

A Framework for Modeling and Detecting Security Vulnerabilities in Human-Machine Pair Programming

Pingyan Wang¹, Shaoying Liu^{1*}, Ai Liu¹, Fatiha Zaidi²

¹ Graduate School of Advanced Science and Engineering, Hiroshima University, Japan

² Laboratoire Méthodes Formelles, {Université Paris-Saclay, CNRS, ENS Paris-Saclay}, France
{pingyanwang, sliu, liuai}@hiroshima-u.ac.jp, zaidi@lri.fr

Abstract

To detect and mitigate security vulnerabilities early in the coding phase is an important strategy for secure software development. Existing solutions typically focus on finding certain vulnerabilities in certain computer systems without giving a systematic way of handling different types of vulnerabilities. In this paper, we present a framework for systematically modeling and detecting potential security vulnerabilities during the construction of programs using a particular programming paradigm known as *Human-Machine Pair Programming*. The framework provides designers with a general way of modeling a class of attacks in detail, and shows how programmers can discover and fix a vulnerability in a timely manner. Specifically, our framework advocates three primary steps: (1) generating an attack tree to model a given security threat, (2) constructing vulnerability-matching patterns based on the result of the attack tree analysis, and (3) detecting corresponding vulnerabilities based on the patterns during the program construction. We also present a case study to demonstrate how it works in practice.

Keywords: Security vulnerabilities, Human-machine pair programming, Attack trees, Static analysis

1 Introduction

Security vulnerabilities can be found in different phases of a software life cycle and might be exploited by attackers who aim to launch attacks against computer 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 for many systems. 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 [1], coding phase [2] and testing phase [3]. 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 solutions, such as *static analysis* techniques [4-5] and *defensive programming* techniques [6-7], are proposed to

achieve this goal, but most of them only focus on certain systems and vulnerabilities instead of addressing the full scope of the problem. Furthermore, since most of the proposed techniques involve considerable manual work and humans' collaboration, the efficiency of their application may not be desirable. This paper tries to mitigate these problems by proposing a framework suitable for computer to adopt to automatically uncover vulnerability problems during the construction of programs.

Vulnerability discovery is a critical step in vulnerability analysis because it indicates what and where the problem is. However, it is a challenging issue for many developers because security expertise is required. Therefore, it is desirable to analyze attacks in a systematic and thorough way so that as many vulnerabilities as possible can more easily be learned by the developers. In this paper, we employ attack trees [8] to this end. Attack trees 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 will be decomposed into a set of relatively simple sub-goals and each sub-goal will be further decomposed into lower-level nodes if possible. Leaf nodes, i.e., the lowest level nodes, will be used to describe all potential ways that can cause the attack goal to occur. This paper makes use of attack trees to model any classes of attacks, each of which will be decomposed into multiple smaller attacks that are identified as potential vulnerabilities. Furthermore, the vulnerabilities will be classified, according to the attack tree analysis, into two categories: one can be fixed at code level, which is of interest to this work, and another is unlikely to be addressed at code level.

Another challenge faced by many existing approaches is that developers need to do considerable manual work such as manually adding assertions to find certain vulnerabilities in programs. As a result, many developers only pay attention to some easy-to-find and easy-to-fix vulnerabilities and thus tend to neglect the important vulnerabilities that require some effort to discover and fix. This paper applies Human-Machine Pair Programming (HMPP) [9] to alleviate the problem. HMPP is characterized by the feature that humans (i.e., developers) create algorithms, data structures, and the architecture of the program whereas the machine (i.e., the computer) acts as an assistant: 1) to monitor the program under construction to identify potential software defects or violation of standards in the program, and 2) to predict useful program segments for enhancing the robustness and the completeness of the program. HMPP has various advantages; for example, no

communication between different developers is needed. Inspired by such a programming paradigm, the developer and the computer in our approach can work interactively and collaboratively, as opposed to the developer that finds and fixes code vulnerabilities manually.

In this paper, we make three contributions. Firstly, we propose a framework for building a computerized technology to systematically and automatically detect vulnerabilities during the construction of programs. This technology can effectively support the new programming paradigm known as Human-Machine Pair Programming. Secondly, we put forward a systematic approach to model vulnerabilities and construct vulnerability-matching patterns in the framework that can be employed to detect corresponding vulnerabilities. Thirdly, we describe a way that the human programmer can effectively interact and collaborate with the computer in the framework.

The rest of this paper is organized as follows. Section 2 introduces the background knowledge necessary for our proposed framework, including attack trees and Human-Machine Pair Programming. Section 3 proposes a framework to systematically deal with security vulnerabilities in the coding phase. Section 4 provides a case study on SQL injection attacks (SQLIAs). Section 5 reviews related work and section 6 presents the conclusion and future work.

2 Background





In this section, we briefly introduce the attack trees and HMPP both of which are related to our framework.

2.1 Attack Trees

An attack tree [8] 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-decompositions can be decomposed as a set of attack sub-goals, any one of which must be achieved for the attack to succeed [10].

Generally, both graphical representation and textual representation can be used to represent an attack tree. In this paper, we use graphical representation and borrow some useful symbols from fault trees [11-12], as shown in Table 1. Note that the meaning of each symbol used in this paper might be slightly changed. For example, while circles represent basic events in a fault tree, they represent atomic attacks in this paper.

Table 1. Symbols used in this paper

Symbol	Fault trees [12]	This paper
	Basic event	Atomic attack
	Intermediate event	Attack goal/sub-goal
	AND	AND
	OR	OR

In this paper, the root node, intermediate nodes, and leaf nodes in an attack tree represent the *attack goal*, *sub-goals*, and *atomic attacks*, respectively (see Figure 1). Formally, an attack tree is defined as follows.

Definition 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 paper we use the term *attack scenario* (also known as *intrusion scenario* [10]) 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 trees [12]. Figure 1 provides a simple example to describe the decomposition of an attack goal. In this tree, for example, to achieve the attack goal G_0 , attackers must achieve one sub-goal either G_1 or G_2 ; similarly, to achieve the sub-goal G_1 , attackers must successfully launch both atomic attack A_1 and A_2 . 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 $\langle A_4 \rangle$.

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 [13]. 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 uses PGP [8].

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 some useful information, such as the severity of the attack and the probability of occurrence, thus allowing one to better evaluate the attack. We will elaborate on that in Section 3.1.

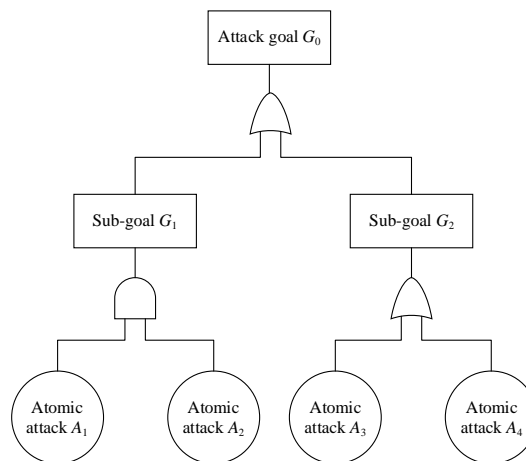


Figure 1. Example of an attack tree

2.2 Human-Machine Pair Programming

HMPP [9], inspired by pair programming [14], is characterized by the feature that the human programmer creates algorithms and data structures for the program under construction while the computer provides a constant checking for detecting bugs and predicting future contents. The bugs can be classified into different categories, such as requirements-related bugs, implementation-related bugs, security-related bugs, and efficiency-related bugs. In this paper, we focus exclusively on security-related bugs.

HMPP can be supported by Software Construction Monitoring (SCM) and Software Construction Predicting

(SCP) [9]. Figure 2 shows the basic framework for SCM. The *Syntactical Analysis* of the current version of software CV_S can help form a set of specific properties that need to be checked. The *property-related knowledge base*, equipped with essential software properties such as software development conventions or standards and common faults, can be updated over time. Checking properties can ensure that CV_S satisfies those specific properties. Once any of the properties are found that are not satisfied, faults will be reported. The fault report will provide some brief but useful information about the faults, such as the location of the faults.

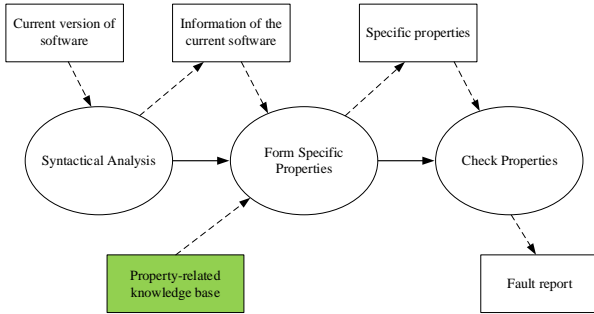


Figure 2. Basic framework for SCM [9]

3 Proposed Approach

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

Figure 3 shows the general overview of the proposed framework, consisting of two phases: *pattern preparation* phase (orange shaded boxes) and *pattern application* phase (blue shaded boxes). In Figure 3, we use D , C and P to represent the *designer*, *computer*, and *programmer*, respectively. 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.

Section 3.1 and 3.2 discuss the pattern preparation and pattern application, respectively.

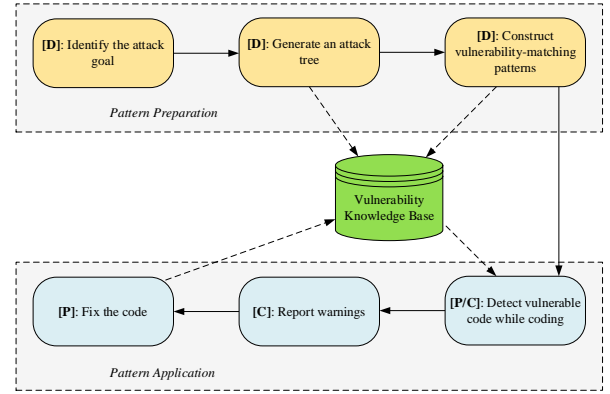


Figure 3. Overview of the proposed framework

3.1 Pattern Preparation

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

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) [15] and Common Weakness Enumeration (CWE) [16]. On the other hand, a designer from an enterprise may focus on identifying a set of attacks that can compromise the systems of the enterprise.

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 4.

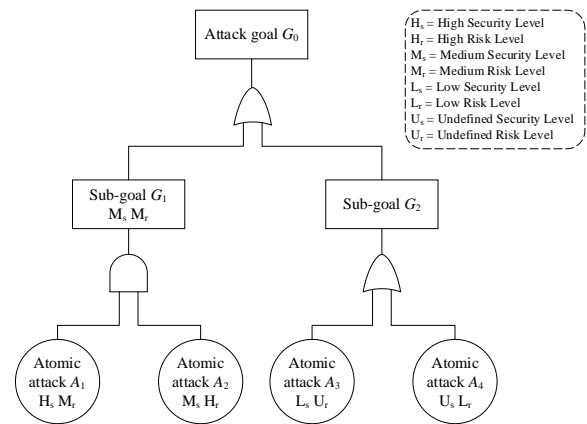


Figure 4. Generation of an attack tree

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 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 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 [17]. 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 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.

- a) *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 [18-19]. 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).
- b) *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.
- c) *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 [20-21] analysis based on the resources and benefits stated 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 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 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 stated 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.

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. 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 paper 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 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 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* [22-23] and *taint analysis* [24]. 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 [23].

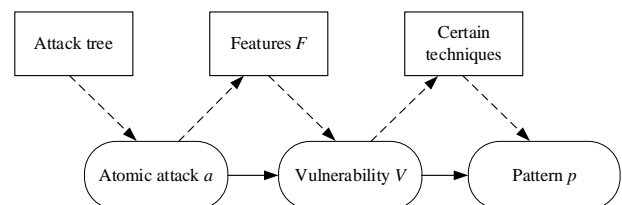


Figure 5. Process of pattern construction

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 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.

As an example, Example 1 and 2 use regular expressions to illustrate false negatives and false positives, respectively.

Example 1. Consider the code snippet in Figure 6(a). Let us use a regular-expression pattern to match any method following `fw..`. If a pattern `fw[\w.]+\ (.+\)` 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 6(b).

Example 2. Consider the code snippet in Figure 6(a). Let us use a regular-expression pattern to match any method following `fw..`. If a pattern `fw[\w.]+.+` 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 6(c).

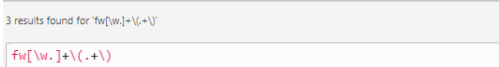
```

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

```

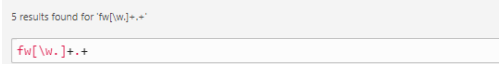
1 // false negatives
2 FileWriter fw = new FileWriter("fw.txt");
3 fw.write("a");
4 fw.write("bcd");
5 fw.write(123);
6 fw.close();
    
```



(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();
    
```



(c) False positives

Figure 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.

3.2 Pattern Application

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

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`.

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 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 7. Example of warning report

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

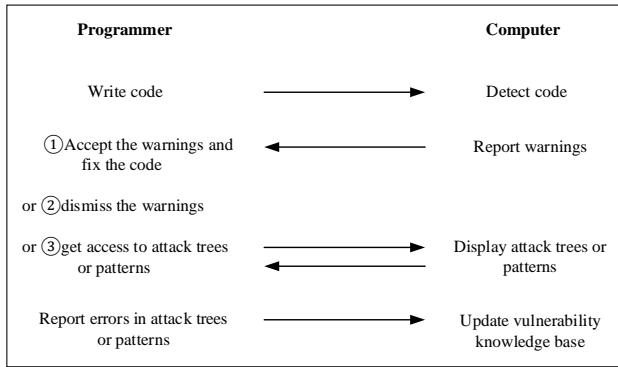


Figure 8. Interaction between programmer and computer during pattern application phase

4 Case Study

In this section, we will illustrate the proposed framework in a case study. To demonstrate the main idea more clearly, we use a fictitious web-based stock exchange trading system as the target system. Figure 9 depicts the architecture of the stock exchange trading system. The system allows customers and companies to register, buy or sell stocks. In this system, most data, such as stock information and trading records, are stored in a database, where data can be accessed and managed with Structured Query Language (SQL).

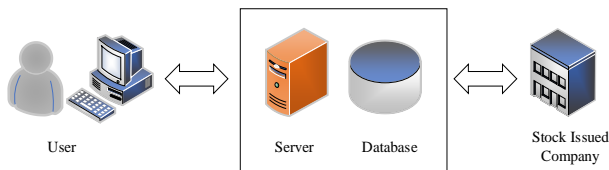


Figure 9. Architecture of stock exchange trading system

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) [16], SQLIAs can pose a serious threat to many web applications.

4.1 Modeling SQLIAs with Proposed Framework

Based on the proposed framework, this subsection describes the entire process for modeling SQLIAs from pattern preparation to pattern application. Each of the 6 steps below corresponds to the steps described in Section 3, respectively.

4.1.1 Identifying Attack Goals

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

4.1.2 Generating Attack Trees

We generate the attack tree against SQLIAs, as shown in Figure 10. Note that a complete attack tree of SQLIAs could be much more complicated as it involves many different types of attacks and countless variations [25-26]. 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 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 10, which is also an attack scenario (1.1.1).

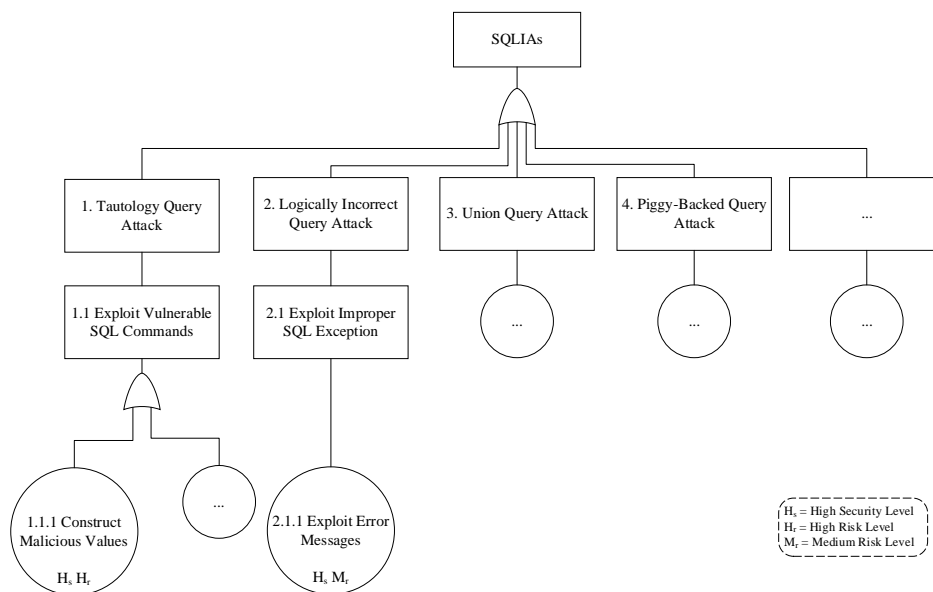


Figure 10. Attack tree against SQL injection

For security level of node 1.1.1, we assess it based on the security metrics. Since this attack is able to bypass the authentication (see next step for detailed discussion), the confidentiality cannot be guaranteed. Moreover, integrity and availability can also be violated because the attacker might modify or delete customers' data via launching this type of attack. 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, but only a computer and some basic security knowledge. Finally, the expected benefits outweigh the potential risks because this type of attack allows the attacker to gain much information from the database. For example, the attack can reveal most, if not all, customers' stock trading information stored in the system. 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 (1.1.1) share the same property values H_s and H_r , i.e., $S_{(1.1.1)} = S_{1.1.1} = \{H_s, H_r\}$.

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 10. To clarify, we take an example of the following code fragment (Figure 11):

```

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 11. Illustrative code fragment

This code is susceptible to (1.1.1) because it creates SQL statements by using string concatenation [27] 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+\s
WHERE\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 [24] 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 $\{\text{getParameter}, \text{data propagation}, \text{execute}\}$, 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 injection because the data-propagation analysis generally involves *pointer analysis* (also known as *points-to analysis*), which clearly shows what a variable may refer to [28]. 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 [27] instead of string concatenation to build queries is one possible solution to avoid this type of SQL injection.

4.1.4 Detecting Vulnerable Code

As shown in Figure 12, 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+\\s+FROM\\s+\\s+\\s+WHERE\\s+\\s+\\s+"[;]*'

(\\w+\\s+\\s*)+"SELECT\\s+\\s+\\s+FROM\\s+\\s+\\s+\\s+WHERE\\s+\\s+\\s+"[;]*'

Figure 12. Detection for vulnerable code

4.1.5 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 13. 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.

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

Figure 13. Warning report for the illustrative code

4.1.6 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 [27].

5 Related Work

There are many attempts to prevent or uncover security vulnerabilities during coding phase. Defensive programming is one possible technique to this end; it is proposed to check whether the code is executing correctly by adding assertions [7]. The idea is based on the fact that an assertion must be evaluated *true* when the program is executing; otherwise, the execution will be terminated. Teto et al [6] apply defensive programming to mitigate I/O cybersecurity attacks by using input validation and escaping (i.e., encoding) techniques. Though defensive programming is promising, there remain critical issues. One of the major challenges is that programmers are required to possess sufficient security knowledge such as adding appropriate assertions.

Static analysis is a more popular and powerful method for uncovering security-related bugs during software development [29]. Static analysis techniques can be employed

to statically examine the source code of a program without executing it [2]. Basic lexical analysis is adopted by practical tools such as ITS4 [4] 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. Yamaguchi et al. [30] introduce code property graphs, which merge abstract syntax tree, control flow graphs and program dependence graphs, to facilitate code vulnerabilities auditing. Laroche and Evans [31, 5] 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. Such static analysis tools or methods encapsulate security knowledge so that the programmer (i.e., the tool operator) is generally not required to possess as much security expertise as the designer (i.e., the tool developer). Another branch of static analysis in security area is using pointer analysis or taint analysis to find code vulnerabilities. Livshits and Lam [24] present a taint-analysis approach based on pointer analysis to finding security vulnerabilities such as SQL injections and cross-site scripting in Java applications. Arzt et al. [32] present a taint-analysis tool called FlowDroid for Android applications. It claims to be fully context, flow, field and object-sensitive, thereby reducing missed leaks and false positives. However, most existing pointer/taint analysis approaches require whole-program availability [33], namely a complete program to be analyzed, which is at odds with the goal of our approach to some extent since our approach performs analysis on the code that is still under construction, namely an incomplete program.

6 Conclusion and Future Work

Detecting security vulnerabilities during software development can be challenging. This paper presents a framework for systematically and automatically identifying and correcting the vulnerability-related bugs during the construction of programs. The framework is expected to serve as the foundation for building an intelligent tool support for Human-Machine Pair Programming. We discuss the whole process of the idea, such as modeling attacks based on attack trees, conducting risk analysis, and constructing patterns. Finally, we conduct a case study on SQL injection attacks to illustrate the proposed framework.

In spite of the important progress we have made in our framework, there are some issues that remain unsolved. For example:

- It is necessary to provide a further discussion on how to choose the most appropriate techniques when constructing vulnerability-matching patterns for given vulnerabilities.
- To better assess security and risk level in practice, we should take account of more factors, such as the independence of atomic attacks.
- It is critical to create a classification of security vulnerabilities in terms of the software life cycle (for example, some vulnerabilities are easier to be fixed in coding phase than in testing phase.), so that resources for addressing a specific security vulnerability can be well allocated.

In future work, we plan to conduct further research on these topics. In addition, to develop a tool that can be applied to practical development is also part of our future work.

Acknowledgment

The research was supported by ROIS NII Open Collaborative Research 2021-(21FS02).

References

- [1] G. Sindre, A. L. Opdahl, Eliciting Security Requirements with Misuse Cases, *Requirements Engineering*, Vol. 10, No. 1, pp. 34-44, January, 2005.
- [2] B. Chess, G. McGraw, Static Analysis for Security, *IEEE Security & Privacy*, Vol. 2, No. 6, pp. 76-79, November-December, 2004.
- [3] P. Vilela, M. Machado, W. E. Wong, Testing for Security Vulnerabilities in Software, *Proceedings of the Sixth IASTED International Conference on Software Engineering and Applications*, Cambridge, Massachusetts, USA, 2002, pp. 460-465.
- [4] J. Viega, J. T. Bloch, Y. Kohno, G. McGraw, ITS4: A Static Vulnerability Scanner for C and C++ Code, *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, New Orleans, Louisiana, USA, 2000, pp. 257-267.
- [5] D. Evans, D. Larochelle, Improving Security Using Extensible Lightweight Static Analysis, *IEEE Software*, Vol. 19, No. 1, pp. 42-51, January-February, 2002.
- [6] J. K. Teto, R. Bearden, D. C. T. Lo, The Impact of Defensive Programming on I/O Cybersecurity Attacks, *Proceedings of the 2017 ACM Southeast Regional Conference*, Kennesaw, GA, USA, 2017, pp. 102-111.
- [7] F. Schindler, Coping with Security in Programming, *Acta Polytechnica Hungarica*, Vol. 3, No. 2, pp. 65-72, 2006.
- [8] B. Schneier, Attack Trees, *Dr. Dobbs's Journal*, Vol. 24, No. 12, pp. 21-29, December, 1999.
- [9] S. Liu, Software Construction Monitoring and Predicting for Human-Machine Pair Programming, *International Workshop on Structured Object-Oriented Formal Language and Method*, Gold Coast, QLD, Australia, 2018, pp. 3-20.
- [10] A. P. Moore, R. J. Ellison, R. C. Linger, *Attack Modeling for Information Security and Survivability*, Technical Note CMU/SEI-2001-TN-001, March, 2001.
- [11] H. S. Lallie, K. Debattista, J. Bal, A Review of Attack Graph and Attack Tree Visual Syntax in Cyber Security, *Computer Science Review*, Vol. 35, Article No. 100219, February, 2020.
- [12] W. E. Vesely, F. F. Goldberg, N. H. Roberts, D. F. Haasl, *Fault Tree Handbook*, Nuclear Regulatory Commission Washington DC, 1981.
- [13] P. A. Khand, System Level Security Modeling Using Attack Trees, *2nd International Conference on Computer, Control and Communication*, Karachi, Pakistan, 2009, pp. 1-6.
- [14] K. Beck, Embracing Change with Extreme Programming, *Computer*, Vol. 32, No. 10, pp. 70-77, October, 1999.
- [15] National Vulnerability Database (NVD), 2021, <https://nvd.nist.gov/>.
- [16] Common Weakness Enumeration, 2021, <https://cwe.mitre.org/data/>.
- [17] Common Vulnerability Scoring System (CVSS), 2021, <https://www.first.org/cvss/>.
- [18] W. E. Wong, X. Li, P. A. Laplante, Be more Familiar with Our Enemies and Pave the Way Forward: A Review of the Roles Bugs Played in Software Failures, *Journal of Systems and Software*, Vol. 133, pp. 68-94, November, 2017.
- [19] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, M. F. Siok, Recent Catastrophic Accidents: Investigating How Software Was Responsible, *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, Singapore, 2010, pp. 14-22.
- [20] T. R. Ingoldsby, *Attack Tree-Based Threat Risk Analysis*, Amenaza Technologies Limited, 2010.
- [21] D. Vose, *Risk Analysis: A Quantitative Guide*, John Wiley & Sons, 2008.
- [22] J. E. F. Friedl, *Mastering Regular Expressions*, O'Reilly Media, Inc., 2006.
- [23] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Pub. Co., 1986.
- [24] V. B. Livshits, M. S. Lam, Finding Security Vulnerabilities in Java Applications with Static Analysis, *14th USENIX Security Symposium*, Baltimore, MD, USA, 2005, pp. 271-286.
- [25] J. Clarke, *SQL Injection Attacks and Defense*, Elsevier, 2009.
- [26] J. Wang, R. C. W. Phan, J. N. Whitley, D. J. Parish, Augmented Attack Tree Modeling of Sql Injection Attacks, *2nd IEEE International Conference on Information Management and Engineering*, Chengdu, China, 2010, pp. 182-186.
- [27] M. Howard, D. LeBlanc, *Writing Secure Code*, Pearson Education, 2003.
- [28] N. Grech, Y. Smaragdakis, P/Taint: Unified Points-to and Taint Analysis, *Proceedings of the ACM on Programming Languages*, Vol. 1, No. OOPSLA, pp. 1-28, October, 2017.
- [29] J. Wilander, M. Kamkar, A Comparison of Publicly Available Tools for Static Intrusion Prevention, *7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, 2002, pp. 68-84.
- [30] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and Discovering Vulnerabilities with Code Property Graphs, *2014 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2014, pp. 590-604.
- [31] D. Larochelle, D. Evans, Statically Detecting Likely Buffer Overflow Vulnerabilities, *10th USENIX Security Symposium*, Washington, D.C., USA, 2001, pp. 177-190.
- [32] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, P. McDaniel, Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-Aware Taint Analysis for Android Apps, *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, Edinburgh, UK, 2014, pp. 259-269.
- [33] Y. Smaragdakis, G. Balatsouras, Pointer Analysis, *Foundations and Trends in Programming Languages*, Vol. 2, No. 1, pp. 1-69, April, 2015.

Biographies



Pingyan Wang received the M.E. degree in computer science from Guangdong University of Technology. He is currently a Ph.D. candidate in data science and informatics at Hiroshima University. His research interests are in Software Engineering, particularly Software Security, Program Analysis and Human-Machine

Pair Programming.



Shaoying Liu is a Professor of Software Engineering at Hiroshima University, Japan, IEEE Fellow, and BCS Fellow. His research interests include Formal Engineering Methods, Specification-based Program Inspection and Testing, and Human-Machine Pair Programming. He has published a book, 12 edited conference

proceedings, and over 250 papers.



Ai Liu is an Assistant Professor of Software Engineering at Hiroshima University, Japan. He received the Ph.D. in Applied Mathematics from Peking University, China in 2020. His research interests include Testing-Based Formal Verification, Quantum Computing and Coalgebra Theory.



Fatiha Zaidi received the PhD degree in computer science from the University of Evry, France, in 2001. Since 2003, she is an associate professor at Paris Sud University (newly renamed Paris Saclay University). Her research interests include model-based testing, runtime verification, attack tolerance, and parameterized model

checking.