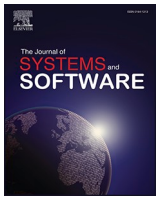
The Journal of Systems and Software 208 (2024) 111902

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



Detecting security vulnerabilities with vulnerability nets

Pingyan Wang, Shaoying Liu *, Ai Liu, Wen Jiang

Hiroshima University, Higashi Hiroshima, Japan

ARTICLE INFO

Editor: Prof W. Eric Wong

Keywords:
Vulnerability
Security
Static analysis
Manual audits
Petri nets

ABSTRACT

Detecting security vulnerabilities is a crucial part in secure software development. Many static analysis tools have proved to be effective in finding vulnerabilities, but generally there are some complex and subtle vulnerabilities that can escape from 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 approaches incorporate the auditor's expertise into a static analysis tool during vulnerability discovery. Following this strategy, this paper presents a representation of source code called a vulnerability net, which is a special Petri net that integrates with data dependence graphs and control flow graphs. The combined representation can facilitate the detection of taint-style vulnerabilities such as buffer overflows and injection vulnerabilities. We test the proposed approach on Securibench Micro and demonstrate that it has the capability to identify a variety of vulnerabilities while keeping the rates of false negatives and positives low.

1. Introduction

In software-based systems, security vulnerabilities, also known as security-related bugs, can lead to malicious attacks, which in turn may cause security failures (Avizienis et al., 2004). Thus, a natural strategy to enhance software security is to find (and then fix) vulnerabilities in code. Indeed, vulnerability discovery is so important for security that it has been discussed in the literature for decades (Chess and West, 2007).

While security vulnerabilities can be detected in various ways, the vast majority of them are discovered by static code analysis (Goseva--Popstojanova and Tyo, 2017). The core idea behind static code analysis is to analyze a program without executing it. Many static analysis tools (e.g., (SonarQube, 2023; Coverity Static Application Security Testing (SAST), 2023)) have been developed and adopted 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 (Cowan et al., 2000), 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 (Heelan, 2011).

Manual audits are a complementary (not alternative) method to using automated tools. In the process of auditing, an analyst manually examines the given code, based on his/her expertise, to find vulnerabilities that escape from detection of tools. To aid analysts in auditing manually, some researchers analyze source code using techniques such as fault trees (Leveson and Harvey, 1983; Leveson et al., 1991), in which certain crucial information of code is made explicit. However, manual audits are tedious, error prone, and costly, so it is normally impractical to manually audit an entire program, though auditing a few critical code fragments is possible.

To benefit from both the static analysis tools and manual audits, work in this area (e.g., (Yamaguchi et al., 2014; Larochelle and Evans, 2001)) has considered incorporating the analyst's security knowledge into a tool during the detection process. The knowledge (such as annotations (Vanegue and Lahiri, 2013)) provided by security auditors can guide the detection for vulnerabilities. To make contributions in this branch of research, this paper presents a novel representation of source code, called a *vulnerability net*, which is in the form of a Petri net structure.

Petri nets are a mathematical representation widely used for modeling and analyzing various types of systems, ranging from chemical systems to computer systems (Peterson, 1981). A program, the analysis target of this paper, can be viewed as a system, where the statements within the program correspond to the components of the system. Therefore, Petri nets have the potential to model a program. To tailor Petri nets specifically for program analysis, we propose vulnerability

* Corresponding author.
E-mail address: sliu@hiroshima-u.ac.jp (S. Liu).

https://doi.org/10.1016/j.jss.2023.111902
Received 15 February 2023; Received in revised form 16 October 2023; Accepted 8 November 2023
Available online 9 November 2023

nets in this paper. While preserving the core principles of standard Petri nets, vulnerability nets introduce some novel features and incorporate data dependence graphs and control flow graphs, aiming at representing source code and modeling the flow of information within it. The combination explicitly describes the key information of a given program and provides a good view for analysts to perform auditing. Like standard Petri nets, vulnerability nets are executable, thus allowing analysts to conveniently track the data of interest to examine if any path execution may cause a security issue. By supplying a vulnerability net with necessary taint information (such as *sources* and *sinks*), the net can be executed to identify potential taint-style vulnerabilities, which include various types of security flaws, such as buffer overflows (Cowan et al., 2000; Zitser et al., 2004), injection vulnerabilities (Halfond et al., 2006; Su and Wassermann, 2006), and cross-site scripting (XSS) vulnerabilities (Jovanovic et al., 2006; Gupta and Gupta, 2017).

To show the feasibility of our approach, we test it on a security benchmark, (Securibench Micro, 2023), and present a case study in analyzing an example adapted from a real-world case. The evaluation results show that our approach has the capability to identify a variety of vulnerabilities while keeping the rates of false negatives and positives low.

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 identifying the presence of vulnerabilities. To formalize this idea, algorithms are also described in this paper.
- Performing evaluation of the proposed approach by conducting experiments on Securibench Micro and presenting a case study.

The remainder of this paper is organized as follows. Section 2 presents general definitions for vulnerability nets. Section 3 describes the use of vulnerability nets for identifying taint-style vulnerabilities and Section 4 evaluates our approach by conducting experiments and a case study. Section 5 discusses limitations and the possible extension of the proposed approach. Section 6 reviews related work and the final section concludes this work.

2. Definition of vulnerability nets

For comparison purpose, in this section we start with the definition of traditional Petri nets. Then, vulnerability nets are formally defined. Most of the notation and terminologies in our discussions are taken or adapted from the work in (Peterson, 1981; Leveson and Stolzy, 1987).

2.1. Petri nets

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

Definition 1. A *Petri net* M is a five-tuple, $M = (P, T, I, O, \mu)$, where

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

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

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

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

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

2.2. Vulnerability nets

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

Definition 2. A *vulnerability net* Φ is a five-tuple, $\Phi = (P, T, I, O, \mu)$, where

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

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

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

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

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

The marking function μ implies that each place holds a number, 0 or 1. By convention, the element that a place holds is called a *token*. That is, each place in a vulnerability net holds zero or one token. The reason why we restrict the number of tokens in a place to only 0 or 1 is that 0 represents the absence and 1 represents the presence of a specific value in our vulnerability analysis. (A detailed discussion of the representations appears in Section 3.) The marking function μ shows the number and distribution of tokens in a vulnerability net. We use $\mu(p_1)$, $\mu(p_2)\ldots$ to represent each specific marking for places in a net. For example, $\mu(p_i)$ is the marking for the place p_i, and $\mu(p_i) = 1$ means p_i holds one token. By contrast, the marking of an entire vulnerability net is denoted by μ, where $\mu = (\mu(p_1), \mu(p_2), \ldots, \mu(p_n))$. In the rest of this paper, a marking refers to a marking of an entire net, unless stated otherwise. When a vulnerability net executes, its marking may change as the number of tokens in each place may change. The conditional expressions in guarded transitions are the Boolean expressions used to constrain the change in the number of tokens.

An example of a vulnerability net is shown in Fig. 1 (a). The net is composed of five places and three guarded transitions (henceforth transitions). The initial marking of the net $\mu_0 = (1, 0, 0, 1, 0)$ states that the place p_1 and p_4 each initially contain a token, while other places do not contain any. The links between places and transitions are revealed by the input and output functions. For example, $I(t_1) = \{p_1, p_2\}$ indicates that p_1 and p_2 are the input places of the transition t_1; $O(t_1) = \{p_3\}$ indicates that p_3 is the output place of the transition t_1. A transition

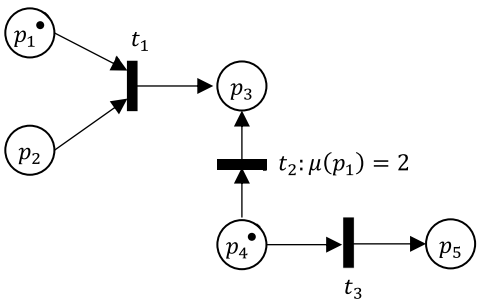
$P = \{p_1, p_2, p_3, p_4, p_5\}$

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

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

$I(t_1) = \{p_1, p_2\}$	$O(t_1) = \{p_3\}$
$I(t_2) = \{p_4\}$	$O(t_2) = \{p_3\}$
$I(t_3) = \{p_4\}$	$O(t_3) = \{p_5\}$

(a)



(b)

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

(e.g., t_2) with an expression indicates that its Boolean value depends on the evaluation of the expression (e.g., $\mu(p_1) = 2$ in t_2), while other transitions (such as t_1 and t_3) without any explicit expression are assigned the value *true* as a default.

We can also use a graphical representation to represent a vulnerability net for a better readability. A vulnerability net graph for Fig. 1 (a) is shown in Fig. 1 (b), where places, transitions, and tokens are denoted by circles, bars, and small dots, respectively. Directed arcs are used to connect places with transitions.

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

Definition 3. A transition $t_j \in T$ in a vulnerability net $\Phi = (P, T, I, O, \mu)$ is *enabled* if

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

For example, the transition t_3 in Fig. 1 is enabled according to the definition. However, t_1 is not enabled since p_2, one of the t_1's input places, does not contain a token. In the case of t_2, we notice that its conditional expression $\mu(p_1) = 2$ always evaluates to false because the place p_1 will never hold two tokens (in fact, none of the places in a vulnerability net can hold two or more tokens, according to the function μ in Definition 2), so t_2 is not enabled.

When a transition fires during the execution of a vulnerability net, tokens *propagate* from its input places to output places, i.e., new tokens are assigned to output places while tokens in input places are retained. We use *state* to describe the change. Every state of a vulnerability net is defined by a marking; for example, the initial state is defined by initial marking μ_0. The state space of a vulnerability net with n places is the set of all markings, i.e., B^n, where B is the set of $\{0, 1\}$. Definition 4 defines a *next-state function* for calculating how a state can change after firing a transition.

Definition 4. 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

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

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

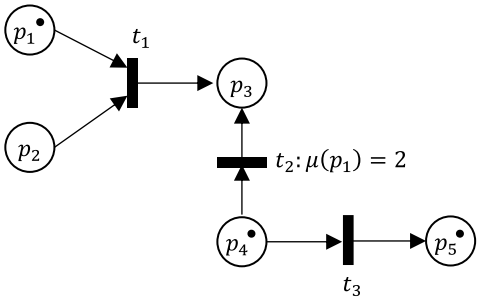


Fig. 2. The next state of Fig. 1.

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

2.3. Vulnerability nets with colored tokens

Colored tokens are used to augment the expressiveness of a vulnerability net. A vulnerability net with colored tokens also meets the definitions given in Section 2.1 as long as we consider each kind of colored token separately. We formally give a definition as follows.

Definition 5. A *vulnerability net with colored tokens* is a vulnerability net structure $C = (P, T, I, O, G)$, where G is a finite set of markings $G = \{\mu^1, \mu^2, \ldots, \mu^k\}$, $k \geq 0$, where each μ^i is the marking for a kind of token with a unique color in a graphical representation.

Fig. 4 shows an example. Initially, while p_2 and p_4 each contain one kind of token, p_1 holds two. During the execution of the net, the various kinds of tokens propagate independently of each other. Therefore, all kinds of colored tokens can propagate to p_3, with the exception of the green token as t_1 requires two green inputs but only one (i.e., the green token in p_1) is available. The initial state of the vulnerability net can be denoted by $\mu_0^1 = (0, 0, 0, 1)$, $\mu_0^2 = (1, 0, 0, 0)$, and $\mu_0^3 = (1, 1, 0, 0)$, where μ_0^1, μ_0^2, and μ_0^3 represent the initial markings for black, green, and red tokens, respectively. The finial state of the net is $\mu_f^1 = (0, 0, 1, 1)$, $\mu_f^2 = (1, 0, 0, 0)$, and $\mu_f^3 = (1, 1, 1, 0)$.

3. Vulnerability discovery

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

The class of security weaknesses this paper focuses on is taint-style vulnerabilities, including several critical vulnerabilities such as buffer overflows, injection vulnerabilities, and cross-site scripting vulnerabilities. In taint analysis, a *tainted value* is derived from external, untrusted input such as command-line arguments or results returned from function calls. While a *source* is the original program location (such as a function or method) that accepts tainted values, a *sink* is the program location that should not receive tainted values. A security weakness may exist

(a) Execution of a standard Petri net

(b) Execution of a vulnerability net

Fig. 3. Example illustrating a distinction between vulnerability nets and standard Petri nets.

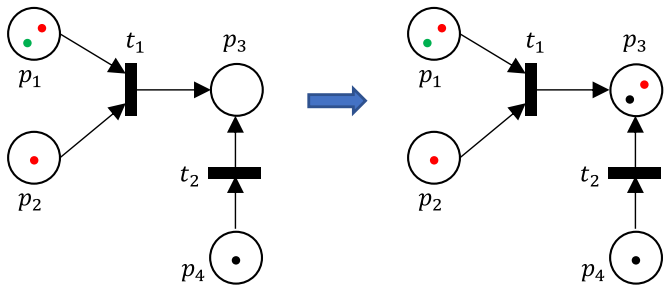


Fig. 4. Execution of a vulnerability net with colored tokens.

when a tainted value from a source reaches a sink through an execution path. *Sanitization* is a step to remove the taint from a value, thereby eliminating the potential security risk. To perform sanitization, we could either replace the tainted value by an untainted value or terminate the path of execution when a tainted value is detected.

For illustration, consider a simple taint-style vulnerability shown in Fig. 5. In this Java code, `x` accepts a tainted value returned from a source function and passes it to `y` when the if-condition is met. Then `x` is sanitized, and finally a sink function receiving `y` as an argument is called.

3.1. Data dependence graphs

Data dependence graphs (DDGs) are a program representation that can explicitly represent data dependences among statements and predicates (Ferrante et al., 1987). A data dependence is present when two statements cannot be switched without changing any variable's value. For example, in Fig. 5, $S3$ depends on $S1$ as $S1$ must be executed first in order for `x`'s value to be properly used in $S3$. Fig. 6 (a) shows the DDG for the example code in Fig. 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, Fig. 6 (a) explicitly shows that the tainted value originated from the source will eventually be passed to the sink, resulting in a leak.

3.2. Control flow graphs

Control flow graphs (CFGs) are another commonly used representation that can explicitly show the execution order of statements and the flow of control determined by conditional expressions (Aho et al., 2007). In a CFG, statements and predicates are denoted by nodes, and the flow of control is indicated by directed edges. Fig. 6 (b) shows the CFG for the code sample in Fig. 5.

The use of CFGs in our approach is important, since DDGs alone often do not suffice to ensure the existence of a vulnerability. For example, if we exchange the order of $S3$ and $S4$ in Fig. 5 (as shown in Fig. 7), while the data dependence graph may remain unchanged (depending on what the sanitization is), the code is no longer flawed because `x`'s value has been sanitized before being passed to `y` and `sink(y)`. That is, using a DDG alone in this case could yield a spurious alarm. To remedy the situation, we leverage a CFG as it can make explicit whether the sanitization for a tainted value is already executed prior to using the value. Furthermore, control-flow information is also essential in the detection of many other types of vulnerabilities, such as use-after-free vulnerabilities.

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

Fig. 5. Example of a taint-style vulnerability in Java.

3.3. Building vulnerability nets

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

As an example, consider a simple Java code fragment "`a = source1(); b = source2(); sink(a, b);`". For this code, we can use places p_1, p_2, and p_3 to denote each statement, respectively. Also, we use two distinct colored tokens (let us say, a black token and a red token) to represent the values of `a` and `b`, respectively. Initially, p_1 holds one black token while p_2 and p_3 do not hold any black token. The initial marking for the black token is $\mu_0^{black} = (1, 0, 0)$. Similarly, the initial marking for the red token is $\mu_0^{red} = (0, 1, 0)$. After `a` is passed into `sink(a, b)`, which indicates that the black token propagates from p_1 to p_3, the marking for the black token changes to $\mu_1^{black} = (1, 0, 1)$; similarly, the marking for the red token changes to $\mu_1^{red} = (0, 1, 1)$ after `b` is passed into `sink(a, b)`. Whether a token can propagate from one place to another is dependent on the corresponding transition(s). We provide more details below after DDGs and CFGs are introduced.

Since DDGs and CFGs can provide essential information for security analysis, we incorporate them into vulnerability nets when representing source code. For brevity, the combination is also referred to as a vulnerability net. To build such a net, a DDG serves as a skeleton on which we gradually add CFG information (CFG edges and conditional expressions). Specifically, the DDG nodes are replaced by places, the DDG edges are replaced by transitions, and the conditional expressions from the CFG are assigned to corresponding transitions. To clarify the idea, consider again the code example shown in Fig. 5. We combine its DDG and CFG, both given in Fig. 6, and produce a vulnerability net, as indicated in Fig. 8 (a).

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

First, we can see that t_1 is immediately enabled, resulting in the next state $\mu_1 = \delta(\mu_0, t_1) = (1, 1, 0, 0, 0)$. Transitions t_2 and t_3 are conditionally enabled as the condition $x > 100$ may hold. To make conservative (or safe) approximations (Aho et al., 2007, Nielson et al., 2004), when we cannot determine whether a condition of a transition will be met, we will consider the worst case, i.e., the condition will be met, and the token can be propagated through the transition successfully. After firing the two transitions, the marking changes to $(1, 1, 1, 1, 0)$. Finally, t_4 is enabled and will fire to yield the marking $(1, 1, 1, 1, 1)$, which indicates that the token can propagate across all places, including the sink function (i.e., p_5). Therefore, the final execution result suggests the existence of a taint-style vulnerability since the tainted value from the source can reach the sink.

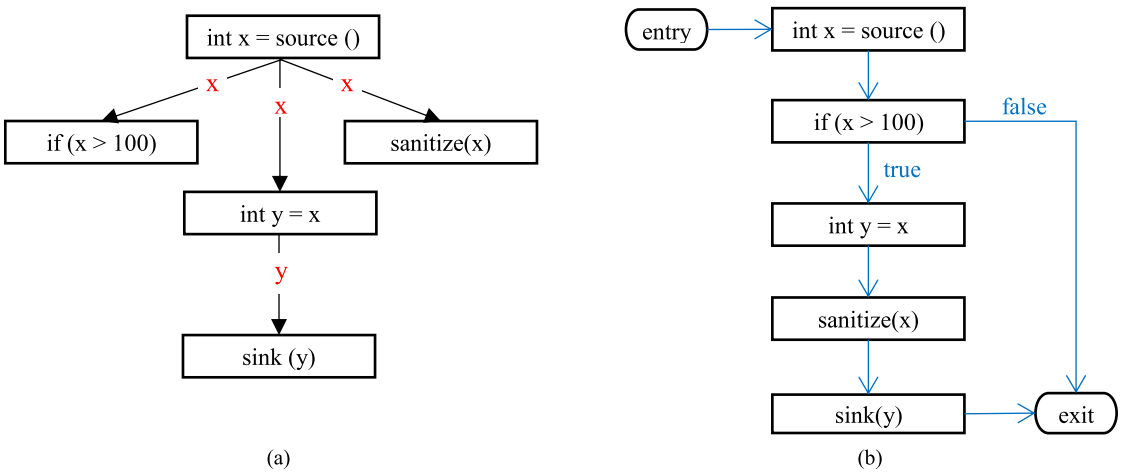


Fig. 6. (a) DDG for the code in Fig. 5; (b) CFG for the code in Fig. 5.

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

Fig. 7. Example code adapted from Fig. 5.

To reduce the risk of taint-style vulnerabilities, programmers may sanitize the tainted values before using them. In this situation, false positives may arise if we do not realize the presence of sanitization. For example, as discussed in Section 3.2, exchanging the order of $S3$ and $S4$ in Fig. 5 will eliminate the vulnerability, so we should not report an alarm in that case. Therefore, it is crucial to perform sanitization identification when spotting taint-style vulnerabilities. To this end, necessary sanitization information can be accommodated in the transitions of a vulnerability net. As an example, Fig. 8 (b) shows the vulnerability net with sanitization identification for the code in Fig. 7. (Let us assume the sanitization used in Fig. 7 is to terminate the path of execution.)

In Fig. 8 (b), each transition adds a Boolean predicate `nonsan(var)` for the corresponding variable `var`, which will return a *true* or *false* value by evaluating whether the variable `var` is sanitized. It returns true if `var` is not sanitized and false otherwise. It is worth noting that such predicates are added and evaluated by hand in this paper, but they can be done in an automatic manner in practice as well. For example, since sanitization within a program can be identified through taint-specification mining techniques (e.g., (Clapp et al., 2015; Chibotaru et al., 2019)), a predicate `nonsan(var)` can be automatically evaluated to false if the corresponding variable `var` is sanitized; otherwise, it is evaluated to true by default.

To illustrate how the sanitization can help, we simulate the execution of the net in Fig. 8 (b) as follows. The initial marking is $\mu_0 = (1, 0, 0, 0, 0)$ as we assign a token to p_1. Since `x` is not sanitized at t_1, the `nonsan(x)` evaluates to true, making t_1 enabled and thus allowing the token to be propagated to p_2. The propagation leads to the next state $\mu_1 = \delta(\mu_0, t_1) = (1, 1, 0, 0, 0)$. Similarly, t_2 could be enabled when $x > 100$, so we have $\mu_2 = (1, 1, 1, 0, 0)$. However, the situation of t_3 is different. The CFG edges in the net show that prior to the propagation path p_1

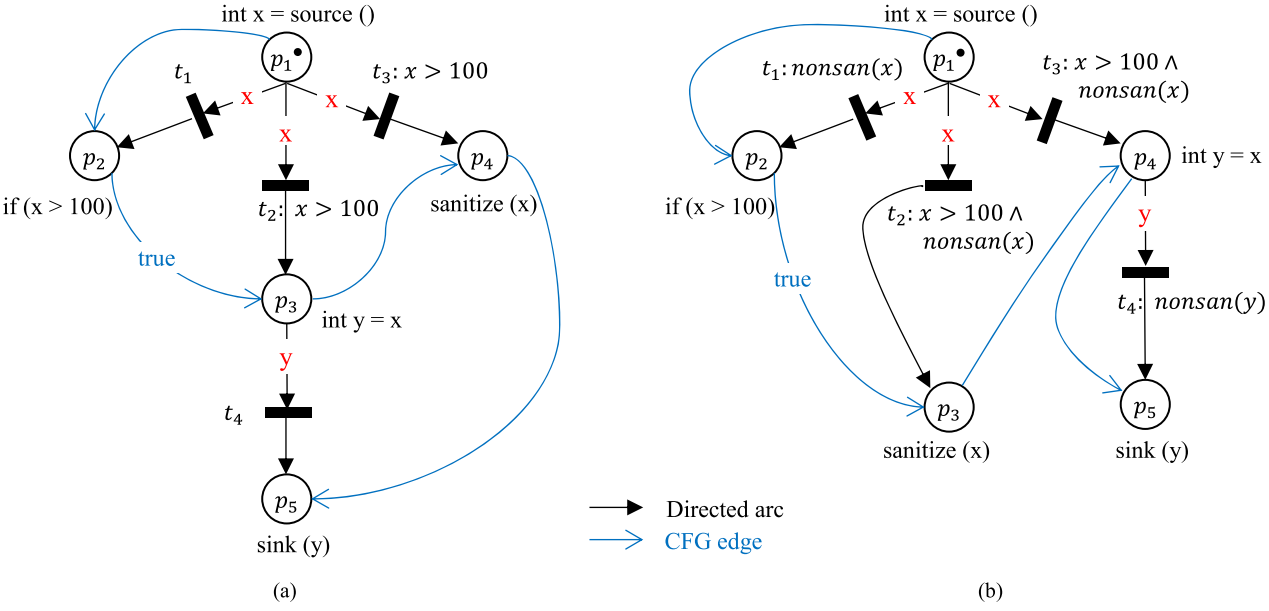


Fig. 8. (a) Vulnerability net for the code in Fig. 5; (b) Vulnerability net with sanitization identification for the code in Fig. 7.

through p_4, `x` must have been sanitized in p_3. Consequently, the `nonsan(x)` at t_3 evaluates to false, making t_3 not enabled and thus preventing the token in p_1 from propagating forward to p_4. Furthermore, t_4 cannot be enabled either as p_4 never receives a token. In summary, there is no further change in the state, i.e., the final state of the net is $\mu_f = \mu_2 = (1, 1, 1, 0, 0)$, which implies that no taint-style vulnerability is present in this code as no tainted value reaches the sink function (i.e., p_5). As a result, no false alarms will be reported in this net.

3.4. Algorithms

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

The process of generating a vulnerability net is formalized in Algorithm 1. Each step can be completed in an automatic manner, though security analysts may manually add information to a net for performance enhancement. At line 1, the merger between a control flow graph and a data dependence graph is the union of their nodes and edges. Lines 2 through 7 transform the merger graph to a vulnerability net. The assignments of conditional expressions to transitions are based on the Boolean expressions given in the control flow graph (e.g., the $x > 100$ in Fig. 8 (a)) or given by security analysts (e.g., the `nonsan(x)` in Fig. 8 (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* is a two-tuple $(source, sink)$, where the source through the sink is a propagation path that may cause a security flaw. The quantities of sources and sinks in a vulnerability net are arbitrary.

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 2.3).

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

4. Evaluation

In this section, we present an evaluation of our approach by conducting a set of small-scale experiments on Securibench Micro (Securibench Micro, 2023), followed by a comparison with SonarQube (SonarQube, 2023). Moreover, we present a case study to illustrate the use of our approach in a real-world case.

4.1. Experiments

Preliminaries: Securibench Micro (Securibench Micro, 2023) is comprised of a series of small Java test cases, which encompass a variety of vulnerabilities such as SQL injection vulnerabilities and XSS vulnerabilities. The code for our experiments appears in three directories of Securibench Micro, namely `basic` (containing various basic vulnerabilities), `aliasing` (containing aliasing-related vulnerabilities), and `inter` (containing vulnerabilities in interprocedural cases), with a total of 62 source programs. The vulnerability types within these programs are categorized and summarized in Table 1. Note that in these programs, many XSS vulnerabilities are accompanied by information disclosure vulnerabilities, and both types of vulnerabilities are caused by the same source-sink propagation paths. To avoid duplicate counting, the information disclosure vulnerabilities have been omitted from our analysis.

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

We also manually predefined a list of sources, sinks, source-sink pairs, sanitizers, and entry points for the test cases. (When analyzing large-scale programs, such taint information can be automatically obtained using taint-specification mining techniques (Clapp et al., 2015; Chibotaru et al., 2019).) For example, one typical source is the method `getParameter` as it accepts attacker-supplied input, while `println` is a typical sink as it might process tainted data and produce unexpected results. The (`getParameter`, `println`) is a typical source-sink pair in the test cases of our experiments.

Implementation: In principle, a vulnerability net itself is executable, but since an automated tool for our approach has not yet been available, we partially implemented our approach as a Datalog program. Specifically, Fig. 9 shows a Soufflé-style Datalog implementation of Algorithm 2. In the program, lines 2 and 7 accept vulnerability net edges and

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 ($i \in 1, 2, \ldots$);
4 **for each** DDG edge e **do**
5 replace e by a transition t_j ($j \in 1, 2, \ldots$) 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 $S = \{(p_{source_1}, p_{sink_1}), \ldots, (p_{source_i}, p_{sink_j}), \ldots, (p_{source_m}, p_{sink_n})\}$, $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$**OUTPUT**: Taint-style vulnerabilities**METHOD**:

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

Table 1
Vulnerability types within the programs for our experiments.

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

source-sink pairs as input, respectively, which in turn serve as facts for computation under certain rules (lines 4, 5, and 9). Line 4 states that a token can be propagated from the place `x` to `y` if there is a vulnerability net edge connecting from `x` to `y` and the Boolean expression in the transition evaluates as true, denoted by "1" in `VNEdge(x, y, 1)`. In line 5, the token is propagated across the entire net as long as two places in the net satisfy the condition in line 4. Finally, line 9 computes insecure source-sink pairs, which are output as discovered vulnerabilities at line 10. (Note that the concepts of Datalog are beyond the scope of this paper; see (Aho et al., 2007, Soufflé: A Datalog Synthesis Tool for Static Analysis, 2023) for discussions about Datalog or Soufflé.)

4.2. Results

After examining the test cases, we summarize the results in Table 2.

The 62 source programs contain a total of 89 vulnerabilities. As mentioned in Section 4.1, we did not analyze *Inter13* and *Inter14* due to the failure to inline procedures in them, so we missed the discovery of two vulnerabilities. For the remaining 60 programs, our approach has correctly discovered 87 vulnerabilities without any false negatives or positives. That is, our approach reported 97.8% (87/89) of vulnerabilities in the test cases.

The results show that our approach can detect various taint-style vulnerabilities in source code, though the intraprocedural analysis hurts performance to some extent. We discuss a solution to this issue in Section 5.

4.3. Comparison with sonarqube

As a popular static analysis tool, SonarQube (4) can serve as a good baseline for measuring the performance of our approach. The version of SonarQube we used is the Community Edition 10.2.1.78527. We first converted the target programs from Securibench Micro into a Maven project for the purpose of analysis and then performed analysis through SonarQube. After an analysis report was generated, we examined if there were any false positives or negatives. Table 3 summarizes the analysis results.

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

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

Table 2
Experiment results.

Programs	Correct warnings	Missed warnings	False warnings
Inter13 and *Inter14*	0	2	0
Remaining 60 programs	87	0	0

Table 3
Comparison between SonarQube and our approach.

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

4.4. Case study

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

Fig. 10 is the code adapted from an abstract code fragment, which is abstracted from a real-world case (Arzt et al., 2014). For brevity, the type information of each variable is omitted in the code. The ellipses that appear in lines 2 and 10 represent some code omitted. The code fragment contains two source-sink pairs, namely $(source1, sink1)$ and $(source2, sink2)$. To identify the potential taint-style vulnerabilities that exist in this code, we want to examine whether a tainted value from a source (e.g., *source*1) may reach its paired sink (e.g., *sink*1).

Our analysis begins by generating the vulnerability net (see Algorithm 1), as shown in Fig. 11. Note that we assume the DDG can resolve aliasing (Ferrante et al., 1987; Aho et al., 2007), thereby allowing the

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

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

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

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

10  .output BugFound
```

Fig. 9. Datalog program implementing Algorithm 2.

```
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  }
```

Fig. 10. Code fragment adapted from a real-world case.

dependences to be generated accurately. For example, in the code given in Fig. 10, since the variable `b` (line 4) and `x` (line 5) are aliases of each other, `b.f` and `x.f` are aliased. Accordingly, p_6 (i.e., `x.f = w`) in Fig. 11 is connected to p_{11} (i.e., `sink(b.f)`).

Then we proceed to analyze the net. As described in Fig. 9, the input for our analysis includes vulnerability net edges and source-sink pairs. The vulnerability net edges of Fig. 11 are represented as follows.

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

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

We next input the source-sink pairs:

p5	p4
p5	p11
p7	p10

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

p5	p11

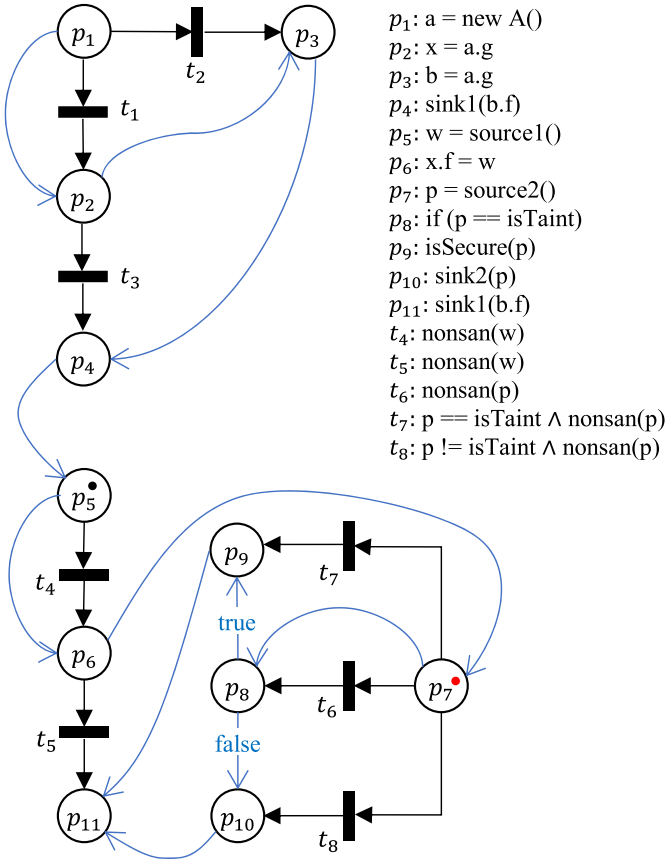


Fig. 11. Vulnerability net for the code in Fig. 10.

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

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

- Vulnerability nets support aliasing analysis. The use of DDG enables our approach to handle aliasing issues, so we can recognize that `b.f` and `x.f` are aliased in Fig. 11. Consequently, the vulnerability (p_5, p_{11}) can be correctly detected.
- Vulnerability nets are flow-sensitive. From the vulnerability net in Fig. 11 we can see that the token in p_5 may propagate to p_{11}, but never propagate to p_4 even though p_4 and p_{11} represent an identical statement (i.e., `sink1(b.f)`). Thus, a spurious vulnerability (p_5, p_4) will not be reported.
- Sanitization is recognized in vulnerability nets. Since the sanitization in t_8 of the net in Fig. 11 is recognized, t_8 is proved to not enabled and thus the token in p_7 cannot propagate to p_{10}. That is, (p_7, p_{10}) will not be incorrectly considered as a vulnerability.

5. Limitations

Our evaluation demonstrates the capability of our approach for uncovering taint-style vulnerabilities in source code. However, several limitations arise if we apply our approach to more sophisticated programs in the real world. Firstly, our approach is intended only for detecting taint-style vulnerabilities, so it is unclear if similar ideas can be applied to discovery of other types of vulnerabilities.

Secondly, the approach discussed in this paper involves only intraprocedural analysis, since a data dependence graph or a control flow graph are built on a single procedure. As mentioned in Section 4, although procedure inlining can make our intraprocedural analysis, in effect, interprocedural analysis, it has certain inherent limitations. Fortunately, our approach can be extended for interprocedural analysis if system dependence graphs (Horwitz et al., 1990) and interprocedural control flow graphs are introduced.

Thirdly, our approach omits the discussion of input validation, just as many other taint analysis approaches do (e.g., (Arzt et al., 2014)). Input validation is often adopted as a complement to sanitization since sanitization alone may not be sufficient to handle external input comprehensively. False alarms can arise when input validation is not successfully recognized. There are many approaches (e.g., (Medeiros et al., 2016, Figueiredo et al., 2020, Medeiros et al., 2022)) in the literature that have shown how to identify input validation and how this can greatly reduce false positives during taint analysis.

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

- Visualization: This component defines graphical representations of the elements in a vulnerability net. For example, places are represented by circles. The component should display a vulnerability net in a structured manner.

- Transformation: This component defines how the statements in a program are mapped to vulnerability net constructs. Techniques such as *rule-based mapping* can be used to guide the transformation process and ensure that the generated vulnerability net can accurately represent the program's characteristics.
- Simulation: This component handles the firing of transitions based on the current markings and firing rules defined in the vulnerability net. It should allow the user to specify the initial marking, track token movements, and observe the changes in the marking. To visualize the simulation process, the Simulation component should be integrated with the Visualization component.
- Analysis: This component performs vulnerability analysis based on the simulation data from the Simulation component. It may include various analyses, such as reachability analysis and taint analysis. Analysis results are generated in this component.

6. Related work

Although purely manual audits are rarely used in practice, some interesting work has been done in this direction. Leveson and Harvey (1983), Leveson et al. (1991) present a method using software fault trees to perform safety analysis at the source code level. The analysis is expressed in a tree form, which starts with determining a fault of interest, followed by a backward analysis to find the set of possible causes. This kind of method relies heavily on the analyst's expertise, making it unlikely to achieve full automation. Similar work (e.g., (Min et al., 1999, Oh et al., 2005)) has also been presented in this branch of research. A major similarity between these approaches and ours is that we both use a graphical node to explicitly represent a statement or predicate in source code, which can assist analysts in auditing the code manually. However, our approach can not only facilitate manual audits, but also support automated static analysis.

In comparison to manual audits, automated static analysis is much more popular for vulnerability discovery. We discuss the work most related to ours in the following. Arzt et al. (2014) present FlowDroid, a taint analysis approach for Android applications, which claims to be fully context, flow, field and object-sensitive, thus reducing both false negatives and false positives. There are also other approaches performing taint analysis on Android, such as AmanDroid (Wei et al., 2018) and DroidSafe (Gordon et al., 2015). To further detect complex and subtle vulnerabilities, some approaches incorporate expert knowledge into the process of vulnerability discovery. Livshits and Lam (2005) suggest a taint analysis method using user-provided specifications to find security vulnerabilities in Java applications.Yamaguchi et al. (2014) merge abstract syntax tree, control flow graphs and program dependence graphs into a joint data structure, in which analysts craft certain rules, known as *traversals*, to facilitate vulnerability auditing. While all these approaches and ours are similar in the sense that we all detect taint-style vulnerabilities based on taint analysis, a clear distinction is that our approach provides a graphical view during the analysis process, which provides analysts with more intuitive information.

It may also be worth mentioning the static analysis approaches focusing on incomplete programs. These approaches (e.g., (Schubert et al., 2021, Rountev and Ryder, 2001; Wang et al., 2022)) enable analysis in the coding phase, making it possible to perform vulnerability discovery in real time. Our approach also supports such kind of idea since a vulnerability net can be generated from the start of coding and incrementally expanded as coding proceeds.

7. Conclusion and future work

In this paper, we present an approach called vulnerability nets for detecting security vulnerabilities in source code. Specifically, the approach assists analysts 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 sanitization 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, as well as their algorithms. The preliminary results of our evaluation demonstrate the feasibility of our approach, yet there lacks a discussion of whether our approach is applicable to large programs.

Despite the important progress we have made in this work, there remains some work to be done for scaling up of the capability of the proposed approach. A few unsolved issues listed in Section 5 are driving our future research.

CRediT authorship contribution statement

Pingyan Wang: Conceptualization, Methodology, Writing – original draft, Software. **Shaoying Liu:** Writing – review & editing, Supervision, Methodology. **Ai Liu:** Formal analysis, Validation. **Wen Jiang:** Investigation, Resources.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

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

References

Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Depend. Secure Comput. 1 (1), 11–33. https://doi.org/10.1109/TDSC.2004.2.

Chess, B., West, J., 2007. Secure Programming with Static Analysis. Pearson Education.

Goseva-Popstojanova, K., Tyo, J., 2017. Experience report: security vulnerability profiles of mission critical software: empirical analysis of security related bug reports. In: 28th IEEE International Symposium on Software Reliability Engineering, pp. 152–163. https://doi.org/10.1109/ISSRE.2017.42.

SonarQube. https://www.sonarsource.com/products/sonarqube/., 2023 (Accessed 26 September, 2023).

Coverity Static Application Security Testing (SAST). https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html., 2023 (Accessed 26 September, 2023).

Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J., 2000. Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Proceedings DARPA Information Survivability Conference and Exposition, pp. 119–129 vol. 2.

Heelan, S., 2011. Vulnerability detection systems: think cyborg, not robot. IEEE Secur. Priv. 9 (3), 74–77. https://doi.org/10.1109/MSP.2011.70.

Leveson, N.G., Harvey, P.R., 1983. Analyzing software safety. IEEE Trans. Softw. Eng. (5), 569–579. https://doi.org/10.1109/TSE.1983.235116.

Leveson, N.G., Cha, S.S., Shimeall, T.J., 1991. Safety verification of ada programs using software fault trees. IEEE Softw. 8 (4), 48–59. https://doi.org/10.1109/52.300036.

Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, pp. 590–604. https://doi.org/10.1109/SP.2014.44.

Larochelle, D., Evans, D., 2001. Statically detecting likely buffer overflow vulnerabilities. In: 10th USENIX Security Symposium.

Vanegue, J., Lahiri, S.K., 2013. Towards practical reactive security audit using extended static checkers. In: 2013 IEEE Symposium on Security and Privacy, pp. 33–47. https://doi.org/10.1109/SP.2013.12.

Peterson, J.L., 1981. Petri Net Theory and the Modeling of Systems. Prentice-Hall.

Zitser, M., Lippmann, R., Leek, T., 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In: 12th ACM SIGSOFT International

Symposium on Foundations of Software Engineering, pp. 97–106. https://doi.org/10.1145/1029894.1029911.
Halfond, W.G., Viegas, J., Orso, A., 2006. A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering, pp. 13–15 vol. 1.
Su, Z., Wassermann, G., 2006. The essence of command injection attacks in web applications. Acm Sigplan Notices 41 (1), 372–382. https://doi.org/10.1145/1111037.1111070.
Jovanovic, N., Kruegel, C., Kirda, E., 2006. Pixy: a static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy, pp. 258–263. https://doi.org/10.1109/SP.2006.29.
Gupta, S., Gupta, B.B., 2017. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. Int. J. Syst. Assur. Eng. Manag. 8 (1), 512–530. https://doi.org/10.1007/s13198-015-0376-0.
Securibench Micro. https://github.com/too4words/securibench-micro. 2023 (Accessed 13 January. 2023).
Leveson, N.G., Stolzy, J.L., 1987. Safety analysis using Petri nets. IEEE Trans. Softw. Eng. (3), 386–397. https://doi.org/10.1109/TSE.1987.233170.
Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Programm. Lang. Syst. (TOPLAS) 9 (3), 319–349. https://doi.org/10.1145/24039.24041.
Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2007. Compilers: Principles, Techniques, & Tools. Pearson Education India.
Nielson, F., Nielson, H.R., Hankin, C., 2004. Principles of Program Analysis. Springer Science & Business Media.
Clapp, L., Anand, S., Aiken, A., 2015. Modelgen: mining explicit information flow specifications from concrete executions. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 129–140. https://doi.org/10.1145/2771783.2771810.
Chibotaru, V., Bichsel, B., Raychev, V., Vechev, M., 2019. Scalable taint specification inference with big code. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 760–774. https://doi.org/10.1145/3314221.3314648.
Soufflé: A Datalog Synthesis Tool for Static Analysis. https://souffle-lang.github.io/. 2023 (Accessed 13 January, 2023).
Arzt, S., et al., 2014. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 49, pp. 259–269. https://doi.org/10.1145/2594291.2594299.
Horwitz, S., Reps, T., Binkley, D., 1990. Interprocedural slicing using dependence graphs. ACM Trans. Programm. Lang. Syst. (TOPLAS) 12 (1), 26–60. https://doi.org/10.1145/77606.77608.
Medeiros, I., Neves, N., Correia, M., 2016. Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Trans. Reliab. 65 (1), 54–69. https://doi.org/10.1109/TR.2015.2457411.
Figueiredo, A., Lide, T., Matos, D., Correia, M., 2020. MERLIN: multi-language web vulnerability detection. In: 19th International Symposium on Network Computing and Applications (NCA), pp. 1–9. https://doi.org/10.1109/NCA51143.2020.9306735.
Medeiros, I., Neves, N., Correia, M., 2022. Statically detecting vulnerabilities by processing programming languages as natural languages. IEEE Transactions on Reliability 71 (2), 1033–1056. https://doi.org/10.1109/TR.2021.3137314.
Min, S.Y., Jang, Y.K., Cha, S.D., Kwon, Y.R., Bae, D.H., 1999. Safety verification of Ada95 programs using software fault trees. In: 18th International Conference on Computer Safety, Reliability and Security (SAFECOMP'99), pp. 226–238. https://doi.org/10.1007/3-540-48249-0_20.
Oh, Y., Yoo, J., Cha, S., Son, H.S., 2005. Software safety analysis of function block diagrams using fault trees. Reliab. Eng. Syst. Saf. 88 (3), 215–228. https://doi.org/10.1016/j.ress.2004.07.019.
Wei, F., Roy, S., Ou, X., Robby, 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Privacy Secur. 21 (14), 1–32. https://doi.org/10.1145/3183575.
Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C., 2015. Information flow analysis of android applications in droidsafe. In: 22nd Annual Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2015.23089.
Livshits, V.B., Lam, M.S., 2005. Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th USENIX Security Symposium, pp. 271–286.
Schubert, P.D., Hermann, B., Bodden, E., 2021. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. In: 35th European Conference on Object-Oriented Programming (ECOOP 2021), pp. 1–31. https://doi.org/10.4230/LIPIcs.ECOOP.2021.2.
Rountev, A., Ryder, B.G., 2001. Points-to and side-effect analyses for programs built with precompiled libraries. In: 10th International Conference on Compiler Construction, pp. 20–36. https://doi.org/10.1007/3-540-45306-7_3.
Wang, P., Liu, S., Liu, A., Zaidi, F., 2022. A framework for modeling and detecting security vulnerabilities in human-machine pair programming. J. Internet Technol. 23 (5), 1129–1138. https://doi.org/10.53106/160792642022092305021.

Pingyan Wang is currently a Ph.D. candidate in data science and informatics at Hiroshima University, Higashi-Hiroshima, Japan. He received the M.E. degree in computer science from Guangdong University of Technology, Guangzhou, China, in 2019. His research interests are in software security, program analysis, and Human-Machine Pair Programming.

Shaoying Liu is a Professor of Software Engineering at Hiroshima University, Japan, IEEE Fellow, BCS Fellow, and AAIA Fellow. He received the Ph.D. in Computer Science from the University of Manchester, U.K in 1992. His research interests include Formal Engineering Methods for Software Development, Specification-based Program Inspection and Testing, Testing-Based Formal Verification, and Human-Machine Pair Programming. He has published a book entitled "Formal Engineering for Industrial Software Development" with Springer-Verlag, 13 edited conference proceedings, and over 250 papers in refereed journals and international conferences. He proposed to use the terminology of "Formal Engineering Methods" in 1997, and has established Formal Engineering Methods as a research area based on his extensive research on the SOFL (Structured Object-Oriented Formal Language) method since 1989, and the development of ICFEM conference series since 1997. In recent years, he has served as the General Chair of QRS 2020 and ICECCS 2022. He is an Associate Editor for IEEE Transactions on Reliability and Innovations in Systems and Software Engineering, and a member of IPSJ and IEICE.

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.

Wen Jiang is a Ph.D. student in the Dependable Systems Laboratory at Hiroshima University, Japan. He obtained his master's degree in Signal and Information Processing at the South China University of Technology, China. His research focuses on investigating how to analyze safety problems in safety-critical systems.