

Modeling and Detecting Security Vulnerabilities with Static Analysis

By

WANG PINGYAN

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Informatics and Data Science

HIROSHIMA UNIVERSITY

2024

© 2024 Wang Pingyan

Abstract

Security vulnerabilities in software can be exploited by attackers to launch malicious attacks, which in turn can cause security failures. Static analysis is a widely used technique for vulnerability discovery in both academia and industry. While many existing static analysis approaches have been proved to be effective, there remain some challenging open problems. This dissertation focuses on two important problems. The first problem is how to perform analysis on incomplete programs and obtain real-time analysis results during program development, which enables vulnerability detection at an early stage. Another interesting problem is that automated static analysis alone tends to miss some complex and subtle vulnerabilities, thus requiring the incorporation of certain manual security expertise to augment its capabilities.

To enable real-time analysis on incomplete programs, we first present a general framework used in a paradigm known as *Human-Machine Pair Programming*. The framework employs attack trees to model a given class of vulnerabilities and then crafts patterns for each individual vulnerability. The programmer will be alarmed during coding when patterns match potentially vulnerable code. To identify specifically taint-style vulnerabilities in Human-Machine Pair Programming, we further present two pointer-analysis-based approaches, namely *exhaustive pointer analysis* and *demand-driven pointer analysis*. Both the approaches support an incremental pointer analysis and provide points-to information for vulnerability discovery during program development. Our experiment results show that the proposed approaches can detect all the potential vulnerabilities in Securibench Micro with low false positives.

To benefit from both manual audits and automated static analysis, we propose *vulnerability nets*, a graphical code representation for modeling source code. With a combination of Petri nets, data dependence graphs, and control flow graphs, vulnerability nets explicitly describe the key information of a given program and provide a graphical view for analysts to perform auditing. Our evaluation shows that the proposed approach outperforms the tool SonarQube in generating fewer false negatives when tested in Securibench Micro.

Acknowledgments

First and foremost, I would like to thank my advisor Shaoying Liu. This thesis would have been impossible without your continuous support and mentoring throughout my PhD in Hiroshima University. I am so grateful for all your valuable comments, suggestions, and encouragement over the last several years.

I would also like to express my gratitude to the committee members of this thesis, Tadashi Dohi, Koji Eguchi, and Jianjun Zhao, for their valuable comments and suggestions. Your feedback has significantly improved the quality of this thesis.

My sincere thanks go to Ai Liu for making substantial insightful comments on my research. I would also like to extend my grateful thanks to every member of Dependable Systems Laboratory for always giving me a hand whenever I need any help. I truly enjoyed the time when we discussed various research problems and exchanged our ideas during group seminars.

Finally, my special thanks go to my family and friends.

This work was supported by JST SPRING (Grant Number JPMJSP2132) and ROIS NII Open Collaborative Research 2021-(21FS02).

Table of Contents

Abstract	iii
Acknowledgments.....	v
List of Figures	xi
List of Tables	xiii
1 Introduction.....	1
1.1 Vulnerability Discovery	2
1.1.1 What Are Vulnerabilities?	2
1.1.2 Vulnerability Discovery Approaches.....	3
1.2 The Problems.....	4
1.3 Contributions	5
1.4 Organization of the Thesis	6
2 Vulnerability Discovery in Human-Machine Pair Programming: A Framework.....	7
2.1 Human-Machine Pair Programming	8
2.2 Attack Trees	9
2.3 The Framework	11
2.3.1 Pattern Preparation.....	12
2.3.1.1 Identifying attack goals.....	12
2.3.1.2 Generating attack trees.....	13
2.3.1.3 Constructing vulnerability-matching patterns.....	15
2.3.2 Pattern Application	18
2.3.2.1 Detecting vulnerable code.....	18
2.3.2.2 Reporting warnings.....	19
2.3.2.3 Fixing the code.....	19
2.4 Case Study.....	20
2.4.1 Modeling SQLIAs.....	20
2.4.1.1 Identifying the attack goal	20
2.4.1.2 Generating attack trees.....	21

2.4.1.3	Constructing vulnerability-matching patterns.....	22
2.4.2	Detecting SQLIAs.....	25
2.4.2.1	Detecting Vulnerable Code.....	25
2.4.2.2	Reporting Warnings.....	25
2.4.2.3	Fixing the Code.....	25
2.5	Discussion.....	26
2.6	Related Work.....	26
3	Vulnerability Discovery in Human-Machine Pair Programming: Using Pointer Analysis.....	28
3.1	Background.....	29
3.1.1	Taint Analysis.....	29
3.1.2	Pointer Analysis.....	30
3.2	Proposed Approaches.....	30
3.2.1	Motivating Example.....	30
3.2.2	Exhaustive Pointer Analysis.....	32
3.2.2.1	Human-Machine Pair Programming fashion.....	32
3.2.2.2	Flow sensitivity.....	33
3.2.2.3	Conservative approximation.....	34
3.2.2.4	Pointer propagation rules.....	36
3.2.3	Demand-Driven Pointer Analysis.....	37
3.2.4	Difference and Equivalence.....	39
3.3	Evaluation.....	41
3.3.1	Experiment.....	41
3.3.1.1	Preliminaries.....	41
3.3.1.2	Implementation.....	42
3.3.2	Results and Discussion.....	43
3.3.3	Case Study.....	45
3.4	Discussion.....	47
3.5	Related Work.....	48
4	Vulnerability Nets for Vulnerability Discovery.....	50

4.1	Definitions	51
4.1.1	Petri Nets.....	51
4.1.2	Vulnerability Nets.....	52
4.1.3	Vulnerability Nets with Colored Tokens	56
4.2	Modeling and Detecting Vulnerabilities	57
4.2.1	Data Dependence Graphs.....	58
4.2.2	Control Flow Graphs.....	59
4.2.3	Building Vulnerability Nets.....	60
4.2.4	Algorithms	63
4.3	Evaluation.....	65
4.3.1	Experiments	65
4.3.1.1	Preliminaries	65
4.3.1.2	Implementation	66
4.3.2	Results.....	67
4.3.3	Comparison with SonarQube.....	68
4.3.4	Case Study	69
4.4	Discussion	72
4.5	Related Work.....	74
5	Conclusion and Outlook	76
5.1	Summary of Results	76
5.2	Future Work	76
6	Bibliography	78
A	Source Code Containing LDAP Injection.....	88
B	Copyright Documentation.....	92

List of Figures

Figure 2.1. SCM framework [34].	8
Figure 2.2. Example of an attack tree.	10
Figure 2.3. Overview of the proposed framework.	12
Figure 2.4. Generation of an attack tree.	13
Figure 2.5. Process of pattern construction.	16
Figure 2.6. Examples of false negatives and false positives.	18
Figure 2.7. Example of warning report.	19
Figure 2.8. Interaction between the programmer and computer.	20
Figure 2.9. Attack tree against SQL injection.	22
Figure 2.10. Illustrative code fragment.	22
Figure 2.11. Detection for vulnerable code.	25
Figure 2.12. Warning report for the illustrative code.	25
Figure 3.1. Motivating code example.	31
Figure 3.2. Overview of our approach.	32
Figure 3.3. Code example adapted from Figure 3.1.	34
Figure 3.4. Demand-driven pointer analysis of the code example in Figure 3.1.	38
Figure 3.5. Code snippet of <i>Interl</i> from Securibench Micro.	44
Figure 3.6. Code snippet from Apache Druid 0.17.0 illustrating LDAP injection.	46
Figure 4.1. (a) Textual representation of a vulnerability net; (b) Graphical representation of Figure 4.1 (a).	53
Figure 4.2. The next state of Figure 4.1.	55
Figure 4.3. (a) Execution of a standard Petri net; (b) Execution of a vulnerability net.	56

Figure 4.4. Execution of a vulnerability net with colored tokens.....	57
Figure 4.5. Example of a taint-style vulnerability in Java.	58
Figure 4.6. (a) DDG for the code in Figure 4.5; (b) CFG for the code in Figure 4.5.	59
Figure 4.7. Example code adapted from Figure 4.5.....	60
Figure 4.8. (a) Vulnerability net for the code in Figure 4.5; (b) Vulnerability net with sanitization identification for the code in Figure 4.7.	61
Figure 4.9. Datalog program implementing Algorithm 4.2.	67
Figure 4.10. Code fragment adapted from a real-world case.....	69
Figure 4.11. Vulnerability net for the code in Figure 4.10.	70
Figure A.1. Source code of the authentication module in Apache Druid 0.17.0.	91

List of Tables

Table 1.1. A summary of commonly used vulnerability discovery methods.	3
Table 2.1. Symbols used in our attack-tree analysis.	9
Table 3.1. Pointer propagation rules.	36
Table 3.2. Vulnerability types within the programs for our experiments.	42
Table 3.3. Analysis example.	43
Table 3.4. Experiment results.	43
Table 4.1. Experiment results.	67
Table 4.2. Comparison between SonarQube and our approach.	68

1 Introduction

As computer systems play an increasingly important role in our lives, there is growing concern over the security of software. Security vulnerabilities in software can be exploited by attackers to launch malicious attacks, which in turn can cause security failures. Numerous security breaches in the past have shown that even a single vulnerability can lead to a catastrophic failure (e.g., [1, 2]). Probably the most notorious example in the last decade is the *Heartbleed vulnerability* [2], which is found in the popular OpenSSL cryptographic software library in 2014. Heartbleed allowed attackers to remotely read sensitive memory from an estimated 24-55% of popular HTTPS sites [3]. Surprisingly, the flaw itself is simple: a single missing bound check enables any attacker-specified length of a user-supplied message. A more recent example is that a vulnerability in Twitter's systems caused substantial account information leakages in 2022 [4]. All these breaches have demonstrated the leading role of vulnerabilities in security. Therefore, a natural strategy to enhance software security is to identify and eliminate the potential vulnerabilities in code.

There are various ways of identifying vulnerabilities, ranging from dynamic analysis (e.g., fuzz testing [5]) to formal methods (e.g., model checking [6]). Of these techniques, static analysis is arguably the most common and effective one [7]. Static program analysis refers to an analysis method that analyzes a program without executing it, as opposed to dynamic program analysis such as unit testing. Many static analysis techniques (e.g., data flow analysis) are derived from compiler technologies. These techniques can be encapsulated in automated tools, freeing developers from manual code review. A considerable number of existing static analysis tools (e.g., [8-12]) have proved to be effective in vulnerability detection in practice. However, many problems remain unsolved and the research in this field still has received considerable attention in recent years [13-15]. In this thesis, we will discuss our solutions to two important problems in the field.

Before we dive into more details, in the rest of this chapter, we first provide some basic background about vulnerability discovery. Then we describe the problems discussed in this thesis, followed by a summary of our contributions and the organization of this thesis.

1.1 Vulnerability Discovery

1.1.1 What Are Vulnerabilities?

According to CVE [16], a vulnerability is defined as “a weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, availability.” Confidentiality refers to a security policy that protects information from unauthorized access, while integrity is a security policy that ensures the trustworthiness and completeness of data, and availability is another policy that guarantees that data are accessible when needed. In this work, we focus exclusively on the code vulnerabilities in software. We use the terms *vulnerability*, *security-related bug*, and *security flaw* interchangeably throughout this thesis.

Many defects in code are not regarded as vulnerabilities because they are security unrelated. To design methods specifically for vulnerability discovery, it is important for us to first understand the features that all vulnerabilities (or a specific class of vulnerabilities) have in common. The work in [17] summaries three key characteristics of vulnerable code: sensitive operation, attacker control, and insufficient validation. To offer a more comprehensive view, we add several other characteristics to the list and provide a brief description of each one in the following.

- **Sensitive operation.** Sensitive operations may include protected functionality invocations, personal data handling, authentication, buffer copying, and memory allocation.
- **Attacker control.** It means the code allows an attacker to influence the behavior of the application through attacker-controlled input or other ways.
- **Improper input validation.** Input validation refers to the action of verifying that the input data are correct and safe or adhere to certain criteria. Failure to provide proper validation on user-supplied input may allow an attacker to gain control over the system through carefully crafted input.

- **Improper input sanitization.** Input sanitization refers to the action of cleaning the input data to prevent dangerous input from being processed. Failure to properly sanitize user-supplied input can lead to problems such as SQL injections.
- **Information leakages.** Responses from the application, such as error messages, may reveal sensitive information and thus can be exploited by an attacker.
- **Use of vulnerable components.** Using vulnerable libraries or other software components can result in the inheritance of their security risks.

1.1.2 Vulnerability Discovery Approaches

Vulnerability discovery is so important for software security that it has been discussed in the literature for decades [18]. There is a large body of work on this topic in the literature. For comparison, we have summarized and categorized some commonly used techniques for vulnerability discovery in Table 1.1. Note that this table is not intended to be comprehensive, but rather to provide a brief overview of the existing techniques and to indicate the category to which our research belongs. The work in this thesis falls primarily into the category of automated static analysis, though it also involves manual code review.

Table 1.1. A summary of commonly used vulnerability discovery methods.

Methods & Techniques		Description
Manual Code Review	Manual audits (e.g., [19])	Identify the code manually to find vulnerabilities.
Automated Static Analysis	Rule/pattern-based analysis (e.g., [10])	Use a set of predefined rules or patterns to find vulnerabilities in code.
	Data flow analysis (e.g., [20])	Track the flow of data through the program to find vulnerabilities.
	Taint analysis (e.g., [21])	Monitor how the untrusted sources propagate through the program and introduce security risks in some points.
	Abstract interpretation (e.g., [22])	Use abstract representations of program behavior to analyze possible code paths.

Methods & Techniques		Description
Dynamic Analysis	Fuzz testing (e.g., [5])	Provide random data as input to the program to find vulnerabilities that occur during execution.
	Penetration Testing (e.g., [23])	Simulate real-world attacks to identify vulnerabilities.
Formal Methods	Model checking (e.g., [6])	Explore all possible states of a program to verify properties and detect vulnerabilities.
Machine Learning-Based	Pattern recognition (e.g., [24])	Identify patterns associated with known vulnerabilities.
Hybrid Approaches	Combination of two or more different methods (e.g., symbolic execution [25])	Combine elements of multiple analysis methods.

1.2 The Problems

Static analysis can be used in multiple phases of the software development lifecycle. Intuitively, performing static analysis in the coding phase is generally efficient because it allows the programmer to review and fix the vulnerable code in a timely manner. However, during the coding phase, the program is generally incomplete and some key information might be missing, which can lead to imprecise analysis and thus introduce false negatives or positives. In particular, false positives are one of the primary reasons that discourage developers from using a static analysis tool [13]. Therefore, it is important for us to study the problem of enabling analysis on incomplete programs while keeping the false positives low. Unfortunately, work in this branch is not extensively presented. For example, as one of the fundamental techniques used in static analysis, pointer analysis generally requires whole-program availability [26], which indicates that most pointer analysis approaches cannot be applied directly to incomplete program.

Another interesting problem is that static analysis tools will always fail to uncover some subtle vulnerabilities (i.e., false negatives), or may produce substantial false alarms (i.e.,

false positives), due to the difficulty of obtaining soundness and completeness. For example, buffer overflows [27], one of the most notorious vulnerabilities, still cannot be fully addressed by using automated tools alone. Instead, significant security expertise is often involved during detection of buffer overflows [28]. To alleviate the problem, some researchers (e.g., [29, 30]) have considered incorporating the analyst’s security knowledge (e.g., manual audits) into a tool during the detection process. Still, the exploration in this research branch is inadequate. For example, little work has been done on graphical representations of source code for supporting manual audits (e.g., [19]).

To summarize, this thesis focuses on two important problems: (1) how to perform analysis on incomplete programs and obtain real-time analysis results during program development, thereby allowing the developer to remove security risks at an early stage; (2) how to benefit from both manual audits and automated static analysis, thereby aiding the developer (or the security analyst) in finding some complex or subtle vulnerabilities.

Interestingly, while the two problems are seemingly unrelated at first glance, addressing one is likely to significantly alleviate another. For example, real-time analysis results can help the developer find some vulnerabilities earlier before they turn to complex later. On the other hand, manual audits can also facilitate the analysis of incomplete programs.

1.3 Contributions

The main contributions of this thesis include:

- We describe a pattern-based framework for timely vulnerability discovery in Human-Machine Pair Programming. In this framework, we describe in detail how patterns are created based on attack-tree analysis and how patterns are applied to real-time vulnerability detection in the coding phase.
- We present two pointer analysis approaches, exhaustive pointer analysis and demand-driven pointer analysis, to identify taint-style vulnerabilities in Human-Machine Pair Programming. Both the approaches support an incremental pointer

analysis and provide points-to information for vulnerability discovery during program development.

- We put forward vulnerability nets, a novel graphical code representation for modeling and detecting vulnerabilities in source code. Vulnerability nets support both automated analysis and manual audits.
- We present evaluation of the proposed approaches by conducting experiments on Securibench Micro and/or real-world case studies. Our evaluation demonstrates the effectiveness of our approaches. For example, our vulnerability net approach outperforms the existing tool SonarQube in generating fewer false negatives when tested in Securibench Micro.

1.4 Organization of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we describe a pattern-based framework for vulnerability discovery in Human-Machine Pair Programming. In Chapter 3, we further present two pointer analysis approaches for identifying taint-style vulnerabilities in Human-Machine Pair Programming. In Chapter 4, we discuss vulnerability nets, a graphical code representation for modeling and detecting vulnerabilities in source code. Chapter 5 concludes this thesis and presents future work.

2 Vulnerability Discovery in Human-Machine Pair Programming: A Framework¹

Security vulnerabilities can be found in different phases of a software lifecycle and exploited by attackers to launch attacks against software-based systems. Although system administrators can install patches after being attacked, systems have been compromised and attackers probably have achieved their goals. For this reason, the traditional penetrate-and-patch approach might not be considered as an effective strategy in many scenarios. For most software-based systems, especially security-critical systems, it is important to detect and tackle the security problems at an early stage since adverse impact can increase rapidly with time. Researchers have explored many approaches for mitigating security problems during different development phases, including requirement phase [31], coding phase [32] and testing phase [33]. Intuitively, identifying the security-related problems in the coding phase is generally efficient because it allows the programmer to review and fix the vulnerable code in a timely manner. Some static analysis techniques are proposed to fulfill this goal. Human-Machine Pair Programming [34] is one such technique that advocates a compiler-like mechanism for bug detection, during which the computer (machine) constantly runs a check to find bugs according to certain rules when the programmer (human) is coding.

Attack trees [8] are considered as a popular method to describe the sequence of events that can result in a specific attack. In an attack tree, an attack goal G_0 will be decomposed into a set of relatively simple sub-goals and each sub-goal will be further decomposed into smaller sub-goals if possible. The smallest sub-goals, i.e., the leaf nodes of the tree, denote the smallest attack actions that can cause the attack goal G_0 to occur. In this chapter, we present an attack trees-based approach in Human-Machine Pair Programming [35]. Specifically, we first create an attack tree to model a specific attack goal G_0 , which will be decomposed into a set of smallest attacks that are easier for analysis. Subsequently, we

¹ This chapter is adapted from our work in [35].

perform vulnerability analysis for each leaf node of an attack tree and craft corresponding patterns. Finally, we apply the patterns to perform real-time vulnerability discovery during coding.

2.1 Human-Machine Pair Programming

Human-Machine Pair Programming (HMPP) [34] is a paradigm characterized by the feature that the programmer (human) creates algorithms and data structures for a program while the computer (machine) monitors the coding process to find bugs and predict future contents. While HMPP supports both Software Construction Monitoring (SCM) and Software Construction Predicting (SCP), the scope of this thesis only involves the former.

Figure 2.1 shows the general process of SCM. Let CV_S be the current version of software under construction. Firstly, a syntactical analysis for CV_S is conducted, which can help extract useful code information, such as variable declarations, function definitions, control structures, and the relationships between program elements. Then, the extracted information together with a property-related knowledge base (equipped with knowledge such as secure development conventions and common faults) will form a set of specific properties of interest p_1, p_2, \dots, p_n . Finally, these properties will be examined to identify if there are any violations. A violation indicates the presence of a potential bug in the program. As CV_S is continuously changed during coding phase, the analysis also continuously proceeds and yields new results.

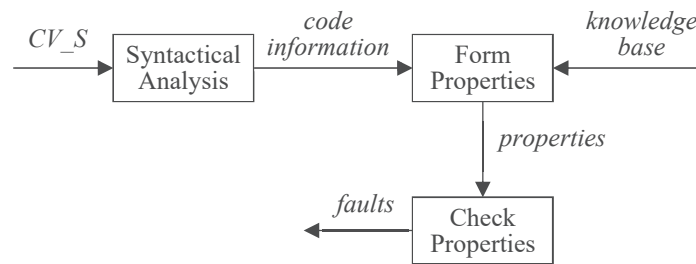






Figure 2.1. SCM framework [34].

2.2 Attack Trees

An attack tree [36] is comprised of AND- and OR-decompositions. An AND-decomposition can be decomposed as a set of attack sub-goals, all of which must be achieved for the attack to succeed, while an OR-decomposition can be decomposed as a set of attack sub-goals, any one of which is achieved is sufficient for the attack to succeed [37].

While both a graphical representation and a textual representation can be used to represent an attack tree, we use the former for readability. The graphical symbols of attack trees are often borrowed from those of fault trees [38, 39], as shown in Table 2.1. Note that the very same symbol in a fault tree and an attack tree can have different meanings. For example, while circles represent basic events in a fault tree, they represent atomic attacks in an attack tree.

Table 2.1. Symbols used in our attack-tree analysis.

Symbols	Fault Trees [39]	Our work
	Basic event	Atomic attack
	Intermediate event	Attack goal/sub-goal
	AND	AND
	OR	OR

In our attack-tree analysis, the root node, intermediate nodes, and leaf nodes in an attack tree represent the *attack goal*, *sub-goals*, and *atomic attacks*, respectively (see Figure 2.2). Formally, an attack tree is defined as follows.

Definition 2.1. An *attack tree* $AT = (G_0, \{G_i\}_{i=1}^n, A, \lambda)$ is a tree structure for modeling an arbitrary attack, where G_0 is the attack goal (root node), $\{G_i\}_{i=1}^n$ is a set of sub-goals (intermediate nodes), A is a set of atomic attacks (leaf nodes), and $\lambda: G_0 \cup \{G_i\}_{i=1}^n \cup A \rightarrow S$ is a function assigning properties to each node where S is the set of property values.

Throughout the thesis we use the term *attack scenario* (also known as *intrusion scenario* [37]) to describe a smallest combination of atomic attacks that can cause the attack goal to occur, which is similar to a minimal cut set in fault-tree analysis [39]. Figure 2.2 provides a simple example to describe the decomposition of an attack goal. In this tree, for example, to achieve the attack goal G_0 , either sub-goal G_1 or G_2 should be achieved first; similarly, to achieve the sub-goal G_1 , both atomic attack A_1 and A_2 need to be fulfilled. Therefore, there are three attack scenarios in this tree, i.e., three different ways to achieve G_0 : $\langle A_1, A_2 \rangle$, $\langle A_3 \rangle$ and $\langle A_4 \rangle$.

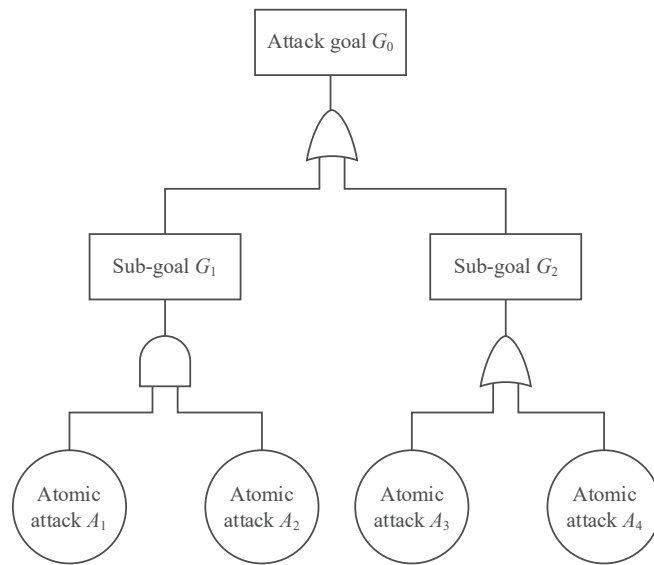


Figure 2.2. Example of an attack tree.

To generate an attack tree, the analyst should think from the perspective of the attacker (instead of the defender) with infinite resources, knowledge, and skill [40]. This could take considerable effort and time because the analyst needs to take account of all possible atomic attacks against the attack goal. Fortunately, attack trees are reusable. For example, once the PGP attack tree has been completed, anyone can use it in any situation that involves PGP [36].

Once all the nodes of an attack tree have been generated, the analyst can assign property values to each of them. The property values contain security information such as the

severity of the attack and the probability of occurrence, thus allowing people to better evaluate the attack. We will elaborate on that in Section 2.3.1.

2.3 The Framework

In this section, we present the main idea of our approach.

Figure 2.3 shows the general overview of the proposed framework, which can be decomposed into two phases: *pattern preparation* phase (shown as orange shaded boxes) and *pattern application* phase (shown as blue shaded boxes). In Figure 2.3, we use D , C , and P to represent the *designer*, *computer*, and *programmer*, respectively. A designer in this context is a security analyst responsible for designing patterns for vulnerabilities of interest.

The designer models attack goals by creating attack trees and constructing vulnerability-matching patterns, all of which will be stored in a *vulnerability knowledge base*. The computer, armed with a tool and the vulnerability knowledge base, detects vulnerable code during the program construction. The programmer interacts with the computer by constructing the program and fixing the vulnerable code. Moreover, the programmer might give useful feedback on the attack trees and patterns to make improvements to the vulnerability knowledge base. In our approach, there is no need for the programmer to possess much security expertise and to manually perform security analysis while coding because the manual work, including creating attack trees and constructing patterns, has been done by the designer in the pattern preparation phase. On the other hand, despite the fact that the manual work may require considerable time and effort from the designer, it is fortunately reusable, which means once the work has been done it can be reused by any other designer such that different designers do not need to repeat the process of pattern preparation for the same vulnerability.

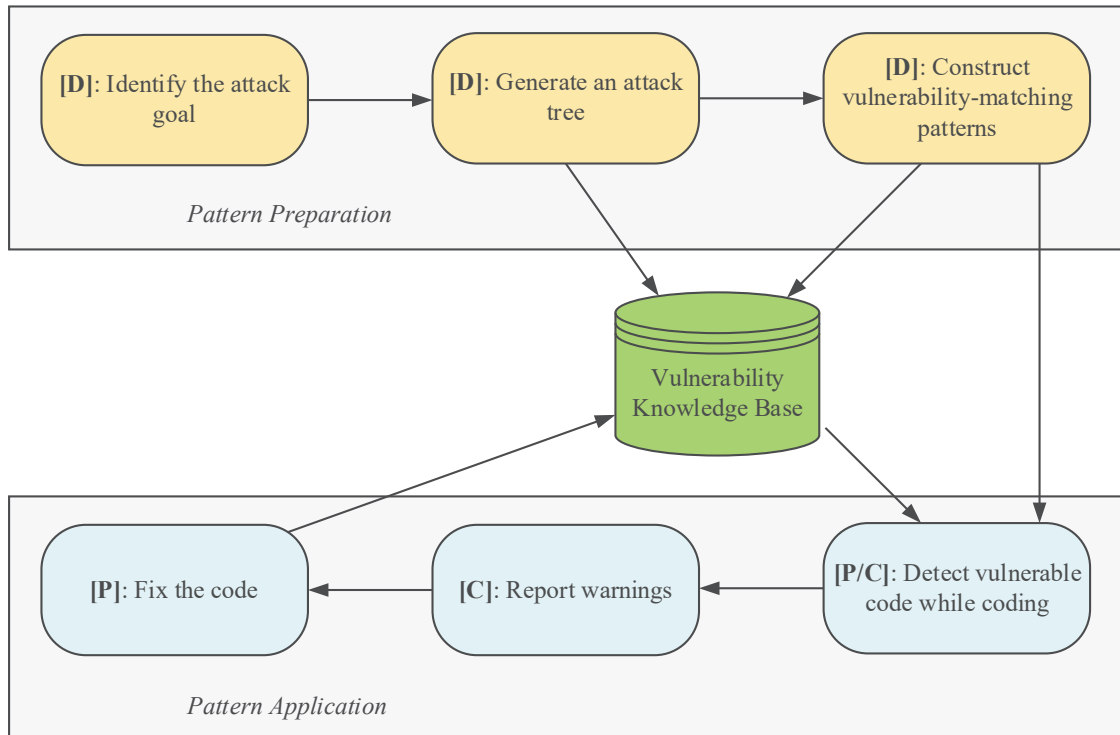


Figure 2.3. Overview of the proposed framework.

2.3.1 Pattern Preparation

This stage includes three activities: *identifying attack goals*, *generating attack trees*, and *constructing vulnerability-matching patterns*.

2.3.1.1 Identifying attack goals

In the activity of identifying an attack goal, the attack goal and the target system will be defined. Generally, the designer would select attack goals from common attacks occurred in the past or based on specific security requirements/specification. For example, the designer may refer to the common attacks listed in security-related databases, such as National Vulnerability Database (NVD) [41] and Common Weakness Enumeration (CWE) [42]. On the other hand, a designer from an enterprise may pay close attention to those attacks that can potentially compromise the systems of the enterprise.

2.3.1.2 Generating attack trees

In the activity of generating an attack tree, the attack goal will be decomposed as a set of sub-goals and atomic attacks, as shown in Figure 2.4.

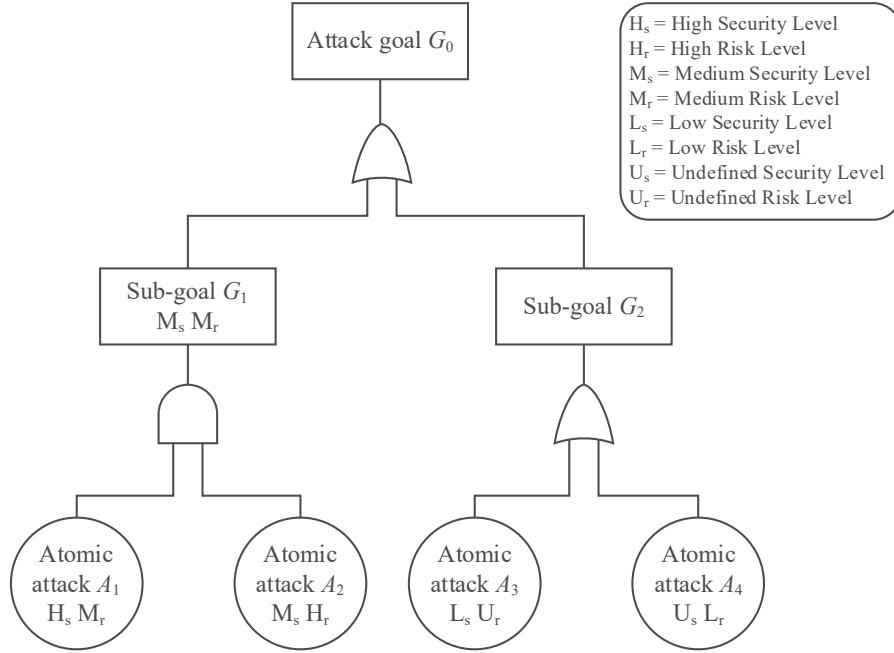


Figure 2.4. Generation of an attack tree.

In order to reflect the characteristics of each attack scenario, we use two property values, *security level* and *risk level*, to show the severity of the scenario and the probability of occurrence. That is, $S_{security} \cup S_{risk} \subseteq S$, where $S_{security}$ is the set of security-level values, S_{risk} the set of risk-level values and S the set of property values (see Definition 2.1).

Commonly, we would first assign the two property values to each atomic attack. To assign security level, we can use qualitative severity rankings of a set of values, such as $\{Low, Medium, High, Undefined\}$. As shown in Figure 2.4, L_s , M_s , H_s and U_s are used to represent *Low*, *Medium*, *High*, and *Undefined* security level, respectively. That is, $S_{security} = \{L_s, M_s, H_s, U_s\}$. The assessment criterion is mainly based on the severity of the attack, which can be measured by security metrics such as *confidentiality* impact, *integrity*

impact, and *availability* impact [43]. A successful attack against confidentiality, for example, may allow an unauthorized attacker to access the sensitive data of a system.

Similarly, L_r , M_r , H_r and U_r are used to represent *Low*, *Medium*, *High*, and *Undefined* risk level, respectively (see Figure 2.4), such that $S_{risk} = \{L_r, M_r, H_r, U_r\}$. The assessment criterion is based on the probability of occurrence of each atomic attack. The following provides a basic risk assessing method for roughly calculating the risk level.

- *Risk identification*: An attack is normally launched by the attacker who exploits certain vulnerability, but in some extreme cases it may be caused by system failures or user's unintentional manipulation. Therefore, there are two types of risk: hostile risk and random risk. To identify the type of risk can help the analyst choose an appropriate assessing method. When considering a hostile risk, for example, we should think mainly from the perspective of the attacker (instead of the defender).
- *Required resources calculation*: Consider performing the analysis for a hostile risk. We should analyze what resources are required for an attacker to exploit the vulnerability. The resources may include money, time, raw materials, knowledge, and skill. It is obvious that the more resources are required for an attack, the lower likelihood that the attacker will launch the attack.
- *Expected benefits calculation*: In this step, we will analyze what expected benefits an attacker can gain from a successful attack, by which attacker's motivation and expected returns can be learned. The more benefits are expected to gain from an attack, the greater likelihood that the attacker will launch the attack.

To calculate the risk level of a given atomic attack, we perform a cost-benefit analysis [44, 45] based on the resources and benefits mentioned above. For example, if an attack is expected to bring substantial benefits but to consume only a few resources, there would be high likelihood that the attack will occur, i.e., the risk level will be considered as H_r .

Once property values have been assigned to atomic attacks, we can then calculate the values for attack scenarios. There are two types of scenarios: AND-decompositions and OR-decompositions. For scenarios of OR-decompositions, we can directly use the property

values of each atomic attack. In Figure 2.4, for example, since the attack scenario $\langle A_3 \rangle$ is the atomic attack A_3 itself, they share the same property values, i.e., $S_{\langle A_3 \rangle} = S_{A_3} = \{L_s, U_r\}$. For scenarios of AND-decompositions, on the other hand, we need to take account of the values of both A_1 and A_2 when calculating the property values. A quick way to perform the calculations is to choose the minimal value between the two, for example, $Min(H_s, M_s) = M_s$. Accordingly, the property values of attack scenario $\langle A_1, A_2 \rangle$ in Figure 2.4 are M_s and M_r (i.e., $S_{\langle A_1, A_2 \rangle} = \{M_s, M_r\}$), as indicated in the higher-level node G_1 . Note that there is no need to show the values in G_2 because its lower-level nodes are OR-decompositions and cannot merge together simplistically.

However, the calculating method for scenarios of AND-decompositions mentioned above is overly simplistic especially when the attack scenario contains multiple atomic attacks. In the case of independent atomic attacks, a more accurate way is to calculate the product of probabilities of them.

2.3.1.3 Constructing vulnerability-matching patterns

In the activity of constructing vulnerability-matching patterns, patterns will be built for detecting vulnerable code during the process of vulnerability matching. Formally, a vulnerability matching is defined as follows.

Definition 2.2. A *vulnerability matching* is a function $cm: P \rightarrow \mathcal{P}(C)$ that maps patterns to vulnerable code, where P is a set of patterns, \mathcal{P} is the power set, and C is the set of vulnerable code fragments.

Note that a *code fragment* mentioned in this thesis can simply be an expression, a statement, or a block of programs. The construction of the patterns relies on the analysis of atomic attacks, which can be launched based on the exploitation of certain vulnerabilities. Therefore, the major concern is how to relate an atomic attack in an attack tree to a vulnerability in a code fragment.

Let V denote a set of vulnerabilities that can lead to the same atomic attack $a \in A$ (i.e., the atomic attack a is caused by any one vulnerability $v \in V$). For example, if the atomic

attack a is caused by a method $foo(int p)$ in Java, then any code fragments that call this method, such as $x.foo(a1)$ and $y.foo(a2)$, will be treated as potential vulnerabilities.

A *vulnerability-matching pattern*, or simply *pattern*, is formally defined as follows.

Definition 2.3. A *vulnerability-matching pattern* $p \in P$ is a pattern that can be used to match a set of code fragments $E \subseteq C$, each of which contains a vulnerability $v \in V$.

Figure 2.5 shows the process of pattern construction. Given an atomic attack $a \in A$, we extract a set of features F which indicate a is caused by a specific type of vulnerability, from which we conclude the vulnerability set V that relates to a . Based on the set V , we construct the pattern p using some certain techniques such as *regular expressions* [46, 47] and *taint analysis* [21]. The technique chosen to construct the pattern depends on the type of vulnerability. For example, regular expressions are efficient for matching vulnerabilities that consist lexical structure of constructs such as identifiers, constants, keywords, and white space, but they are unlikely to deal with nested structures [47].

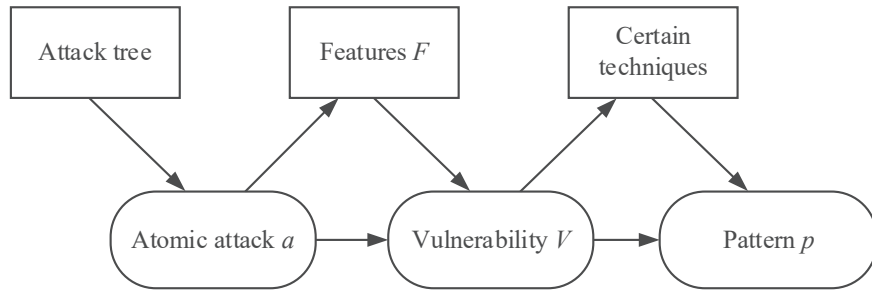


Figure 2.5. Process of pattern construction.

Once the pattern p is obtained, the designer would typically pay attention to the fact that whether it can reduce *false negatives* and *false positives*. False negatives mean that the pattern fails to match the real vulnerability while false positives mean that the pattern reports false alarms. Our approach is expected to achieve relatively low false negatives and positives because the original attack goal has been decomposed as a set of relatively simple and fine-grained atomic attacks that are easier to model. We formally define the false negative and positive as follows.

Definition 2.4. Let E be the set of code fragments that a pattern $p \in P$ should match in theory and let E' be the set of code fragments that the pattern p does match in practice. If there exists a code fragment $c \in E - E'$ that the pattern p fails to match, then a *false negative* occurs. If there exists a code fragment $c \in E' - E$ that the pattern p does match, then a *false positive* occurs.

To clarify the idea, Examples 2.1 and 2.2 use regular expressions to illustrate false negatives and false positives, respectively.

Example 2.1. Consider the code snippet in Figure 2.6 (a). Let us use a regular-expression pattern to match any method following $fw..$ If a pattern $fw[\backslashw.]+\backslash(.+\backslash)$ is used, then a false negative occurs due to the fact that it fails to match the method $fw.close()$ in this code, as shown in Figure 2.6 (b).

Example 2.2. Consider the code snippet in Figure 2.6 (a). Let us use a regular-expression pattern to match any method following $fw..$ If a pattern $fw[\backslashw.]+.+$ is used, then a false positive occurs due to the fact that it mismatches the filename $fw.txt$, which is not a method, as shown in Figure 2.6 (c).

```

1 // code sample
2 FileWriter fw = new FileWriter("fw.txt");
3 fw.write("a");
4 fw.write("bcd");
5 fw.write(123);
6 fw.close();

```

(a) Code sample.

```

1 // false negatives
2 FileWriter fw = new FileWriter("fw.txt");
3 fw.write("a");
4 fw.write("bcd");
5 fw.write(123);
6 fw.close();

```

3 results found for 'fw[\w.]+\(.+\)'

fw[\w.]+\(.+\)

(b) False negatives.

```
1 // false positives
2 FileWriter fw = new FileWriter("fw.txt");
3 fw.write("a");
4 fw.write("bcd");
5 fw.write(123);
6 fw.close();
```

5 results found for 'fw[\w.]+-'

fw[\w.]+.

(c) False positives.

Figure 2.6. Examples of false negatives and false positives.

After constructing the patterns, the designer should also work out a countermeasure against each corresponding vulnerability at this stage, so that the programmer can take it as a code fix suggestion. Ideally, the countermeasure is also expected to provide a secure code example, thus allowing the programmer to adopt it directly.

2.3.2 Pattern Application

This stage includes three activities: *detecting vulnerable code*, *reporting warnings*, and *fixing the code*.

2.3.2.1 Detecting vulnerable code

In the activity of detecting vulnerable code, particular code that contains the vulnerabilities will be automatically detected while the program is under construction. The detection will be performed by the computer based on the patterns constructed in the pattern preparation phase. In practice, the patterns will be stored in a vulnerability knowledge base, which can be read by a tool. We assume such a knowledge base and tool already exist when discussing pattern application. The vulnerable code will be captured in real time once it triggers the corresponding pattern, which is similar to searching specific strings using Unix `grep`.

2.3.2.2 Reporting warnings

In the activity of reporting warnings, the programmer will be informed of what and where the vulnerability is, and how to fix it. The warning report should include the location of the vulnerability, security and risk level information, and countermeasures. The security and risk level have been discussed in the pattern preparation phase. The countermeasures should also be prepared in the pattern preparation phase, and they will serve as suggestions for the programmer. Figure 2.7 shows an example of warning report. Also, the computer will give the programmer access to the attack trees and patterns for more details about the warnings.

<p>Warning(s): The code contains sensitive information Location: Line 20-30 Possible attack(s): SQL injection Security level: High Risk level: High Countermeasure(s): Do not contain any sensitive information</p>

Figure 2.7. Example of warning report.

2.3.2.3 Fixing the code

In the activity of fixing the code, the programmer can promptly examine and fix the vulnerable code according to the warnings provided by the computer. The programmer can also decide to dismiss the warnings if a false positive is found. Moreover, if the programmer is interested in viewing the attack trees and patterns, he/she can check them in the vulnerability knowledge base and give feedback. For example, if there is a false positive caused by an inaccurate pattern, the programmer can dismiss the warning and report the problem to the computer such that the computer may update the vulnerability knowledge base by revising the pattern.

Figure 2.8 shows the interaction between the programmer and computer during pattern application phase.

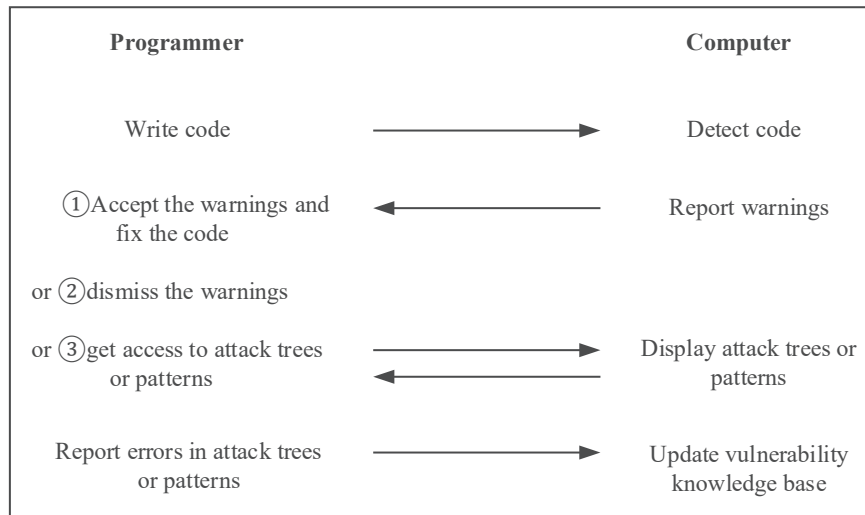


Figure 2.8. Interaction between the programmer and computer.

2.4 Case Study

In this section, we will illustrate the proposed framework in a case study.

We focus on a common security issue called SQL injection attacks (SQLIAs), which is mainly caused by insecure code or lack of input validation. As one of the *Most Dangerous Software Weaknesses* listed in the 2020 Common Weakness Enumeration (CWE) [42], SQLIAs can pose a serious threat to many web applications. Suppose the target system in our analysis is a web-based stock exchange trading system. This system allows customers and companies to register, buy or sell stocks.

Based on the proposed framework, we describes the entire process for modeling and detecting SQLIAs from pattern preparation to pattern application. The steps given below correspond to the ones described in the preceding section.

2.4.1 Modeling SQLIAs

2.4.1.1 Identifying the attack goal

We select the SQLIAs as the attack goal and the web-based stock exchange trading system as the target system.

2.4.1.2 Generating attack trees

We generate the attack tree against SQLIAs, as shown in Figure 2.9. Note that a complete attack tree of SQLIAs could be much more complicated as it involves many different types of attacks and countless variations [48, 49]. For the sake of illustration, we omit some details and generate a simplified, incomplete version.

Once the nodes of the attack tree have been generated, we calculate the property values of security level and risk level for each atomic attack and attack scenario in the tree. As an example, the following uses the assessing methods described in Section 2.3.1 to illustrate how to calculate the security and risk level for the atomic attack *Construct Malicious Values*, i.e., node 1.1.1 of Figure 2.9, which is also an attack scenario $\langle 1.1.1 \rangle$.

For security level of node 1.1.1, we assess it based on the security metrics. Since such a successful attack can easily bypass the authentication (see next step for detailed discussion), the confidentiality will be violated. Moreover, integrity and availability will be violated because the attacker might modify or delete customers' data via launching this type of attack [48]. Therefore, we would assign the value H_s to indicate the security level of this threat.

For risk level of node 1.1.1, we first identify that this type of risk is a hostile risk. Second, we analyze what resources are required to perform this atomic attack. Since this type of SQL injection is common and easy to perform (see next step), it does not involve many resources such as considerable time or money. All resources the attacker needs are a computer and some basic security knowledge. Finally, the expected benefits are good enough for the attacker to risk because this type of attack allows the attacker to gain much information from the database. For example, some customers' stock trading information stored in the system will be revealed. Based on this cost-benefit analysis, we would consider the risk level of this atomic attack as H_r .

As discussed, the atomic attack 1.1.1 and the attack scenario $\langle 1.1.1 \rangle$ share the same property values H_s and H_r , i.e., $S_{\langle 1.1.1 \rangle} = S_{1.1.1} = \{H_s, H_r\}$.

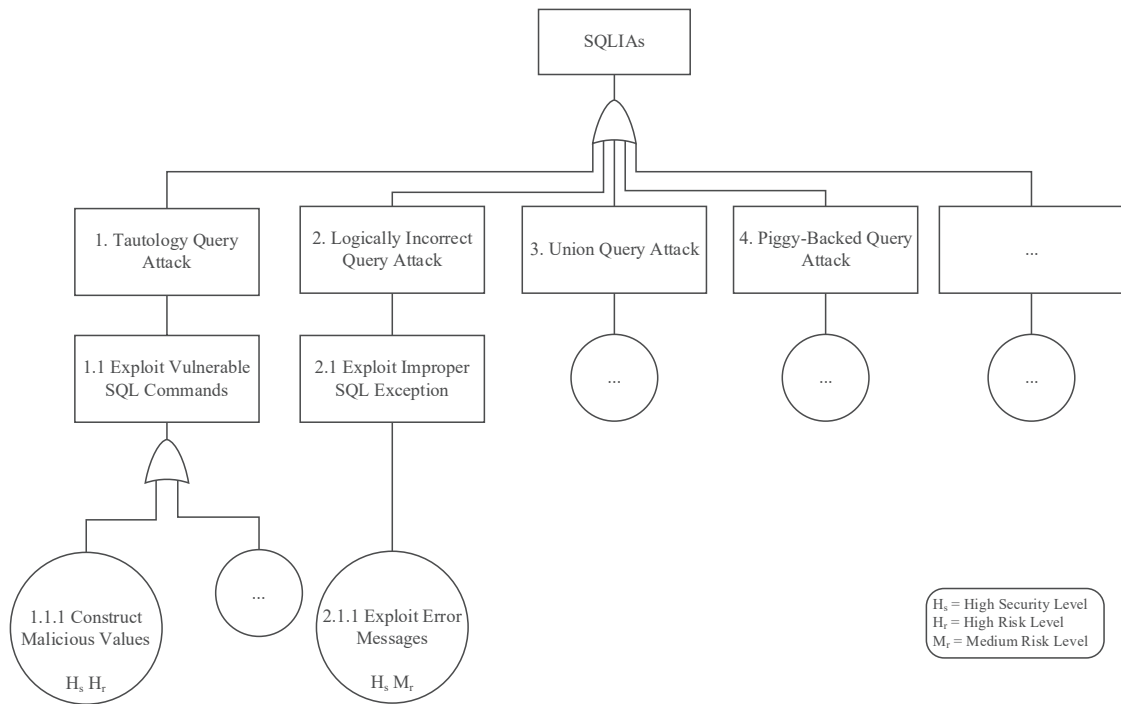


Figure 2.9. Attack tree against SQL injection.

2.4.1.3 Constructing vulnerability-matching patterns

In this step, we use an example to show how to construct a pattern based on certain techniques, including regular expressions and taint analysis.

Consider that we want to construct a pattern for capturing vulnerabilities related to the attack scenario (1.1.1) in Figure 2.9. For illustration, we take an example of the code fragment in Figure 2.10.

```

1 //Use string concatenation to build the query
2 String name, password, query;
3 name = getParameter("name");
4 password = getParameter("pwd");
5 Connection con ...
6 query = "SELECT * FROM customer WHERE name = '" + name +
7         "' AND pwd = '" + password + "'";
8 con.execute(query);

```

Figure 2.10. Illustrative code fragment.

The code in Figure 2.10 is susceptible to <1.1.1> because it creates SQL statements by using string concatenation [50] and the attacker can thus dynamically construct and execute a malicious SQL query. For example, the attacker can enter the string `abc' OR 1 = 1 --` – for the name input field and the query becomes:

```
SELECT * FROM customer WHERE name = 'abc' OR 1 = 1 --' AND
pwd = ' ';
```

The comment operator `--` makes the `pwd` input field irrelevant. Since `1 = 1` is always true, the `WHERE` clause will always evaluate to true. In other words, the `WHERE` clause will be transformed into a tautology and the attacker can finally bypass the authentication even if he/she does not know what the name or password is.

a) *Regular-expression-based pattern*: Based on the analysis above, we extract a set of key features F from the query: keywords such as `SELECT`, concatenation (using single quotes), and semicolon. Accordingly, a regular-expression-based pattern for this type of vulnerability might be created as follows:

```
(\w+\s*=\s*)+"SELECT\s\S+\sFROM\s\S+\sWHERE\s\S+\s*=\s*" [^;
]*
```

The following is a more readable way to describe it:

```
(\w+\s*=\s*)+ /* variable name and equal sign */

"SELECT /* matches ", followed by SELECT */

\s\S+\s /* whitespace, anything not whitespace, and whitespace */

FROM /* keyword FROM */

\s\S+\s /* whitespace, anything not whitespace, and whitespace */

WHERE /* keyword WHERE */
```

```

\s\S+\s* /* whitespace, anything not whitespace, and 0 or more whitespace */

=\s* /* matches =, followed by 0 or more whitespace */

'[^;]* /* matches ', followed by anything not ; */

```

b) *Taint-analysis-based pattern*: In this example, the input variables name and password are considered tainted [21] because they are returned from a method *getParameter* (called the *source*) that gets unchecked input. The tainted variables name and password are passed to the variable query on line 6 and finally *con.execute* on line 8. Since the original source data are untrusted, the call to the method *execute* (called the *sink*) on line 8 is potentially unsafe. In this case, the key features F that we extract for constructing a pattern should contain the source method (*getParameter*), the sink method (*execute*) and the data-propagation information. Accordingly, the taint-analysis-based pattern can be formulated as $\{getParameter, \text{data propagation}, execute\}$, where data propagation gives information about whether the tainted data from *getParameter* can be passed to *execute*.

Compared with regular expressions, taint analysis is more effective and precise to detect SQL injections because the data-propagation analysis generally involves *pointer analysis*, which clearly shows what a variable may refer to [51]. For illustration, the rest of the case study will only show the case of using a regular-expression pattern.

Finally, after constructing a pattern, we should work out a countermeasure against the attack scenario at this stage so that the programmer can take it as a code fix suggestion. For example, using parameterized queries [50] instead of string concatenation to build queries is one possible solution to avoid this type of SQL injection.

2.4.2 Detecting SQLIAs

2.4.2.1 Detecting Vulnerable Code

As shown in Figure 2.11, line 6-7 is the corresponding vulnerable code captured by the regular-expression pattern indicated at the bottom of the figure.

```
1 //Use string concatenation to build the query
2 String name, password, query;
3 name = getParameter("name");
4 password = getParameter("pwd");
5 Connection con ...
6 query = "SELECT * FROM customer WHERE name = '" + name +
7       "' AND pwd = '" + password + "'";
```

1 result found for '(\\w+\\s*=\\s*)+"SELECT\\s\\s+\\sFROM\\s\\s+\\sWHERE\\s\\s+\\s*=\\s"[^"]*'

(\\w+\\s*=\\s*)+"SELECT\\s\\s+\\sFROM\\s\\s+\\sWHERE\\s\\s+\\s*=\\s"*[^\"]*

Figure 2.11. Detection for vulnerable code.

2.4.2.2 Reporting Warnings

The warnings include the location of vulnerable code, the type of possible attack, security and risk level information, and countermeasures, as shown in Figure 2.12. The location is revealed in step 4 while the security and risk level have been discussed in step 2 respectively. The countermeasure, as mentioned in step 3, is also given.

<p>Warning(s): The SQL query uses string concatenation Location: Line 6-7 Possible attack(s): SQL injection Security level: High Risk level: High Countermeasure(s): Consider using parameterized queries</p>

Figure 2.12. Warning report for the illustrative code.

2.4.2.3 Fixing the Code

Finally, the programmer can examine and fix the code according to the warning report. For example, the programmer might accept the suggestion and use a parameterized query as follows:

```
query = "SELECT * FROM customer WHERE name = ? AND pwd = ?";
```

This query uses question marks as placeholders, which can help avoid SQL injection. For example, if the attacker tries to enter `abc' OR 1 = 1 --` for the name input field, the entire input will be inserted into the name field as a name and no SQL injection will occur [50].

2.5 Discussion

The framework discussed in this chapter attempts to give a generic way of modeling vulnerabilities. We use attack trees to model a given class of vulnerabilities and then crafts patterns for each individual vulnerability. However, one significant weakness is that such pattern construction may require substantial preliminary work especially when we need to do so from scratch. Therefore, it may not be practical when used to model some vulnerabilities that require significant pattern preparation.

In addition, the patterns discussed in this chapter are mainly syntax-based, such as regular expressions-based. Loosely speaking, syntax-based analysis is efficient but may not be precise enough since it does not explore the behavior of the code. By contrast, semantic-based analysis is generally less efficient but more precise. Semantic-based analyses, such as taint analysis and pointer analysis, have been mentioned in this chapter, but not extensively discussed yet. In Chapter 3 we will study thoroughly how to perform such semantic-based analyses in Human-Machine Pair Programming.

Finally, as a first step towards vulnerability discovery in Human-Machine Pair Programming, it lacks a more concrete evaluation of the framework.

2.6 Related Work

Static analysis is a popular method for uncovering security-related bugs during software development [52]. Static analysis techniques can be employed to statically examine the source code of a program without executing it [32]. In the following we discuss some closely related work, i.e., pattern-based static analysis approaches. Basic lexical analysis is

adopted by practical tools such as ITS4 [10] for identifying security vulnerabilities in C and C++ code. The tool ITS4 breaks the source code into a set of lexical tokens and then matches vulnerable functions from a database. Larochelle and Evans [30, 53] use annotations to syntactically perform static analysis for detecting buffer overflow vulnerabilities. The annotations can be exploited to check whether the code is consistent with certain properties. Yamaguchi et al. [29] merge abstract syntax tree, control flow graphs and program dependence graphs into a joint data structure, in which analysts craft certain rules, known as *traversals*, to facilitate vulnerability auditing. While these approaches are effective and promising, they provide little discussion on analysis of incomplete programs during coding phase.

Another branch of research is to perform bug detection during coding phase. It is beneficial yet challenging. Compilers set a good example for us in the sense of real-time error detection, but the errors they report, such as syntax errors and semantic errors [47], are different from the security vulnerabilities discussed in this thesis. Human-Machine Pair Programming [34] aims to provide a compiler-like way to statically find bugs in the coding phase. Though the work in [34] discusses how to find semantic faults that are not detected by compilers, it does not discuss a way for vulnerability discovery. Following this strategy, our work in this chapter presents a generic framework specifically designed for vulnerability discovery in Human-Machine Pair Programming.

3 Vulnerability Discovery in Human-Machine Pair Programming: Using Pointer Analysis²

In Chapter 2, we have described a framework for vulnerability discovery in Human-Machine Pair Programming. As discussed, the framework intends to provide a general idea but may not be readily used under certain circumstances. This chapter will delve deeper into the problem of vulnerability discovery in Human-Machine Pair Programming.

As mentioned previously, static analysis is one of the predominant techniques for vulnerability discovery. One of the fundamental methods used in static analysis is *pointer analysis*, which models heap objects and computes what a pointer variable or expression can refer to [26]. Such pointer information can be used to facilitate the detection of various types of bugs. For example, Use-After-Free vulnerabilities, one of the leading weaknesses in NVD database [41], can be detected by pointer-analysis-based approaches [54]. Moreover, taint analysis [21], a technique often closely associated with pointer analysis, supports the detection of taint-style vulnerabilities [18], such as buffer overflow vulnerabilities [27], command injection vulnerabilities [55], and cross-site scripting (XSS) vulnerabilities [56].

Generally, pointer analyses are performed on complete programs, since analysis of partial programs may be imprecise and can thus introduce a considerable number of false positives and negatives. Nonetheless, analyzing incomplete programs can be beneficial and has been attractive to researchers (e.g., [57, 58]). The major benefit is that one can obtain the analysis results of a program fragment of interest even if the whole program is unavailable. The nature of such analysis enables early vulnerability detection during program development. Ideally, it can be used to support real-time detection during coding, analogous to how a compiler reports errors during compilation.

² This chapter is adapted from our work in [59] and [60].

In this chapter, we will discuss a combination of taint and pointer analyses in HMPP specifically for identifying taint-style vulnerabilities [59, 60].

3.1 Background

This section briefly introduces the concepts of taint analysis and pointer analysis, both of which will be incorporated into our approaches in this chapter.

3.1.1 Taint Analysis

In *taint analysis* [21, 61], a tainted value refers to a value derived from untrusted external input such as a command-line argument or a returned value from a procedure call. An original program location (such as a call to a method in Java) that receives tainted values is called a *source*, while a *sink* is another program location that should not receive tainted values. A security flaw can be introduced when a tainted value reaches a sink from a source via an execution path. For example, in a web application an SQL Injection may occur if a source method accepts attacker-controlled input (i.e., tainted data) and the input is directly incorporated into an SQL query executed by a sink method. Such a taint-style vulnerability is also referred to as a *leak* since it can cause sensitive information leakage.

To further illustrate, consider the following simple Java code as an example:

```
a = source();    // variable a is tainted  
[...]  
sink(b);        // Is this a leak?
```

In this example, the variable *a* accepts a tainted value from the source method. What we care about is whether the tainted value would be eventually passed to the sink method and result in a leak. In essence, performing taint analysis of this example is to confirm whether *b* is tainted due to its potential connection to variable *a* in the omitted code [...] above. For instance, *b* may be an alias of *a*.

3.1.2 Pointer Analysis

Most taint analysis approaches are based on data-flow analysis or pointer analysis. In fact, some researchers have unified taint analysis and pointer analysis under certain circumstances [51, 62]. Pointer analysis [26, 63], also known as *points-to analysis* [64, 65], is to statically compute the possible objects that each program variable or expression may point to. In addition, *alias (or aliasing) analysis* [66], often used as a synonym for pointer analysis, aims to determine whether two variables are aliased by observing if they may point to the same object.

The points-to information allows security analysts to observe whether certain sensitive program locations may receive unexpected values and introduce security risks. For example, a pointer analysis of the above code example (Section 3.1.1) can compute points-to results to answer the query “may the variable *b* point to a tainted object?”. (A query in this work is a request for the points-to information of a particular variable.) Specifically, if *b* and the known taint *a* point to the same object, which implies *b* is an alias of *a*, then *b* is also tainted. Consequently, the call to *sink(b)* may cause a leak because the tainted value reaches a sink.

3.2 Proposed Approaches

3.2.1 Motivating Example

Consider a Java code fragment in Figure 3.1. To spot taint-style vulnerabilities, our analysis objective is to confirm whether the tainted data from the source method (line 6) may be passed into the sink methods (lines 5 and 7).

```

1  void main() {
2      a = new A();
3      b = a;
4      c = new C();
5      sink(b.f);
6      a.f = source();
7      sink(b.f);
8      d = c;
9      [...]
10 }

```

Figure 3.1. Motivating code example.

A typical pointer analysis (e.g., Andersen-style analysis [29]) of this code might show points-to relations as follows:

$$\begin{aligned}
 a &\hookrightarrow o_2 \\
 b &\hookrightarrow o_2 \\
 c &\hookrightarrow o_4 \\
 d &\hookrightarrow o_4 \\
 o_2.f &\hookrightarrow source.ret
 \end{aligned}$$

The symbol \hookrightarrow simply means “points to”. We use o_i to denote an object created at the allocation site i . In this example, o_2 represents an object created at line 2, namely `new A()`. We can see that the variables b and a are aliases of each other because they point to the same object o_2 , which also implies $b.f$ is an alias of $a.f$, both of which are denoted by $o_2.f$. The same observation applies to the variables c and d . Since the details of the source (line 6) are unclear and unimportant in the code fragment, we use `source.ret` to denote the taint object pointed by the return value.

The above points-to results and the taint information together are sufficient to uncover the security flaws. As `source.ret` is a taint pointed by $o_2.f$, any `sink(o_2.f)` would be recognized as a leak. Consequently, two leaks are reported at lines 5 and 7, respectively.

However, it is not hard to notice that the leak reported at line 5 is a false alarm as $b.f$ is not tainted until line 6 is executed. This is a common drawback of many pointer analysis approaches (e.g., [51, 64]) as they typically treat all statements as a set and ignore the statement order.

Another limitation of the analysis is that the points-to results of the variables c and d in this code are irrelevant to the vulnerability analysis, thus causing extra computation. Furthermore, this analysis does not support vulnerability discovery in real time, i.e., identify the flaw immediately when it appears in the code during coding. We will discuss all these issues in the subsequent subsections.

3.2.2 Exhaustive Pointer Analysis

3.2.2.1 Human-Machine Pair Programming fashion

Figure 3.2 shows the overview of our approach. The pointer analysis plays the role of the syntactical analysis in HMPP (see Figure 2.1 in Chapter 2 for comparison), and the taint information serves as the knowledge base. The core idea is to perform pointer analysis on the ever-changing CV_S .

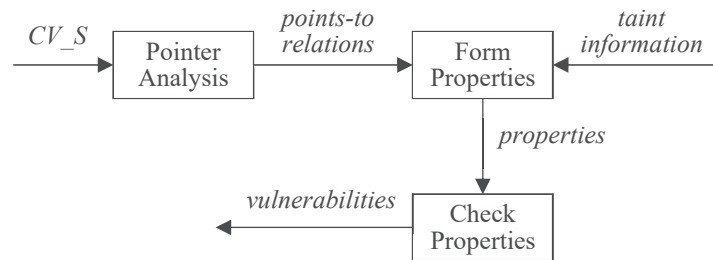


Figure 3.2. Overview of our approach.

To perform pointer analysis in such an HMPP fashion, it requires us to compute points-to results in an incremental fashion, i.e., start from the computation of the first line of code and then incrementally add new results as coding progresses. Meanwhile, we immediately combine the points-to information with necessary taint information to find potential vulnerabilities, without concerning with the succeeding code. For example, when the

programmer completes the coding at line 7 in the motivating example (Figure 3.1), our approach should immediately report a leak there and need not wait for more code to come out because the flaw has been determined at that point. As a result, vulnerabilities can, in principle, be discovered in real time, allowing the programmer to cope with them during program development.

3.2.2.2 *Flow sensitivity*

Since our objective is to perform vulnerability discovery in the coding phase, we cannot tolerate too many spurious alarms. (Otherwise, programmers would be overwhelmed by the alarms and unable to concentrate on programming.) To this end, flow sensitivity is introduced for better precision. Instead of lumping all the statements together as a set, a flow-sensitive analysis takes into account the control flow and analyzes statements in order. In the context of HMPP, it means to perform a pointer analysis on statements in order and in an incremental manner as coding proceeds. To illustrate, consider again the code in Figure 3.1. A flow-sensitive analysis of this code example in HMPP is performed as follows (note that we input the code line by line and compute the points-to results incrementally):

```
line 2:  $a \hookrightarrow o_2$   
line 3:  $b \hookrightarrow o_2$   
line 4:  $c \hookrightarrow o_4$   
line 5: \  
line 6:  $o_2.f \hookrightarrow source.ret$   
line 7: \  
...
```

The symbol `\` means that a points-to relation is not involved or unknown at that point. As the results are computed, we feed the taint information to detect vulnerabilities at the same time:

```
...
```

line 5: $sink(o_2.f)$
 line 6: $o_2.f \hookrightarrow source.ret$
 line 7: $sink(o_2.f)$
 ...

Unlike the analysis in Section 3.2.1, since the statement order is provided here, we can clearly see that only the call to $sink(o_2.f)$ at line 7 will trigger a leak alarm. Thus, the false alarm at line 5 is eliminated. More importantly, in this analysis the leak at line 7 can be immediately identified prior to the input of succeeding statements (i.e., need not consider line 8, line 9, ...), indicating a real-time discovery.

3.2.2.3 Conservative approximation

Since static analysis is, in general, undecidable [30], we discuss how to make a conservative approximation in our approach. Consider the code fragment given in Figure 3.3, which is adapted from Figure 3.1. Though the two code fragments have equivalent functionality, an additional difficulty may arise when analyzing the code in Figure 3.3. That is, suppose the programmer has completed the first six lines of code but the foo method (lines 8-11) has not been constructed, can we discover the leak at line 6?

```

1  void main() {
2      a = new A();
3      b = a;
4      sink(b.f);
5      a.f = foo(p);
6      sink(b.f); ①
7  }
8  Object foo(Object q) {
9      q = source();
10     return q; ②
11 }

```

Figure 3.3. Code example adapted from Figure 3.1.

We address this problem by making a conservative approximation: if a method may introduce taints, then we consider it as a source. For example, though the body of the *foo* method is unknown at line 5, the programmer should have realized whether the *foo* method would accept tainted values according to requirements or other information (e.g., whether it may accept user-supplied input). Once the *foo* method is marked as a source at line 5, then the leak at line 6 can be discovered immediately even though the succeeding code, i.e., lines 7-11, is still under construction.

More formally, we use a placeholder procedure p_ph_i to denote an unknown procedure, which can be a procedure that has not been created, or a procedure from an external module whose source is unavailable. Similarly, an unknown variable can be denoted by a placeholder variable v_ph_i . Such representations enable us to conceptually treat an analyzed code fragment as a complete program. The placeholder p_ph_i will be treated as a variable, so its points-to information will be computed and used during our analysis. For example, when the *foo* method at line 5 of Figure 3.3 is unknown, we denote the statement as $a.f = p_ph_1$, and the points-to results can be yielded as:

line 5: $p_ph_1 \hookrightarrow o_5$

line 5: $o_2.f \hookrightarrow o_5$

For illustration, we use a function $taint(var) = \{tainted, untainted, unknown\}$ to define the taint status of an arbitrary variable var . In this example, since p_ph_1 is unknown, it is safe to denote its taint status as $taint(p_ph_1) = unknown$, and therefore $taint(o_2.f) = unknown$. However, unknown taint status may not be helpful in vulnerability analysis. Hence, we try to further determine the taint status of any unknown v_ph_i or p_ph_i ; for instance, p_ph_1 here can be marked as tainted if it may involve tainted values and marked as untainted otherwise.

Depending on demand, we can make an *optimistic* or *pessimistic assumption* about the unknown v_ph_i or p_ph_i . Making an optimistic assumption is to “optimistically” regard the missing code as untainted whenever possible, which can typically lead to fewer false positives but more false negatives, whereas a pessimistic assumption tends to yield fewer

false negatives but more false positives. In this work, we choose to make a pessimistic assumption because in many cases missing a leak is more undesirable than producing a false alarm.

3.2.2.4 Pointer propagation rules

We proceed by giving deduction rules for handling pointer analysis in the coding phase. There are several different ways of computing points-to relations, such as using Datalog-based rules [26] or rules for pointer assignment graphs [64]. Table 3.1 gives our rules adapted from the work in [64].

Table 3.1. Pointer propagation rules.

	Instruction	Edge	Rule
<i>Alloc</i>	$i: a = \text{new } C$	$o_i \rightarrow a$	$\frac{o_i \rightarrow a}{i: \{o_i\} = / \in pt(a)}$
<i>Assign</i>	$j: a = b$	$b \rightarrow a$	$\frac{\begin{array}{l} b \rightarrow a \\ \{o_i\} = / \in pt(b) \end{array}}{j: \{o_i\} = / \in pt(a)}$
<i>Store</i>	$k: a.f = b$	$b \rightarrow a.f$	$\frac{\begin{array}{l} b \rightarrow a.f \\ \{o_i\} = / \in pt(a) \\ \{o_j\} = / \in pt(b) \end{array}}{k: \{o_j\} = / \in pt(o_i.f)}$
<i>Load</i>	$l: a = b.f$	$b.f \rightarrow a$	$\frac{\begin{array}{l} b.f \rightarrow a \\ \{o_i\} = / \in pt(b) \\ \{o_j\} = / \in pt(o_i.f) \end{array}}{l: \{o_j\} = / \in pt(a)}$
<i>Call</i>	$m: a = b.f(p_1, p_2, \dots, p_n)$	$\begin{array}{l} p_1 \rightarrow \text{meth}_{q1} \\ p_2 \rightarrow \text{meth}_{q2} \\ \dots \\ p_n \rightarrow \text{meth}_{qn} \\ \text{meth}_{ret} \rightarrow a \end{array}$	$\frac{\begin{array}{l} \{o_i\} = / \in pt(b) \\ \text{meth} = \text{dispatch}(o_i, f) \\ \{o_{u_j}\} = / \in pt(p_j), 1 \leq j \leq n \\ \{o_v\} = / \in pt(\text{meth}_{ret}) \end{array}}{m: \begin{array}{l} \{o_{u_j}\} = / \in pt(\text{meth}_{qj}), 1 \leq j \leq n \\ \{o_i\} = / \in pt(\text{meth}_{this}) \\ \{o_v\} = / \in pt(a) \end{array}}$

In Java, there are usually five kinds of pointer-related instructions, namely *Alloc* (allocations), *Assign* (assignments), *Load* (field loads), *Store* (field stores), and *Call* (method calls). The arrow symbol \rightarrow is the edge that reflects the pointer flow in a pointer assignment graph. $Pt(var)$ denotes the points-to set of var . The operator “ $= / \in$ ” means

either “=” or “ \in ”; for example, $\{x\} =/\in pt(y)$ denotes that either $pt(y) = \{x\}$ or $\{x\} \in pt(y)$ will be computed. This allows us to determine an exact points-to relation whenever possible; for example, if we can be sure that y can only point to x at a certain point, then we denote the relation as $pt(y) = \{x\}$, and as $\{x\} \in pt(y)$ otherwise.

The upper part of a deduction rule is the *hypothesis* (or *premise*), and the lower part is the *conclusion*. The first four rules are relatively straightforward and similar to those described in [26], with a major difference that we try to determine the exact object pointed by a pointer at a particular program location. For example, the *Alloc* rule states that if we create an object o_i and assign it to the variable a , denoted by the edge $o_i \rightarrow a$, then we can conclude that $i:\{o_i\} =/\in pt(a)$. In contrast, in [26] (and other subset-based pointer analysis solutions), the deduction conclusion is in the form of $o_i \in pt(a)$, i.e., add o_i to the points-to set of a . We make such changes in the rules in order for a more precise analysis in HMPP.

The rule of *Call* may be less intuitive, so we illustrate it by taking the code in Figure 3.3 as an example. The statement $a.f = foo(p)$ at line 5 invokes a method. According to the *Call* rule in Table 3.1, we first resolve the method in question with the assistance of its method signature, and then relate the arguments to the parameters, i.e., add an edge from p to q (edge ①). There is also an edge from q to $a.f$ (edge ②) since $a.f$ is the variable that receives a return value. Finally, the points-to relations propagate along the edges. In this example, after the propagation the points-to results will show us that since *source.ret* is pointed by the return variable q (lines 9-10), it is also pointed by $a.f$ (or more precisely, $o_2.f$).

3.2.3 Demand-Driven Pointer Analysis

Though the exhaustive pointer analysis described above can successfully yield the expected results, it is often too costly since it computes the points-to information for all program variables, including irrelevant ones. For example, the points-to computation of the variables c and d in Figure 3.1 is irrelevant and thus incurs extra overhead. A solution to this issue is to employ the demand-driven pointer analysis [67, 68], which computes only

the results that may contribute to answering a particular query. In the example in Figure 3.1, the information of c and d is not necessary for answering the query “is $b.f$ at line 7 tainted?”, so we can exclude it from the computation.

Figure 3.4 shows how a demand-driven pointer analysis can be performed in an HMPP fashion.

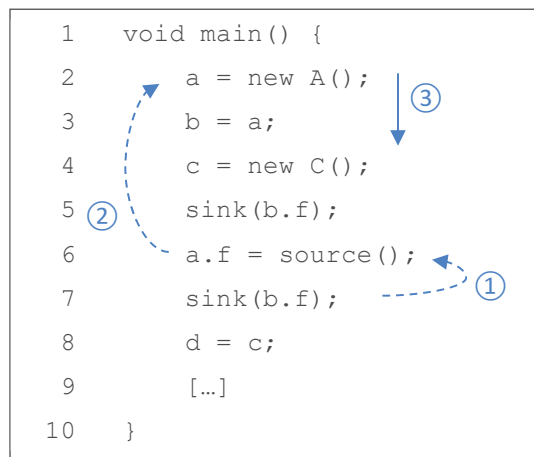


Figure 3.4. Demand-driven pointer analysis of the code example in Figure 3.1.

The analysis starts from a sink, rather than from the first line of code. For the sink at line 5, we search backward and notice that no source is involved in preceding statements, so the $b.f$ is not tainted. Similarly, to answer the query “is $b.f$ at line 7 tainted?”, we search backward for a source, as shown in step ①. Then in step ②, we trace the variable a to line 2. Step ③ indicates that we perform a pointer analysis forward for all the statements involved a (and its aliases) based on the pointer propagation rules in Table 3.1. The points-to results are as follows:

- line 2: $a \hookrightarrow o_2$
- line 3: $b \hookrightarrow o_2$
- line 6: $o_2.f \hookrightarrow source.ret$

As explained in Section 3.2.1, the results indicate the presence of a leak at line 7. Note that the variables c and d are not involved in this on-demand points-to analysis, thus avoiding unnecessary computation. The idea is formalized in Algorithm 3.1.

Algorithm 3.1 Demand-driven Analysis.

INPUT: A source program P under construction and an ordered n -tuple $S = ()$

OUTPUT: Points-to relations

```

1:  foreach sink statement  $a.\text{sink}()$  in  $P$  do
2:      search backward for the source;
3:      foreach source statement  $x = b.\text{src}()$  do
4:          Trace backward for all statements  $S'$  involved  $a, b, x$ , or their aliases;
5:          add  $S'$  to  $S$ ;
6:          Track the propagation of  $x$  forward between the source and the sink;
7:          if (any variable is tainted by the propagation of  $x$ )
8:              add the involved statements to  $S$ ;
9:      foreach statement  $s \in S$  do
10:         apply pointer propagation rules and compute points-to results

```

In the algorithm, the ordered statements in our analysis will be kept in an ordered n -tuple, as opposed to a set, since it can represent ordered collections. Note that the order of statements in the tuple S should be consistent with that in the program P , i.e., if a statement s_i precedes another statement s_j in P , then s_i also precedes s_j in S , where $s_i, s_j \in P \cap S$.

3.2.4 Difference and Equivalence

The exhaustive pointer analysis is performed in an incremental fashion and the analysis results are accumulated as the coding process progresses, making the response time for a vulnerability short. The drawback of this approach is that it may incur extra computation of irrelevant variables. In contrast, the demand-driven approach eliminates the irrelevant computation, at the expense of a relatively delayed response time as the on-demand analysis does not start until a sink appears in the code.

Despite the difference stated, we emphasize that the two approaches are equivalent in the sense that they compute identical necessary results for a particular query. As shown previously, to answer the query “is $b.f$ at line 7 tainted?”, both approaches compute the same necessary points-to relations, although the exhaustive approach computes extra irrelevant information. A more formal description of the equivalence is provided as follows.

Let V be the set of pointer variables and let O be the set of abstract objects in a program P . The entire points-to relations of our exhaustive approach can be encoded by a mapping $pt: V \rightarrow \wp(O)$, where $\wp(O)$ is the power set of O . Similarly, let U be the set of pointer variables involved in our demand-driven approach and let O_D be the set of abstract objects, where $U \subseteq V$ and $O_D \subseteq O$. Then we can encode its points-to relations by a mapping $pt_D: U \rightarrow \wp(O_D)$, where $\wp(O_D) = \{pt(u) | u \in U\}$. Compared with the demand-driven approach, the exhaustive approach computes extra points-to results for the variables in $(V - U)$, which are, however, irrelevant information in our vulnerability analysis. Given a specific version of program, let us suppose that $v \hookrightarrow o$ is an arbitrary points-to relation contributed to vulnerability analysis. We need to show that the exhaustive approach computes $v \hookrightarrow o$ if and only if the demand-driven approach computes $v \hookrightarrow o$ [31].

Lemma 1. *If the exhaustive approach computes $v \hookrightarrow o$, then the demand-driven approach computes $v \hookrightarrow o$.*

Proof. Since $v \in U \subseteq V \wedge o \in O_D \subseteq O$ and variables in $(V - U)$ do not affect points-to results of v , the demand-driven approach computes the same results for v as the exhaustive approach does, i.e., v also points to o . \square

Lemma 2. *If the demand-driven approach computes $v \hookrightarrow o$, then the exhaustive approach computes $v \hookrightarrow o$.*

Proof. According to proof by contraposition, this is equivalent to prove that “if the exhaustive approach does not compute $v \hookrightarrow o$, then the demand-driven approach does not compute $v \hookrightarrow o$ ”. Since the demand-driven’s points-to results are a subset of the

exhaustive's, the demand-driven approach will not compute $v \hookrightarrow o$ when the exhaustive approach does not. Thus, the lemma is true. \square

Combining Lemma 1 and 2 we can prove Theorem 1.

Theorem 1. *The exhaustive approach computes $v \hookrightarrow o$ if and only if the demand-driven approach computes $v \hookrightarrow o$.*

Therefore, our two approaches are equivalent in the context of our vulnerability analysis.

3.3 Evaluation

In this section, we present an evaluation of our two approaches on a security benchmark, followed by a discussion of our experiment results and a real-world case study.

3.3.1 Experiment

3.3.1.1 Preliminaries

The benchmark we used is Securibench Micro [69], which includes a series of small test cases that are susceptible to a variety of vulnerabilities such as SQL injections and cross-site scripting vulnerabilities. A total of 62 source programs for our experiment are available in three folders of Securibench Micro, including *basic* (containing various basic vulnerabilities), *aliasing* (containing aliasing-related vulnerabilities), and *inter* (containing vulnerabilities in interprocedural cases). The vulnerability types within these programs are categorized and summarized in Table 3.2. Note that in these programs, many XSS vulnerabilities are accompanied by information disclosure vulnerabilities, and both types of vulnerabilities are caused by the same source-sink propagation paths. To avoid duplicate counting, the information disclosure vulnerabilities have been excluded from our analysis.

To begin, we predefined a list of sources, sinks, and entry points for the test cases. For example, *getParameter* is a typical source in these test cases as it accepts user-supplied input, while *println* is a typical sink as it can directly process tainted data and might produce unexpected results.

Table 3.2. Vulnerability types within the programs for our experiments.

Vulnerability types	Description	#
XSS	Occurs when user-supplied input is not properly handled, resulting in injection of malicious scripts into web pages.	77
SQL Injection	Occurs when user-supplied input is not properly handled, allowing malicious SQL statements to be manipulated by a database.	6
Path Traversal	Occurs when user-supplied input is not properly handled, enabling attackers to access files or directories outside of the designated scope.	4
Open Redirect	Occurs when a web application redirects users to a specified URL without properly handling the target.	1
Null Pointer Dereference	Occurs when a null pointer is dereferenced, typically causing a crash or exit.	1
Information Disclosure	Occurs when an application unintentionally reveals sensitive information to unauthorized users.	-
Total		89

3.3.1.2 Implementation

We followed the steps in each of our approaches to analyze the test cases. To simulate the real process of spotting vulnerabilities during coding, we manually analyzed the code line by line as if we were the programmer who was writing the code line by line. Moreover, we followed the common programming principles when pretending to be the programmer; for example, as mentioned in Section 3.2.3, when the programmer just creates the statement at line 5 of Figure 3.3, the *foo* method probably has not yet been constructed, so our analysis at that point has no knowledge of *foo*'s details (lines 9-12).

Table 3.3 shows an example of our analysis for a test case in the folder *basic*. While the *leak location* is the program location where a sink occurs and causes a leak, the *reported location* is where we identify and report the leak. The two locations are identical implies

that the vulnerability can be found immediately, and knowledge of succeeding code is not required.

Table 3.3. Analysis example.

Programs	Correct Warnings	Missed Warnings	False Warnings	Leak Location	Reported Location
<i>Basic1</i>	1	0	0	Line 39	Line 39

3.3.2 Results and Discussion

Our two approaches have obtained the same experiment results, as summarized in Table 3.4. We reported 98 warnings in total, including 89 real vulnerabilities and 9 false alarms.

Table 3.4. Experiment results.

Programs	Correct Warnings	Missed Warnings	False Warnings
<i>basic</i>	61	0	0
<i>aliasing</i>	12	0	0
<i>inter</i>	16	0	9
Total	89	0	9

On the one hand, we did not miss any vulnerabilities in the test cases. In particular, every vulnerability was reported immediately once it appeared in code, obviating the need for succeeding code. This demonstrates that our approaches support (near) real-time detection. On the other hand, our approaches yielded some false alarms, all of which occurred in the interprocedural-related test cases. The reason for this problem is that when analyzing an incomplete program, we often have little knowledge of an invoked procedure but to make a conservative assumption. As an example, consider the code snippet of *Inter1* in Figure 3.5. In line 4 we may regard *s3* as a taint in the absence of the details of the *id* method (lines 9-11), and consequently report a false alarm at line 7.


```

1  protected void doGet(HttpServletRequest req, HttpServletResponse
   resp) throws IOException {
2      String s1 = req.getParameter(FIELD_NAME);
3      String s2 = id(s1);
4      String s3 = id("abc");
5      PrintWriter writer = resp.getWriter();
6      writer.println(s2);          /* BAD */
7      writer.println(s3);          /* OK */
8  }
9  private String id(String string) {
10     return string;
11 }

```

Figure 3.5. Code snippet of *InterI* from Securibench Micro.

The number of such false alarms depends on the precision of our conservative approximation. Making an optimistic or pessimistic assumption about the missing code can make a difference to the analysis, as discussed in Section 3.2.2. Fortunately, the problem can be mitigated when the invoked procedures become available. For example, as coding proceeds and more details of the invoked procedures are revealed, we can gradually obtain more precise points-to results and eliminate the false alarms. In the above code example, our approaches will not report the false alarm in the presence of the `id` method (lines 9-11). In fact, our approaches can eventually gain the same precision as a whole-program pointer analysis approach when the coding process is completed. (To compare with a whole-program solution in this sense, our approaches should also be augmented with capabilities such as context sensitivity, as mentioned in Section 3.4.)

Overall, our evaluation shows that our approaches can successfully detect a variety of vulnerabilities in an HMPP fashion, though false alarms may arise when handling procedure calls. The false alarms may be automatically corrected as coding and analysis proceed incrementally.

3.3.3 Case Study

Since the test cases in Securibench Micro are intentionally vulnerable, our experiment above may provide little demonstration of the practical effectiveness of our approaches in real-world cases. In addition, those test cases contain mainly the necessary instructions that contribute to the potential vulnerabilities and few irrelevant instructions are included, thus making it difficult for us to showcase how our demand-driven approach computes less points-to relations than our exhaustive approach does. To alleviate these limitations, we present an in-depth, real-world case study in this subsection.

This case study discusses a security flaw known as *LDAP (Lightweight Directory Access Protocol) injection* (CVE-2020-1958), which was first discovered in Apache Druid 0.17.0 [70]. Apache Druid is a real-time analytics database that supports fast queries on large data sets. When enabling LDAP authentication in Apache Druid 0.17.0, unintended consequences can arise.

Figure 3.6 shows the potentially vulnerable code snippet extracted from Apache Druid 0.17.0. The security flaw lies in line 12, where the user-supplied input, `username`, is used without properly sanitized or validated. The LDAP user search filter handles searching using a template “`(&(uid=%s)(memberof=cn=users,dc=example,dc=org))`”, which is implemented in `ldapConfig.getUserSearch()` (line 12); if the user supplies “`user)(uid=*)`”(“`(uid=*`” as the `username`, the search filter will become “`(&(uid=user)(uid=*))`”(“`(uid=*)`”(“`(uid=*)(memberof=cn=users,dc=example,dc=org))`”, which in turn will be treated as two separate filters, namely “`(&(uid=user)(uid=*))`” and “`(uid=*)(memberof=cn=users,dc=example,dc=org))`”. Only the first filter of the two will be executed and that will allow a malicious attacker to gain unauthorized access [71]. Moreover, it can cause information disclosure when the attacker supplies a set of carefully crafted queries and observes the corresponding responses from the system.

```

1  public AuthenticationResult validateCredentials(String
    authenticatorName, String authorizerName, String username,
    char[] password){
2  SearchResult userResult;
3  [...] //omitted code
4  userResult = getLdapUserObject(this.ldapConfig, dirContext,
    username);
5  }
6  [...]
7  SearchResult getLdapUserObject(BasicAuthLDAPConfig ldapConfig,
    DirContext context, String username){
8  try {
9      SearchControls sc = new SearchControls();
10     sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
11     sc.setReturningAttributes(new String[]
        {ldapConfig.getUserAttribute(), "memberOf" });
12     NamingEnumeration<SearchResult> results =
        context.search(ldapConfig.getBaseDn(),
            StringUtils.format(ldapConfig.getUserSearch(), username), sc);
13 }
14 }

```

Figure 3.6. Code snippet from Apache Druid 0.17.0 illustrating LDAP injection.

We next discuss how our approaches are used to detect the flaw in question. Since `username` is user-supplied input and can be inserted into the string created by `ldapConfig.getUserSearch()`, `username` is recognized as a source and `ldapConfig.getUserSearch()` a sink. Our approaches compute points-to results for `username` and check if it may still point to a tainted object when it reaches the sink (line 10). Since Figure 3.6 shows only a small code snippet, we provide an extended version in Appendix. We perform analyses on the extended version and summarize the key points as follows:

- Parameter `username` in line 30 is the entry that receives potentially tainted user-supplied input. Since no sanitization or input validation for `username` is included in the code, its received value can be eventually passed into `ldapConfig.getUserSearch()` in line 64. The following two points-to relations can reflect this process (o_{30} is tainted):
line 30: `username` \hookrightarrow o_{30}
line 64: `username` \hookrightarrow o_{30}
- To discover the LDAP vulnerability in the code in Appendix, our exhaustive approach performs points-to computation on 42 statements, while our demand-driven approach performs on 10. The statements involved are indicated in the appendix.

Overall, the case study demonstrates that while both the proposed approaches can discover the LDAP injection in Apache Druid 0.17.0, the demand-driven approach performs less computation than the exhaustive approach does.

3.4 Discussion

Although we have extensively discussed the proposed approaches in preceding sections, this work does not cover every technical issue and may have simplified the description of some problems. In the following we discuss several important limitations.

Firstly, it may be considered as a big limitation of our approaches that a programmer may make extensive modifications to his/her code in the coding phase and accordingly the points-to results need to be recomputed many times, thus incurring substantial overhead. We would argue that since most medium-sized or large programs are built from small modules, we can perform analysis of each module separately and reuse analysis results whenever possible. Such a mechanism obviates the need for recomputing the points-to results of most statements when a change is made to the code. We would also argue that the potential overhead is a worthwhile trade-off as it enables us to discover and fix vulnerabilities at an early stage, rather than to address the problems at a much higher price

later. In fact, a similar trade-off exists in compilers, where certain types of errors (such as type mismatches) are constantly detected and reported during compilation.

Secondly, this work omits the discussion of how to handle issues such as *context sensitivity* [26] and *access paths* [61], both of which are critical for a pointer analysis approach to gain precision and scale to large programs. For example, while access paths in a real-world program have an arbitrary length in the form of $a(.fld) *$, e.g., $a.f.g$, the case discussed in this work only involves the form $a.f$. We will discuss these issues in our future work.

Thirdly, the approaches in this work ignore the identification of sanitization when performing a taint analysis. *Sanitization* is a solution to a potential taint-style vulnerability, in which the taint is removed from a tainted value. For example, sanitization can be performed by replacing tainted values with untainted ones, or simply by terminating the potentially insecure paths of execution. Failure to recognize the presence of sanitization in a program can cause an analysis to produce false alarms. The very same limitation applies to our approaches in this work, although it does not manifest itself in our evaluation results. To be applied to more sophisticated programs, our approaches need to accommodate the sanitization identification in the future.

3.5 Related Work

As discussed previously, to perform bug detection during programming is beneficial yet challenging. Similar work [34, 35, 47] in this direction has its own limitations (to avoid duplicate discussion, see Section 2.6 in Chapter 2). To make improvements in the research branch Human-Machine Pair Programming, we adopt a different strategy in the work in this chapter. To the best of our knowledge, this work is the first attempt to perform pointer analysis for vulnerability discovery during coding. In addition, some commercial static analysis tools also support analysis of incomplete programs [72, 73]. For example, SonarQube [9] is a popular static analysis tool that supports real-time bug detection based on a predefined ruleset. It is designed for identifying a wide range of bugs and not specifically for security-related bugs. As commercial tools, however, the underlying technologies are often not (fully) public, though techniques such as data-flow analysis and

symbolic execution are believed to be adopted [73]. It may also be worth noting that although *defensive programming* (e.g., [74]) also aims to avoid possible bugs at the stage of coding (by using guard statements or assertions), it is a technique that falls outside the category of static analysis as it will only report bugs at run time.

Another branch of related work is on taint and pointer analyses. We discuss some most closely related work in the following. Heintze and Tardieu [67] introduce a demand-driven pointer analysis for C that computes only the necessary points-to information for a particular query using certain deduction rules. Sridharan *et al.* [68] extend the demand-driven pointer analysis to Java and show that their approach can greatly improve the precision. Livshits and Lam [21] present a taint analysis approach for finding taint-style security vulnerabilities in Java applications. Their approach handles taint analysis in Java code by means of a pointer analysis. For Android applications, Arzt *et al.* [61] present FlowDroid, a context, flow, field, and object-sensitive taint analysis approach, which reduces both false positives and negatives. Due to the close relationship between taint analysis and pointer analysis, Grech and Smaragdakis [51] suggest a unification of them, in which points-to algorithms are effectively used to implement taint analysis. In contrast to the above approaches (and most other pointer-analysis-based approaches) that require whole-program availability [8], there exist approaches that specifically target the analysis of incomplete programs. Rountev *et al.* [58] propose two frameworks for fragment analyses based on existing whole-program analyses. Rountev and Ryder [75] discuss a worst-case analysis and a summary-based analysis to analyze library modules and client modules separately. While our work in this chapter is inspired by all these approaches, our emphasis is on analysis of incomplete programs during coding, which implies that analysis targets are changing over time. For this purpose, we employ an incremental pointer analysis that can keep track of the coding process and produce (near) real-time analysis results.

4 Vulnerability Nets for Vulnerability Discovery³

In Chapters 2 and 3, we have explored the approaches suitable for Human-Machine Pair Programming. Those approaches can be further implemented as automated tools. Automated static analysis tools can encapsulate certain security knowledge for vulnerability discovery, thereby freeing developers from manually spotting security flaws during software development. However, due to the difficulty of obtaining soundness and completeness, tools will always fail to uncover some subtle vulnerabilities (i.e., false negatives), or may produce substantial false alarms (i.e., false positives). For example, buffer overflows [27], one of the most notorious vulnerabilities, still cannot be fully addressed by using automated tools alone. Instead, significant security expertise is often involved during detection of buffer overflows [28].

Manual audits are a complementary (not alternative) method to automated tools. In the process of auditing, an analyst manually examines the given code, based on his/her expertise, to find vulnerabilities that escape from detection of tools. To aid analysts in auditing manually, some researchers analyze source code using techniques such as fault trees [19, 76], in which certain crucial information of code is made explicit. However, manual audits are tedious, error prone, and costly, so it is normally impractical to manually audit an entire program, though auditing a few critical code fragments is possible.

To benefit from both the static analysis tools and manual audits, work in this area (e.g., [29, 30]) has considered incorporating the analyst's security knowledge into a tool during the detection process. The knowledge (such as annotations [77]) provided by security auditors can guide the detection for vulnerabilities. To make contributions in this branch of research, this chapter presents a novel representation of source code, called a *vulnerability net* [78, 79], which is in the form of a Petri net structure.

³ This chapter is adapted from our work in [78] and [79].

Petri nets are a mathematical representation widely used for modeling and analyzing various types of systems, ranging from chemical systems to computer systems [80]. A program, the analysis target of this thesis, can be viewed as a system, where the statements within the program correspond to the components of the system. Therefore, Petri nets have the potential to model a program. To tailor Petri nets specifically for program analysis, we propose vulnerability nets in this chapter. While preserving the core principles of standard Petri nets, vulnerability nets introduce some novel features and incorporate data dependence graphs and control flow graphs, aiming at representing source code and modeling the flow of information within it. The combination explicitly describes the key information of a given program and provides a good view for analysts to perform auditing. Like standard Petri nets, vulnerability nets are executable, thus allowing analysts to conveniently track the data of interest to examine if any path execution may cause a security issue. By supplying a vulnerability net with necessary taint information (such as *sources* and *sinks*), the net can be executed to identify potential taint-style vulnerabilities, which include various types of security flaws, such as buffer overflows [27, 81], injection vulnerabilities [48, 55], and cross-site scripting (XSS) vulnerabilities [20, 56].

4.1 Definitions

For comparison purpose, in this section we start with the definition of traditional Petri nets. Then, vulnerability nets are formally defined. Most of the notation and terminologies in our discussions are taken or adapted from the work in [80, 82].

4.1.1 Petri Nets

A Petri net is defined by its places, transitions, input function, output function, and marking.

Definition 4.1. A *Petri net* M is a five-tuple, $M = (P, T, I, O, \mu)$, where

$P = \{p_1, p_2, \dots, p_n\}$ is a finite set of *places*, $n \geq 0$.

$T = \{t_1, t_2, \dots, t_m\}$ is a finite set of *transitions*, $m \geq 0$. $P \cap T = \emptyset$.

$I: T \rightarrow P^\infty$ is the *input* function, a mapping from transitions to bags of places.

$O: T \rightarrow P^\infty$ is the *output* function, a mapping from transitions to bags of places.

$\mu: P \rightarrow N$ is the *marking* of the net, a mapping from the set of places P to N , where N is the set of nonnegative integers.

4.1.2 Vulnerability Nets

A *vulnerability net* is in the form of a Petri net, with some special properties. We formally define it as follows.

Definition 4.2. A *vulnerability net* Φ is a five-tuple, $\Phi = (P, T, I, O, \mu)$, where

$P = \{p_1, p_2, \dots, p_n\}$ is a finite set of *places*, $n \geq 0$.

$T = \{t_1, t_2, \dots, t_m\}$ is a finite set of *guarded transitions*, $m \geq 0$. $P \cap T = \emptyset$.

$I: T \rightarrow \wp(P)$ is the *input* function, a mapping from transitions to sets of places. $\wp(P)$ is the power set of P .

$O: T \rightarrow \wp(P)$ is the *output* function, a mapping from transitions to sets of places.

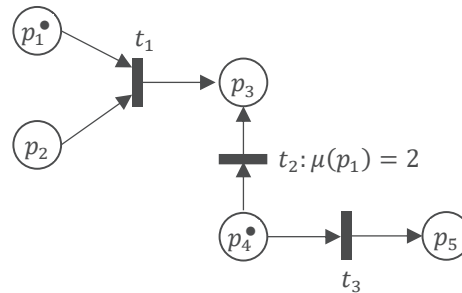
$\mu: P \rightarrow \{0, 1\}$ is the *marking* of the net, a mapping from the set of places P to the set $\{0, 1\}$.

The marking function μ implies that each place holds a number, 0 or 1. By convention, the element that a place holds is called a *token*. That is, each place in a vulnerability net holds zero or one token. The reason why we restrict the number of tokens in a place to only 0 or 1 is that 0 represents the absence and 1 represents the presence of a specific value in our vulnerability analysis. (A detailed discussion of the representations appears in Section 4.2.) The marking function μ shows the number and distribution of tokens in a vulnerability net. We use $\mu(p_1), \mu(p_2), \dots$ to represent each specific marking for places in a net. For example, $\mu(p_i)$ is the marking for the place p_i , and $\mu(p_i) = 1$ means p_i holds one token. By contrast, the marking of an entire vulnerability net is denoted by μ , where $\mu = (\mu(p_1), \mu(p_2), \dots, \mu(p_n))$. In the rest of this work, a marking refers to a marking of an entire net, unless stated otherwise. When a vulnerability net executes, its marking may change as the number of tokens in each place may change. The conditional expressions in guarded transitions are the Boolean expressions used to constrain the change in the number of tokens.

An example of a vulnerability net is shown in Figure 4.1 (a). The net is composed of five places and three guarded transitions (henceforth transitions). The initial marking of the net $\mu_0 = (1, 0, 0, 1, 0)$ states that the place p_1 and p_4 each initially contain a token, while other places do not contain any. The links between places and transitions are revealed by the input and output functions. For example, $I(t_1) = \{p_1, p_2\}$ indicates that p_1 and p_2 are the input places of the transition t_1 ; $O(t_1) = \{p_3\}$ indicates that p_3 is the output place of the transition t_1 . A transition (e.g., t_2) with an expression indicates that its Boolean value depends on the evaluation of the expression (e.g., $\mu(p_1) = 2$ in t_2), while other transitions (such as t_1 and t_3) without any explicit expression are assigned the value *true* as a default.

$P = \{p_1, p_2, p_3, p_4, p_5\}$	
$T = \{t_1, t_2: \mu(p_1) = 2, t_3\}$	
$\mu_0 = (1, 0, 0, 1, 0)$	
$I(t_1) = \{p_1, p_2\}$	$O(t_1) = \{p_3\}$
$I(t_2) = \{p_4\}$	$O(t_2) = \{p_3\}$
$I(t_3) = \{p_4\}$	$O(t_3) = \{p_5\}$

(a)



(b)

Figure 4.1. (a) Textual representation of a vulnerability net; (b) Graphical representation of Figure 4.1 (a).

We can also use a graphical representation to represent a vulnerability net for a better readability. A vulnerability net graph for Figure 4.1 (a) is shown in Figure 4.1 (b), where

places, transitions, and tokens are denoted by circles, bars, and small dots, respectively. Directed arcs are used to connect places with transitions.

Vulnerability nets are executed by *firing* transitions. A transition is ready for firing if it is *enabled*. As formally defined in Definition 4.3, a transition t_j is enabled if all of the following three conditions hold: 1) each of the t_j 's input places contains a token, 2) there exists an output place of t_j that does not contain any token, and 3) the conditional expression of t_j evaluates to true.

Definition 4.3. A transition $t_j \in T$ in a vulnerability net $\Phi = (P, T, I, O, \mu)$ is *enabled* if

1. $\forall p_i \in I(t_j) (\mu(p_i) = 1)$,
2. $\exists p_k \in O(t_j) (\mu(p_k) = 0)$, and
3. the conditional expression of t_j evaluates to true.

For example, the transition t_3 in Figure 4.1 is enabled according to the definition. However, t_1 is not enabled since p_2 , one of the t_1 's input places, does not contain a token. In the case of t_2 , we notice that its conditional expression $\mu(p_1) = 2$ always evaluates to false because the place p_1 will never hold two tokens (in fact, none of the places in a vulnerability net can hold two or more tokens, according to the function μ in Definition 4.2), so t_2 is not enabled.

When a transition fires during the execution of a vulnerability net, tokens *propagate* from its input places to output places, i.e., new tokens are assigned to output places while tokens in input places are retained. We use *state* to describe the change. Every state of a vulnerability net is defined by a marking; for example, the initial state is defined by initial marking μ_0 . The state space of a vulnerability net with n places is the set of all markings, i.e., B^n , where B is the set of $\{0, 1\}$. Definition 4.4 defines a *next-state function* for calculating how a state can change after firing a transition.

Definition 4.4. The *next-state function* $\delta: \{0, 1\}^n \times T \rightarrow \{0, 1\}^n$ for a vulnerability net $\Phi = (P, T, I, O, \mu)$ and transition $t_j \in T$ is defined if and only if t_j is enabled. If $\delta(\mu, t_j)$ is defined, then $\delta(\mu, t_j) = \mu'$, where

$$\mu'(p_i) = \begin{cases} 1, & \text{for all } p_i \in O(t_j) \\ \mu(p_i), & \text{for all } p_i \in P - O(t_j). \end{cases} \quad (1)$$

According to (1), we can calculate the next-state result of μ_0 in Figure 4.1. As discussed previously, of the three transitions only t_3 is enabled, so that we have the next state $\mu' = \delta(\mu_0, t_3) = (1, 0, 0, 1, 1)$. The result suggests that a token has propagated to p_5 . Figure 4.2 shows the change. After the change, since no transition is enabled, the execution must halt and μ' is in fact the final state of the net.

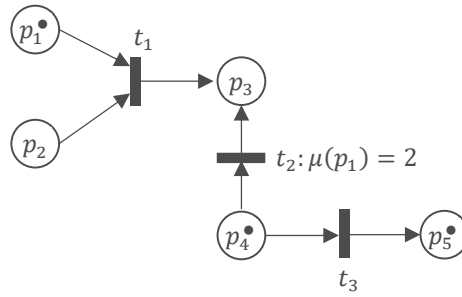


Figure 4.2. The next state of Figure 4.1.

One may notice that a vulnerability net executes differently from a standard Petri net. As a simple example, Figure 4.3 shows the distinct results of executing two nets that look initially identical. In Figure 4.3 (a), the initial Petri net fires t_1 to flow a token from p_1 to p_2 and then the execution halts since no transition is enabled anymore. In contrast, the initial vulnerability net of Figure 4.3 (b) fires t_1 to propagate a token to p_2 and then the execution halts. More distinctions between vulnerability nets and standard Petri nets can be revealed by their own definitions. In Section 4.2 we will see how the special properties of vulnerability nets can benefit vulnerability analysis.

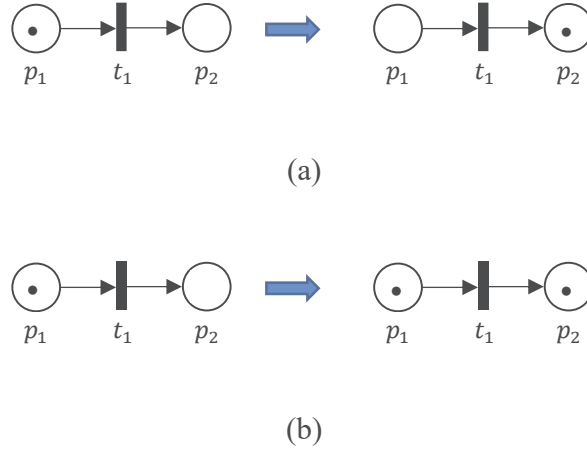


Figure 4.3. (a) Execution of a standard Petri net; (b) Execution of a vulnerability net.

4.1.3 Vulnerability Nets with Colored Tokens

Colored tokens are used to augment the expressiveness of a vulnerability net. A vulnerability net with colored tokens also meets the definitions given in Section 4.1.2 as long as we consider each kind of colored token separately. We formally give a definition as follows.

Definition 4.5. A *vulnerability net with colored tokens* is a vulnerability net structure $C = (P, T, I, O, G)$, where G is a finite set of markings $G = \{\mu^1, \mu^2, \dots, \mu^k\}$, $k \geq 0$, where each μ^i is the marking for a kind of token with a unique color in a graphical representation.

Figure 4.4 shows an example. Initially, while p_2 and p_4 each contain one kind of token, p_1 holds two. During the execution of the net, the various kinds of tokens propagate independently of each other. Therefore, all kinds of colored tokens can propagate to p_3 , with the exception of the green token as t_1 requires two green inputs but only one (i.e., the green token in p_1) is available. The initial state of the vulnerability net can be denoted by $\mu_0^1 = (0, 0, 0, 1)$, $\mu_0^2 = (1, 0, 0, 0)$, and $\mu_0^3 = (1, 1, 0, 0)$, where μ_0^1 , μ_0^2 , and μ_0^3 represent the initial markings for black, green, and red tokens, respectively. The final state of the net is $\mu_f^1 = (0, 0, 1, 1)$, $\mu_f^2 = (1, 0, 0, 0)$, and $\mu_f^3 = (1, 1, 1, 0)$.

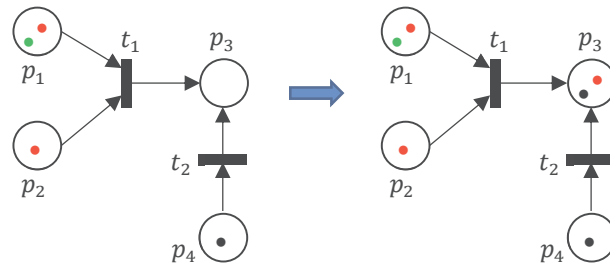


Figure 4.4. Execution of a vulnerability net with colored tokens.

4.2 Modeling and Detecting Vulnerabilities

In this section, we start by discussing the characteristics of taint-style vulnerabilities and giving a simple code example. Then we briefly introduce data dependence graphs and control flow graphs, followed by a description of the incorporation of them into vulnerability nets for vulnerability-discovery purposes. Finally, algorithms for vulnerability nets generation and vulnerability discovery are described.

The class of security weaknesses this chapter focuses on is taint-style vulnerabilities, including several critical vulnerabilities such as buffer overflows, injection vulnerabilities, and cross-site scripting vulnerabilities. In taint analysis, a *tainted value* is derived from external, untrusted input such as command-line arguments or results returned from function calls. While a *source* is the original program location (such as a function or method) that accepts tainted values, a *sink* is the program location that should not receive tainted values. A security weakness may exist when a tainted value from a source reaches a sink through an execution path. *Sanitization* is a step to remove the taint from a value, thereby eliminating the potential security risk. To perform sanitization, we could either replace the tainted value by an untainted value or terminate the path of execution when a tainted value is detected.

For illustration, consider a simple taint-style vulnerability shown in Figure 4.5. In this Java code, *x* accepts a tainted value returned from a source function and passes it to *y* when the

if-condition is met. Then x is sanitized, and finally a sink function receiving y as an argument is called.

```
void foo() {  
S1:     int x = source();  
S2:     if (x > 100) {  
S3:         int y = x;  
S4:         sanitize(x);  
S5:         sink(y);  
        }  
}
```

Figure 4.5. Example of a taint-style vulnerability in Java.

4.2.1 Data Dependence Graphs

Data dependence graphs (DDGs) are a program representation that can explicitly represent data dependences among statements and predicates [83]. A data dependence is present when two statements cannot be switched without changing any variable's value. For example, in Figure 4.5, $S3$ depends on $S1$ as $S1$ must be executed first in order for x 's value to be properly used in $S3$. Figure 4.6 (a) shows the DDG for the example code in Figure 4.5.

In our approach, the use of DDGs makes data dependences explicit and visible, which enables analysts to find taint-style vulnerabilities more easily. For example, Figure 4.6 (a) explicitly shows that the tainted value originated from the source will eventually be passed to the sink, resulting in a leak.

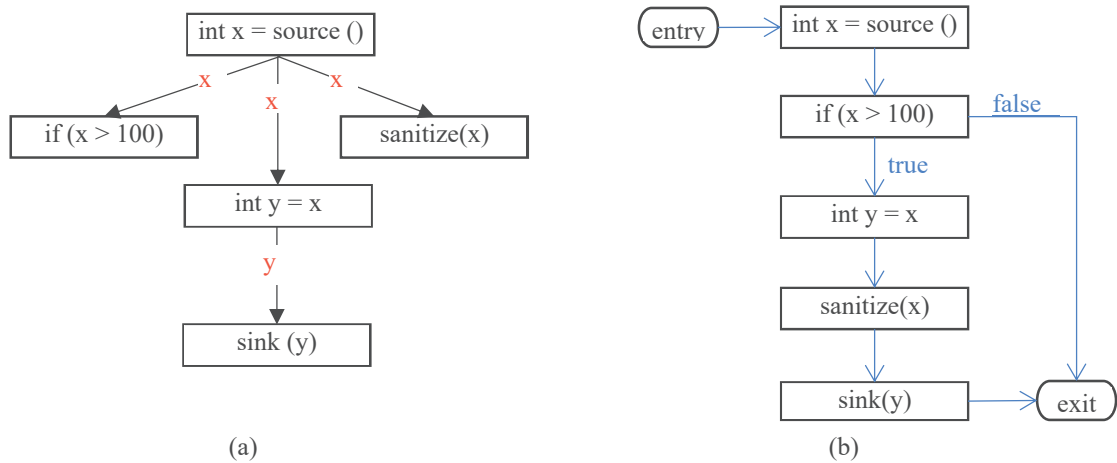


Figure 4.6. (a) DDG for the code in Figure 4.5; (b) CFG for the code in Figure 4.5.

4.2.2 Control Flow Graphs

Control flow graphs (CFGs) are another commonly used representation that can explicitly show the execution order of statements and the flow of control determined by conditional expressions [47]. In a CFG, statements and predicates are denoted by nodes, and the flow of control is indicated by directed edges. Figure 4.6 (b) shows the CFG for the code sample in Figure 4.5.

The use of CFGs in our approach is important, since DDGs alone often do not suffice to ensure the existence of a vulnerability. For example, if we exchange the order of *S3* and *S4* in Figure 4.5 (as shown in Figure 4.7), while the data dependence graph may remain unchanged (depending on what the sanitization is), the code is no longer flawed because *x*'s value has been sanitized before being passed to *y* and *sink(y)*. That is, using a DDG alone in this case could yield a spurious alarm. To remedy the situation, we leverage a CFG as it can make explicit whether the sanitization for a tainted value is already executed prior to using the value. Furthermore, control-flow information is also essential in the detection of many other types of vulnerabilities, such as use-after-free vulnerabilities.


```

void foo() {
S1:    int x = source();
S2:    if (x > 100) {
S3:        sanitize(x);
S4:        int y = x;
S5:        sink(y);
        }
}

```

Figure 4.7. Example code adapted from Figure 4.5.

4.2.3 Building Vulnerability Nets

When using a vulnerability net to represent source code, places denote statements (or predicates), while transitions denote Boolean expressions that are used to control the propagation of tokens. A token models the existence of a tainted value, and different colored tokens represent different tainted values. Markings in a vulnerability net are used to show the number and position of tainted values. In the context of representing source code, the initial marking of a vulnerability net is the state after all sources have been assigned corresponding tokens. Once a vulnerability net is initialized, we examine whether the tokens can propagate across places by computing the changes in the markings of the net. If a token can propagate to a sensitive place, i.e., a tainted value can be passed into a sensitive sink, then there potentially exists a vulnerability in the program.

As an example, consider a simple Java code fragment “`a = source1(); b = source2(); sink(a, b);`”. For this code, we can use places p_1 , p_2 , and p_3 to denote each statement, respectively. Also, we use two distinct colored tokens (let us say, a black token and a red token) to represent the values of a and b , respectively. Initially, p_1 holds one black token while p_2 and p_3 do not hold any black token. The initial marking for the black token is $\mu_0^{black} = (1, 0, 0)$. Similarly, the initial marking for the red token is $\mu_0^{red} = (0, 1, 0)$. After a is passed into `sink(a, b)`, which indicates that the black token propagates from p_1 to p_3 , the marking for the black token changes to $\mu_1^{black} = (1, 0, 1)$; similarly, the

marking for the red token changes to $\mu_1^{red} = (0, 1, 1)$ after b is passed into $sink(a, b)$. Whether a token can propagate from one place to another is dependent on the corresponding transition(s). We provide more details below after DDGs and CFGs are introduced.

Since DDGs and CFGs can provide essential information for security analysis, we incorporate them into vulnerability nets when representing source code. For brevity, the combination is also referred to as a vulnerability net. To build such a net, a DDG serves as a skeleton on which we gradually add CFG information (CFG edges and conditional expressions). Specifically, the DDG nodes are replaced by places, the DDG edges are replaced by transitions, and the conditional expressions from the CFG are assigned to corresponding transitions. To clarify the idea, consider again the code example shown in Figure 4.5. We combine its DDG and CFG, both given in Figure 4.6, and produce a vulnerability net, as indicated in Figure 4.8 (a).

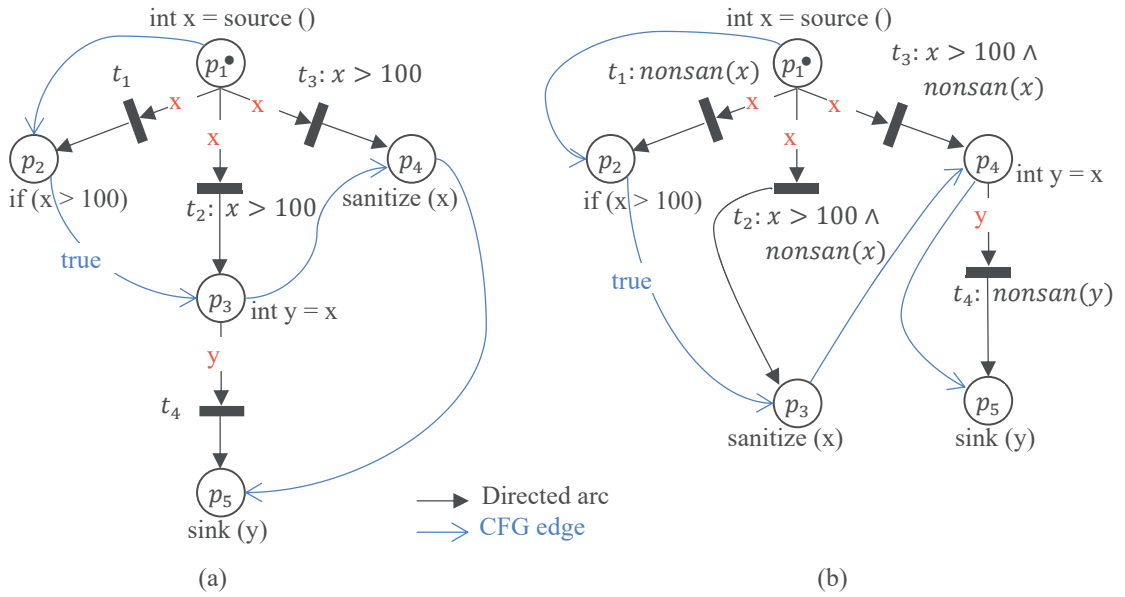


Figure 4.8. (a) Vulnerability net for the code in Figure 4.5; (b) Vulnerability net with sanitization identification for the code in Figure 4.7.

In this example net, since the statement $int\ x = source()$ (denoted by p_1) contains a source method, a token is placed in p_1 to represent a tainted value originated from there and the initial marking of the net is $\mu_0 = (1, 0, 0, 0, 0)$. In the following we analyze how the marking changes during the execution of the net.

First, we can see that t_1 is immediately enabled, resulting in the next state $\mu_1 = \delta(\mu_0, t_1) = (1, 1, 0, 0, 0)$. Transitions t_2 and t_3 are conditionally enabled as the condition $x > 100$ may hold. To make conservative (or safe) approximations [47, 84], when we cannot determine whether a condition of a transition will be met, we will consider the worst case, i.e., the condition will be met, and the token can be propagated through the transition successfully. After firing the two transitions, the marking changes to $(1, 1, 1, 1, 0)$. Finally, t_4 is enabled and will fire to yield the marking $(1, 1, 1, 1, 1)$, which indicates that the token can propagate across all places, including the sink function (i.e., p_5). Therefore, the final execution result suggests the existence of a taint-style vulnerability since the tainted value from the source can reach the sink.

To reduce the risk of taint-style vulnerabilities, programmers may sanitize the tainted values before using them. In this situation, false positives may arise if we do not realize the presence of sanitization. For example, as discussed in Section 4.2.2, exchanging the order of $S3$ and $S4$ in Figure 4.5 will eliminate the vulnerability, so we should not report an alarm in that case. Therefore, it is crucial to perform sanitization identification when spotting taint-style vulnerabilities. To this end, necessary sanitization information can be accommodated in the transitions of a vulnerability net. As an example, Figure 4.8 (b) shows the vulnerability net with sanitization identification for the code in Figure 4.7. (Let us assume the sanitization used in Figure 4.7 is to terminate the path of execution.)

In Figure 4.8 (b), each transition adds a Boolean predicate $nonsan(var)$ for the corresponding variable var , which will return a *true* or *false* value by evaluating whether the variable var is sanitized. It returns true if var is not sanitized and false otherwise. It is worth noting that such predicates are added and evaluated by hand in this work, but they can be done in an automatic manner in practice as well. For example, since sanitization

within a program can be identified through taint-specification mining techniques (e.g., [85, 86]), a predicate $nonsan(var)$ can be automatically evaluated to false if the corresponding variable var is sanitized; otherwise, it is evaluated to true by default.

To illustrate how the sanitization can help, we simulate the execution of the net in Figure 4.8 (b) as follows. The initial marking is $\mu_0 = (1, 0, 0, 0, 0)$ as we assign a token to p_1 . Since x is not sanitized at t_1 , the $nonsan(x)$ evaluates to true, making t_1 enabled and thus allowing the token to be propagated to p_2 . The propagation leads to the next state $\mu_1 = \delta(\mu_0, t_1) = (1, 1, 0, 0, 0)$. Similarly, t_2 could be enabled when $x > 100$, so we have $\mu_2 = (1, 1, 1, 0, 0)$. However, the situation of t_3 is different. The CFG edges in the net show that prior to the propagation path p_1 through p_4 , x must have been sanitized in p_3 . Consequently, the $nonsan(x)$ at t_3 evaluates to false, making t_3 not enabled and thus preventing the token in p_1 from propagating forward to p_4 . Furthermore, t_4 cannot be enabled either as p_4 never receives a token. In summary, there is no further change in the state, i.e., the final state of the net is $\mu_f = \mu_2 = (1, 1, 1, 0, 0)$, which implies that no taint-style vulnerability is present in this code as no tainted value reaches the sink function (i.e., p_5). As a result, no false alarms will be reported in this net.

4.2.4 Algorithms

In the previous subsection we illustrate how a vulnerability net is constructed and used for vulnerability discovery. To formalize the ideas, algorithms are described in this subsection.

The process of generating a vulnerability net is formalized in Algorithm 4.1. Each step can be completed in an automatic manner, though security analysts may manually add information to a net for performance enhancement. At line 1, the merger between a control flow graph and a data dependence graph is the union of their nodes and edges. Lines 2 through 7 transform the merger graph to a vulnerability net. The assignments of conditional expressions to transitions are based on the Boolean expressions given in the control flow graph (e.g., the $x > 100$ in Figure 4.8 (a)) or given by security analysts (e.g., the $nonsan(x)$ in Figure 4.8 (b)).

Algorithm 4.1: Generation of a vulnerability net

INPUT: A source program and its DDG and CFG**OUTPUT:** A vulnerability net

- 1: Merge CFG and DDG;
 - 2: **for each** statement s (or predicate) in DDG **do**
 - 3: replace s by a place p_i ($i \in 1, 2, \dots$);
 - 4: **for each** DDG edge e **do**
 - 5: replace e by a transition t_j ($j \in 1, 2, \dots$) and directed arcs connected to places;
 - 6: **for each** transition t_j **do**
 - 7: assign conditional expressions;
-

The algorithm for detecting taint-style vulnerabilities in a vulnerability net is described in Algorithm 4.2. We start by providing a vulnerability net and specifying the sources and sinks. A *source-sink pair* is a two-tuple ($source, sink$), where the source through the sink is a propagation path that may cause a security flaw. The quantities of sources and sinks in a vulnerability net are arbitrary.

Algorithm 4.2: Vulnerabilities discovery

INPUT: A vulnerability net and a set of source-sink pairs $S = \{(p_{source_1}, p_{sink_1}), \dots, (p_{source_i}, p_{sink_j}), \dots, (p_{source_m}, p_{sink_n})\}$, $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$ **OUTPUT:** Taint-style vulnerabilities

- 1: Generate the set of initial markings $G_0 = \{\mu_0^1, \mu_0^2, \dots, \mu_0^m\}$;
 - 2: Execute the vulnerability net and generate the set of final markings $G_f = \{\mu_f^1, \mu_f^2, \dots, \mu_f^m\}$;
 - 3: **for each** $(p_{source_i}, p_{sink_j})$ in S **do**
 - 4: **if** $\mu_f^i(p_{source_i}) = 1 \wedge \mu_f^j(p_{sink_j}) = 1$ **then**
 - 5: flag $(p_{source_i}, p_{sink_j})$ as a vulnerability;
-

In line 1 of Algorithm 4.2, the set of initial markings G_0 is generated according to the location of the source places, i.e., $\mu_0^i(p_{source_i}) = 1$ for all $p_{source_i} \in P$, where P is the set

of places. Recall that when represented in a graph, different sources are denoted by tokens with different colors (see Section 4.1.3).

Lines 2 through 5 of Algorithm 4.2 state that the vulnerability net is then executed, followed by the generation of the set of final markings, which in turn are used to iteratively examine the markings of all the source-sink pairs. If the markings of a source-sink pair are both 1, meaning a source and its sink each contains a token, then a vulnerability is detected.

4.3 Evaluation

In this section, we present an evaluation of our approach by conducting a set of small-scale experiments on Securibench Micro [69], followed by a comparison with SonarQube [9]. Moreover, we present a case study to illustrate the use of our approach in a real-world case.

4.3.1 Experiments

4.3.1.1 Preliminaries

As mentioned in Chapter 3, Securibench Micro [69] is comprised of a series of small Java test cases, which encompass a variety of vulnerabilities such as SQL injection vulnerabilities and XSS vulnerabilities. The code for our experiments appears in three directories of Securibench Micro, namely *basic* (containing various basic vulnerabilities), *aliasing* (containing aliasing-related vulnerabilities), and *inter* (containing vulnerabilities in interprocedural cases), with a total of 62 source programs. The vulnerabilities in these programs have been summarized in Table 3.2 in Chapter 3.

We preprocessed the source programs prior to analyses, since many of them contain procedure invocations, but our approach supports only intraprocedural analysis. The preliminary processing transformed each program into a single procedure using procedure inlining, i.e., replacing an invocation by the body of the invoked procedure. We found that two programs, *Inter13* and *Inter14* in *inter*, are not suitable for inlining because the loops in them can heavily expand the code size, so we omitted the analyses of them.

We also manually predefined a list of sources, sinks, source-sink pairs, sanitizers, and entry points for the test cases. (When analyzing large-scale programs, such taint information can be automatically obtained using taint-specification mining techniques [85, 86].) For example, one typical source is the method *getParameter* as it accepts attacker-supplied input, while *println* is a typical sink as it might process tainted data and produce unexpected results. The *(getParameter, println)* is a typical source-sink pair in the test cases of our experiments.

4.3.1.2 Implementation

In principle, a vulnerability net itself is executable, but since an automated tool for our approach has not yet been available, we partially implemented our approach as a Datalog program. Specifically, Figure 4.9 shows a Soufflé-style Datalog implementation of Algorithm 4.2. In the program, lines 2 and 7 accept vulnerability net edges and source-sink pairs as input, respectively, which in turn serve as facts for computation under certain rules (lines 4, 5, and 9). Line 4 states that a token can be propagated from the place x to y if there is a vulnerability net edge connecting from x to y and the Boolean expression in the transition evaluates as true, denoted by “1” in $VNEdge(x, y, 1)$. In line 5, the token is propagated across the entire net as long as two places in the net satisfy the condition in line 4. Finally, line 9 computes insecure source-sink pairs, which are output as discovered vulnerabilities at line 10. (Note that the concepts of Datalog are beyond the scope of this thesis; see [47, 87] for discussions about Datalog or Soufflé.)

```

1  .decl VNEdge(from: symbol, to: symbol, boolean: number)
2  .input VNEdge

3  .decl Reachable(from: symbol, to: symbol)
4  Reachable(x, y) :- VNEdge(x, y, 1).
5  Reachable(x, y) :- VNEdge(x, z, 1), Reachable(z, y).

6  .decl SrcSink(src: symbol, sink: symbol)
7  .input SrcSink

8  .decl BugFound(src: symbol, sink: symbol)
9  BugFound(x, y) :- Reachable(x, y), SrcSink(x, y).

10 .output BugFound

```

Figure 4.9. Datalog program implementing Algorithm 4.2.

4.3.2 Results

After examining the test cases, we summarize the results in Table 4.1.

Table 4.1. Experiment results.

Programs	Correct Warnings	Missed Warnings	False Warnings
<i>Inter13</i> and <i>Inter14</i>	0	2	0
Remaining 60 programs	87	0	0
Total	87	2	0

The 62 source programs contain a total of 89 vulnerabilities. As mentioned in Section 4.3.1, we did not analyze *Inter13* and *Inter14* due to the failure to inline procedures in them, so we missed the discovery of two vulnerabilities. For the remaining 60 programs, our

approach has correctly discovered 87 vulnerabilities without any false negatives or positives. That is, our approach reported 97.8% (87/89) of vulnerabilities in the test cases.

The results show that our approach can detect various taint-style vulnerabilities in source code, though the intraprocedural analysis hurts performance to some extent. We discuss a solution to this issue in Section 4.4.

4.3.3 Comparison with SonarQube

As a popular static analysis tool, SonarQube [9] can serve as a good baseline for measuring the performance of our approach. The version of SonarQube we used is the Community Edition 10.2.1.78527. We first converted the target programs from Securibench Micro into a Maven project for the purpose of analysis and then performed analysis through SonarQube. After an analysis report was generated, we examined if there were any false positives or negatives. Table 4.2 summarizes the analysis results.

Table 4.2. Comparison between SonarQube and our approach.

Approaches	Correct Warnings	Missed Warnings	False Warnings
SonarQube	71	18	0
Our Approach	87	2	0

As shown in the table, while both approaches did not generate any false positives, our approach yielded less false negatives compared with SonarQube. One possible explanation is that the rulesets in SonarQube are not tailored specifically to taint-style vulnerabilities. Instead, SonarQube is intended for detecting a broader range of bugs, including security vulnerabilities and other types of bugs. Indeed, in this analysis it also reported some other bugs associated with reliability or maintainability, which fall outside the scope of our vulnerability analysis. Note that since SonarQube supports user-customized configuration, its performance can be improved if we configure its rulesets properly.

If we compare the two approaches from the perspective of severity of the missed vulnerabilities, we can see that our approach suffers less harm as it missed two XSS vulnerabilities, while SonarQube missed multiple XSS and SQL vulnerabilities.

Overall, the comparison has shown that our approach outperforms SonarQube when performing vulnerability discovery on Securibench Micro.

4.3.4 Case Study

While the experiment allows us to study the performance of our method in test cases, the details are difficult to present comprehensibly within a limited space. For this reason, we further present a case study to illustrate how our approach is used in detail to help the reader understand our method better.

```
1  void main() {
2      [...]
3      a = new A();
4      b = a.g;
5      x = a.g;
6      sink1(b.f);
7      w = source1();
8      x.f = w;
9      p = source2();
10     [...]
11     if (p == isTaint) {
12         isSecure(p);
13     } else {
14         sink2(p);
15     }
16     sink1(b.f);
17 }
```

Figure 4.10. Code fragment adapted from a real-world case.

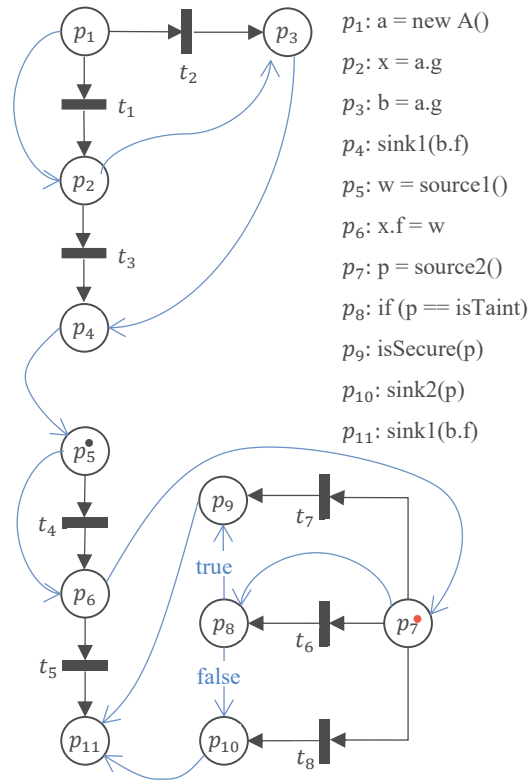


Figure 4.11. Vulnerability net for the code in Figure 4.10.

Figure 4.10 is the code adapted from an abstract code fragment, which is abstracted from a real-world case [61]. For brevity, the type information of each variable is omitted in the code. The ellipses that appear in lines 2 and 10 represent some code omitted. The code fragment contains two source-sink pairs, namely $(source1, sink1)$ and $(source2, sink2)$. To identify the potential taint-style vulnerabilities that exist in this code, we want to examine whether a tainted value from a source (e.g., $source1$) may reach its paired sink (e.g., $sink1$).

Our analysis begins by generating the vulnerability net (see Algorithm 4.1), as shown in Figure 4.11. Note that we assume the DDG can resolve aliasing [21, 22], thereby allowing the dependences to be generated accurately. For example, in the code given in Figure 4.10, since the variable b (line 4) and x (line 5) are aliases of each other, $b.f$ and $x.f$ are

aliased. Accordingly, p_6 (i.e., $x.f = w$) in Figure 4.11 is connected to p_{11} (i.e., $sink(b.f)$).

Then we proceed to analyze the net. As described in Figure 4.9, the input for our analysis includes vulnerability net edges and source-sink pairs. The vulnerability net edges of Figure 4.11 are represented as follows.

p1	p2	1
p1	p3	1
p2	p4	1
p5	p6	1
p6	p11	1
p7	p8	1
p7	p9	1
p7	p10	0

It is often not hard to determine the Boolean values, but if sanitization is present, it may require the CFG information and analysts' expertise. In this example, transitions t_7 (from p_7 to p_9) and t_8 (from p_7 to p_{10}) need attention. We notice that the expression $p \neq isTaint \wedge nonsan(p)$ in t_7 can be true, so "1" is assigned to the edge $VNEdge(p7, p9, 1)$. By contrast, the expression in t_8 , i.e., $p \neq isTaint \wedge nonsan(p)$, is different; the sub-condition $p \neq isTaint$ is the sanitization of p , while another sub-condition $nonsan(p)$ states that no sanitization of p exists. That is, $p \neq isTaint \wedge nonsan(p)$ is a contradiction and is always false, so "0" is assigned to the edge $VNEdge(p7, p10, 0)$.

We next input the source-sink pairs:

p5	p4
p5	p11
p7	p10

Finally, we run the Datalog program to process the input data and output the following results:

p5 p11

Therefore, a vulnerability (p_5, p_{11}) is reported, while (p_5, p_4) and (p_7, p_{10}) are not regarded as vulnerabilities. Alternatively, we can also obtain the same results if we manually analyze the changes in state of the vulnerability net. The initial markings of the net are $\mu_0^1 = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$ (for the black token) and $\mu_0^2 = (0, 0, 0, 0, 0, 0, 1, 0, 0, 0)$ (for the red token). After running the net, we will eventually obtain the corresponding final markings $\mu_f^1 = (0, 0, 0, 0, 1, 1, 0, 0, 0, 1)$ and $\mu_f^2 = (0, 0, 0, 0, 0, 0, 1, 1, 1, 0)$ and yield a vulnerability (p_5, p_{11}) . For brevity, we omit the detailed discussion here.

In presenting the case study we emphasize that our approach reduces false negatives and positives because:

- Vulnerability nets support aliasing analysis. The use of DDG enables our approach to handle aliasing issues, so we can recognize that $b.f$ and $x.f$ are aliased in Figure 4.11. Consequently, the vulnerability (p_5, p_{11}) can be correctly detected.
- Vulnerability nets are flow-sensitive. From the vulnerability net in Figure 4.11 we can see that the token in p_5 may propagate to p_{11} , but never propagate to p_4 even though p_4 and p_{11} represent an identical statement (i.e., $sink1(b.f)$). Thus, a spurious vulnerability (p_5, p_4) will not be reported.

Sanitization is recognized in vulnerability nets. Since the sanitization in t_8 of the net in Figure 4.11 is recognized, t_8 is proved to not be enabled and thus the token in p_7 cannot propagate to p_{10} . That is, (p_7, p_{10}) will not be incorrectly considered as a vulnerability.

4.4 Discussion

Our evaluation demonstrates the capability of our approach for uncovering taint-style vulnerabilities in source code. However, several limitations arise if we apply our approach

to more sophisticated programs in the real world. Firstly, our approach is intended only for detecting taint-style vulnerabilities, so it is unclear if similar ideas can be applied to discovery of other types of vulnerabilities.

Secondly, the approach discussed in this chapter involves only intraprocedural analysis, since a data dependence graph or a control flow graph are built on a single procedure. As mentioned in Section 4.3, although procedure inlining can make our intraprocedural analysis, in effect, interprocedural analysis, it has certain inherent limitations. Fortunately, our approach can be extended for interprocedural analysis if system dependence graphs [88] and interprocedural control flow graphs are introduced.

Thirdly, our approach omits the discussion of input validation, just as many other taint analysis approaches do (e.g., [61]). Input validation is often adopted as a complement to sanitization since sanitization alone may not be sufficient to handle external input comprehensively. False alarms can arise when input validation is not successfully recognized. There are many approaches (e.g., [89-91]) in the literature that have shown how to identify input validation and how this can greatly reduce false positives during taint analysis.

Lastly, this work lacks a tool to support the proposed approach, thus restricting our evaluation to small-scale experiments. We discuss some basic ideas about the implementation of vulnerability nets in the following, which can serve as a starting point for developing a tool:

- **Visualization.** This component defines graphical representations of the elements in a vulnerability net. For example, places are represented by circles. The component should display a vulnerability net in a structured manner.
- **Transformation.** This component defines how the statements in a program are mapped to vulnerability net constructs. Techniques such as *rule-based mapping* can be used to guide the transformation process and ensure that the generated vulnerability net can accurately represent the program's characteristics.

- **Simulation.** This component handles the firing of transitions based on the current markings and firing rules defined in the vulnerability net. It should allow the user to specify the initial marking, track token movements, and observe the changes in the marking. To visualize the simulation process, the Simulation component should be integrated with the Visualization component.
- **Analysis.** This component performs vulnerability analysis based on the simulation data from the Simulation component. It may include various analyses, such as reachability analysis and taint analysis. Analysis results are generated in this component.

4.5 Related Work

Although purely manual audits are rarely used in practice, some interesting work has been done in this direction. Leveson et al. [19, 76] present a method using software fault trees to perform safety analysis at the source code level. The analysis is expressed in a tree form, which starts with determining a fault of interest, followed by a backward analysis to find the set of possible causes. This kind of method relies heavily on the analyst's expertise, making it unlikely to achieve full automation. Similar work (e.g., [92, 93]) has also been presented in this branch of research. A major similarity between these approaches and ours is that we both use a graphical node to explicitly represent a statement or predicate in source code, which can assist analysts in auditing the code manually. However, our approach discussed in this chapter can not only facilitate manual audits, but also support automated static analysis.

In comparison to manual audits, automated static analysis is much more popular for vulnerability discovery. We discuss the work most related to ours in the following. Arzt et al. [61] present FlowDroid, a taint analysis approach for Android applications, which claims to be fully context, flow, field and object-sensitive, thus reducing both false negatives and false positives. There are also other approaches performing taint analysis on Android, such as AmanDroid [94] and DroidSafe [95]. To further detect complex and subtle vulnerabilities, some approaches incorporate expert knowledge into the process of

vulnerability discovery. Livshits and Lam [21] suggest a taint analysis method using user-provided specifications to find security vulnerabilities in Java applications. Yamaguchi et al. [29] merge abstract syntax tree, control flow graphs and program dependence graphs into a joint data structure, in which analysts craft certain rules, known as *traversals*, to facilitate vulnerability auditing. While all these approaches and ours are similar in the sense that we all detect taint-style vulnerabilities based on taint analysis, a clear distinction is that our approach provides a graphical view during the analysis process, which provides analysts with more intuitive information.

It may also be worth mentioning the static analysis approaches focusing on incomplete programs. These approaches, as we have discussed in Chapters 2 and 3, enable analysis in the coding phase, making it possible to perform vulnerability discovery in real time. Our vulnerability net approach also supports such kind of idea since a vulnerability net can be generated from the start of coding and incrementally expanded as coding proceeds.

5 Conclusion and Outlook

In this chapter, we conclude the main findings of this thesis and present future work.

5.1 Summary of Results

- We present a general framework for timely vulnerability discovery in Human-Machine Pair Programming. The framework is expected to be applicable to model a range of vulnerabilities. We illustrate its feasibility through a case study.
- We present two pointer analysis approaches, exhaustive pointer analysis and demand-driven pointer analysis, to identify taint-style vulnerabilities in Human-Machine Pair Programming. Both the approaches support an incremental pointer analysis and provide points-to information for vulnerability discovery during program development. Our evaluation includes experiments on Securibench Micro and a real-world case study: LDAP Injection in Apache Druid 0.17.0.
- We put forward vulnerability nets, a novel graphical code representation for modeling and detecting vulnerabilities in source code. Vulnerability nets support both automated analysis and manual audits. To demonstrate the effectiveness, we have also tested the approach on Securibench Micro.

5.2 Future Work

The work in this thesis is expected to lay the foundation for developing more advanced and intelligent static analysis approaches in the future. Some of the interesting future research topics in this branch may include:

- Incorporating AI-based methods to improve some aspects of this work. For example, as mentioned in Section 4.3.1, some AI-based methods can greatly improve the identification of taint information (such as taint sources and sinks) in large-scale programs.
- To reduce false positives in our vulnerability net approach, we need to obtain more precise information from a program, so it is essential to represent finer grained

elements of a program (e.g., variables' values or types) in a vulnerability net. To this end, it might be necessary to incorporate techniques such as abstract syntax trees (ASTs) in our vulnerability nets.

- Our pointer analysis and vulnerability net approaches focus exclusively on taint-style vulnerabilities. It might be worth exploring if similar ideas can be applied to the detection of other types of vulnerabilities.

6 Bibliography

- [1] H. Moore, "Security flaws in universal plug and play: Unplug. don't play," *Rapid7, Ltd*, vol. 8, 2013.
- [2] CVE-2014-0160. "The Heartbleed Bug." <https://heartbleed.com/>. (accessed November 2023).
- [3] Z. Durumeric *et al.*, "The matter of heartbleed," In *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475-488.
- [4] Twitter Privacy Center. <https://privacy.twitter.com/en/blog/2022/an-issue-affecting-some-anonymous-accounts>. (accessed November 2023).
- [5] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," In *NDSS*, 2008, vol. 8, pp. 151-166.
- [6] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.
- [7] K. Goseva-Popstojanova and J. Tyo, "Experience report: security vulnerability profiles of mission critical software: empirical analysis of security related bug reports," In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 152-163.
- [8] FindBugs - Find Bugs in Java Programs. <https://findbugs.sourceforge.net/>. (accessed January 2023).
- [9] SonarQube. <https://www.sonarsource.com/products/sonarqube/> (accessed November 2023).
- [10] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000, pp. 257-267.

- [11] ESLint. <https://eslint.org/> (accessed November 2023).
- [12] Coverity Static Application Security Testing (SAST). <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>. (accessed November 2023).
- [13] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," In *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 672-681.
- [14] Y. Pan, X. Ge, C. Fang, and Y. Fan, "A systematic literature review of android malware detection using static analysis," *IEEE Access*, vol. 8, pp. 116363-116379, 2020.
- [15] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspán, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58-66, 2018.
- [16] CVE Numbering Authority (CNA) Rules. <https://www.cve.org/ResourcesSupport/AllResources/CNARules>. (accessed November 2023).
- [17] F. Yamaguchi, "Pattern-Based Vulnerability Discovery," 2015.
- [18] B. Chess and J. West, *Secure Programming With Static Analysis*. Pearson Education, 2007.
- [19] N. G. Leveson, S. S. Cha, and T. J. Shimeall, "Safety verification of ada programs using software fault trees," *IEEE software*, vol. 8, no. 4, p. 48, 1991.
- [20] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006, pp. 6-pp.

- [21] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," In *USENIX security symposium*, 2005, vol. 14, pp. 18-18.
- [22] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238-252.
- [23] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84-87, 2005.
- [24] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1-36, 2017.
- [25] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1-39, 2018.
- [26] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends in Programming Languages*, vol. 2, no. 1, pp. 1-69, 2015.
- [27] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," In *Proceedings DARPA Information Survivability Conference and Exposition (DISCEX'00)*, 2000, vol. 2, pp. 119-129.
- [28] S. Heelan, "Vulnerability detection systems: Think cyborg, not robot," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 74-77, 2011.
- [29] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," In *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590-604.

- [30] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," In *10th USENIX Security Symposium (USENIX Security 01)*, 2001, pp. 177-190.
- [31] G. Sindre and A. L. Opdahl, "Eliciting security requirements with misuse cases," *Requirements engineering*, vol. 10, pp. 34-44, 2005.
- [32] B. Chess and G. McGraw, "Static analysis for security," *IEEE security & privacy*, vol. 2, no. 6, pp. 76-79, 2004.
- [33] B. Potter and G. McGraw, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81-85, 2004.
- [34] S. Liu, "Software Construction Monitoring and Predicting for Human-Machine Pair Programming," In *Structured Object-Oriented Formal Language and Method: 8th International Workshop, SOFL+ MSVL 2018*, 2018, pp. 3-20.
- [35] P. Wang, S. Liu, A. Liu, and F. Zaidi, "A Framework for Modeling and Detecting Security Vulnerabilities in Human-Machine Pair Programming," *Journal of Internet Technology*, vol. 23, no. 5, pp. 1129-1138, 2022.
- [36] B. Schneier, "Attack trees," *Dr. Dobbs's journal*, vol. 24, no. 12, pp. 21-29, 1999.
- [37] A. P. Moore, R. J. Ellison, and R. C. Linger, "Attack modeling for information security and survivability," *Technical Note CMU/SEI-2001-TN-001*, 2001.
- [38] H. S. Lallie, K. Debattista, and J. Bal, "A review of attack graph and attack tree visual syntax in cyber security," *Computer Science Review*, vol. 35, p. 100219, 2020.
- [39] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault tree handbook*. Nuclear Regulatory Commission Washington DC, 1981.

- [40] P. A. Khand, "System level security modeling using attack trees," In *2nd International Conference on Computer, Control and Communication*, 2009, pp. 1-6.
- [41] National Vulnerability Database. <https://nvd.nist.gov/>. (accessed January 2023).
- [42] Common Weakness Enumeration. <https://cwe.mitre.org/data/>. (accessed November 2023).
- [43] Common Vulnerability Scoring System (CVSS). <https://www.first.org/cvss/>. (accessed November 2023).
- [44] T. R. Ingoldsby, "Attack tree-based threat risk analysis," *Amenaza Technologies Limited*, pp. 3-9, 2010.
- [45] D. Vose, *Risk analysis: a quantitative guide*. John Wiley & Sons, 2008.
- [46] J. E. F. Friedl, *Mastering regular expressions*. O'Reilly Media, Inc., 2006.
- [47] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.
- [48] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," In *Proceedings of the IEEE international symposium on secure software engineering*, 2006, vol. 1, pp. 13-15.
- [49] J. Wang, R. C. W. Phan, J. N. Whitley, and D. J. Parish, "Augmented attack tree modeling of SQL injection attacks," In *2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 182-186.
- [50] M. Howard and D. LeBlanc, *Writing secure code*. Pearson Education, 2003.
- [51] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1-28, 2017.

- [52] J. Wilander and M. Kamkar, "A comparison of publicly available tools for static intrusion prevention," In *7th Nordic Workshop on Secure IT Systems*, 2002, p. 68.
- [53] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE software*, vol. 19, no. 1, pp. 42-51, 2002.
- [54] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," In *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 327-337.
- [55] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372-382, 2006.
- [56] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512-530, 2017.
- [57] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008, pp. 313-328.
- [58] A. Rountev, B. G. Ryder, and W. Landi, "Data-flow analysis of program fragments," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 235-252, 1999.
- [59] P. Wang and S. Liu, "Detecting Security Vulnerabilities in Human-Machine Pair Programming with Pointer Analysis," In *27th International Conference on Engineering of Complex Computer Systems (ICECCS 2023)*, 2023, pp. 152-156.
- [60] W. Pingyan and L. Shaoying, "Towards Pointer-Analysis-Based Vulnerability Discovery in Human-Machine Pair Programming," *International Journal of Software Engineering and Knowledge Engineering*, In Press. 2024.

- [61] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259-269, 2014.
- [62] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 87-97, 2009.
- [63] N. Dor, M. Rodeh, and M. Sagiv, "Detecting memory errors via static pointer analysis (preliminary experience)," *ACM SIGPLAN Notices*, vol. 33, no. 7, pp. 27-34, 1998.
- [64] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," In *Compiler Construction: 12th International Conference*, 2003, pp. 153-169.
- [65] B. Steensgaard, "Points-to analysis in almost linear time," In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1996, pp. 32-41.
- [66] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural aliasing," *ACM SIGPLAN Notices*, vol. 27, no. 7, pp. 235-248, 1992.
- [67] N. Heintze and O. Tardieu, "Demand-driven pointer analysis," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 24-34, 2001.
- [68] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for Java," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 59-76, 2005.
- [69] Securibench Micro. <https://github.com/too4words/securibench-micro>. (accessed January 2023).
- [70] Apache Druid 0.17.0. <https://github.com/apache/druid/tree/druid-0.17.0> (accessed November 2023).

- [71] LDAP Injection in Apache Druid. <https://ggolawski.github.io/2020/08/06/cve-2020-1958-ldap-injection-druid.html>. (accessed November 2023).
- [72] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5-21, 2008.
- [73] N. Imtiaz, B. Murphy, and L. Williams, "How do developers act on static analysis alerts? an empirical study of coverity usage," In *IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 323-333.
- [74] J. K. Teto, R. Bearden, and D. C.-T. Lo, "The impact of defensive programming on i/o cybersecurity attacks," In *Proceedings of the 2017 ACM Southeast Regional Conference*, 2017, pp. 102-111.
- [75] A. Rountev and B. G. Ryder, "Points-to and side-effect analyses for programs built with precompiled libraries," In *International Conference on Compiler Construction*, 2001, pp. 20-36.
- [76] N. G. Leveson and P. R. Harvey, "Analyzing software safety," *IEEE Transactions on Software Engineering*, no. 5, pp. 569-579, 1983.
- [77] J. Vanegue and S. K. Lahiri, "Towards practical reactive security audit using extended static checkers," In *IEEE Symposium on Security and Privacy*, 2013, pp. 33-47.
- [78] P. Wang, S. Liu, A. Liu, and W. Jiang, "Detecting Security Vulnerabilities with Vulnerability Nets," *Journal of Systems and Software*, vol. 208, p. 111902, 2024.
- [79] P. Wang, S. Liu, A. Liu, and W. Jiang, "Detecting Security Vulnerabilities with Vulnerability Nets," in *IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2022, pp. 375-383.
- [80] J. L. Peterson, *Petri net theory and the modeling of systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

- [81] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 2004, pp. 97-106.
- [82] N. G. Leveson and J. L. Stolzy, "Safety analysis using Petri nets," *IEEE Transactions on software engineering*, no. 3, pp. 386-397, 1987.
- [83] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319-349, 1987.
- [84] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.
- [85] L. Clapp, S. Anand, and A. Aiken, "Modelgen: mining explicit information flow specifications from concrete executions," In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 129-140.
- [86] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev, "Scalable taint specification inference with big code," In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 760-774.
- [87] Soufflé: A Datalog Synthesis Tool for Static Analysis. <https://souffle-lang.github.io/>. (accessed January 2023).
- [88] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26-60, 1990.
- [89] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54-69, 2015.

- [90] A. Figueiredo, T. Lide, D. Matos, and M. Correia, "MERLIN: multi-language web vulnerability detection," In *IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1-9.
- [91] I. Medeiros, N. Neves, and M. Correia, "Statically detecting vulnerabilities by processing programming languages as natural languages," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 1033-1056, 2022.
- [92] S.-Y. Min, Y.-K. Jang, S.-D. Cha, Y.-R. Kwon, and D.-H. Bae, "Safety verification of Ada95 programs using software fault trees," In *18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'99)*, 1999, pp. 226-238.
- [93] Y. Oh, J. Yoo, S. Cha, and H. S. Son, "Software safety analysis of function block diagrams using fault trees," *Reliability Engineering & System Safety*, vol. 88, no. 3, pp. 215-228, 2005.
- [94] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1-32, 2018.
- [95] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," In *22nd Annual Network and Distributed System Security Symposium*, 2015, vol. 15, no. 201, p. 110.

A Source Code Containing LDAP Injection

The source code provided below is from an authentication module (*LDAPCredentialsValidator.java*) in Apache Druid 0.17.0 [70], with some lines of code removed due to space restrictions. We have highlighted the statements that are directly involved in our pointer analyses. The statements in red are considered necessary for computation in our exhaustive pointer analysis, and the underlined statements are the ones necessary for our demand-driven analysis. (Note that we treat “String username” in line 30 as a statement and the entry source for the sake of discussion.).

```
1 package org.apache.druid.security.basic.authentication.validator;
2 import ...
3 public class LDAPCredentialsValidator implements
    CredentialsValidator {
4     private static final Logger LOG = new
        Logger(LDAPCredentialsValidator.class);
5     private static final ReentrantLock LOCK = new ReentrantLock();
6     private final LruBlockCache cache;
7     private final BasicAuthLDAPConfig ldapConfig;
8     public LDAPCredentialsValidator(...) {
9         this.ldapConfig = new BasicAuthLDAPConfig(...);
10        this.cache = new LruBlockCache(...);
11        Properties bindProperties(BasicAuthLDAPConfig ldapConfig) {
12            Properties properties = commonProperties(ldapConfig);
13            properties.put(Context.SECURITY_PRINCIPAL,
                ldapConfig.getBindUser());
14            properties.put(Context.SECURITY_CREDENTIALS,
                ldapConfig.getBindPassword().getPassword());
15            return properties;}
16        Properties userProperties(BasicAuthLDAPConfig ldapConfig,
            LdapName
                userDn, char[] password) {
17            Properties properties = commonProperties(ldapConfig);
18            properties.put(Context.SECURITY_PRINCIPAL, userDn.toString());
```

```

19 properties.put(Context.SECURITY_CREDENTIALS,
    String.valueOf(password));
20 return properties;}
21 Properties commonProperties(BasicAuthLDAPConfig ldapConfig){
22 Properties properties = new Properties();
23 properties.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");
24 properties.put(Context.PROVIDER_URL, ldapConfig.getUrl());
25 properties.put(Context.SECURITY_AUTHENTICATION, "simple");
26 if
    (StringUtils.toLowerCase(ldapConfig.getUrl()).startsWith("ldaps:/
    /")) {
27     properties.put(Context.SECURITY_PROTOCOL, "ssl");
28     properties.put("java.naming.ldap.factory.socket",
        BasicSecuritySSLSocketFactory.class.getName());}
29     return properties;}
30 public AuthenticationResult validateCredentials(String
    authenticatorName, String authorizerName, String username,
    char[]
        password) {
31 SearchResult userResult;
32 LdapName userDn;
33 Map<String, Object> contextMap = new HashMap<>();
34 LdapUserPrincipal principal = this.cache.getOrExpire(username);
35 if (principal != null && principal.hasSameCredentials(password))
    {
36     contextMap.put(BasicAuthUtils.SEARCH_RESULT_CONTEXT_KEY,
        principal.getSearchResult());
37     return new AuthenticationResult(username, authorizerName,
        authenticatorName, contextMap);
38 } else {
39     try {InitialDirContext dirContext = new
        InitialDirContext(bindProperties(this.ldapConfig));
40     try {userResult = getLdapUserObject(this.ldapConfig,
        dirContext, username);
41         if (userResult == null) {

```

```

42     LOG.debug("User not found: %s", username);
43     return null;}
44     userDn = new LdapName(userResult.getNameInNamespace());}
45     finally {
46         try {dirContext.close();}
47         catch (Exception ignored) {// ignored}}
48     catch (NamingException e) {
49         LOG.error(e, "Exception during user lookup");
50         return null;}
51     if (!validatePassword(this.ldapConfig, userDn, password)) {
52         LOG.debug("Password incorrect for LDAP user %s", username);
53         throw new BasicSecurityAuthenticationException("User LDAP
54             authentication failed username[%s].", userDn.toString());}
54     byte[] salt = BasicAuthUtils.generateSalt();
55     byte[] hash = BasicAuthUtils.hashPassword(password, salt,
56         this.ldapConfig.getCredentialIterations());
56     LdapUserPrincipal newPrincipal = new
57     LdapUserPrincipal(username,
58         new BasicAuthenticatorCredentials(salt, hash,
59             this.ldapConfig.getCredentialIterations()), userResult);
57     this.cache.put(username, newPrincipal);
58     contextMap.put(BasicAuthUtils.SEARCH_RESULT_CONTEXT_KEY,
59         userResult);
59     return new AuthenticationResult(username, authorizerName,
60         authenticatorName, contextMap);}}
60     SearchResult getLdapUserObject(BasicAuthLDAPConfig ldapConfig,
61     DirContext context, String username){
61     try {SearchControls sc = new SearchControls();
62         sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
63         sc.setReturningAttributes(new String[]
64             {ldapConfig.getUserAttribute(), "memberOf"});
64         NamingEnumeration<SearchResult> results =
65             context.search(ldapConfig.getBaseDn(),
66                 StringUtils.format(ldapConfig.getUserSearch(), username),
67                 sc);
65     try {if (!results.hasMore()) {

```

```

66     return null;}
67     return results.next();}
68     finally {results.close();}}
69 catch (NamingException e) {
70     LOG.debug(e, "Unable to find user '%s'", username);
71     return null;}}
72 boolean validatePassword(BasicAuthLDAPConfig ldapConfig, LdapName
    userDn, char[] password) {
73     InitialDirContext context = null;
74     try {context = new InitialDirContext(userProperties(ldapConfig,
    userDn, password));
75     return true;}
76     catch (AuthenticationException e) {
77     return false;}
78     catch (NamingException e) {
79     LOG.error(e, "Exception during LDAP authentication
    username[%s]", userDn.toString());
80     return false;}
81     finally {
82     try {if (context != null) {
83     context.close();}}
84     catch (Exception ignored) {
85     LOG.warn("Exception closing LDAP context");}}}}

```

Figure A.1. Source code of the authentication module in Apache Druid 0.17.0.

B Copyright Documentation

This thesis is extended from our published papers. Chapter 2 is adapted from [35], Chapter 3 is adapted from [59, 60], and Chapter 4 is adapted from [78, 79].