Detecting Security Vulnerabilities with Vulnerability Nets

Pingyan Wang, Shaoying Liu*, Ai Liu, and Wen Jiang
Hiroshima University, Higashi-Hiroshima, Hiroshima, Japan
{pingyanwang, sliu, liuai}@hiroshima-u.ac.jp, bonjourjw@gmail.com
*corresponding author

Abstract— **Detecting code vulnerabilities is a crucial part in secure software development. Many static analysis tools have been proven useful in finding vulnerabilities, but generally there are some complex and subtle vulnerabilities that can escape detection. Manual audits are a complementary approach to using tools. Unfortunately, most manual analyses are tedious and error prone. To benefit from both the tools and manual audits, some work incorporates the auditor's expertise into a static analysis tool during vulnerability discovery. Following this strategy, this paper presents vulnerability nets, which are a special Petri net that integrates with data dependence graphs and control flow graphs. Specifically, the proposed approach is intended for detecting taint-style vulnerabilities such as buffer overflows and injection vulnerabilities. In this paper, the construction and use of vulnerability nets are discussed in detail. Furthermore, we show the feasibility by presenting a case study in analyzing an example adapted from a real-world case.**

Keywords-vulnerability; security; static analysis; manual audits; petri nets

I. INTRODUCTION

Security threats are increasingly prevalent in today's computer systems [1]. In software-based systems, code vulnerabilities (also known as security-related bugs) can lead to malicious attacks, which in turn cause security failures [2]. Thus, to find (and then fix) the potential vulnerabilities in the code is a natural tactic to enhance software security.

The vast majority of security vulnerabilities are discovered by static code analysis [3]. The core idea behind static code analysis is to analyze a program without actually executing it. Many static analysis tools (e.g., [4, 5]) have been developed and adopted by developers to facilitate code review. Tools 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 [6], one of the most notorious vulnerabilities, still cannot be fully addressed using automated tools alone. Instead, significant security expertise is often involved during detection of buffer overflows [7].

Manual audits are a complementary (not alternative) method to using automated tools. In the process of auditing, an analyst (i.e., the auditor) manually examine a code, based on his/her expertise, to find vulnerabilities that escape detection of tools. To aid analysts in auditing manually, some researchers analyze source code using techniques such as fault trees [8, 9], in which some crucial information of code is made explicit. However, manual audits are tedious, error prone, and costly, so it is normally impractical to manually audit a whole program.

To benefit from both the static analysis tools and manual audits, some work (e.g., [10, 11]) has considered to incorporate the analyst's security knowledge into a tool during the detection process. The knowledge (such as annotations [12]) provided by security auditors can guide the detection for vulnerabilities. To make contributions in this branch of research, in this paper, we present a novel representation of source code, called a *vulnerability net*, which is in the form of a Petri net structure. Our approach incorporates data dependence graphs and control flow graphs into a vulnerability net. The combination explicitly describes the key information of a code and provides a good view for analysts to audit source code. Analysts can add certain knowledge to the net to augment vulnerability discovery. Like standard Petri nets, vulnerability nets are executable, thus allowing analysts to conveniently track the data of interest to examine the existence of a vulnerability. Specifically, our approach is intended for finding taint-style vulnerabilities, which includes a number of critical vulnerabilities such as buffer overflows [6, 13], injection vulnerabilities [14, 15], and cross-site scripting (XSS) vulnerabilities [16, 17].

To show the feasibility of our approach, we present a case study in analyzing an example adapted from a real-world case. The case study illustrates how our approach reduces false negatives and false positives, yet it is unclear whether the idea can be applied to large-scale programs.

In summary, the main contributions of this paper include:

- Proposing a novel representation of source code called vulnerability nets.
- Illustrating how a vulnerability net that integrates with data dependence graphs and control flow graphs can aid in identifying the presence of vulnerabilities. To formalize this idea, algorithms are also described in this paper.
- Conducting a case study to demonstrate the feasibility of the proposed approach.

The remainder of this paper is organized as follows. Section II presents general definitions for vulnerability nets. Section III describes the use of vulnerability nets for identifying taint-style vulnerabilities and Section IV demonstrates the feasibility by presenting a case study.

2693-9371/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS-C57518.2022.00062

Section V discusses limitations and the possible extension of the proposed approach. Section VI reviews related work and the final section concludes this work.

II. Definition of Vulnerability Nets

In this section, vulnerability nets are formally defined. Most of the notation and terminology are taken or adapted from [18, 19].

A. Vulnerability Nets

A *vulnerability net* is a Petri net structure with some special properties. We formally define it as follows.

Definition 1. 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*, $m \geq 0$. $P \cap T = \emptyset$.

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

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

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

The function of the marking implies that each place holds a number, 0 or 1. By convention, the element that a place holds is called a *token*. Therefore, each place in the vulnerability net holds zero or one token. A marking μ shows the number and distribution of tokens in a vulnerability net. When a vulnerability net executes, the marking may change as the number of tokens may change. The conditional expressions in guarded transitions are the Boolean expressions used to constrain the change in the number of tokens. We use ϵ to denote the absence of a conditional expression.

An example of a vulnerability net is shown in Figure 1 (a). The net is composed of five places and three guarded transitions (henceforth transitions). The initial marking $\mu_0 = (1, 0, 0, 1, 0)$ 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.

We can also use a graphical representation to represent a vulnerability net for ease of reading. A vulnerability net graph for Figure 1 (a) is shown in Figure 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 execute by *firing* transitions. A transition is ready for firing if it is *enabled*. As formally defined in Definition 2, 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) none of the conditional expressions of t_j evaluates to false. For example,

$P = \{p_1, p_2, p_3, p_4, p_5\}$

$T = \{t_1: \epsilon, t_2: \mu(p_1) = 2, t_3: \epsilon\}$

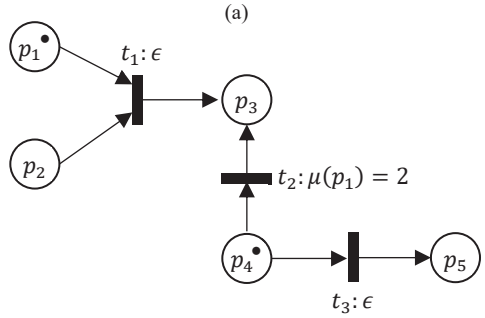
$\mu_0 = (1, 0, 0, 1, 0)$

$I(t_1) = \{p_1, p_2\}$ $\quad O(t_1) = \{p_3\}$

$I(t_2) = \{p_4\}$ $\quad O(t_2) = \{p_3\}$

$I(t_3) = \{p_4\}$ $\quad O(t_3) = \{p_5\}$

(a)



(b)

Figure 1. (a) Textual representation of a vulnerability net; (b) Graphical representation of Figure 1 (a)

the transition t_3 in Figure 1 is enabled because of no violation of 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 contains two tokens, so t_2 is not enabled.

Definition 2. 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. none of the conditional expressions of t_j evaluates to false.

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 3 defines a *next-state function* for calculating how a state can change after firing a transition.

Definition 3. The *next-state function* $\delta: \{0, 1\}^n \times T \to \{0, 1\}^n$ 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

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

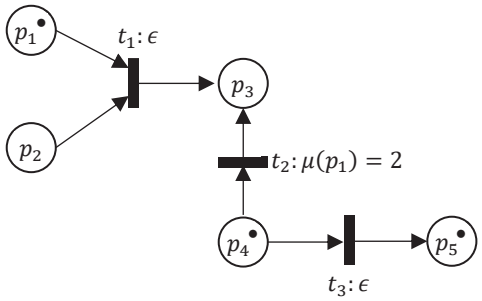


Figure 2. The next state of Figure 1

(a) Execution of a standard Petri net

(b) Execution of a vulnerability net

Figure 3. An example for showing a distinction between vulnerability nets and standard Petri nets

According to (1), we can calculate the next-state result of μ_0 in Figure 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 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.

One may notice that a vulnerability net executes differently from a standard Petri net. As a simple example, Figure 3 shows the distinct results of executing two nets that look initially identical. In Figure 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 3 (b) fires t_1 to propagate a token to p_2 and then the execution halts. More distinctions between a vulnerability net and a standard Petri net can be revealed by their own definitions. In Section III we will see how the special properties of vulnerability nets can benefit vulnerability analyses.

B. *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 II-A as long as we consider each kind of colored tokens separately. We formally give a definition as follows.

Definition 4. 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 $G = \{\mu^1, \mu^2, \ldots, \mu^k\}$, $k \geq 0$, where each μ^i is the marking for a kind of token with a unique color in a graphical representation.

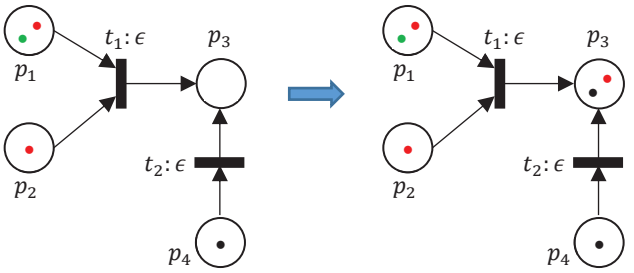


Figure 4. Execution of a vulnerability net with colored tokens

Figure 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 the 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 finial state of the net is $\mu_f^1 = (1,1,0,0)$, $\mu_f^2 = (0,0,1,0)$, and $\mu_f^3 = (1,0,1,1)$.

III. Vulnerability Discovery

In this section, we start by discussing the characteristics of taint-style vulnerabilities and give 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 that this paper focuses on is taint-style vulnerabilities, including a number of critical vulnerabilities such as buffer overflows, injection vulnerabilities, and cross-site scripting vulnerabilities. In taint analysis, a *tainted value* is derived from external input such as results returned from function calls, and command-line arguments. While the *source* is the original program location (such as a function or method) that accepts tainted values, the *sink* is another program location that the tainted values propagate to. Security weaknesses may exist when tainted values reach the sinks. A *sanitization* is a step to remove the taint from a value, thereby eliminating the potential security risk. To perform sanitization, we could

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

Figure 5. Example of a taint-style vulnerability

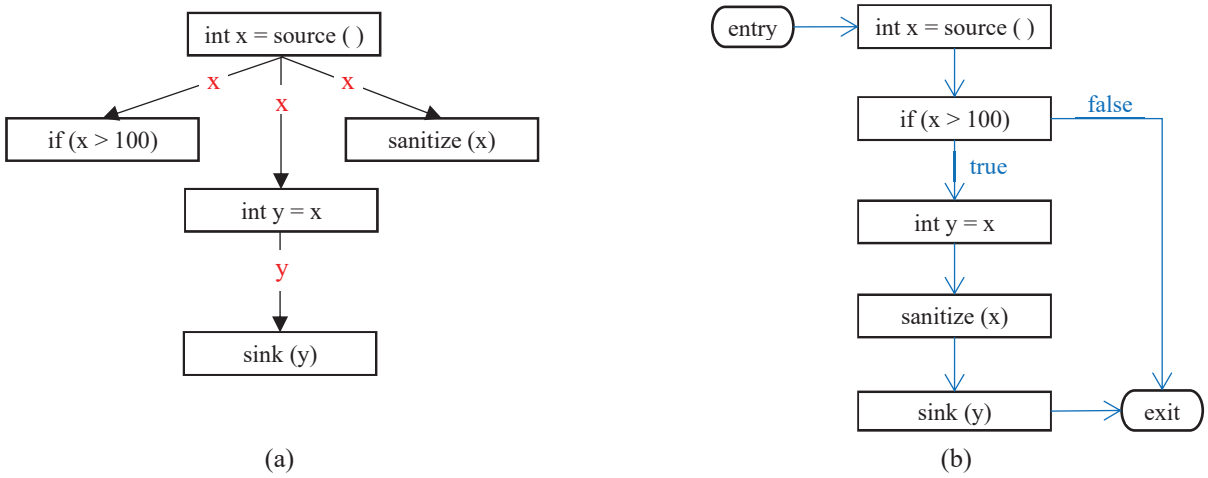


Figure 6. (a) DDG for the example given in Figure 5; (b) CFG for the example given in Figure 5

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 5. In this 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 (let us assume the sanitization used is to terminate the program) and finally a sink function that receives `y` as an argument is called.

A. *Data dependence graphs*

Data dependence graphs (DDGs) are a program representation that can explicitly represent data dependences among statements and predicates [20]. A data dependence is present when two statements cannot be switched without changing any variable's value. For example, in Figure 5, *S*3 depends on *S*1 since *S*1 must be executed first in order for the proper use of x's value in *S*3. Figure 6 (a) shows the DDG for the example code given in Figure 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 5 explicitly shows that the tainted value originated from the source will eventually be passed to the sink and could cause some security issue.

B. *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 [21]. In a CFG, statements and predicates are denoted by nodes, and the flow of control is indicated by directed edges. Figure 6 (b) shows the CFG for the code sample given in Figure 5.

The use of CFGs in our approach is important since data dependence graphs alone often do not suffice to ensure the existence of a vulnerability. For example, if we exchange the order of *S*3 and *S*4 in Figure 5, while the data dependence graph remains unchanged, as shown in Figure 6 (a), the code is no longer a vulnerability because `x`'s value has been sanitized before being passed to `y` and `sink(y)`. As a remedy, a CFG can make explicit whether a sanitization is already executed prior to a sink function. Moreover, control-flow information is also essential in the detection of many other types of vulnerabilities, such as use-after-free vulnerabilities.

C. *Building Vulnerability Nets*

To perform security analysis at code level, we incorporate DDGs and CFGs into vulnerability nets. For brevity, the combination is also referred to as a vulnerability net.

When using a vulnerability net to represent source code, places are used to denote statements and predicates according to a DDG while transitions are used to control the propagation of tokens according a CFG or rules specified by security analysts. A token models the existence of a tainted value. The color of a token represents a specific value; for example, if a black token and a red one are deposited in a place p_i, then the statement denoted by p_i contains two different tainted values. Markings are used to show the number and position of tainted values. Once a token is assigned to a place, we examine whether it can propagate to other places by calculating the changes in the states of the net. If a token can propagate to some sensitive places, i.e., a tainted value can be passed into some sensitive functions, then there exists a code vulnerability in the program. In addition to the above elements, we may add some textual descriptions to the net as needed.

To clarify the idea, consider again the code example shown in Figure 5. We combine the DDG and CFG, both given in Figure 6, and produce a vulnerability net, as indicated in Figure 7 (a). In the net, the places correspond to the statements and predicates while transitions are the conditions controlling the propagation of tokens.

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 $\mu_0 = (1, 0, 0, 0, 0)$. Now we analyze how the marking changes during the execution of the net. First, we can see that t_1 is immediately enabled, resulting in

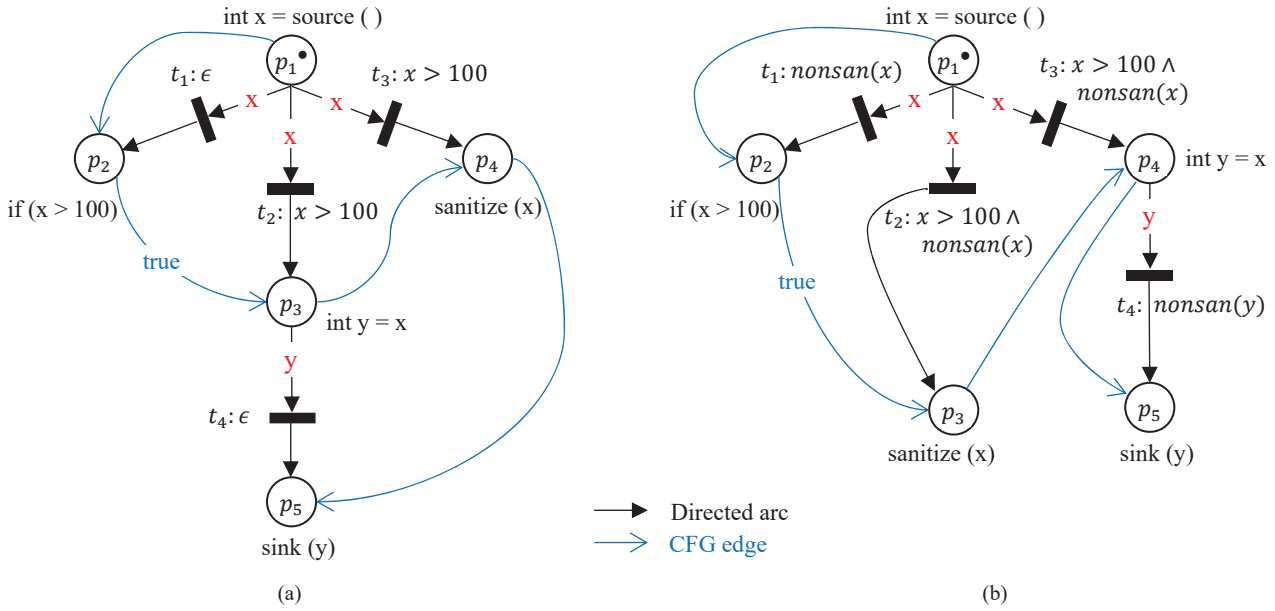


Figure 7. (a) Vulnerability net for the example given in Figure 5; (b) Vulnerability net for the code in Figure 5 that exchanges the order of S3 and S4

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 [21, 22], when we cannot determine whether a condition of a transition will be met, we would consider the worst case, i.e., the condition will be met and the token can propagate successfully. After firing the two transitions, the marking changes to $(1, 1, 1, 1, 0)$. Finally, t_4 is enabled, and its firing yields the marking $(1, 1, 1, 1, 1)$. This means the token can propagate to all places, including the sink function (i.e., p_5). Therefore, the execution result suggests the existence of a taint-style vulnerability: 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 a sanitization. For example, as discussed in Section III-B, exchanging the order of *S*3 and *S*4 in Figure 5 will eliminate the vulnerability, so we should not report an alarm in that case. Therefore, it is crucial to confirm sanitizations' existence when spotting taint-style vulnerabilities. To this end, necessary sanitization information may be accommodated in the transitions of a vulnerability net. As an example, Figure 7 (b) shows the vulnerability net for the code that exchanges the order of *S*3 and *S*4 in Figure 5.

In Figure 7 (b), each transition adds a Boolean predicate `nonsan(var)`, which will return *true* or *false* value by evaluating that whether the variable `var` is sanitized. It will return *true* if `var` is not sanitized, and *false* otherwise. To illustrate this, we simulate the execution of the net given in Figure 7 (b). The initial marking is $\mu_0 = (1, 0, 0, 0, 0)$ as we assign a token to p_1. Since `x` is not sanitized before reaching t_1, the `nonsan(x)` evaluates to *true*, which implies that t_1 is enabled, and the token will propagate to p_2. We have 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 to p_4 from p_1, `x` must have been sanitized in p_3. Consequently, the `nonsan(x)` at t_3 evaluates to *false* and t_3 is thus not enabled, preventing the token in p_3 from propagating to p_4. Furthermore, t_4 will not be enabled either since p_4 never contains a token. In summary, there is no further change in the state of the net. That is, 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). In this example we can see that taking into account the sanitization information helps us recognize the absence of vulnerabilities, thereby reducing spurious alarms.

D. Algorithm

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

The process of generating a vulnerability net is formalized in Algorithm 1. Each step can be completed in an automatic manner. At line 1, the merger between a control flow graph and a data dependence graph is the union of the two graphs (their nodes and edges). From line 2 to line 7, the merger graph is transformed 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 7 (a)) or given by the security analysts (e.g., the `nonsan(x)` in Figure 7 (b)).

The algorithm for detecting taint-style vulnerabilities in a vulnerability net is described in Algorithm 2. We start by providing a vulnerability net and specifying the sources and

sinks. A *source-sink pair* denotes a source and a sink, where data propagation from the source to the sink may cause a security flaw. The number of sources and sinks is arbitrary.

In line 1 of Algorithm 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. Note that when represented in a graph, different sources are denoted by tokens with different colors (see Section II-B).

Line 2 of Algorithm 2 states that the vulnerability net is then executed, followed by the generation of the set of final markings. The final markings are used to iteratively examine the markings of all the source-sink pairs. If the markings of a source and its sink are both 1, meaning each of them contains a token (i.e., the tainted data reach the sink), then a vulnerability is detected. The process is formalized at line 3 to line 5.

Algorithm 1: Generation of a vulnerability net

INPUT: A source program and its DDG and CFG

OUTPUT: A vulnerability net

METHOD:

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;

Algorithm 2: Vulnerabilities discovery

INPUT: A vulnerability net and a set of source-sink pairs $S = \left\{\langle p_{source_1}, p_{sink_1}\rangle, \dots, \langle p_{source_i}, p_{sink_j}\rangle, \dots, \langle p_{source_m}, p_{sink_n}\rangle\right\}$, $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$

OUTPUT: Taint-style vulnerabilities

METHOD:

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** $\langle p_{source_i}, p_{sink_j}\rangle$ in S **do**
4: **if** $\mu_f^i(p_{source_i}) = 1 \wedge \mu_f^i(p_{sink_j}) = 1$ **then**
5: report $\langle p_{source_i}, p_{sink_j}\rangle$ as a vulnerability;

IV. Case Study

In this section we illustrate the feasibility of our approach by conducting a case study.

Figure 8 is the code adapted from an abstract code fragment, which is abstracted from a real-world case [23]. For brevity, the type information of each variable is omitted in the code. The ellipses appear in line 2 and line 10 represent some code omitted. The code fragment contains two source-sink pairs, namely $\langle source1, sink1\rangle$ and $\langle source2, sink2\rangle$. 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 the Algorithm 1), as shown in Figure 9. Note that we assume the DDG can handle the aliasing issue [20, 21], thereby allowing the dependences to be generated accurately. For example, in the code given in Figure 8, 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 9 is connected to p_{11} (i.e., `sink(b.f)`).

Then we proceed to execute the net (see the Algorithm 2). Since p_5 and p_7 are distinct sources, we assign each of them a unique colored token. 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 obtain the corresponding final markings $\mu_f^1 = (0, 0, 0, 0, 1, 1, 0, 0, 0, 1)$ and $\mu_f^2 = (0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0)$ and a vulnerability $\langle p_5, p_{11}\rangle$ is reported.

Let us take a closer look at the changes in state during the execution of the vulnerability net. The token in p_5 propagates to p_6 when t_4 fires and then propagates to p_{11} when t_5 fires. In other words, the tainted data from the source p_5 can finally reach its sink p_{11}, resulting in a potential vulnerability.

Similarly, the token in p_7 propagates to p_8 when t_6 fires and to p_9 when t_7 fires. However, the token reaches p_9 will not lead to a vulnerability since p_9 (i.e., `isSecure(p)`) is not a call to a sink method. Instead, it is a call to a secure method.

Finally, we examine whether a token will propagate to p_{10}. Consider the condition of t_8, i.e., $p\,!= isTaint \wedge nonsan(p)$. The sub-condition $p\,!= isTaint$ is a sanitization of p, while another sub-condition $nonsan(p)$ states that no sanitization of p exists. Thus, the condition $p\,!= isTaint \wedge nonsan(p)$ is a contradiction and is always false. Consequently, transition t_8 will never fire, and thus no token (i.e., tainted data) will reach p_{10}. In other words,

```
 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 8. Code fragment adapted from a real-world case

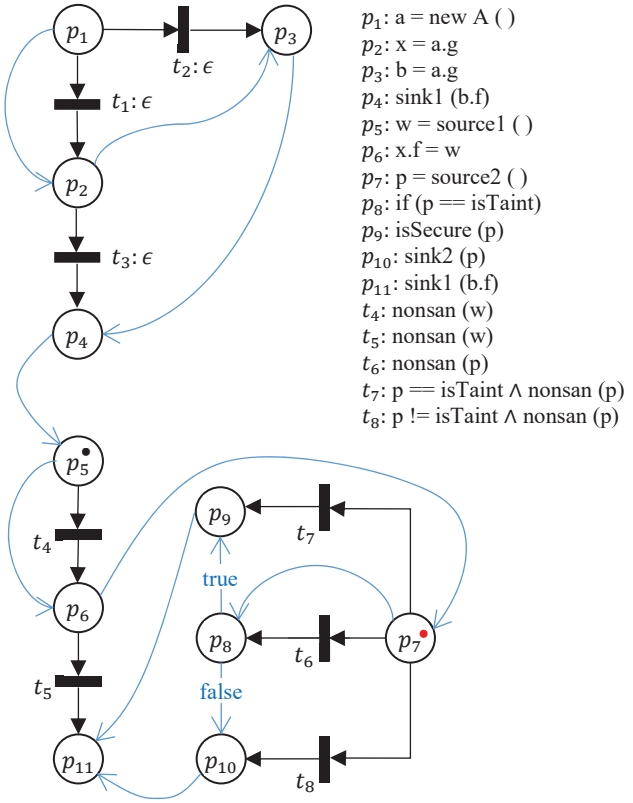


Figure 9. Vulnerability net for the code given in Figure 8

$\langle p_7, p_{10} \rangle$ is not a vulnerability due to the presence of a sanitization.

The case study shows that our approach reduces false negatives and false 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 9. Consequently, the vulnerability $\langle p_5, p_{11} \rangle$ can be correctly detected.
- Vulnerability nets are flow-sensitive. From the vulnerability net given in Figure 9 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 $\langle p_5, p_4 \rangle$ will not be reported.
- Sanitizations are recognized in vulnerability nets. Since the sanitization in t_8 of the net in Figure 9 is recognized, t_8 is proved to not enabled and thus the token in p_7 cannot propagate to p_{10}. That is, $\langle p_7, p_{10} \rangle$ will not be considered incorrectly as a vulnerability.

V. Limitations

The discovery of taint-style vulnerabilities in the case study demonstrates the feasibility of our approach. However, several limitations arise if we apply our approach to more sophisticated programs in real world. Firstly, our approach is intended only for detecting taint-style vulnerabilities and it is unclear if similar techniques can be applied to discovery of other types of vulnerabilities.

Secondly, the approach discussed in this paper involves only intraprocedural analysis since a data dependence graph or a control flow graph are built on a single procedure. Fortunately, it can be extended for interprocedural analysis if system dependence graphs [24] and interprocedural control flow graphs are introduced.

Thirdly, this paper handles only statement-level analysis, neglecting the analysis of the arguments of methods. For example, given a sink method `sink (a, b, c)` where `a` is tainted but `b` and `c` are untainted, our approach would simply treat the whole sink method as tainted rather than consider the arguments separately. In the future, we will explore the solutions to this issue.

VI. Related Work

Although the purely manual audits are rarely used in practice, some interesting work has been done in this direction. Leveson et al. [8, 9] present a method using software fault trees for safety analysis at 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, so it is unlikely to achieve automated analysis. A major similarity between this work and our work is that we both use a graphical node to explicitly represent a statement or predicate in the source code, which can assist analysts during auditing. Some other similar work (e.g., [25, 26]) is also presented in this branch of research.

In comparison to manual audits, using static analysis tools to perform vulnerability discovery is much more popular. Viega et al. [4] present ITS4, a practical tool, using basic lexical analysis to identify security vulnerabilities in C and C++ code. The idea is to extract lexical tokens from the source code and then matches the vulnerable functions. Bush et al. [27] propose PREfix to perform analysis on a set of execution paths to track information. PREfix can find security-related bugs, such as null pointer references and memory leaks, in large programs. To find complex and subtle vulnerabilities, many tools incorporate expert knowledge into the identification of vulnerabilities. For example, to detect buffer overflow vulnerabilities, Larochelle and Evans [11, 28] employ annotations to check whether the code being analyzed is consistent with a set of particular properties. Yamaguchi et al. [10] 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 vulnerabilities auditing.

Applying pointer analysis or taint analysis [29] to vulnerability discovery is an important direction in static program analysis. Dor et al. [30] first use pointer analysis techniques to detect memory errors. Livshits and Lam [31] suggest a taint-analysis method to find taint-style security vulnerabilities, such as SQL injections and cross-site scripting, in Java applications. Arzt et al. [23] present FlowDroid for Android applications, which claims to be fully context, flow, field and object-sensitive, thus reducing both

false negatives and false positives. While pointer/taint analysis approaches are valuable, most of them require whole-program availability [32], i.e., a program being analyzed must be complete.

To perform analyses on an incomplete program is another branch of research. For example, defensive programming [33, 34] is a technique for uncovering bugs during the construction of programs. Human-Machine Pair Programming [35, 36] allows the developer (i.e., the human) and the computer (i.e., the machine) to work collaboratively to discover vulnerabilities in the coding phase. Our approach supports this kind of idea since a vulnerability net can be generated when a program is still under construction.

VII. Conclusion and Future Work

In this paper, we present an approach called vulnerability nets, which can be used to detect security vulnerabilities in source code. Specifically, the approach assists in finding taint-style vulnerabilities such as buffer overflows, injection vulnerabilities, and cross-site scripting vulnerabilities. Vulnerability nets can perform analyses in an automatic manner except that the analyst needs to help identify the sanitizations for the purpose of reducing false alarms. On the other hand, vulnerability nets provide a graphical view that explicitly shows code information, which also benefits manual audits. In the paper we describe the idea in detail, including providing the definitions of vulnerability nets, the ways of building and using vulnerability nets, plus their algorithms. To demonstrate the feasibility, we also present a case study on an example adapted from a real-world case, yet there lacks a discussion of whether our approach is applicable to large programs.

Despite the progress we have made in this work, there remains a significant amount of work to be done. A number of unsolved issues listed in the following are driving our future research.

- Discussion of interprocedural analysis.
- Discussion of distinguishing the tainted values between arguments in a method/function.
- More practical evaluation of the proposed approach.
- Extension of the proposed approach to handle incomplete programs.

Acknowledgments

This work was supported by JST SPRING, Grant Number JPMJSP2132.

References

[1] C. Easttom, Computer security fundamentals. Pearson IT Certification, 2019.

[2] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11-33, October 2004.

[3] K. Goseva-Popstojanova and J. Tyo, "Experience report: security vulnerability profiles of mission critical software: empirical analysis of security related bug reports," IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp. 152-163, October 2017.

[4] J. Viega, J.-T. Bloch, Y. Kohno, and G. McGraw, "ITS4: A static vulnerability scanner for C and C++ code," Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00), pp. 257-267, December 2000.

[5] J. R. Larus et al., "Righting software," IEEE Software, vol. 21, no. 3, 2004, pp. 92-100.

[6] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," Proceedings DARPA Information Survivability Conference and Exposition, DISCEX'00, vol. 2, pp. 119-129, January 2000.

[7] S. Heelan, "Vulnerability detection systems: Think cyborg, not robot," IEEE Security & Privacy, vol. 9, no. 3, 2011, pp. 74-77.

[8] N. G. Leveson and P. R. Harvey, "Analyzing software safety," IEEE Transactions on Software Engineering, no. 5, 1983, pp. 569-579.

[9] 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, 1991, pp. 48-59.

[10] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," IEEE Symposium on Security and Privacy, pp. 590-604, May 2014.

[11] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," 10th USENIX Security Symposium, pp. 177-190, August 2001.

[12] J. Vanegue and S. K. Lahiri, "Towards practical reactive security audit using extended static checkers," IEEE Symposium on Security and Privacy, pp. 33-47, May 2013.

[13] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 97-106, October 2004.

[14] W. G. Halfond, J. Viegas, and A. Orso, "A classification of SQL-injection attacks and countermeasures," Proceedings of the IEEE International Symposium on Secure Software Engineering, March 2006.

[15] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 372-382, January 2006.

[16] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," IEEE Symposium on Security and Privacy (S&P'06), pp. 258-263, May 2006.

[17] 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, 2017, pp. 512-530.

[18] J. L. Peterson, Petri net theory and the modeling of systems. Prentice-Hall, 1981.

[19] N. G. Leveson and J. L. Stolzy, "Safety analysis using Petri nets," IEEE Transactions on Software Engineering, no. 3, 1987, pp. 386-397.

[20] 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, 1987, pp. 319-349.

[21] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: principles, techniques, & tools. Pearson Education India, 2007.

[22] F. Nielson, H. R. Nielson, and C. Hankin, Principles of program analysis. Springer Science & Business Media, 2004.

[23] S. Arzt et al., "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014), pp. 259-269, June 2014.

[24] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 12, no. 1, 1990, pp. 26-60.

[25] 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," Computer Safety, Reliability and Security, 18th International Conference, SAFECOMP'99, pp. 226-238, September 1999.

[26] 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, 2005, pp. 215-228.

[27] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," Software: Practice and Experience, vol. 30, no. 7, 2000, pp. 775-802.

[28] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," IEEE Software, vol. 19, no. 1, 2002, pp. 42-51.

[29] N. Grech and Y. Smaragdakis, "P/taint: Unified points-to and taint analysis," Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, 2017, pp. 1-28.

[30] N. Dor, M. Rodeh, and M. Sagiv, "Detecting memory errors via static pointer analysis (preliminary experience)," Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, pp. 27-34, June 1998.

[31] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," Proceedings of the 14th USENIX Security Symposium, pp. 271-286, July-August 2005.

[32] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," Foundations and Trends in Programming Languages, vol. 2, no. 1, 2015, pp. 1-69.

[33] F. Schindler, "Coping with security in programming," Acta Polytechnica Hungarica, vol. 3, no. 2, 2006, pp. 65-72.

[34] J. K. Teto, R. Bearden, and D. C.-T. Lo, "The impact of defensive programming on i/o cybersecurity attacks," Proceedings of the 2017 ACM Southeast Regional Conference, pp. 102-111, April 2017.

[35] S. Liu, "Software construction monitoring and predicting for Human-Machine Pair Programming," 8th International Workshop, SOFL+MSVL 2018, pp. 3-20, November 2018.

[36] 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, 2022, pp. 1129-1138.