbb{X}$ with previously-classified IVECs. If we discover a $Z = (X, Y)$ such that $X \subseteq \mathbb{X}$, we verify that Z is present in the environment. Once verified, we output Z to step 1.

This phase receives the set $\mathbb{X}$ as input from step 3 only when a new characteristic has been added. Since IVECs are unique, there will only ever be one new IVEC discovered in this phase. If no such IVEC is discovered, we skip step 1 and take a shortcut directly to step 2.

§6.4.5 Phase 2 Summary

We summarize each step in phase 2 in Figure 6.2. This phase of the Policy-Based Vulnerability Analysis Framework is highly iterative, potentially continuing until all resources have been exhausted. However, at the end of the analysis phase, we have a set $\mathbb{Z}$ of confirmed IVECs complete with characteristics and symptoms.

§6.5 Phase 3: Mitigation

The final phase in the Policy-Based Vulnerability Analysis Framework is the mitigation phase. We attempt to mitigate the vulnerabilities discovered in phase 2 by disabling the underlying characteristics. We break this process into three steps:

1. *Characteristic Identification.* Identifies a target set of the most common or dangerous characteristics.
2. *Characteristic Mitigation.* Identifies how to disable or partially-disable each characteristic in the target set.
3. *Vulnerability Mitigation.* Compares the set of disabled or partially-disabled characteristics with the set of known vulnerabilities.

We repeat this process with characteristics from the set of unmitigated vulnerabilities until all vulnerabilities are mitigated, or until we reach some threshold (depicted as n in Figure 6.1).

§6.5.1 Step 1: Characteristic Identification

We maximize the impact our mitigations have by focusing on those characteristics that will disable the largest number of vulnerabilities, or are associated with the most severe symptoms. To build the set X^T of target characteristics, we first identify a subset of the most commonly occurring characteristics from the set $\mathbb{Z}$. Let Z_{X_i} be the set of IVECs containing characteristic X_i:

$$Z_{X_i} = \{ Z = (X, Y) : X_i \in X \}$$

Let $g \leq |\mathbb{Z}|$ be an integer threshold. We build the set of common characteristics X_g as follows:

$$X_g = \{ X_i \in \mathbb{X} : |Z_{X_i}| \geq g \}$$

Disabling a characteristic from the set X_g will help mitigate the affects of at least g different IVECs. However, we are also concerned with characteristics that are associated with particularly troublesome symptoms. For example, we may wish to focus on blocking a vulnerability that disables privileges for all subjects in an environment. Therefore, we also build a set of target symptoms Y^T from the set $\mathbb{Y}$ of confirmed symptoms. To determine which symptoms we should target, we must look at the policy violations captured by these symptoms. For example, we may only be interested in symptoms that escalate privileges, or disable privileges for a large set of subjects.

We use the set Y^T of target symptoms to build a set Z^T of target IVECs. Informally, we include any IVEC that shares a symptom with the target set. Specifically:

$$Z^T = \{ Z = (X, Y) : Y \cap Y^T \neq \emptyset \}$$

We build the resulting set of target characteristics, based on the set of target symptoms, by taking the union of all characteristics from the set of target vulnerabilities. More formally:

$$X_{Z^T} = \bigcup_{(X,Y) \in Z^T} X$$

This allows us to include characteristics associated with the most troublesome symptoms. We form the entire set X^T of target characteristics as follows:

$$X^T = X_g \cup X_{Z^T}$$

Ideally, the set of target characteristics should be kept to a practical size. If we end up with too many characteristics to analyze, we can reduce the size of the target set by increasing the threshold g or reducing the size of Y_T. Once an appropriate set of target characteristics has been identified, we output X^T to the next step.

§6.5.2 Step 2: Characteristic Mitigation

This step of the mitigation process attempts to mitigate all of the characteristics in the target set. This step receives as input the set X^T, and outputs the set of disabled or partially disabled characteristics X^D to the next step.

Ideally, we completely disable each characteristic in the target set such that $X^D = X^T$. However, in practice, this process is more complex than simply disabling or removing the characteristic. We must compare each characteristic with known defenses, and determine whether these defenses are viable in our environment. This may involve modifications to the code, installing additional security mechanisms, or changing the security mechanisms in use. The defenses themselves may only partially disable the characteristic, and may require extensive resources.

In cases where an appropriate system-level mitigation is not available, we must resort to higher-level procedures to address these characteristics. For example, we may have to shift from a "disable and prevent" mindset to a "monitor and react" approach to dealing with some of these characteristics. We only include those characteristics that we are able to appropriately mitigate in the set X^D of characteristics, which we output to the next step.

§6.5.3 Step 3: Vulnerability Mitigation

This step compares the set $\mathbb{Z}$ of known IVECs with the set X^D of mitigated characteristics to form a set $\mathbb{Z}'$ of unmitigated vulnerabilities. Recall from chapter 5 that disabling a characteristic disables the associated vulnerability. Therefore, we determine if an IVEC is mitigated by seeing if any characteristics from the basic characteristic set are mitigated in step 2. Specifically, the set of mitigated vulnerabilities is the set Z^D such that:

$$Z^D = \{ Z = (X, Y) \in \mathbb{Z} : X \cap X^D \neq \emptyset \}$$

We form the set of unmitigated vulnerabilities as follows:

$$\mathbb{Z}' = \mathbb{Z} \setminus Z^D$$

Ideally, $\mathbb{Z}' = \emptyset$ and we are able to discontinue our analysis. Otherwise, we must determine whether to continue iterating through this phase to reduce the size of $\mathbb{Z}'$. We identify a threshold n and continue iterations until either $|\mathbb{Z}'| \geq n$ or we are no longer able to mitigate any more characteristics in step 2.

§6.5.4 Phase 3 Summary

This phase of the Policy-Based Vulnerability Analysis Framework focuses on mitigating the discovered IVECs by mitigating the associated characteristics. We may iterate through this phase multiple times, until we are satisfied by the mitigations identified in step 2.

§6.6 Analysis Example

We illustrate this implementation vulnerability analysis framework with a hypothetical example. We use the same electronic voting environment introduced in section 4.4. A registered voter makes his or her selections on an electronic touch screen device, which prints those selections onto a paper ballot. The registered voter then places this ballot in a ballot box, which is taken by the poll workers to the election headquarters for counting. We focus our analysis on the electronic ballot printers. We simplify the operation of these electronic ballot printers so that we may demonstrate the entire analysis process.

§6.6.1 Phase 1: Preparation

The first step in the preparation phase is to identify the global policy event space. Recall that the configured oracle is only able to return a policy decision for system-defined subjects, objects, and actions. As such, we only need to consider the set of subjects, objects, actions, and conditions defined by an electronic ballot printer for the global policy event space.

There are two user accounts defined on this device: `root` and `anon`. We also include the process `init` in our universal set of subjects and objects. This process runs when the device is powered on, and handles most of the functionality on the ballot printer. There is a single system-defined `ballot` object, and two actions that may be performed on that object: `config()` and `print()` to configure or print the ballot respectively. We also define two other actions: `auth()` to authenticate as a particular user account, and `input()` to get a password string from the user. We specify the Boolean conditions in the universal condition set as needed in our analysis.

With the global policy event space defined, we must now approximate the configured oracle. We assume a closed policy; a policy event is not allowed unless otherwise specified. For example, the electronic touch screen ballot printer is configured such that the `root` user account requires a password, but the `anon` user account does not. The set of allowed policy events are:

$$\mathcal{P}_{\text{co}}(\texttt{init}, \texttt{root}, \texttt{auth()}, \texttt{input()} \equiv \texttt{passwd}) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\texttt{init}, \texttt{anon}, \texttt{auth()}, \texttt{true}) = \texttt{yes}$$

We are primarily concerned with how the `ballot` object may be manipulated. The printer is configured such that only the `root` user account is allowed to perform the `config()` on the

`ballot` object. However, both the `root` and `anon` user accounts may perform the `print()` action on the `ballot` object, which triggers the device to print the current set of selections onto a paper ballot. The allowed policy events include:

$$\mathcal{P}_{\text{co}}(\texttt{root}, \texttt{ballot}, \texttt{config()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\texttt{root}, \texttt{ballot}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\texttt{anon}, \texttt{ballot}, \texttt{print()}, \texttt{true}) = \texttt{yes}$$

This completes our approximation of the configured oracle for the global policy event space. We output $\mathbb{E}$ and $\mathcal{P}_{\text{co}}$ to the next phase of this framework.

§6.6.2 Phase 2: Analysis

The analysis phase iterates between four steps: instantiated oracle analysis, characteristic analysis, environment analysis, and vulnerability analysis. In the first iteration of this process, we do not have the necessary information to perform an instantiated oracle analysis. As such, we initialize the set of confirmed IVECs and characteristics to the empty set, and move to the characteristic analysis step.

The first time we perform characteristic analysis, we must draw from an external source to initialize the list of suspected characteristics. We assume we have access to a database with the characteristic-based classifications of several implementation vulnerability equivalence classes. The ballot printer software is written in the C language. Therefore, we initialize the set of suspected characteristics to include all characteristics that are known to exist in C programs. This includes the set of buffer overflow characteristics developed in section 5.3.7.

The next step, environment analysis, confirms whether the suspected characteristics exist in the environment. In this example, suppose we discover that the `x:buff` characteristic exists anywhere the user is allowed to enter a write-in candidate. We know this characteristic is associated with buffer overflow vulnerabilities, and begin to focus our search for those characteristics. We also know that with this architecture, the characteristics associated with the `X:CONTAINS` and `X:EXECUTE` characteristic classes exist. At the end of this step, we are able to confirm that several of the characteristics for a buffer overflow vulnerability exist in the environment. The next step is to determine whether a specific buffer overflow vulnerability exists.

The third step in this process is to take the set of confirmed characteristics in our environment, and determine if these lead to an actual vulnerability. This requires is to match these characteristics to specific instances in the code. Suppose we discover that the `init` process sets a Boolean flag `isroot` to `true` when the `root` user account properly authenticates. Whether the `config()` action may be triggered depends on this flag. By overflowing the buffer for the first write-in candidate, an attack is able to change the contents of this flag. Specifically, we verify specific instances of the following buffer overflow characteristics:

- `x:buff` exists for the write-in buffer
- `x:path` exists for the variable `isroot`
- `x:type` exists for the write-in user input
- `x:vval` exists for the variable `isroot`

This matches the definition of a direct data buffer overflow IVEC from section 5.3.7. Therefore, we are able to add this buffer overflow vulnerability to our list of confirmed IVECs.

Once we have confirmed the presence of an IVEC, we must update our approximation of the instantiated oracle. In this case, the direct data buffer overflow vulnerability allows the `anon` user account to trigger the `config()` action. Therefore:

$\mathcal{P}_{\text{co}}(\texttt{anon}, \texttt{ballot}, \texttt{config()}, \texttt{true}) = \texttt{no}$

$\mathcal{P}_{\text{in}}(\texttt{anon}, \texttt{ballot}, \texttt{config()}, \texttt{true}) = \texttt{yes}$

Once we are satisfied with the approximation of the instantiated oracle, we may move on to the mitigation phase of this framework.

§6.6.3 Phase 3: Mitigation

In the analysis phase, we confirmed the presence of a direct data buffer overflow IVEC. We attempt to mitigate this confirmed vulnerability in the final phase of this analysis framework. We begin by examining the characteristics associated with this IVEC. Normally, we may have to prioritize which characteristics we consider for mitigation. However, for this hypothetical example, we may consider all four of the characteristics: `x:buff`, `x:type`, `x:vvar`, `x:path`.

There are several changes that may be made to the source code on the ballot printer to disable this vulnerability. The first should be to address the buffer overflow bug and remove the `x:buff` characteristic. While we only need to disable a single characteristic, we may also wish

to address the others as well. For example, we may restrict the type of characters that may be entered by the user for write-in candidates. This mitigates the characteristic `x:type` by reducing the ability of an attacker to upload the same data type as the flow variable into memory.

Until the code changes are completed and installed on this device, we may also decide to enact new security procedures to limit, detect, and react to successful attacks. For example, we attempt to detect a successful attack at the polling place by emphasizing the need for voters to verify their selections on the ballot. These procedures mitigate the discovered vulnerability by disabling related policy events at higher levels in the policy hierarchy. See section 4.4 for an example of this type of higher-level analysis.

§6.6.4 Example Summary

We demonstrated the Policy-Based Vulnerability Analysis Framework using the scenario developed in chapter 4 and characteristics introduced in chapter 5. If the same IVEC database is used to develop the set of suspected characteristics, the set of confirmed characteristics should be consistent between different analysts given enough time. However, the order that characteristics and IVECs are confirmed will vary from analyst to analyst in practice. If the analysis is cut short due to limited resources, the final set of confirmed IVECs may differ slightly.

§6.7 Summary

This chapter introduces the Policy-Based Vulnerability Analysis Framework, which is comprised of the Policy-Based Vulnerability Hierarchy, Formal Implementation Vulnerability Model, and Characteristic-Based Vulnerability Classification Scheme. The Policy-Based Vulnerability Analysis Framework consists of three primary phases:

1. Preparation
2. Analysis
3. Mitigation

The preparation phase identifies the scope of the global policy event space and approximates the configured oracle. The analysis phase identifies a set of implementation vulnerability equivalence classes by approximating the instantiated oracle in an iterative process. Finally, the miti-

FIGURE 6.3: TERMINOLOGY AND NOTATION

Name	Notation	Reference
Global Policy Event Space	$\mathbb{E} = \mathbb{S} \times \mathbb{O} \times \mathbb{A} \times \mathbb{B}$	3.3 (p39)
Configured Policy Oracle	$\mathcal{P}_{\text{co}}$	4.5 (p48)
Instantiated Policy Oracle	$\mathcal{P}_{\text{in}}$	4.6 (p49)
Implementation Vulnerability Equivalence Class (IVEC)	$Z = (X, Y)$	5.22 (p80)
Characteristics	X	5.11 (p75)
Symptoms	Y	5.13 (p76)
Confirmed Characteristics	$\mathbb{X}$	
Suspected Characteristics	X_S	§6.4 (p96)
Targeted Characteristics	X_T	§6.5 (p99)
Disabled Characteristics	X_D	§6.5 (p99)
Confirmed IVECs	$\mathbb{Z}$	§6.4 (p96)
Unmitigated IVECs	$\mathbb{Z}'$	§6.5 (p99)

Summary of the terminology and notation used throughout chapter 6.

gation phase attempts to mitigate the discovered vulnerabilities by disabling the associated characteristics. Figure 6.1 and Figure 6.3 summarize the overall framework.

We have designed the Policy-Based Vulnerability Analysis Framework to be both repeatable and practical, but our results demonstrating these properties are preliminary. We discuss how well this framework meets our objectives in the next chapter.

CHAPTER 7
Conclusion

This dissertation introduces the Policy-Based Vulnerability Analysis Framework to provide a repeatable and practical framework for vulnerability analysis. We summarize this framework and our contributions below, reflect on how well we met our original objectives, and discuss directions for future research.

§7.1 Summary

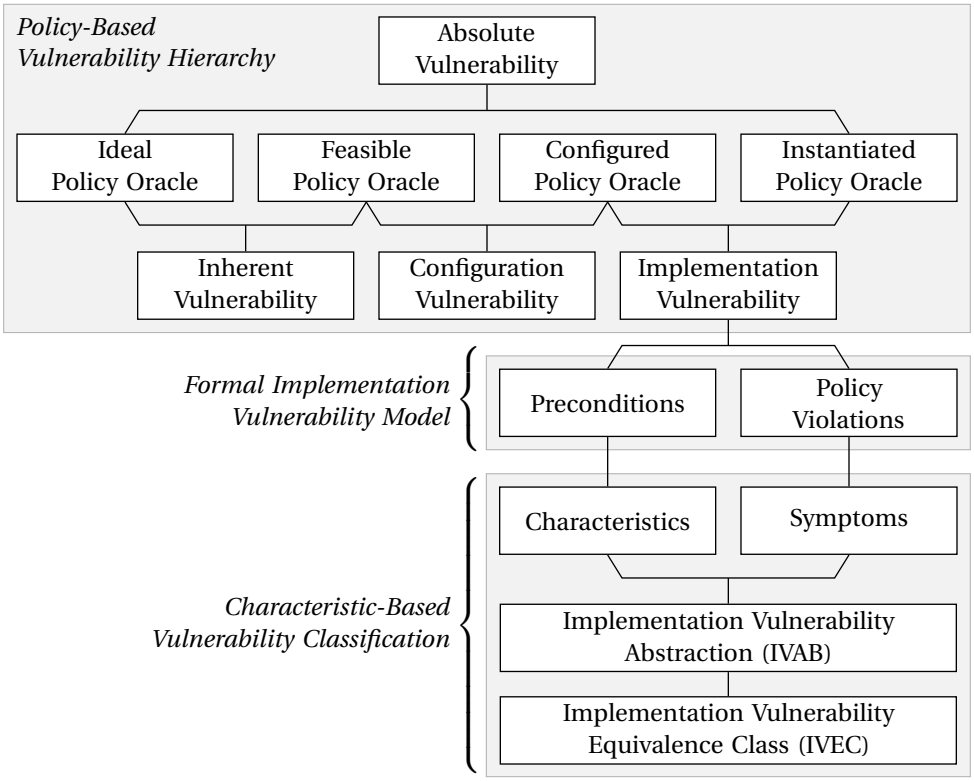
This dissertation introduces the Policy-Based Vulnerability Analysis Framework, which provides a policy-based analysis framework for the discovery and mitigation of implementation vulnerabilities. We build this framework from three major components: the Policy-Based Vulnerability Hierarchy, the Formal Implementation Vulnerability Model, and the Characteristic-Based Vulnerability Classification Scheme.

The Policy-Based Vulnerability Hierarchy in chapter 4 defines a security policy at four levels of abstraction: ideal, feasible, configured, and instantiated. We explore the gaps between these levels of a security policy to define three different types of a vulnerability: inherent, configuration, and implementation. This provided a hierarchical policy-based notion of a vulnerability that is able to capture both security procedures and security mechanisms.

We examine the lower levels of the Policy-Based Vulnerability Hierarchy with the Formal Implementation Vulnerability Model in Appendix A. This model defines a security policy as a language of authorized configurations, instead of a partition of states. We use this model to motivate a shift in thinking from determining *if* a system is secure with respect to a security policy, to determining *when* a system is non-secure.

The Formal Implementation Vulnerability Model provides the formal foundation for the Characteristic-Based Vulnerability Classification Scheme in chapter 5. We formally define im-

FIGURE 7.1: TERMINOLOGY OVERVIEW



An overview of the major concepts introduced in this dissertation. Most of this terminology is introduced in chapter 4 and chapter 5.

plementation vulnerabilities using preconditions and policy violations, and then abstract these notions to characteristics and symptoms. We rely on these reversible levels of abstraction to move away from the "perfect knowledge assumption" that comes with theoretical environments, and perform hierarchical characteristic-based vulnerability classification at a practical level. We use these results in the framework to inform where we should focus our vulnerability analysis.

The Policy-Based Vulnerability Analysis Framework integrates these components into a cohesive implementation vulnerability analysis framework in chapter 6. The entire process consists of three phases: preparation, analysis, and mitigation. The preparation phase identifies the scope of the global policy event space, and approximates the configured oracle. The analysis phase iterates between instantiated oracle analysis, characteristic analysis, environment analysis, and vulnerability analysis, to identify confirmed implementation vulnerability equivalence classes and their associated characteristics and symptoms. The mitigation phase attempts to mitigate these discovered implementation vulnerability equivalence classes by disabling prevalent or severe characteristics.

§7.2 Contributions

This dissertation introduces the Policy-Based Vulnerability Hierarchy, Formal Implementation Vulnerability Model, and Characteristic-Based Vulnerability Classification Scheme, and integrates these components into a cohesive framework for implementation vulnerability analysis. We discuss the contributions of each of these components and the entire framework in this section, and reflect on how these contributions address our overall objectives.

§7.2.1 Vulnerability Hierarchy

The Policy-Based Vulnerability Hierarchy reflects the different levels of security policy encountered in practice, and captures both security procedures and security mechanisms. This approach separates intention from implementation, providing valuable insight into where and why vulnerabilities occur without breaking well-established intuition. As a result, we are able to use the Policy-Based Vulnerability Hierarchy to inform the type of techniques we may use to prevent, mitigate, or defend against these vulnerabilities. For example, Klein et al. used formal verification to eliminate the class of implementation vulnerabilities for the seL4 microkernel [KLE09]. However, the work on enforceable security policies illustrates that only a subset of security policy is enforceable [SCH00, HAM06]. As a result, vulnerabilities at higher-levels in the hierarchy may be impossible or impractical to fully eliminate.

The Policy-Based Vulnerability Hierarchy expands the Unifying Policy Hierarchy [CAR06]. We discuss the specific contributions we made to the original model in section 4.5. This hierarchy allows us to meet our primary objective by providing a policy-based notion of a vulnerability.

§7.2.2 Vulnerability Model

The Formal Implementation Vulnerability Model demonstrates a different approach for formally defining a security policy for a state machine. Specifically, we define a security policy as a partition of configurations instead of a partition of states. This approach allows us to express a security policy at this level of abstraction without having to modify the underlying state machine, and better models how a security policy is expressed in a realistic environment. As such, the Formal Implementation Vulnerability Model allows us to achieve our primary objective by providing a strong theoretical foundation for our framework that reflects practice.

§7.2.3 Vulnerability Classification

The Characteristic-Based Vulnerability Classification Scheme builds on prior work to provide a hierarchical characteristic-based approach for classification of implementation vulnerabilities at a practical level of abstraction [BIS99, WHA05]. We introduce the perfect knowledge assumption, which makes explicit the theoretical assumptions that may not hold in a realistic environment. We use reversible layers of abstraction to develop characteristics and symptoms, and move away from the perfect knowledge assumption. We specify where ambiguity may be introduced in this process, and the affects this has on the soundness and completeness of this model. Finally, we expand on our previous buffer overflow work [BIS10] by providing revised buffer overflow characteristics and a master characteristic tree.

The Characteristic-Based Vulnerability Classification Scheme allows us to achieve our objectives by providing a practical level of abstraction for implementation vulnerabilities, and repeatability with reversible layers of abstraction.

§7.2.4 Vulnerability Analysis

The Policy-Based Vulnerability Analysis Framework brings together the policy-based approach of Carlson [CAR06], our earlier theoretical model [ENG08B], the characteristic-based approach of Bishop [BIS99], and the hierarchical approach of Whalen et al. [WHA05] into a single, cohesive framework for practical and repeatable implementation vulnerability analysis. To achieve this cohesion, we have refined each of these approaches to use consistent terminology based on a strong theoretical foundation.

The Policy-Based Vulnerability Analysis Framework enables targeted implementation vulnerability analysis for stable, small-scale environments. As a result, we are able to address the gap between repeatability and practicality, and encapsulate where and when ambiguity is introduced when moving between theoretical and realistic settings.

§7.3 Future Work

We meet our objectives with the Policy-Based Vulnerability Analysis Framework in this dissertation, but there are still several interesting research directions to pursue. We discuss these directions for future research in this section.

§7.3.1 Theoretical Directions

We used the Formal Implementation Vulnerability Model in chapter 5 to provide a theoretical foundation for implementation vulnerabilities, and expand this model in Appendix A to demonstrate the computability of security. Exploring additional theoretical results using this model is outside the scope of this dissertation, but various opportunities for future research exist.

For example, one possible direction for future research is to explore other theoretical system models. We used Turing machines to model the computational capability of modern systems. However, in practice, systems do not have infinite time and memory for computations. Instead, we could use **linear bounded automata** as a foundation for our vulnerability model. The size of the tape for these automata is linearly bounded by the length of the input, and have a finite number of distinct configurations [SIP97, $p178$]. Some problems, such as the acceptance problem, are undecidable for Turing machines but decidable for linearly bounded automata [SIP97, $p177$].

Whether we are able to decide if a system is secure with respect to a security policy depends both on the type of machine and computability class of the security policy. Another future research direction is to investigate other computability classes for security policies. For example, we use regular languages for the policy condition examples in chapter 3.

Even if a security problem is decidable, it may not be tractable. We investigate security, non-security, and real-time security in Appendix A, and find that real-time security is decidable. However, real-time security is likely intractable. We may wish to extend this research to cover time and space complexity as well, and explore other problems in security in search for problems that are both decidable and tractable in practice.

§7.3.2 Vulnerability Database

We are able to improve our ability to perform targeted implementation vulnerability analysis in chapter 6 by providing more characteristic-based classification results. An important future direction for this research is to develop a database of vulnerabilities and their resulting classification for a large number and wide array of vulnerabilities. We may use as a starting point the protocol vulnerability classification work introduced in chapter 2, the work on buffer overflow characteristics introduced in chapter 5, and several existing vulnerability databases such as the Common Weakness Enumeration (CWE) Dictionary [CWE09], the Common Attack Pattern

Enumeration and Classification (CAPEC) Catalog [CAP10], and the Database Of Vulnerabilities, Exploits, and Signatures (DOVES) [DOV98].

§7.3.3 Extended Case Study

We demonstrated preliminary results that the Policy-Based Vulnerability Analysis Framework is capable of repeatable analysis results in a realistic environment. The next step is to apply the framework on a larger scale with an extended case study.

However, we wish to avoid the obstacles associated with performing this type of analysis on a live environment. Instead, we focus on applying the framework to a detailed, hypothetical environment. We propose forming four disjoint teams, consisting of an environment team, and three analysis teams: alpha, beta, and control. The environment team develops the hypothetical environment, complete with specific open-source systems and software, procedures, and high-level security policies. Each analysis team receives the same information and analysis toolkits from the environment team. Both the alpha and beta analysis teams must perform an independent analysis using the Policy-Based Vulnerability Analysis Framework, whereas the control team may use any other framework. The environment team collects and compares the analysis results, examining both the number and consistency of the results across all three teams.

Electronic voting provides an ideal environment for this type of case study, especially since there has already been work on the detailed definition of an election process [SIM08]. For a specific county, the environment is generally well-defined with specific security requirements, specific operating environments (including both the polling places and election headquarters), and specific subjects, actions, and objects. Additionally, there are several open-source electronic voting systems available for analysis.

FIGURE 7.2: GENERAL NOTATION (ALPHABETICALLY)

Aa	a action, $\mathbb{A}$ universal action set
Bb	b Boolean condition, $\mathbb{B}$ universal condition set
Cc	C classification tree (vulnerability, characteristic, or symptom),
	$\mathbb{C}$ master classification tree
Dd	$d \in D$ policy decision
Ee	E policy event, $\mathbb{E}$ global policy event space
Ff	f function, F computable function
Gg	G graph
Hh	h configuration, H computation history
Ii	i integer e.g. ith element, $\mathcal{I}$ system oracle
Jj	j integer e.g. jth element
Kk	k integer e.g. kth element
Ll	L language
Mm	M Turing machine, $\mathbb{M}$ representative system set
Nn	n size (usually of a set or string)
Oo	o object, $\mathbb{O}$ universal object set
Pp	P policy set, $\mathcal{P}$ policy oracle
Qq	q Turing machine state, Q set of states
Rr	$r \in R$ policy response
Ss	s subject, S policy statement, $\mathbb{S}$ universal subject set
Tt	t precondition, T precondition set, $\mathbb{T}$ system precondition set
Uu	u policy violation, U policy violation set, $\mathbb{U}$ system violation set
Vv	V implementation vulnerability
Ww	w word or input string
Xx	$x \in \Sigma$ symbol or character, X characteristic or characteristic set,
	$\mathcal{X}$ characteristic oracle, $\mathbb{X}$ universal characteristic set
Yy	Y symptom or symptom set, $\mathcal{Y}$ symptom oracle,
	$\mathbb{Y}$ universal symptom set
Zz	Z vulnerability equivalence class
$\Lambda\alpha$	α access attribute, δ transition function
	Σ input alphabet, Γ tape alphabet, Λ access control matrix

General notation used throughout dissertation. When discussing specific publications and models, there may be overlapping notation.

APPENDIX A

Formal Implementation Vulnerability Model

We formally define implementation vulnerabilities in chapter 5 for use in the Characteristic-Based Vulnerability Classification Scheme. We expand this formalization in this chapter with the Formal Implementation Vulnerability Model.

§A.1 Introduction

We present a model of security for Turing machines in this chapter. Turing machines are the most powerful and widely used theoretical model of computation. While Turing machines may not necessarily mimic the operation of modern systems, they are often used to model the computational capability of modern systems. We provide a brief introduction to Turing machines and how they may be used to study decidability in chapter 2.

The Formal Implementation Vulnerability Model reflects our policy-based approach for theoretical settings, as discussed in chapter 3. Instead of defining a security policy as a partition of states, we define a security policy as a language of configurations. This approach better reflects the type of scenarios we target for vulnerability analysis.

The primary purpose of this model is to provide a formal definition of implementation vulnerabilities and a theoretical foundation for the Characteristic-Based Vulnerability Classification Scheme in chapter 5. However, we are also able to use this model to explore the decidability of security without making assumptions about the underlying security mechanism. We demonstrate this by providing several decidability results in this chapter.

§A.2 Terminology

We introduced the notions of a computation trace, partial trace, and language of valid configurations in chapter 2. We formally define these notions in this section, and show their decidability

using the theorems and definitions from Sipser [SIP97, p191–193]. We use the notation ϵ to denote the empty string. The notation w_j refers to the jth string in the lexicographical ordering of Σ^*. For example, if $\Sigma = \{ 0, 1 \}$, the lexicographical ordering of Σ^* is the sequence:

ϵ, 0, 1, 00, 01, 10, 11, ...

§A.2.1 Computation Trace

We use the Turing machine computation trace to track all of the configurations entered by a Turing machine on an input. If the Turing machine halts on the input, the trace is equivalent to the computation history. However, if the machine does not halt, the trace tracks the potentially infinite set of configurations entered by that machine. Formally:

DEFINITION A.1: Let M be a Turing machine. The **trace** of M on input w is the potentially infinite sequence of configurations $(h_0, h_1, \ldots, h_i, h_{i+1}, \ldots)$ that M enters when computing w such that:

- h_0 is the start configuration $q_0\, w$, and
- each h_i yields h_{i+1}.

DEFINITION A.2: Let M be a Turing machine. The language ***TRACE(M , w)*** is the set of configurations:

$$TRACE(M, w) = \{ h_i : h_i \text{ is in the trace of } M \text{ on input } w \}$$

The language $TRACE(M, w)$ captures the unique configurations entered regardless of order. We find that the language is recursively enumerable and undecidable. We demonstrate each of these results individually, starting with the following theorem:

THEOREM 1.1: The language $TRACE(M, w)$ for a Turing machine M and input w is recursively enumerable.

We demonstrate that $TRACE(M, w)$ is recursively enumerable with an enumerator E_{trace} that is able to enumerate this language. We define E_{trace} as follows:

E_{trace} = "On input $\langle M, w \rangle$, where M is a Turing machine:

1. For $i = 0, 1, 2, \ldots$
2. Simulate input w on M for i steps.
3. Output the current configuration uqv of M.
4. If uqv is a halting configuration, halt."

We demonstrate that E_{trace} is a valid enumerator for *TRACE*(M, w) as follows:

- Suppose $uqv \in$ *TRACE*(M, w). We know Turing machine M eventually enters configuration uqv when computing w. Let uqv be the ith configuration entered by M on input w. The enumerator E_{trace} on input $\langle M, w \rangle$ will output uqv on the ith iteration. Therefore:

  $$uqv \in \textit{TRACE}(M, w) \implies uqv \in L(E_{\text{trace}}(M, w))$$

- Suppose $uqv \notin$ *TRACE*(M, w). We know Turing machine M never enters configuration uqv when computing w. The enumerator E_{trace} on input $\langle M, w \rangle$ only outputs configurations entered by M on input w, and will never output configuration uqv. Therefore:

  $$uqv \notin \textit{TRACE}(M, w) \implies uqv \notin L(E_{\text{trace}}(M, w))$$

Therefore, *TRACE*(M, w) is recursively enumerable by definition. Next, we examine the undecidability of *TRACE*(M, w):

THEOREM 1.2: The language *TRACE*(M, w) for a Turing machine M and input w is undecidable.

The "Instantaneous Description Problem" is a similar to *TRACE*(M, w) and is known to be undecidable [DEN96]. Denning et al. define this problem as:

$$\text{ID}(M, h_i, h_j) = \begin{cases} 1 & \text{if } M \text{ ever reaches configuration } h_j \text{ when started in } h_i \\ 0 & \text{otherwise} \end{cases}$$

We use a similar approach to demonstrate *TRACE*(M, w) is undecidable. Specifically, we show a reduction from A_{TM} to *TRACE*(M, w). Let M' be a computable function:

M' = "On input $\langle M, w \rangle$, where M is a Turing machine:

1. Simulate M on w.
2. If M accepts w, then:
3. Clear the tape.
4. Enter accept state q'_{a}.
5. Else, enter reject state q'_{r}."

We demonstrate that $A_{\text{TM}} \leq_{\text{m}}$ *TRACE*(M, w) as follows:

- Suppose that $\langle M, w \rangle \in A_{\text{TM}}$. In this case, M' will clear the tape and halt on state q'_{a}. Since the tape is empty, the final configuration is $\epsilon\, q'_{\text{a}}\, \epsilon$, or equiva-

lently the string q'_{a}. Therefore, the trace of M' computing w will include the configuration q'_{a}. As a result, we claim:

$$\langle\, M, w \,\rangle \in A_{\text{TM}} \implies q'_{\text{a}} \in \mathit{TRACE}(\, M', w \,)$$

- Alternatively, suppose that $\langle\, M, w \,\rangle \notin A_{\text{TM}}$. In this case, M' will not enter the accept state q'_{a}. As a result, the tape is not cleared and the accept state q'_{a} is never entered by M' when computing w. Therefore:

$$\langle\, M, w \,\rangle \notin A_{\text{TM}} \implies q'_{\text{a}} \notin \mathit{TRACE}(\, M', w \,)$$

Therefore, $\mathit{TRACE}(\, M, w \,)$ is undecidable by mapping reducibility.

§A.2.2 Partial Trace

In situations where we need a decidable language of configurations, we define the notion of a *partial trace*. A partial trace only tracks the computation of a Turing machine for a fixed number of steps. More specifically:

DEFINITION A.3: Let M be a Turing machine. The **partial trace** of M on input w is the finite sequence of configurations $(\, h_0, h_1, \ldots, h_k \,)$ that M enters when computing w for n steps such that:

- h_0 is the start configuration $q_0\, w$,
- each h_i yields h_{i+1} for $i < n$, and
- h_k is a halting configuration (such that $k < n$), or
- h_k is the nth configuration (such that $k = n$)

DEFINITION A.4: Let M be a Turing machine. The language ***PARTIAL(M, w, n)*** is the set of configurations:

$$\mathit{PARTIAL}(\, M, w, n \,) = \{\, h_i \,:\, h_i \text{ is in the partial trace of } M \text{ on input } w \text{ for } n \text{ steps} \,\}$$

Like a computation history, the partial trace is always finite. Specifically:

THEOREM 1.3: The language $\mathit{PARTIAL}(\, M, w, n \,)$ for a Turing machine M, input w, and positive integer n is decidable.

Any finite language is decidable, and $\mathit{PARTIAL}(\, M, w, n \,)$ is finite, as:

$$|\, \mathit{PARTIAL}(\, M, w, n \,) \,| \leq n + 1$$

§A.2.3 Valid Configurations

For a Turing machine $M = (\, Q,\, \Sigma,\, \Gamma,\, \delta,\, q_0,\, q_\text{a},\, q_\text{r}\,)$, we know that any valid configuration must be in the form $\Gamma^* \, Q \, \Gamma^*$. However, this does not mean all of these configurations are entered when the Turing machine M computes. For example, suppose the only transition to our accepting state q_a is from q_i as follows:

$$\delta(\, q_i,\, x\,) \;=\; (\, q_\text{a},\, \sqcup,\, \text{R}\,)$$

We expect M to only enter configurations of the form $\Gamma^* \;\sqcup\; q_\text{a}\, \Gamma^*$ during computation. Therefore, the configuration $0\, q_\text{a}\, 0$ is considered an invalid configuration for M. This illustrates that a valid configuration for one Turing machine may not be valid for another. We state that any valid configuration with respect to a specific Turing machine M must be one that M is capable of entering during computation. Formally:

> DEFINITION A.5: Let M be a Turing machine. A **valid configuration** with respect to M is any configuration entered by M during computation.

> DEFINITION A.6: Let M be a Turing machine. The language ***VALID*(*M*)** is the set of configurations:
>
> $$\mathit{VALID}(\, M\,) \;=\; \{\, h_i \;:\; h_i \;\in\; \mathit{TRACE}(\, M,\, w\,) \text{ for some input } w \,\}$$

We can restate the language $\mathit{VALID}(\, M\,)$ as the union of $\mathit{TRACE}(\, M,\, w\,)$ for all input strings w:

$$\bigcup_{w \,\in\, \Sigma^*} \mathit{TRACE}(\, M,\, w\,)$$

The language $\mathit{TRACE}(\, M,\, w\,)$ is countable, and the union of all countable sets is countable. Therefore, we expect this language to be enumerable:

> THEOREM 1.4: The language $\mathit{VALID}(\, M\,)$ for a Turing machine M is recursively enumerable.

We demonstrate that $\mathit{VALID}(\, M\,)$ is recursively enumerable with an enumerator E_valid that is able to enumerate this language. Let w_j be the jth string in the lexicographical ordering of Σ^*. We define E_valid as follows:

E_valid = "On input $\langle\, M\,\rangle$, where M is a Turing machine:

1. For $i \;=\; 1,\, 2,\, \ldots,\, \infty$:
2. For $j \;=\; 1,\, \ldots,\, i$:

3. Simulate M on w_j for $k = i - j$ steps, or
   until M halts on w_j.
4. Output the current configuration h_k of M."

During the first iteration, E_{valid} will output configuration h_0 for string w_1. In the second iteration, E_{valid} will output h_1 for string w_1 and configuration h_0 for string w_2. By dovetailing the different inputs, we are able to output a diagonalization of $\mathit{VALID}(M)$ as illustrated in Figure A.1. Halting configurations may be output multiple times by E_{valid}, but do not affect the language $L(E_{\text{valid}})$.

We demonstrate that $E_{\text{valid}}(M)$ enumerates the language $\mathit{VALID}(M)$ as follows:

- Suppose $uqv \in \mathit{VALID}(M)$. There exists some input w_j such that $uqv \in \mathit{TRACE}(M, w_j)$. Let uqv be the kth computation in the trace of M on w_j. The enumerator E_{valid} will output uqv when $i = k + j$. Therefore, we state:
  $$uqv \in \mathit{VALID}(M) \implies uqv \in L(E_{\text{valid}}(M))$$
- Suppose $uqv \notin \mathit{VALID}(M)$. The Turing machine M never enters configuration C_k on any output. The enumerator E_{valid} on input $\langle M \rangle$ only outputs configurations entered by M on some input w_j, and will never output configuration uqv. Therefore, we state:
  $$uqv \notin \mathit{VALID}(M) \implies uqv \notin L(E_{\text{valid}}(M))$$

By definition, the language $\mathit{VALID}(M)$ is recursively enumerable. Next, we examine the undecidability of $\mathit{VALID}(M)$:

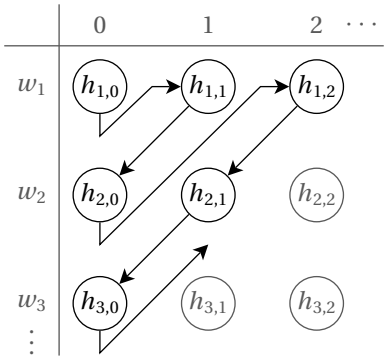
THEOREM 1.5: The language $\mathit{VALID}(M)$ is undecidable.

We can use the same reduction as $\mathit{TRACE}(M, w)$ to demonstrate that $\mathit{VALID}(M)$ is undecidable. Specifically, we can reduce A_{TM} to $\mathit{VALID}(M)$ using the same computable function M' from section A.2.1. For example, if M accepts w, then the configuration q'_{a} is a valid configuration of M'. The machine M' never enters q'_{a} on any other input. As a result, q'_{a} is not a valid configuration if M does not accept input w. By mapping reducibility, $\mathit{VALID}(M)$ is undecidable.

§A.3 Security Problems

We reduce security problems, such as determining if or when a system is secure, to determining if a system belongs to a specific class of languages. This allows us to apply techniques from

FIGURE A.1: DIAGONALIZATION OF *VALID(M)*



Let $h_{j,k}$ be the kth element of the trace of M on input w_j. The enumerator E_{valid} outputs the sequence ($h_{1,0}$, $h_{1,1}$, $h_{2,0}$, $h_{1,2}$, $h_{2,1}$, $h_{3,0}$, ...) and so on.

the theory of computation to determine if our problem is solvable. For example, consider the **emptiness problem**, which tries to decide whether the language of a Turing machine is empty. We restate the emptiness problem as the language $\boldsymbol{E_{\mathbf{TM}}}$:

$$E_{\text{TM}} = \{ \langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset \}$$

The language for a Turing machine M is empty if and only if $M \in E_{\text{TM}}$. Since we know that the language E_{TM} is undecidable, we know the emptiness problem is unsolvable.

§A.3.1 System Security

Intuitively, a Turing machine is secure when it has no vulnerabilities. If there are no vulnerabilities in the machine, then there are no policy violations. This is only true when the machine is unable to enter an unauthorized configuration. Following this line of reasoning, we state a machine is *secure* if and only if it enters only those configurations authorized by its security policy:

> DEFINITION A.7: The language $\boldsymbol{SECURE_{\mathbf{TM}}}$ is the language of Turing machines and security policies such that:
>
> $$SECURE_{\text{TM}} = \{ \langle M, P \rangle : VALID(M) \subseteq L(P) \}$$

This reduces the problem of determining if a Turing machine is secure to determining if the machine and its security policy belong to this language. Specifically:

> DEFINITION A.8: A Turing machine M is **secure** with respect to a security policy P if and only if $\langle M, P \rangle \in SECURE_{\text{TM}}$.

We know from pre-established decidability results that security is undecidable [HAR76], and expect the same to hold here:

THEOREM 1.6: The language $SECURE_{\text{TM}}$ is not recursively enumerable, and hence undecidable.

We demonstrate that $SECURE_{\text{TM}}$ is undecidable with the mapping reduction from the language E_{TM} to the language $SECURE_{\text{TM}}$. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{a}}, q_{\text{r}})$, and M'' be a computable function defined as follows:

M'' = "On input $\langle M \rangle$, where M is a Turing machine:

1. Construct a Turing machine P as follows:
2. P = 'On input $\langle M, w \rangle$:
   a. If $w \in \Gamma^* q_{\text{a}} \Gamma^*$, reject w.
   b. Otherwise, accept w.'
3. Output $\langle M, P \rangle$."

We construct P such that $w \in L(P)$ only if w is *not* an accepting configuration. The membership test in step 2a is decidable, since regular expressions are decidable languages. Therefore, P is a decider Turing machine and may be used as the security policy for M. We demonstrate that $A_{\text{TM}} \leq_{\text{m}} TRACE(M, w)$ as follows:

- Notice that if $L(M)$ is empty, then it never enters an accepting configuration. Therefore, M will never violate the security policy P. This demonstrates that:
  $$\langle M \rangle \in E_{\text{TM}} \implies \langle M, P \rangle \in SECURE_{\text{TM}}$$
- Suppose that $L(M)$ is not empty. The Turing machine M must be able to enter an accepting configuration, which would violate its security policy P. This demonstrates that:
  $$\langle M \rangle \notin E_{\text{TM}} \implies \langle M, P \rangle \notin SECURE_{\text{TM}}$$

Therefore, the language $SECURE_{\text{TM}}$ is undecidable by mapping reducibility. This language is also not recursively enumerable:

THEOREM 1.7: The language $SECURE_{\text{TM}}$ is not recursively enumerable.

We are able to use the same mapping reduction $E_{\text{TM}} \leq_{\text{m}} SECURE_{\text{TM}}$ and computable function M'' to demonstrate that the language $SECURE_{\text{TM}}$ is not recursively enumerable. Given these results, we know that we are unable to determine if a Turing machine M is secure with respect to a decidable security policy P.

§A.3.2 System Non-Security

The complement to system security is system non-security, which asks if our system is *not* secure. Specifically, it is the language:

DEFINITION A.9: The language $\boldsymbol{UNSECURE_{TM}}$ is the defined as:

$$UNSECURE_{\text{TM}} = \overline{SECURE_{\text{TM}}} = \{ \langle M, P \rangle : VALID(M) \not\subseteq L(P) \}$$

We already know from Theorem 1.6 that security is undecidable, and hence non-security is undecidable. However, while the language $SECURE_{\text{TM}}$ is not recursively enumerable, the complement $UNSECURE_{\text{TM}}$ is:

THEOREM 1.8: The language $UNSECURE_{\text{TM}}$ is recursively enumerable.

We demonstrate that $UNSECURE_{\text{TM}}$ with a Turing machine $M_{\overline{\text{sec}}}$ that is able to recognize this language. Let E_{valid} be the enumerator from section A.2.3. We define $M_{\overline{\text{sec}}}$ as:

$M_{\overline{\text{sec}}}$ = "On input $\langle M, P \rangle$, where M is a TM and P is a decider:

1. Enumerate $VALID(M)$ using E_{valid}.
2. If E_{valid} outputs a configuration $uqv \notin L(P)$, reject."

Since P is a decider, we know that step 2 is decidable. If M does eventually enter an unauthorized configuration, the Turing machine $M_{\overline{\text{sec}}}$ will reject. This demonstrates that:

$$\langle M, P \rangle \in L(M_{\overline{\text{sec}}}) \implies \langle M, P \rangle \in UNSECURE_{\text{TM}}$$

Therefore, $M_{\overline{\text{sec}}}$ recognizes the language $UNSECURE_{\text{TM}}$. By definition, $UNSECURE_{\text{TM}}$ is recursively enumerable.

The problem of system security asks the unsolvable question, "Is my system secure with respect to my policy?" The problem of computer non-security asks the recursively enumerable question, "Is my system non-secure with respect to my policy?" By refining our question, we increase our ability to find an answer. We take this a step further and explore the question, "When is my system non-secure?" We capture this with *real-time security* in the next section.

§A.3.3 Real-Time Security

We may be unable to determine if a system is secure, but we are able to determine if our systems are non-secure. However, in practice, the answer to this question is a forgone conclusion. It is safe assumption that any modern general purpose system is non-secure.

In light of this, it is more useful to determine *when* a system is non-secure. The answer to this question provides actionable information on how to better protect our systems. Our notion of *real-time security* is one step towards providing this answer. Specifically, we define the language:

DEFINITION A.10: The language $\boldsymbol{RTSECURE_{TM}}$ is the set:

$$RTSECURE_{\text{TM}} = \{ \langle M, P, w, i \rangle : PARTIAL(M, w, i) \subseteq L(P) \}$$

Essentially, a machine is real-time secure if the Turing machine is secure up to the current computation step. More formally:

DEFINITION A.11: Let i be the current computation step of a Turing machine M on input w. We say M is **real-time secure** with respect to a security policy P if and only if:

$$\langle M, P, w, i \rangle \in RTSECURE_{\text{TM}}$$

As we know from Theorem 1.3, the language $PARTIAL(M, w, i)$ is decidable. Therefore, we expect the language $RTSECURE_{\text{TM}}$ to be decidable as well:

THEOREM 1.9: The language $RTSECURE_{\text{TM}}$ is decidable.

We know that by definition P is a decider Turing machine, and that $L(P)$ is decidable. For every configuration $uqv \in PARTIAL(M, w, i)$ we are able to decide if $uqv \in L(P)$. We also know that $PARTIAL(\ )$ is finite from Theorem 1.3, and therefore we only have a finite number of these membership tests. Therefore, $RTSECURE_{\text{TM}}$ decidable.

As a result, we are able to detect exactly when a system is non-secure. We are able to find the earliest computation step that causes a Turing machine to violate its security policy, and provide the partial trace of the Turing machine up to that point. We use this information to motivate our approach to implementation vulnerability analysis in chapter 6.

§A.4 Discussion

In practice, we do not have infinite time and computers do not have infinite memory. Even those problems that are solvable by Turing machines may require too much space or time to solve in practice. Furthermore, modern systems and security policies are too complex to formally define as Turing machines.

FIGURE A.2: DECIDABILITY RESULTS

Language	Recursively Enumerable	Co-Recursively Enumerable	Decidable
VALID(M)	Yes	No	No
TRACE(M, w)	Yes	No	No
PARTIAL(M, w, n)	Yes	Yes	Yes
$SECURE_{\mathrm{TM}}$	No	Yes	No
$UNSECURE_{\mathrm{TM}}$	Yes	No	No
$RTSECURE_{\mathrm{TM}}$	Yes	Yes	Yes

Decidability results for each language. For example, $SECURE_{\mathrm{TM}}$ is not recursively enumerable, co-recursively enumerable, and undecidable.

However, theoretical work does place an upper bound on what is achievable in computer security and suggests where to focus research. Our initial theoretical results demonstrate that some questions are more practical to explore than others.

For example, theoretical work often tackles the question, "Is my system secure?" Depending on the underlying system and environment, this may be the appropriate question to ask. However, even in cases where this question may be decidable, it is not necessarily feasible to solve in practice. And, for most generic-purpose modern computers, we already know the answer—these systems are *not* secure.

This leads to a binary notion of security. Either a system is a member of $SECURE_{\mathrm{TM}}$ or it is not. A single vulnerability can compromise an entire system. As long as a single vulnerability exists, our systems are not secure. However, this approach hides the complexity of security. As Bishop states, "evaluating security in this fashion evades the purpose of providing it" [BIS03B].

Instead, asking the question, "When is my system not secure?" better matches how security is addressed in practice. We identify the issues that weaken our system security, and attempt to address those weaknesses with prevention and detection mechanisms. This provides actionable information that allows us to increase our confidence in the security of these systems. Instead of a binary notion of security, we treat security as an upper bound and strive to get as close to that bound as possible.

The notion of real-time security reflects this shift in thinking. With real-time security, we are trying to understand *when* our systems are not secure. We use this intuition when performing implementation vulnerability analysis in chapter 6.

FIGURE A.3: TERMINOLOGY AND NOTATION

Name	Notation		Reference
Turing Machine	$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$		§2.4.3 (p25)
Computation History	$H = (h_0, h_1, \ldots, h_k)$		§2.4.6 (p28)
Policy Condition	$\{ uqv : uqv$ is authorized $\}$		3.12 (p41)
State Condition	and $u \circ v \in \Sigma^*$		3.15 (p42)
Tape Condition	and $q \in Q$		3.16 (p43)
Trace	$TRACE(M, w)$	for input w	A.2 (p115)
Partial Trace	$PARTIAL(M, w, n)$	for input w and n steps	A.4 (p117)
Valid Configurations	$VALID(M)$	for Turing machine M	A.6 (p118)
Security Policy	$L(P)$ where P is a decider TM		3.14 (p42)
Security Problems			§A.3 (p119)
Security	$SECURE_{TM} = \{ \langle M, P \rangle : VALID(M) \subseteq L(P) \}$		A.7 (p120)
Non-Security	$UNSECURE_{TM} = \overline{SECURE_{TM}}$		A.9 (p122)
Real-Time	$RTSECURE_{TM} = \{ \langle M, P, w, i \rangle : PARTIAL(M, w, i) \subseteq L(P) \}$		A.10 (p123)
Vulnerability	$V = (T, U)$	(*implementation-level*)	5.3 (p69)
Preconditions	$T \subseteq PARTIAL(M, P, w, i-1)$		5.2 (p69)
Policy Violations	$U \subseteq (VALID(M) \ominus L(P))$		5.1 (p69)

Summary of the terminology and notation used throughout chapter A. Languages and security problems are typeset in capital italics with the base system model in the subscript. For example, A_{TM} is the acceptance problem for Turing machines (TM).

§A.5 Summary

The Formal Implementation Vulnerability Model provides the theoretical foundation necessary for the Characteristic-Based Vulnerability Classification Scheme, which is an integral part of our vulnerability analysis framework. The Formal Implementation Vulnerability Model defines security policy in chapter 3 as a language of configurations, and implementation vulnerabilities in chapter 5 as a combination of preconditions and policy violations. In this chapter, we utilize this model to define real-time security for a Turing machine, and demonstrate various decidability results. We use these decidability results to motivate a shift from determining *if* a system is secure to detecting *when* a system is non-secure in chapter 6.

APPENDIX B

Discussion

We expand our discussion on various topics in this chapter. Specifically, we illustrate how to approximate the configured oracle introduced in chapter 4 for an ACM-based security mechanism, motivate our choice in chapter 5 for formalizing the basic characteristic set, and detail the original and revised buffer overflow characteristics used in chapter 5.

§B.1 Approximating Access Control Matrix Models

We introduced the Access Control Matrix (ACM) Model [LAM71, GRA71] in chapter 2. An access control matrix is often used to capture the access control policy of security mechanisms. While our framework does not use an access control matrix, we are able to capture the policy represented by one with the configured oracle.

Consider the protection system model of Graham and Denning and the eight rules they present [GRA71]. We use the same notation as the original paper, except that we use the notation Λ for the the access control matrix. Let S be the set of subjects in that matrix, and X be the set of objects in the matrix. We use $s \in S$ to denote a single subject and $x \in X$ to denote an object defined in the access control matrix. We use α to denote an access attribute and α^* to denote an access attribute with the copy flag set, which indicates a subject may copy or transfer the attribute to another user. We demonstrate how to form the global policy event space and approximate the configured oracle for this ACM-based protection system model next.

§B.1.1 Global Policy Event Space

We must approximate the global policy event space before we are able to approximate the configured oracle. We derive the universal sets of subjects and objects directly from the access control

matrix, and derive the universal set of actions from access attributes and the rules that govern how the access control matrix may be manipulated.

We assume the access control matrix initially includes the *universal subject*. The universal subject is a subject without an owner and has every possible access attribute to every object, similar to the notion of an administrator or root user. Therefore, the initial set S will contain at least one subject. We derive the universal subject set directly from this set:

$$\mathbb{S} = S$$

Subjects are also considered objects by the access control matrix. For example, a process may both perform an action, or be the target of an action. The access control matrix itself is also considered an object. This allows us to capture actions which operate on the access control matrix by adding new rows, columns, or modifying entries. Given this, we initialize the universal set of objects as follows:

$$\mathbb{O} = S \cup X \cup \{ \Lambda \}$$

We infer actions from the access attributes present in the access control matrix Λ, and from the set of eight rules governing how the access control matrix is modified. The access attributes indicate the access privileges for a particular subject/object pair. For example, the access attribute $\texttt{w} \in \Lambda[\, s, x \,]$ may indicate that subject s is authorized to perform the `write` action on the object x. We state that for every unique access attribute α, we add the associated set of actions to our universal action set $\mathbb{A}$.

The exact set of access attributes depends on the specific protection system in question. However, there is a minimal set of actions and attributes inferable from the set of rules. These rules control how the access control matrix may be modified. For each rule, we add the associated command to our universal action set $\mathbb{A}$. More specifically, we add the following actions for modifying access attributes in the matrix:

- `read`$(\, s, \ x \,)$: read the set of access attributes for subject s and object x from the matrix $\Lambda[\, s, x \,]$ (inferred from rule R4)
- `grant`$(\, \alpha, \ s, \ x \,)$: grant the access attribute α to subject s and object x such that $\alpha \in \Lambda[\, s, x \,]$ (inferred from rule R2)
- `transfer`$(\, \alpha, \ s, \ x \,)$: transfer the access attribute α to subject s and object x such that $\alpha \in \Lambda[\, s, x \,]$ (inferred from rule R1)

- `delete`(α, s, x): delete the attribute α (or α^* if present) from subject s and object x such that $\alpha \notin \Lambda[\, s, x \,]$ and $\alpha^* \notin \Lambda[\, s, x \,]$ (inferred from rule R3)

The `grant` and `transfer` actions perform as expected when dealing with access attributes with the copy flag. We also include actions that allow for the structure of the matrix to be modified. We use the following actions to remove elements from the matrix:

- `destroy_obj`(x): deletes the column for object x in Λ (inferred from rule R6)
- `destroy_sbj`(s): deletes the row for subject s in Λ and performs the action `destroy_obj`(s) to delete object s (inferred from rule R8)

To add new subjects and objects, we track which subject is performing the action such that we may update the `owner` attribute in access control matrix. Specifically, we add:

- `create_obj`(s, x): creates a column in Λ for object x and adds the `owner` attribute such that `owner` $\in \Lambda[\, s, x \,]$ (inferred from rule R5)
- `create_sbj`(s, $\acute{s}$): creates a new row in Λ for subject $\acute{s}$, performs the action `create_obj`(s, $\acute{s}$) to create a new column in Λ for object $\acute{s}$, and adds the `control` access attribute such that `control` $\in \Lambda[\, \acute{s}, \acute{s} \,]$ (inferred from rule R7)

All of these actions are added to our universal action set $\mathbb{A}$.

§B.1.2 Configured Oracle

We are now able to infer the configured oracle for this protection system from the set of rules. These rules define when specific actions are authorized based on the contents of the access control matrix. All of these actions operate on the access control matrix itself.

Rule R1 allows the transfer of access attributes if the copy flag of that attribute is set. This is a modification of the access control matrix itself, and thus Λ is our object. The configured oracle captures this behavior with the following:

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{transfer}(\, \alpha,\, \acute{s},\, x \,),\, \alpha^* \in \Lambda[\, s, x \,]\,) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{transfer}(\, \alpha,\, \acute{s},\, x \,),\, \alpha^* \notin \Lambda[\, s, x \,]\,) = \texttt{no}$$

This states that the `transfer` action is allowed under the Boolean condition that the access attribute α^* is in the access control matrix. For example, subject `yasmin` may transfer the `read` attribute for object `log.txt` to `xander` if `yasmin` has `read`* privileges for `log.txt`. If `yasmin` transfers the `read`* attribute to `xander`, then `xander` may transfer the privilege to other subjects.

The Rule R2 allows a subject to grant access attributes, with the copy flag if desired, to others for any object it holds the `owner` attribute. Specifically:

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{grant}(\, \alpha,\, \acute{s},\, x\,),\, \texttt{owner} \in \Lambda[\, s, x\,]\,) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{grant}(\, \alpha,\, \acute{s},\, x\,),\, \texttt{owner} \notin \Lambda[\, s, x\,]\,) = \texttt{no}$$

We handle transferring or granting access attributes with the copy flag in the same way:

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{grant}(\, \alpha^*,\, \acute{s},\, x\,),\, \texttt{owner} \in \Lambda[\, s, x\,]\,) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{grant}(\, \alpha^*,\, \acute{s},\, x\,),\, \texttt{owner} \notin \Lambda[\, s, x\,]\,) = \texttt{no}$$

Rule R3 allows the deletion of attributes based on the `owner` and `control` attributes.† We capture this with the statements:

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{delete}(\, \alpha,\, \acute{s},\, x\,),\, \texttt{control} \in \Lambda[\, s,\, \acute{s}\,]\,) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{delete}(\, \alpha,\, \acute{s},\, x\,),\, \texttt{owner} \quad \in \Lambda[\, s,\, x\,]\,) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\, s,\, \Lambda,\, \texttt{delete}(\, \alpha,\, \acute{s},\, x\,),\, \texttt{control} \notin \Lambda[\, s,\, \acute{s}\,] \wedge \texttt{owner} \notin \Lambda[\, s,\, x\,]\,) = \texttt{no}$$

If `yasmin` has the `owner` attribute for `log.txt`, then `yasmin` may grant or delete the `read` attribute from `xander`. Similarly, if `root` has the `control` attribute for the subject `yasmin`, then the user `root` may grant or delete attributes from `yasmin`. We use the logical and operator $\wedge$ to determine when the `delete` option should be disallowed, but other logically equivalent methods for defining the configured oracle for this rule exist. We are able to capture the remaining rules using a similar approach.

§B.1.3 Other Access Attributes

The configured policy thus far captures how the access control matrix may be modified. However, the protection system is likely to have other access attributes which indicate when an action should be allowed. In general, if an access control matrix Λ has an attribute α that indicates action a is authorized for some subject s and object $o \in \mathbb{O}$, we are able to capture this with:

$$\mathcal{P}_{\text{co}}(\, s,\, o,\, a,\, \alpha \in \Lambda[\, s, o\,]\,) = \texttt{yes}$$

$$\mathcal{P}_{\text{co}}(\, s,\, o,\, a,\, \alpha \notin \Lambda[\, s, o\,]\,) = \texttt{no}$$

This allows us to illustrate the gap between the feasible and configured policies. For example, suppose we authorize `yasmin` to `read` the `log.txt` file. Therefore:

$$\mathcal{P}_{\text{fe}}(\, \texttt{yasmin},\, \texttt{log.txt},\, \texttt{read},\, \texttt{true}\,) = \texttt{yes}$$

† The `control` attribute only applies to subjects.

At the configured level, the policy depends on the access control matrix:

$$\mathcal{P}_{\text{co}}(\texttt{yasmin}, \texttt{log.txt}, \texttt{read}, \texttt{r} \in \Lambda[\texttt{yasmin}, \texttt{log.txt}]) = \texttt{yes}$$

If the access control matrix is properly populated such that $\texttt{r} \in \Lambda[\texttt{yasmin}, \texttt{log.txt}]$, then the instantiated policy matches our feasible policy. However, suppose that the access control matrix becomes corrupted (either accidentally or maliciously), and as a result of this corruption the attribute $\texttt{r} \notin \Lambda[\texttt{yasmin}, \texttt{log.txt}]$. The instantiated policy becomes:

$$\mathcal{P}_{\text{in}}(\texttt{yasmin}, \texttt{log.txt}, \texttt{read}, \texttt{true}) = \texttt{no}$$

Notice that this does *not* violate the configured policy. Since the attribute `r` is no longer in the access matrix for `yasmin`, she should not have the ability to `read` the file `log.txt`. However, this does not match the policy we intended to enact. The policy hierarchy is able to capture this as a problem with the configuration, and not necessarily with the implementation.

§B.2 Basic Set Formalization

Bishop introduces the notion of a *basic characteristic set* as a "unique, sound characteristic set of minimal size" [Bis99]. We formalize this notion in section 5.3.5 using concepts from the Formal Implementation Vulnerability Model. However, there are multiple possible formalizations of a basic characteristic set, each with different advantages and disadvantages. We expand on these different formalizations in this chapter. We focus on characteristic sets in our discussion, but the same holds true for symptom sets.

§B.2.1 Minimal Set Cover

One possible formalization is to define a basic characteristic set as a minimal set cover of preconditions. We introduce the notion of a set cover in section 2.4.7. This results in the following definition of a basic set:

> Definition B.1: Given a set of preconditions $T \subseteq \mathbb{T}_M$ for system $M \in \mathbb{M}$, the **basic characteristic set** is the set $X_T \subseteq \mathbb{X}$ such that X_T is a minimal set cover of T for system M; i.e. the following holds:
>
> $$\bigcup_{X_i \in X_T} \mathcal{X}(M, X_i) = T$$

Using a minimal set cover to define a basic characteristic set, we perfectly capture the precondition set. For example, suppose we have a set of preconditions T defined for system M, the

associated basic characteristic set X_T, and the set of preconditions T' that are mapped by the characteristics in X_T for system M. We may calculate T' as follows:

$$T' = \bigcup_{X_i \in X_T} \mathcal{X}(M, X_i)$$

Using Definition B.1, the sets T and T' are exactly equal:

$$\bigcup_{X_i \in X_T} \mathcal{X}(M, X_i) = T$$

Therefore, whether we take a top-down or bottom-up approach to classification, we are dealing with the same set of underlying preconditions.

However, in practice, this approach has disadvantages. There are several *NP*-complete problems when it comes to dealing with subsets, including the set covering problem (including the exact cover problem) and the set packing problem [KAR72]. We introduce the set covering problem in chapter 2, which attempts to find a minimal set cover. The set packing problem attempts to find the largest pairwise disjoint cover of a set of sets. Both of these problems are related to the process of finding a basic characteristic or symptom set—a sound basic characteristic set is a pairwise disjoint minimal set cover.

We must avoid having to solve any *NP*-complete problems if we want our abstractions to be used in practice. We may turn to approximation algorithms in practice if necessary, but we believe that the soundness property of our universal sets allows us to avoid solving an *NP*-complete problem altogether. Specifically, since the universal sets are sound, each element may appear at most once in any subset. We may attempt to find a set cover by examining the preconditions mapped by every characteristic in our universal set. We claim that any set cover we find will be both pairwise disjoint and minimal, as there is only one unique set cover possible.

This highlights the importance of having a sound universal characteristic set. However, a sound universal characteristic set reduces the likelihood that a set cover exists for our set of preconditions. Consider the following example:

$$\mathbb{X} = \{ \mathbb{X}_a, \mathbb{X}_d, \mathbb{X}_e, \mathbb{X}_g \} \qquad \mathcal{X}(M, X_a) = \{ a, b, c \}$$
$$\mathbb{T} = \{ (M, \mathbb{T}_M) \} \qquad \mathcal{X}(M, X_d) = \{ d \}$$
$$\mathbb{T}_M = \{ a, b, c, d, e, f, g, h \} \qquad \mathcal{X}(M, X_e) = \{ e, f \}$$
$$T = \{ c, d, e, f \} \qquad \mathcal{X}(M, X_g) = \{ g, h \}$$

The universal characteristic set $\mathbb{X}$ is both sound and complete, and has fewer characteristics than preconditions in the universal precondition set $\mathbb{T}$. However, we are unable to form a minimal set cover for T in this example. If we use the set $\{ X_d, X_e \}$, then precondition c remains uncovered. If we include X_a in the set, we also include preconditions a and b which are not in set T. Either way, there is no set cover of characteristics for the preconditions T. To find a set cover, we must modify the universal characteristic set by splitting X_a into two characteristics—a process that may be time consuming in practice.

It may be too difficult in practice to develop a universal characteristic set that is both sound and able to cover the basic characteristic sets for every vulnerability being analyzed. We may loosen the restriction placed by using a minimal set cover for the basic characteristic set, but this loosens the relationship between preconditions and characteristics. We discuss the advantages and disadvantages of this approach next.

§B.2.2 Minimal Superset Cover

In the earlier example, we are unable to perfectly cover the set of preconditions T. If we do not modify the universal characteristic set, we have two options:

- Cover a maximal subset of T. For example, we are able to perfectly cover the subset of preconditions $\{ d, e, f \}$ with the set cover $\{ X_d, X_e \}$. In this case, we do not include precondition $c \in T$.
- Cover a minimal superset of T. For example, we are able to perfectly cover the superset $\{ a, b, c, d, e, f \}$ with the set cover $\{ X_a, X_d, X_e \}$. In this case, we include preconditions $a, b \notin T$.

Both approaches have downsides—but we choose the second approach due to practical considerations. Suppose we cover a minimal superset of T. Then the characteristic X_a is included in the basic characteristic set. If we choose to disable the characteristic X_a in the system, we still disable the associated vulnerability. However, we remove more preconditions than necessary. Alternatively, assume we choose the first approach. The characteristic X_a is not included in the basic characteristic set, removing a possible defense vector. We may also miss similarities between vulnerabilities that would have otherwise included characteristic X_a.

We capture the second approach by including *any* characteristic that intersects with our precondition set:

DEFINITION B.2: Given a set of preconditions $T \subseteq \mathbb{T}_M$ for system $M \in \mathbb{M}$, the **basic characteristic set** is the set $X_T \subseteq \mathbb{X}$ such that:

$$X_T = \{ X_i \in \mathbb{X} : \mathcal{X}(M, X_i) \cap T \neq \emptyset \}$$

This formalization is more practical, as it requires less modification to the universal characteristic set. We examine each characteristic in our universal characteristic set, and test whether the intersection is empty. We avoid any *NP*-complete problems as before, and do not need to refine the universal characteristic set. There will always be a basic characteristic set given our universal characteristic set is sound and complete.

However, we no longer perfectly capture the underlying precondition set. For example, consider the precondition set:

$$T = \{ c, d, e, f \}$$

Using Definition B.2, the set of preconditions covered by X_T for system M are:

$$T' = \{ a, b, c, d, e, f \}$$

This approach captures more preconditions than necessary in the basic characteristic set. Therefore, the set of underlying preconditions differs whether we take a top-down or bottom-up approach to classification. We believe the advantages outweigh the disadvantages in practice, especially since we rarely explicitly define preconditions in practice.

§B.3 Buffer Overflow Characteristics

We introduced the initial work by Bishop et al. on developing buffer overflow characteristics in chapter 2 [BIS10], and introduced revised buffer overflow characteristics in chapter 5. We summarize the original characteristics from this work here, and map them to our revised characteristics. We assume these characteristics were developed for the representative system set:

$$\mathbb{M} = \{ M : M \text{ runs a Unix-like operating system} \}$$

§B.3.1 Original Characteristics

The original buffer overflow work defines eighteen characteristics for four types of buffer overflow vulnerabilities: direct executable, indirect executable, direct data, and indirect data. A **di-**

rect executable buffer overflow vulnerability modifies the function pointer on the stack, and are captured by the following five characteristics:

- P1: The length of the uploaded string is longer than that of the buffer.
- P2: The string may contain instructions and/or addresses.
- P3: Input can modify the stored return address.
- P4: The program can jump to memory in the stack.
- P5: The program can execute instructions stored in the stack.

An **indirect executable** buffer overflow vulnerability occurs when a function pointer variable on the heap is modified. We capture these vulnerabilities with:

- P6: The length of the uploaded string is longer than that of the buffer.
- P7: The string may contain addresses.
- P8: Input can modify the value in the function pointer.
- P9: The program can jump to the heap.
- P10: The program can execute instructions in the heap.

A **direct data** buffer overflow vulnerability occurs when the program flow is altered by modifying the value of that variable. The following characteristics capture this:

- P11: The length of the uploaded string is longer than that of the buffer.
- P12: The string may contain data of the same type of the variable.
- P13: The value stored in the variable may be modified.
- P14: The variable determines which execution path is taken at a future point in the execution of the process.

Finally, a **indirect data** buffer overflow vulnerability occurs when the address or pointer to a control flow variable is modified, altering the program flow. Five characteristics are defined for this type of vulnerability:

- P15: The length of the uploaded string is longer than that of the buffer.
- P16: The string may contain addresses.
- P17: The address stored in the pointer variable can be modified.
- P18: The value pointed to by the pointer variable determines which execution path is taken at a future point in the execution of the process.

Even though the characteristics are defined informally, each of these characteristics are based on aspects of the code—and thus we are reasonably sure that they may be translated into implementation preconditions. For example, let n be the size of the destination buffer. The characteristic P1 may be represented by the precondition:

$$P1 = \{\, uqv : q \in Q \text{ and } |\, u \circ v \,| \geq n \,\}$$

We are able to refine these characteristics to increase our chances for consistent and repeatable vulnerability classification results. We present one such refinement next.

§B.3.2 Revised Characteristics

Many of the original characteristics share similarities. For example, P1, P6, P11, and P15 all capture the overflowing of the buffer, but do so for different types of buffer overflow vulnerabilities. We modify the original characteristics of the buffer overflow work to reflect these similarities.

We first observe that P1, P6, P11, and P15 are identical characteristics. We collapse these into a single new characteristic `x:buff` as follows:

```
x:buff:  len(input) > len(buffer)
```

Informally, this captures the original characteristic "the length of a (possibly transformed) uploaded string is longer than that of the destination buffer." We use `input` to refer to the uploaded string (post any transformation), and `buffer` to refer to the destination buffer. We assume the `len()` function returns the length of a string or the buffer. We chose to describe the characteristic in pseudo-code, but a grammar may also be created similar to that of Whalen et al. [WHA05].

Each type of buffer overflow vulnerability also contains a characteristic based on the type of information that may be included in the uploaded string. We define a function `may_contain()` to return the data types allowed for a particular string.

```
x:addr:  may_contain(input, addr) ≡ true

x:inst:  may_contain(input, inst) ≡ true
```

Informally, the characteristic `x:addr` captures "the uploaded (and possibly transformed) string may contain addresses," as described in P7 and P16. The characteristic `x:inst` captures "the uploaded (and possibly transformed) string may contain instructions," which when combined with `x:addr` captures P2.

Characteristic P16 is described as "the uploaded (and possibly transformed) string may contain data of the type of the particular variable." To capture this, we define the variable `flowvar` to represent the "particular variable" being referenced that affects the program flow, and the function `type()` to return the type of that variable. Using these, we describe P16 with the following new characteristic:

```
x:type:  may_contain(input, type(flowvar)) ≡ true
```

We also observe that each type of buffer overflow vulnerability involves some sort of modification. Let the function `may_modify()` test whether this modification may occur. Let `retnptr` be the return address referenced in P3, `funcptr` be the function pointer referenced in P8, `flowvar` be the variable referenced in P13, and `flowptr` be the pointer for the program flow variable referenced in P17. This results in the following new characteristics:

```
x:rval:  may_modify(retnptr) ≡ true

x:fptr:  may_modify(funcptr) ≡ true

x:vvar:  may_modify(flowvar) ≡ true

x:vptr:  may_modify(flowptr) ≡ true
```

Executable buffer overflow vulnerabilities also have other similarities. For example, both P4 and P9 involve jumping into either a stack or heap, and characteristics P5 and P10 involve executing instructions in the stack or heap. We capture these four characteristics with the following:

```
x:jmps:  can_jump(stack) ≡ true

x:jmph:  can_jump(heap ) ≡ true

x:exes:  can_exec(stack) ≡ true

x:exeh:  can_exec(heap ) ≡ true
```

Finally, data buffer overflow vulnerabilities both require a variable that affects which execution path is taken at a future point of execution. Specifically, P14 is described as "the particular variable determines which execution path is to be taken at a future point in the execution of the process," and P18 is described as "the value pointed to by the particular pointer variable determines which execution path is to be taken at a future point in the execution of the process." Let `flowvar` be the variable that affects the program flow, and let `flowptr` be a pointer to `flowvar`. We may capture both P14 and P18 with the characteristic:

```
x:path:  affects_flow(flowvar) ≡ true
```

FIGURE B.1: BUFFER OVERFLOW CHARACTERISTICS

Old	New	Description
P1	x:buff	The length of the (possibly transformed) uploaded string is longer than that of the destination buffer.
P2	x:addr, x:inst	The uploaded (and possibly transformed) string may contain instructions and/or addresses.
P3	x:rval	Input can change the stored return address without the change being countered.
P4	x:jmps	The program can jump to memory in the stack.
P5	x:exes	The program can execute instructions stored in the stack.
P6	x:buff	The length of the (possibly transformed) uploaded string is longer than that of the destination buffer.
P7	x:addr	The uploaded (and possibly transformed) string may contain addresses.
P8	x:fptr	Input can change the value in the function pointer variable without being countered.
P9	x:jmph	The program can jump to the heap.
P10	x:exeh	The program can execute instructions in the heap.
P11	x:buff	The length of the (possibly transformed) uploaded string is longer than that of the destination buffer.
P12	x:type	The uploaded (and possibly transformed) string may contain data of the type of the particular variable.
P13	x:vval	The value stored in the particular variable can be changed without being countered.
P14	x:path	The particular variable determines which execution path is to be taken at a future point in the execution of the process.
P15	x:buff	The length of the uploaded string is longer than that of the destination buffer.
P16	x:addr	The uploaded (and possibly transformed) string may contain addresses.
P17	x:vptr	The address stored in the particular pointer variable can be changed without being countered.
P18	x:path	The value pointed to by the particular pointer variable determines which execution path is to be taken at a future point in the execution of the process.

The original 18 buffer overflow characteristics defined by Bishop et al. [BIS10], and which characteristics they map to in the Characteristic-Based Vulnerability Classification Scheme.

Bibliography

ABB76 Abbott, Robert P., Janet S. Chin, James. E. Donnelley, William L. Konigsford, Shigeru Tokubo, and Douglas A. Webb. "Security Analysis and Enhancements of Computer Operating Systems." Final Report NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

ALM06 Alm, Christopher and Michael Drouineaud. "Analysis of Existing Policy Languages." Deliverable AP 2.3, ORKA Consortium, June 2006.

AND96 Anderson, Ross J. "A Security Policy Model for Clinical Information Systems." In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996, pages 30–43.

AND04 Andrews, Mike and James A. Whittaker. "Computer Security." *IEEE Security & Privacy, volume 2* (5), September/October 2004: pages 68–71.

ASL95 Aslam, Taimur. *A Taxonomy of Security Faults in the UNIX Operating System.* Master's Thesis, Purdue University, August 1995.

AVE07 Aven, Terje. "A Unified Framework for Risk and Vulnerability Analysis Covering Both Safety and Security." *Reliability Engineering & System Safety, volume 92* (6), June 2007: pages 745–754.

BAR92 Barnes, Greg, Jonathan F. Buss, Walter L. Ruzzo, and Baruch Schieber. "A Sublinear Space, Polynomial Time Algorithm for Directed s-t Connectivity." In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference.* IEEE Computer Society Press, June 1992, pages 27–33.

BEZ09 Beznosov, Konstantin, Philip Inglesant, Jorge Lobo, Rob Reeder, and Mary Ellen Zurko. "Usability Meets Access Control: Challenges and Research Opportunities." In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2009, pages 73–74. (Panel Session).

BIS78 Bisbey, Richard and Dennis Hollingworth. "Protection Analysis: Final Report." Final Report ISI/SR-78-13, Information Sciences Institute, University of Southern California, May 1978.

BIS84 Biskup, Joachim. "Some Variants of the Take-Grant Protection Model." *Information Processing Letters, volume 19* (3), October 1984: pages 151–156.

BIS95 Bishop, Matt. "Theft of Information in the Take-Grant Protection Model." *Journal of Computer Security, volume 3* (4), 1994/1995: pages 283–308.

BIS96 Bishop, Matt and David Bailey. "A Critical Analysis of Vulnerability Taxonomies." Technical Report CSE-96-11, Department of Computer Science, University of California, Davis, September 1996.

BIS99 Bishop, Matt. "Vulnerability Analysis: An Extended Abstract." In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 1999, pages 125–136.

BIS03A Bishop, Matt. *Computer Security: Art and Science*. Addison-Wesley, 2003.

BIS03B Bishop, Matt. "What Is Computer Security?" *IEEE Security and Privacy, volume 1* (1), January/February 2003: pages 67–69.

BIS06 Bishop, Matt and Sean Peisert. "Your Security Policy is *What??*" Technical Report CSE-2006-20, Department of Computer Science, University of California, Davis, 2006.

BIS08 Bishop, Matt, Sophie Engle, Sean Peisert, Sean Whalen, and Carrie Gates. "We Have Met the Enemy and He is Us." In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW)*, September 2008, pages 1–12.

BIS09A Bishop, Matt. "Reflections on UNIX Security." In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, December 2009.

BIS09B Bishop, Matt, Sophie Engle, Sean Peisert, Sean Whalen, and Carrie Gates. "Case Studies of an Insider Framework." In *Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS)*, January 2009, pages 1–10.

BIS10 Bishop, Matt, Damien Howard, Sophie Engle, and Sean Whalen. "A Taxonomy of Buffer Overflow Preconditions." Technical Report CSE-2010-01, Department of Computer Science, University of California, Davis, 2010.

BISAR Bishop, Matt, Sophie Engle, Deborah A. Frincke, Carrie Gates, Frank L. Greitzer, Sean Peisert, and Sean Whalen. "A Risk Management Approach to the "Insider Threat"." In *Insider Threats in Cybersecurity—And Beyond*. Springer Verlag, Berlin, 2010 (to appear).

BLA07 Blaze, Matt, Arel Cordero, Sophie Engle, Chris Karlof, Naveen Sastry, Micah Sherr, Till Stegers, and Ka-Ping Yee. "Source Code Review of the Sequoia Voting System." Final report, Office of the California Secretary of State, June 2007. Top-to-Bottom Review of California Voting Systems.

BOY00 Boyle, Robert. *The Unsuccessful Experiment* in *Certain Physiological Essays*, volume 2 of *The Works of Robert Boyle*. Edited by Michael Hunter and Edward B. Davis. Pickering & Chatto, London, 1999–2000.

CAP10 The MITRE Corporation, http://capec.mitre.org/. *Common Attack Pattern Enumeration and Classification (CAPEC)*, January 2010. (Last Accessed).

CAR06 Carlson, Adam. *The Unifying Policy Hierarchy Model*. Master's Thesis, Department of Computer Science, University of California, Davis, June 2006.

CHE04 Chen, Hao, Drew Dean, and David Wagner. "Model Checking One Million Lines of C Code." In *Proceedings of the 11th Annual Network and Distributed Systems Security Symposium (NDSS)*, February 2004, pages 171–185.

CHR07 Christey, Steve and Robert A. Martin. "Vulnerability Type Distributions in CVE." Document Version 1.1, Common Weakness Enumeration (CWE), http://cwe.mitre.org/documents/vuln-trends/index.html, May 22 2007. (Last Modified).

CHU01 Chung, Christina Yip, Michael Gertz, and Karl Levitt. "Discovery of Multi-Level Security Policies." In *Proceedings of the IFIP TC11/WG11.3 Fourteenth Annual Working Conference on Database Security*. ACM Special Interest Group on Management of Data (SIGMOD), 2001, pages 173–184.

COH97A Cohen, Fred. "Information System Attacks: A Preliminary Classification Scheme." *Computers & Security, volume 16* (1), 1997: pages 29–46.

COH97B Cohen, Fred. "Information System Defences: A Preliminary Classification Scheme." *Computers & Security, volume 16* (2), 1997: pages 94–114.

COR03 Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2003.

COR07 Core Security Technologies, http://www.coresecurity.com/?action=item&id=1703. *OpenBSD's IPv6 `mbufs` Remote Kernel Buffer Overflow*, March 2007. (Posted).

COW98 Cowan, Crispin, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, Berkeley, California, 1998, page 5.

CVE09 The MITRE Corporation, http://cve.mitre.org/about/terminology.html. *Common Vulnerabilities and Exposures (CVE)*, November 2009. (Last Accessed).

CWE09 The MITRE Corporation, http://cwe.mitre.org/about/. *Common Weakness Enumeration (CWE)*, November 2009. (Last Accessed).

DEM95 Demillo, Richard A. and Aditya P. Mathur. "A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TEX." Technical Report SERC-TR-165-P, Software Engineering Research Center, September 1995.

DEN96 Denning, Peter J. and Jack B. Dennis. "A Simple Proof of the Correspondence Theorem." Memo 29, Computation Structures Group, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 196?

DEN99 Denning, Dorothy E. "The Limits of Formal Security Models." National Computer Systems Security Award Acceptance Speech, October 18 1999.

DOV98 "Database of Vulnerabilities, Exploits, and Signatures (DOVES)." http://seclab.cs.ucdavis.edu/projects/vulnerabilities, November 1998. (Last Modified).

D'S08 D'Souza, Deepak, Raveendra Holla, Janardhan Kulkarni, Raghavendra K. Ramesh, and Barbara Sprick. *On the Decidability of Model-Checking Information Flow Properties* in *Information Systems Security*, volume 5352/2008 of *Lecture Notes in Computer Science*. Springer, Berlin, December 2008, pages 26–40.

DUN02 Dunlap, George W., Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay." *ACM SIGOPS Operating Systems Review, volume 36* (SI), Winter 2002: pages 211–224.

END75 Endres, Albert. "An Analysis of Errors and Their Causes in System Programs." *ACM SIGPLAN Notices, volume 10* (6), June 1975: pages 327–336.

ENG06A Engle, Sophie, Sean Whalen, Damien Howard, and Matt Bishop. "Tree Approach to Vulnerability Classification." Technical Report CSE-2006-10, Department of Computer Science, University of California, Davis, 2006.

ENG06B Engle, Sophie, Sean Whalen, Damien Howard, Adam Carlson, Elliot Proebstel, and Matt Bishop. "A Practical Formalism for Vulnerability Comparison." Technical Report CSE-2006-11, Department of Computer Science, University of California, Davis, 2006.

ENG08A Engle, Sophie and Matt Bishop. "A Model for Vulnerability Analysis and Classification." Technical Report CSE-2008-05, Department of Computer Science, University of California, Davis, 2008.

ENG08B Engle, Sophie, Sean Whalen, and Matt Bishop. "Modeling Computer Insecurity." Technical Report CSE-2008-14, Department of Computer Science, University of California, Davis, 2008.

FEY74 Feynman, Richard P. "Cargo Cult Science." *Engineering and Science, volume 37* (7), June 1974: pages 10–13.

FIT04 Fithen, William L., Shawn V. Hernan, Paul F. O'Rourke, and David A. Shinberg. "Formal Modeling of Vulnerability." *Bell Labs Technical Journal, volume 8* (4), February 2004: pages 173–186.

GOG82 Goguen, Joseph A. and José Meseguer. "Security Policies and Security Models." In *Proceedings of the IEEE Symposium on Security and Privacy*, 1982, pages 11–20.

GRA71 Graham, G. Scott and Peter J. Denning. "Protection — Principles and Practice." In *Proceedings of the Fall Joint Computer Conference*. American Federation of Information Processing Societies (AFIPS), November 1971, pages 417–429.

HAL08 Halderman, J. Alex, Eric Rescorla, Hovav Shacham, and David Wagner. "You Go to Elections with the Voting System You Have: Stop-Gap Mitigations for Deployed Voting Systems." In *Proceedings of the Conference on Electronic Voting Technology (EVT'08)*. USENIX Association, Berkeley, California, 2008.

HAM06 Hamlen, Kevin W., Greg Morrisett, and Fred B. Schneider. "Computability Classes for Enforcement Mechanisms." *ACM Transactions on Programming Languages and Systems (TOPLAS), volume 28* (1), January 2006: pages 175–205.

HAR76 Harrison, Michael, Walter Ruzzo, and Jeffery Ullman. "Protection in Operating Systems." *Communications of the ACM, volume 19* (8), August 1976: pages 461–471.

HER06 Herzog, Pete. *Open-Source Security Testing Methodology Manual*. Institute for Security and Open Methodologies (ISECOM), New York, osstmm 2.2 edition, December 2006.

HOW97 Howard, John D. *An Analysis of Security Incidents on the Internet 1989–1995*. Ph.D. Dissertation, Department of Engineering and Public Policy, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1997.

HOW05 Howard, Michael, Jon Pincus, and Jeannette M. Wing. "Measuring Relative Attack Surfaces." In *Computer Security in the 21st Century*. Springer, March 2005, pages 109–137.

HUL94 Huling, George. "Introduction to Use of Formal Methods in Software and Hardware." In *Conference Record of Wescon'94 Idea/Microelectronics*, September 1994, pages 48–52.

IRV07 Irvine, Cynthia E. and Karl Levitt. "Trusted Hardware: Can it be Trustworthy?" In *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007, pages 1–4.

JAJ97A Jajodia, Sushil, Pierangela Samarati, and Venkatramanan S. Subrahmanian. "A Logical Language for Expressing Authorizations." In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997, pages 31–42.

JAJ97B Jajodia, Sushil, Pierangela Samarati, Venkatramanan S. Subrahmanian, and Eliza Bertino. "A Unified Framework for Enforcing Multiple Access Control Policies." In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. ACM Special Interest Group on Management of Data (SIGMOD), 1997, pages 474–485.

JEC03 Jech, Thomas. *Set Theory*. Springer Monographs in Mathematics. Springer, Berlin, third millennium edition, 2003.

JHA02 Jha, Somesh, Oleg Sheyner, and Jeannette M. Wing. "Two Formal Analyses of Attack Graphs." In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, June 2002, pages 49–63.

JØ07 Jøsang, Audun, Bander AlFayyadh, Tyrone Grandison, Mohammed AlZomai, and Judith McNamara. "Security Usability Principles for Vulnerability Analysis and Risk Assessment." In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference (ACSAC)*, December 2007, pages 269–278.

JON76 Jones, Anita K., Richard J. Lipton, and Lawrence Snyder. "A Linear Time Algorithm for Deciding Security." In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science (SFCS)*, October 1976, pages 33–41.

KAR72 Karp, Richard M. "Reducibility Among Combinatorial Problems." In Raymond E. Miller and James W. Thatcher (Editors) *Complexity of Computer Computations*. Plenum Press, New York, March 1972, pages 85–103.

KLE07 Kleiner, Eldar and Tom C. Newcomb. "On the Decidability of the Safety Problem for Access Control Policies." *Electronic Notes in Theoretical Computer Science, volume 185*, 2007: pages 107–120.

KLE09 Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. "seL4: Formal Verification of an OS Kernel." In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, October 2009, pages 207–220.

KRS98 Krsul, Ivan Victor. *Software Vulnerability Analysis*. Ph.D. Dissertation, Department of Computer Sciences, Purdue University, May 1998. COAST Technical Report 98-09.

LAM71 Lampson, Butler W. "Protection." In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, March 1971, pages 437–443. Reprinted in *Operating Systems Review, volume 8* (1), January 1974: pages 18–24.

LAN81 Landwehr, Carl. "Formal Models for Computer Security." *ACM Computing Surveys, volume 13* (3), September 1981: pages 247–278.

LAN94 Landwehr, Carl E., Alan R. Bull, John P. McDermott, and William S. Choi. "A Taxonomy of Computer Program Security Flaws." *ACM Computing Surveys, volume 26* (3), September 1994: pages 211–254.

LI05 Li, Ninghui and Mahesh V. Tripunitara. "On Safety in Discretionary Access Control." In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005, pages 96–109.

LIG09 Ligatti, Jay, Lujo Bauer, and David Walker. "Run-Time Enforcement of Nonsafety Policies." *ACM Transactions on Information and System Security (TISSEC), volume 12* (3), January 2009: pages 1–41.

LIN75 Linde, Richard R. "Operating System Penetration." In *Proceedings of the National Computer Conference And Exposition*. American Federation of Information Processing Societies (AFIPS), May 1975, pages 361–368.

MCP08 McPherson, Amanda, Brian Proffitt, and Ron Hale-Evans. "Estimating the Total Development Cost of a Linux Distribution." Whitepaper, The Linux Foundation, http://www.linuxfoundation.org/publications/estimatinglinux.php, October 2008.

MEE05 Meenakshi, Balasubramanian. "Formal Verification." *Resonance, volume 10* (5), May 2005: pages 26–38.

NEU78 Neumann, Peter. *Computer Security Evaluation*, volume 47 of *National Computer Conference, AFIPS Conference Proceedings*, 1978, pages 1087–1095.

OLI06 de Oliveira, Daniela A. S., Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. "ExecRecorder: VM-based Full-System Replay for Attack Analysis and System Recovery." In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID)*. ACM, New York, New York, October 2006, pages 66–71.

OPE06 Open Information Systems Security Group (OSISSG). *Information Systems Security Assessment Framework (ISSAF)*, draft 0.2 edition, April 2006.

OST84 Ostrand, Thomas J. and Elaine J. Weyuker. "Collecting and Categorizing Software Error Data in an Industrial Environment." *Journal of Systems and Software, volume 4* (4), November 1984: pages 289–300.

OWA09 Open Web Application Security Project (OWASP), http://www.owasp.org/index.php/Category:Vulnerability. *Category:Vulnerability*, November 2009. (Last Accessed).

PEI07A Peisert, Sean and Matt Bishop. "How to Design Computer Security Experiments." In *Proceedings of the Fifth World Conference on Information Security Education (WISE)*, June 2007, pages 141–148.

PEI07B Peisert, Sean, Matt Bishop, Sidney Karin, and Keith Marzullo. "Analysis of Computer Intrusions Using Sequences of Function Calls." *IEEE Transactions on Dependable and Secure Computing (TDSC), volume 4* (2), April–June 2007: pages 137–150.

PEI07C Peisert, Sean P. *A Model of Forensic Analysis Using Goal-Oriented Logging*. Ph.D. Dissertation, Department of Computer Science and Engineering, University of California, San Diego, March 2007.

RAM98 Ramakrishnan, Coimbatore R. and R. Sekar. "Model-Based Vulnerability Analysis of Computer Systems." In *Proceedings of the 2nd International Workshop on Verification, Model Checking and Abstract Interpretation (VMCAI)*, September 1998.

RAM02 Ramakrishnan, Coimbatore R. and R. Sekar. "Model-Based Analysis of Configuration Vulnerabilities." *Journal of Computer Security (JCS), volume 10* (1-2), January 2002: pages 189–209.

ROC89 Rochlis, Jon A. and Mark W. Eichin. "With Microscope and Tweezers: The Worm from MIT's Perspective." *Communications of the ACM, volume 32* (6), June 1989: pages 689–698.

ROS07 Rosen, Kenneth H. *Discrete Mathematics and Its Applications*. McGraw-Hill, New York, 6th edition, 2007.

RUS92 Rushby, John M. "Noninterference, Transitivity, and Channel-Control Security Policies." Technical Report CSL-92-02, Computer Science Laboratory, SRI International, Menlo Park, California, December 1992.

SAK06 Sakaki, Hiroshi, Kazuo Yanoo, and Ryuichi Ogawa. *A Model-Based Method for Security Configuration Verification* in *Advances in Information and Computer Security*, volume 4266/2006 of *Lecture Notes in Computer Science*. Springer, Berlin, October 2006, pages 60–75.

SCH99 Schneider, Fred B. (Editor) *Trust in Cyberspace*. National Academies Press, 1999.

SCH00 Schneider, Fred B. "Enforceable Security Policies." *ACM Transactions on Information and System Security (TISSEC), volume 3* (1), February 2000: pages 30–50.

SCH05 Schwarz, Benjamin, Hao Chen, David Wagner, Geoff Morrison, Jacob West, Jeremy Lin, and Wei Tu. "Model Checking an Entire Linux Distribution for Security Violations." In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, December 2005.

SIM08 Simidchieva, Borislava I., Matthew S. Marzilli, Lori A. Clarke, and Leon J. Osterweil. "Specifying and Verifying Requirements for Election Processes." In *Proceedings of the International Conference on Digital Government Research*. Digital Government Society of North America, 2008.

SIN08 Sinclair, Sara and Sean W. Smith. *Preventative Directions For Insider Threat Mitigation Via Access Control* in *Insider Attack and Cyber Security: Beyond the Hacker*, volume 39 of *Advances in Information Security*. Springer, 2008, pages 165–194.

SIP97 Sipser, Michael. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

SNY77 Snyder, Lawrence. "On the Synthesis and Analysis of Protection Systems." *ACM SIGOPS Operating Systems Review, volume 11* (5), November 1977: pages 141–150.

STE91 Sterne, Daniel F. "On the Buzzword 'Security Policy'." In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*. ACM Special Interest Group on Management of Data (SIGMOD), 1991, pages 219–230.

THO84 Thompson, Ken. "Reflections on Trusting Trust." *Communications of the ACM, volume 27* (8), August 1984: pages 761–763.

VAU01 Vaught, Robert L. *Set Theory: An Introduction*. Birkhäuser, 2nd edition, 2001.

VEN97 Venkatesan, Ramkuniar M. and Sourav Bhattacharya. "Threat-Adaptive Security Policy." In *Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC)*, February 1997, pages 525–531.

VVS07 "Voluntary Voting System Guidelines (VVSG) Recommendations to the Election Assistance Commission (EAC)." Prepared at the Direction of the Technical Guidelines Development Committee (TGDC), August 2007.

WEB98 Weber, Daniel J. *A Taxonomy of Computer Intrusions*. Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1998.

WEI95 Weissman, Clark. "Security Penetration Testing Guideline." NRL Technical Memorandum 5540:082A, chapter 10 of the *Handbook for the Computer Security Certification of Trusted Systems*. Naval Research Laboratory, Washington, D.C., January 1995.

WHA05 Whalen, Sean, Sophie Engle, and Matt Bishop. "Protocol Vulnerability Analysis." Technical Report CSE-2005-04, Department of Computer Science, University of California, Davis, 2005.

WHI99 Whitten, Alma and J. Doug Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0." In *Proceedings of the 8th USENIX Security Symposium*, 1999.

WIN98 Wing, Jeannette M. "A Symbiotic Relationship Between Formal Methods and Security." In *Computer Security, Dependability and Assurance: From Needs to Solutions*, November 1998, pages 26–38.

WOO04 Wool, Avishai. "A Quantitative Study of Firewall Configuration Errors." *Computer, volume 37* (6), June 2004: pages 62–67.

YUA06 Yuan, Lihua, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. "FIREMAN: A Toolkit for FIREwall Modeling and ANalysis." In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, June 2006, pages 199–213.

ZHA97 Zhang, Kan. "A Theory for System Security." In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW)*, June 1997, pages 148–155.

Terminology Index