

A Framework for Automatic Detection of Vulnerabilities in Human-Machine Pair Programming

Pingyan Wang¹ Shaoying Liu¹ Ai Liu¹

Abstract: In order to mitigate the severe consequences of security threats, many software-based systems are endeavoring to detect security vulnerabilities as early as possible in the software life cycle. In this paper, we present a framework for systematically detecting and mitigating potential security vulnerabilities during the construction of programs using a particular programming paradigm known as *Human-Machine Pair Programming*. The framework allows developers to address the vulnerability problem in the coding phase rather than fix it at a high price when the system is in operation. Our framework advocates three critical steps: (1) generate an attack tree to model a specific security threat, (2) construct code-matching patterns based on the result of the attack tree analysis, and (3) detect corresponding vulnerable code 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

1. Introduction

Security vulnerabilities can be found in different phases of a software life cycle. 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, such as 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 examine and fix the vulnerable code timely. 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 require considerable manual work and humans' collaboration, the efficiency of their application may not be desirable. This paper tries to address these problems by proposing a framework suitable for computer to adopt to automatically uncover vulnerability problems during the construction of programs. It can efficiently support the Human-Machine Pair Programming paradigm (the details of the framework will be discussed in Section 3).

Since code vulnerabilities can be exploited by attackers, it is important to find out as many potential vulnerabilities as possible. Attack trees [8] are considered as a popular method to describe the sequence of events that can result in a specific attack. In this paper, we make use of this technique to analyze and identify all the possible code vulnerabilities of an arbitrary attack, so that we are able to fix them one by one at code level if possible.

Human-Machine Pair Programming (HMPP) [9] 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 required. Inspired by such a programming paradigm, this paper intends to present an approach that the developer and the computer can work collaboratively instead of finding 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 constructing vulnerable code patterns in the framework that can be used to detect specific vulnerable code. Thirdly, we describe a way that the human programmer can effectively 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

Attack trees provide a way to describe attacks against a system in a tree structure, with the goal as the root node and different ways of achieving that goal as leaf nodes [8]. An attack tree 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].

¹ Hiroshima University, Higashi-Hiroshima City, Hiroshima 739-8511, Japan

Note that although attack trees have been studied for decades, there is no standard way to represent an attack tree [11]; for example, either graphical representation or textual representation can be used. In this paper, we use graphical representation and borrow some useful symbols from fault trees [12]. Table 1 shows several typical symbols and their meanings that will be used in our framework. 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 represent the *attack goal*, *sub-goals*, and *atomic attacks*, respectively (see Fig. 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 a set of property values.

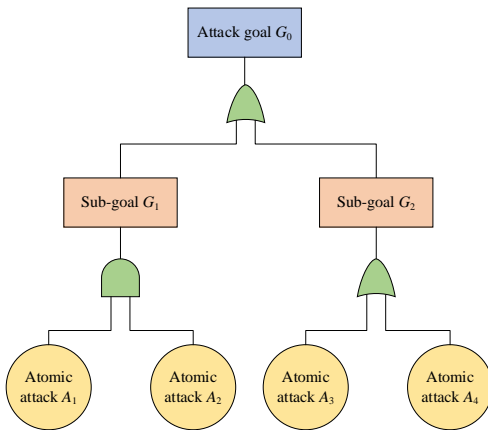


Fig. 1. Example of attack trees.

This paper uses the term *attack scenario* (also intrusion scenario) 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]. Fig. 1 provides a simple example to describe the decomposition of an attack goal. In this tree, for instance, to achieve the attack goal G_0 , attackers must achieve sub-goal G_1 or G_2 ; similarly, to achieve sub-goal G_1 , attackers must successfully launch both atomic attack A_1 and A_2 .

Therefore, there are three attack scenarios, i.e., three different ways to achieve G_0 : $\langle A_1, A_2 \rangle$, $\langle A_3 \rangle$ and $\langle A_4 \rangle$.

Once an attack tree has been generated, the designer can assign values to the leaf nodes. These values enable people to better evaluate the attack. We will elaborate on that in Section 3.

To generate a sound attack tree could take much effort and time because the designer needs to consider all atomic attacks against the attack goal. The designer should think from the perspective of the attacker (instead of the defender) with infinite resources, knowledge and skill [13]. Hence, it depends largely on the experience and expertise of the designer. 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].

2.2 HMPP

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), which aim to automatically and dynamically observe and verify the current version of the software for fault detection and for future contents prediction, respectively. For brevity, we only introduce the basic idea of SCM as follows [9].

Fig. 2 shows the basic framework for SCM. The *Syntactical Analysis* of the current version of software CV_S can help form specific properties that need to be checked. The *property-related knowledge base*, stored essential properties (e.g., the properties based on software development conventions or standards, common faults, etc.) of the software beforehand, can be updated over time.

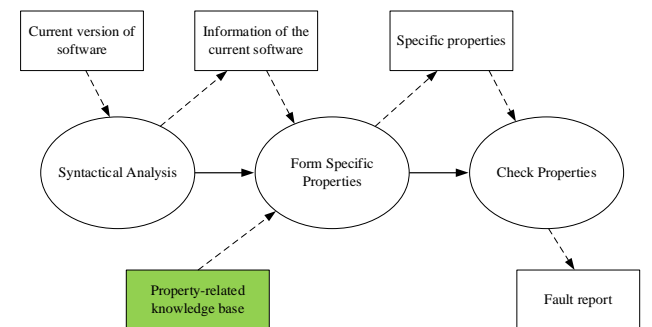


Fig. 2. Basic framework for SCM (taken from [9]).

3. Proposed Approach

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

Fig. 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 Fig. 3, we use D , C and P

to represent the *designer*, *computer* and *programmer*, respectively. The designer (or the analyst) models targeted attacks by creating attack trees and constructing code-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 may check the attack trees and patterns according to the warnings and give feedback (if any) on them. The vulnerability knowledge base may thus be updated based on the feedback.

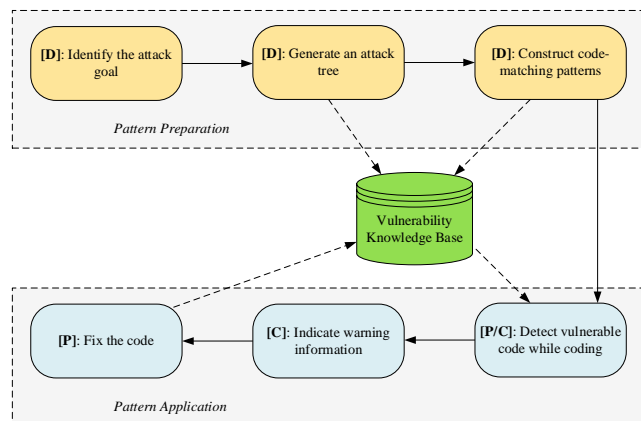


Fig. 3. Overview of the proposed framework.

In our approach, there is no need for the programmer to possess much security expertise or to manually perform security analysis while coding because the manual work (creating attack trees, constructing patterns, etc.) has been done by the designer in the pattern preparation phase. Hence, our approach is programmer-friendly and the programmer can focus on writing the code. Although the manual work may take much time and effort, it is fortunately a one-time event (to a large extent), 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 pattern preparation for the same vulnerability. For this reason, the more designers from the security community adopt our approach, the easier we can build a sound vulnerability database and a powerful tool.

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

3.1 Pattern Preparation

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

3.1.1 Identifying attack goals. In the activity of identifying an attack goal, the attack goal G_0 and the targeted system will be defined. Very often 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]. Subsequently, we need to assign a value of security level to G_0 . For example, we can use

qualitative severity rankings of a set of values $\{Low, Medium, High\}$ for assessing the security level, as described in Common Vulnerability Scoring System (CVSS) [17]. 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. A successful attack against availability, for example, may allow an attacker to launch denial-of-service (DoS) attacks while successful attacks against confidentiality and integrity may allow an attacker to read and modify some sensitive data of a system, respectively.

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 Fig. 4. Meanwhile, risk level should be assigned to each atomic attack and then to each attack scenario. Like security level, risk level in this paper ranges from *Low* to *High* (although any other ranking mechanism can be used). The assessment criterion is based on the probability of occurrence of each atomic attack. However, it is unlikely to calculate the exact probability because the availability of resources (e.g., money, time, etc.) of each attacker varies [18]. Nonetheless, it is possible to conduct a risk analysis to obtain relatively accurate results. To conduct a thorough risk analysis, experience and expertise are required. For the sake of simplicity, the following provides a basic risk assessing method for roughly calculating the risk level [18] [19].

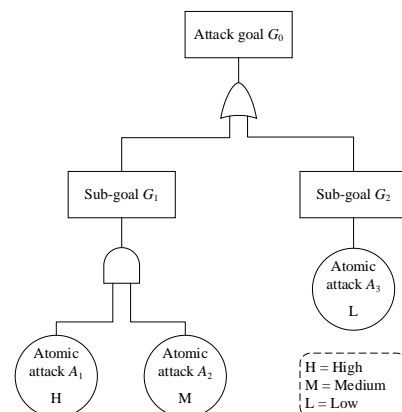


Fig. 4. Generation of an attack tree.

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, user's unintentional manipulation, etc. 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 that we need to do the analysis for a hostile risk. We will analyze what resources are required for an attacker to exploit the vulnerability. The resources may include money, time, raw materials, knowledge, skill, etc.

c) Expected benefits calculation: We will analyze what expected benefits an attacker can gain from a successful attack. That will help us understand attacker's motivation and expected returns.

In order to calculate the risk level of the attack, we can do a

cost-benefit (risk-return) analysis using the resources and benefits mentioned above. For example, if a successful attack is expected to bring considerable benefits but only a few resources are required to launch the attack, there would be high likelihood that the attack will occur such that the risk level will be considered as High.

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

Definition 2. A *code 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 monad, and C is the set of vulnerable code fragments.

The construction of the patterns depends on the analysis of atomic attacks, which can be launched based on the exploitation of certain vulnerabilities. Let V denote a set of vulnerabilities. In this paper, we assume that one atomic attack $a \in A$ is caused by one vulnerability $v \in V$. Let E ($E \subset C$) be a set of equivalent but differently formulated expressions or code fragments, each of which contains the vulnerability v . E may be an infinite set. Let K be the subset of E to denote the expressions or code fragments that are known by the designer. A set of features F can be extracted based on K and be used to construct a pattern $p \in P$. We call the pattern *derived pattern*, and the definition is formally given as follows.

Definition 3. A *derived pattern* $p \in P$ is a pattern that matches a set of equivalent but differently formulated expressions or code fragments E , each of which contains the vulnerability $v \in V$.

The pattern p can be derived from a set of features F , which are extracted based on K ($K \subseteq E$), where K is the known subset of E . K is achieved based on the analysis of the vulnerability $v \in V$ while v is derived from the atomic attack $a \in A$. Fig. 5 shows the process of pattern construction.

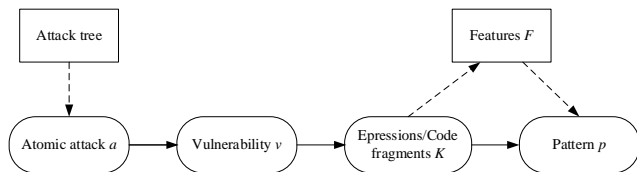


Fig.5. Process of pattern construction.

The features F determine whether the pattern p is powerful enough to reduce *false negatives* (the pattern fails to match the real vulnerable code). On the other hand, the pattern p_i should not be too “powerful”; otherwise *false positives* (the pattern reports false warnings) could occur. We formally define the false negative and false positive as follows.

Definition 4. Let M be the set of expressions or code fragments that a pattern $p \in P$ should match in theory, and let N be the set of expressions or code fragments that the pattern p can match in practice. If there exists an expression or code fragment $c \in M - N$ that the pattern p fails to match, then a *false negative* occurs. If there exists an expression or code fragment $c \in N - M$ that the pattern p does match, then a *false positive*

occurs.

To the best of our knowledge, there is no standard way to construct such derived patterns in practice, but some typical techniques, such as *Regular Expressions* [20] and *Context-Free Grammar* [21], are widely considered as at least a partial solution to this problem. Regular expressions, for example, are efficient for describing lexical structure of constructs such as identifiers, constants, keywords, and white space [21]. These typical techniques can be easily applied to our approach because the targeted attack has been decomposed as fine-grained atomic attacks using attack trees and the atomic attacks become smaller and simpler to model. This paper will use regular expressions as an example to illustrate some steps of the proposed framework. (The detailed regular-expression syntax will not be covered in this paper. Interested readers can see [20] for reference.)

Example 1 and 2 illustrate false negatives and false positives, respectively.

Example 1. Consider a code snippet in Fig. 6. Let us use regular expressions to match any method following *myWriter*.. If a pattern “myWriter[\w.]+\.(.+)\)” is used, then a false negative occurs due to the fact that it fails to match the method *myWriter.close()* in this code, as shown in Fig. 7.

```

1 FileWriter myWriter = new FileWriter("myWriter.txt");
2 myWriter.write("a");
3 myWriter.write("bcd");
4 myWriter.write(123);
5 myWriter.close();
  
```

Fig.6. A code sample.

```

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

```

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

Fig.7. Example of false negatives.

Example 2. Consider a code snippet in Fig. 6. Let us use regular expressions to match any method following *myWriter*.. If a pattern “myWriter[\w.]+.” is used, then a false positive occurs due to the fact that it mismatches the string *myWriter.txt* in this code, as shown in Fig. 8.

```

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

```

5 results found for 'myWriter[\"a\"]<+>'

myWriter[\"a\"]<+>

Fig.8. Example of false positives.

After constructing the patterns for detecting certain vulnerable code, the designer should also work out a solution (i.e., countermeasure) to each corresponding atomic attack at this stage, so that the programmer can take it as a code fix suggestion. Ideally, the solution 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*, *indicating warning information*, and *fixing the code*.

3.2.1 Detecting vulnerable code. In the activity of detecting vulnerable code, particular code that can lead to possible atomic attacks 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 database, which can be read by a tool. Generally, one pattern can be used to monitor and capture one kind of vulnerability. The vulnerable code will be captured in real time once it triggers the corresponding pattern, which is the same as searching specific strings using Unix grep.

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

Fig. 9. Example of warning information.

3.2.2 Indicating warning information. In the activity of indicating warning information, the programmer will be informed of what and where the vulnerability is, and how to fix the vulnerability. The warning information should include the location of the vulnerability, security and risk level information, solutions, etc. The security and risk level have been discussed in the pattern preparation phase. The solutions are the countermeasures for corresponding atomic attacks, which should also be prepared in the pattern preparation phase, and they will serve as suggestions for the programmer. Fig. 9 shows an example of warning information.

3.2.3 Fixing the code. In the activity of fixing the code, the

programmer can examine and fix the vulnerable code timely according to the warning information provided by the computer. The programmer can also decide to fix the code based on his/her own experience and expertise or to completely dismiss the warnings for some reason (e.g., false positives). Moreover, if the programmer is interested in viewing the attack trees and patterns of the warnings, he/she can check them in the vulnerability knowledge base and give feedback (if any) so that the knowledge base will thus be updated based on the those useful feedback from different programmers.

4. Case Study

In this section, we will illustrate the proposed approach in a case study. To demonstrate the main idea of the framework more clearly, we use a simple web-based stock exchange trading system and focus on some common issues that might be familiar to most researchers and practitioners. The stock exchange trading system allows customers and companies to register, buy or sell stocks, etc. Fig. 10 depicts the architecture of the stock exchange trading system.

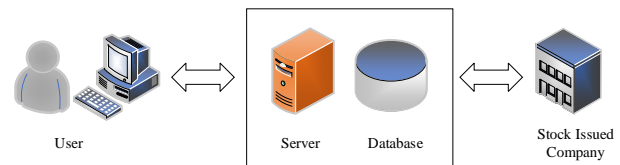


Fig. 10. The architecture of the stock exchange trading system.

We focus on a common vulnerability 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], SQL injection attacks 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 Section 3.1.1 -3.1.3 and Section 3.2.1-3.2.3, respectively.

(1) **Identifying attack goals.** We select the SQLIAs as the attack goal G_0 and the web-based stock exchange trading system as the targeted system. We then assign a value of security level to this type of threat. Since SQLIAs can seriously affect a web-based system [16] [22] (e.g., violating the confidentiality, integrity, and availability of the system), we would assign the value *High* to indicate the security level of this threat.

(2) **Generating attack trees.** We generate the attack tree against SQLIAs, as shown in Fig. 11. Note that a complete attack tree of SQLIAs could be much more complicated as it involves many different types of the attack and countless variations [22] [23]. For the sake of simplicity, we omit some details and generate a simplified, incomplete version based on the properties of the stock exchange trading system.

Once the attack tree is generated, we calculate the risk level of each atomic attack and attack scenario. For example, the following uses the assessing method described in Section 3.1.2 to

illustrate how to calculate the risk level of the atomic attack *Construct Malicious Values* (i.e., node 1.1.1 of Fig. 11), which is also an attack scenario {1.1.1}.

First, we 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 for details), it does not require much time,

money, etc. 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 *High*.

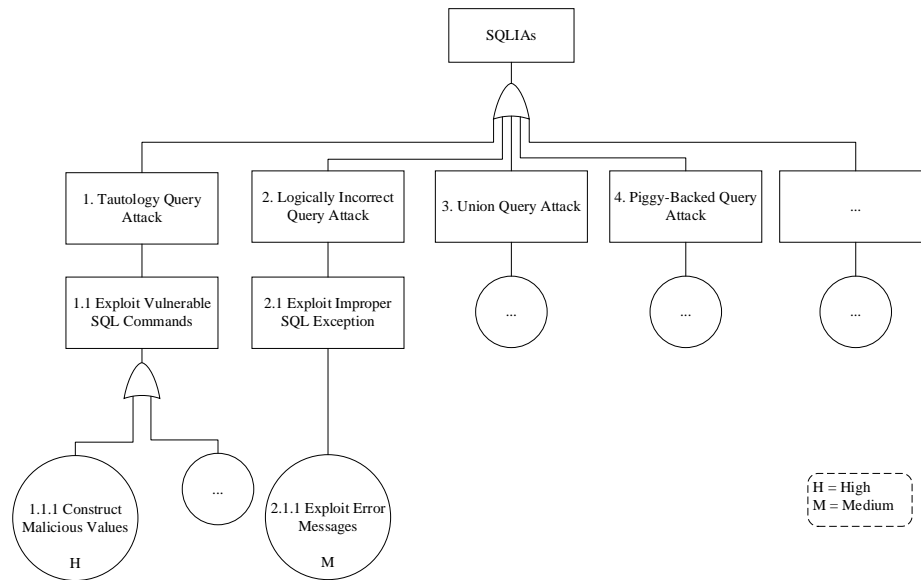


Fig. 11. The attack tree against SQL injection.

(3) **Constructing code-matching patterns.** The construction of a pattern using regular expressions is illustrated by the example below.

Consider that we want to construct a pattern for capturing the vulnerable code related to the attack scenario {1.1.1} (i.e., the atomic attack *Construct Malicious Values*). To clarify, we take an example of the following code fragment:

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

Fig. 12. A sample code fragment.

This code is vulnerable to SQLIAs because it creates SQL statements by using string concatenation [24] 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.

To match such type of SQL query in code, we first extract a set

of important features F based on known queries set K (see Section 3.1.3): keywords (e.g., SELECT), concatenation (using single quotes), and semicolon. In contrast, strings like “customer” and “name” are irrelevant. Accordingly, we can create a pattern shown as follows:

$(\backslash w+\backslash s*=\backslash s*)+\text{"SELECT"}\backslash s\backslash S+\backslash s\text{FROM}\backslash s\backslash S+\backslash s\text{WHERE}\backslash s\backslash S+\backslash s*=\backslash s*\text{[}\wedge\text{;]}*$

Note that this pattern is just an illustrative example and it is not necessarily accurate. Then we can verify the pattern by writing some simple code in any programming development environment (see next step). Finally, we should double check the pattern to avoid false positives.

In addition, as a designer, we should work out a solution to the atomic attack at this stage. For example, using parameterized queries [24] instead of string concatenation to build queries is one possible solution to avoid this type of SQL injection attack. The programmer can take it as a code fix suggestion if necessary (see step 6).

(4) **Detecting vulnerable code.** As shown in Fig. 13, line 5-6 is the corresponding vulnerable code captured in real time by the pattern indicated at the bottom of the figure. Here we use a text editor to show the result although any programming development environment can be used.

```
1 //Use string concatenation to build the query
2 String name, password, query;
3 name = getParameter("name");
4 password = getParameter("pwd");
5 query = "SELECT * FROM customer WHERE name = '" + name + "' AND pwd = '" +
6         password + "'";
7
8
9
10
11
12
13
```

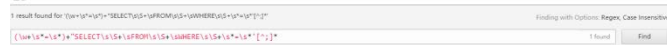


Fig. 13. Detect vulnerable code based on a pattern.

(5) **Indicating warning information.** The warnings indicate information including the location of vulnerable code, the type of possible attack, security and risk level information, etc., as shown in Fig. 14. The location is revealed in step 4. The security and risk level have been discussed in step 1 and step 2, respectively. The solution, as mentioned in step 3, is also given.

Warning(s): The SQL query uses string concatenation
Location: Line 5-6
Possible attack(s): SQL injection
Security level: High
Risk level: High
Solution(s): Consider using parameterized queries

Fig. 14. The warning information for the illustrative code.

(6) **Fixing the code.** Finally, the programmer can examine and fix the code according to the warning information. 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. For more information on parameterized queries, see [24].

5. Related Work

To develop secure code, defensive programming techniques are proposed to check whether the code is executing correctly by adding assertions [21] [25]. This is due to the fact that an assertion must be evaluated *true* when the program is executing; otherwise, the execution will be terminated [21]. 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 of using defensive programming is that programmers are required to possess sufficient security knowledge such as adding appropriate assertions.

Static analysis is a popular method for uncovering security-related bugs during software development [26]. Static

analysis techniques can be used 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. Larochelle and Evans [27] [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. Livshits and Lam [28] present a static analysis approach based on points-to analysis for finding security vulnerabilities such as SQL injections and cross-site scripting in Java applications. In [28], to find as many vulnerabilities as possible, complete user-provided specifications of vulnerabilities should be prepared and translated into static analyzers. Compared with manual security analysis, most static analysis approaches encapsulate security knowledge so that the programmer (i.e., the tool operator) is not required to possess as much security expertise as the designer (i.e., the tool developer). However, most existing static analysis methods are not systematic and thorough due to not decomposing targeted attacks into atomic attacks.

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 an attack 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.

However, there are some issues that remain unsolved. For example:

- Since some tasks are unlikely or even impossible to be done with regular expressions due to their intrinsic restrictions, more methods for constructing code-matching patterns should be explored.
- What other factors should be taken into account when conducting risk analysis in practice?
- Some security vulnerabilities are able to be addressed in various phases of the software life cycle, but resources (e.g., money, time, etc.) required to detect and mitigate these vulnerabilities must vary from phase to phase. Therefore, a classification of security vulnerabilities in terms of the software life cycle is critical.

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

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

Reference

- [1] Sindre, Guttorm, and Andreas L. Opdahl. "Eliciting security

- requirements with misuse cases." *Requirements engineering* 10, no. 1 (2005): 34-44.
- [2] Chess, Brian, and Gary McGraw. "Static analysis for security." *IEEE security & privacy* 2, no. 6 (2004): 76-79.
- [3] Potter, Bruce, and Gary McGraw. "Software security testing." *IEEE Security & Privacy* 2, no. 5 (2004): 81-85.
- [4] Viega, John, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. "ITS4: A static vulnerability scanner for C and C++ code." In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pp. 257-267. IEEE, 2000.
- [5] Evans, David, and David Larochelle. "Improving security using extensible lightweight static analysis." *IEEE software* 19, no. 1 (2002): 42-51.
- [6] Teto, Joel Kamdem, Ruth Bearden, and Dan Chia-Tien Lo. "The impact of defensive programming on I/O Cybersecurity attacks." In *Proceedings of the SouthEast Conference*, pp. 102-111. 2017.
- [7] Rossi, Maria Teresa, Renan Greca, Ludovico Iovino, Giorgio Giacinto, and Antonia Bertoli. "Defensive Programming for Smart Home Cybersecurity." In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 600-605. IEEE, 2020.
- [8] Schneier, Bruce. "Attack trees." *Dr. Dobbs's journal* 24, no. 12 (1999): 21-29.
- [9] Liu, Shaoying. "Software Construction Monitoring and Predicting for Human-Machine Pair Programming." In *International Workshop on Structured Object-Oriented Formal Language and Method*, pp. 3-20. Springer, Cham, 2018.
- [10] Moore, Andrew P., Robert J. Ellison, and Richard C. Linger. *Attack modeling for information security and survivability*. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2001.
- [11] Lallie, Harjinder Singh, Kurt Debattista, and Jay Bal. "A review of attack graph and attack tree visual syntax in cyber security." *Computer Science Review* 35 (2020): 100219.
- [12] Vesely, William E., Francine F. Goldberg, Norman H. Roberts, and David F. Haasl. *Fault tree handbook*. Nuclear Regulatory Commission Washington DC, 1981.
- [13] Khand, Parvaiz Ahmed. "System level security modeling using attack trees." In *2009 2nd International Conference on Computer, Control and Communication*, pp. 1-6. IEEE, 2009.
- [14] Beck, Kent. "Embracing change with extreme programming." *Computer* 32, no. 10 (1999): 70-77.
- [15] "National Vulnerability Database (NVD)." <https://nvd.nist.gov/>, (accessed 2021-07-05)
- [16] MITRE, *Common Weakness Enumeration*. <https://cwe.mitre.org/data/index.html>.
- [17] "Common Vulnerability Scoring System (CVSS)." <https://www.first.org/cvss/>, (accessed 2021-07-05)
- [18] Ingoldsby, Terrance R. "Attack tree-based threat risk analysis." *Amenaza Technologies Limited* (2010): 3-9.
- [19] Vose, David. *Risk analysis: a quantitative guide*. John Wiley & Sons, 2008.
- [20] Friedl, Jeffrey EF. *Mastering regular expressions*. "O'Reilly Media, Inc.", 2006.
- [21] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. "Compilers, principles, techniques." *Addison wesley* 7, no. 8 (1986): 9.
- [22] Halfond, William G., Jeremy Viegas, and Alessandro Orso. "A classification of SQL-injection attacks and countermeasures." In *Proceedings of the IEEE international symposium on secure software engineering*, vol. 1, pp. 13-15. IEEE, 2006.
- [23] Wang, Jie, Raphael C-W. Phan, John N. Whitley, and David J. Parish. "Augmented attack tree modeling of SQL injection attacks." In *2010 2nd IEEE International Conference on Information Management and Engineering*, pp. 182-186. IEEE, 2010.
- [24] Howard, Michael, and David LeBlanc. *Writing secure code*. Pearson Education, 2003.
- [25] Schindler, Frank. "Coping with security in programming." *Acta Polytechnica Hungarica* 3, no. 2 (2006): 65-72.
- [26] Wilander, John, and Mariam Kamkar. "A comparison of publicly available tools for static intrusion prevention." In *7th Nordic Workshop on Secure IT Systems, "Towards Secure and Privacy-Enhanced Systems", 7-8 November 2002, Karlstad University, Sweden*, p. 68. Karlstad University Studies, 2002.
- [27] Larochelle, David, and David Evans. "Statically detecting likely buffer overflow vulnerabilities." In *10th USENIX Security Symposium*. 2001.
- [28] Livshits, V. Benjamin, and Monica S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In *USENIX security symposium*, vol. 14, pp. 18-18. 2005.