FIRE: Combining Multi-Stage Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection

Siyue Feng, *National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab; School of Cyber Science and Engineering, Huazhong University of Science and Technology;* Yueming Wu, *Nanyang Technological University;* Wenjie Xue and Sikui Pan, *National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab; School of Cyber Science and Engineering, Huazhong University of Science and Technology;* Deqing Zou, *National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab; School of Cyber Science and Engineering, Huazhong University of Science and Technology; Jinyinhu Laboratory;* Yang Liu, *Nanyang Technological University;* Hai Jin, *National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab; School of Computer Science and Technology, Huazhong University of Science and Technology*

https://www.usenix.org/conference/usenixsecurity24/presentation/feng-siyue

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

FIRE: Combining Multi-Stage Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection

Siyue Feng[1,2], Yueming Wu[4]*, Wenjie Xue[1,2], Sikui Pan[1,2], Deqing Zou[1,2,5], Yang Liu[4], Hai Jin[1,3]

[1]*National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, Cluster and Grid Computing Lab*
[2]*School of Cyber Science and Engineering, Huazhong University of Science and Technology, China*
[3]*School of Computer Science and Technology, Huazhong University of Science and Technology, China*
[4]*Nanyang Technological University, Singapore*
[5]*Jinyinhu Laboratory, China*

Abstract

With the continuous development of software open-sourcing, the reuse of open-source software has led to a significant increase in the occurrence of recurring vulnerabilities. These vulnerabilities often arise through the practice of copying and pasting existing vulnerabilities. Many methods have been proposed for detecting recurring vulnerabilities, but they often struggle to ensure both high efficiency and consideration of semantic information about vulnerabilities and patches. In this paper, we introduce FIRE, a scalable method for large-scale recurring vulnerability detection. It utilizes multi-stage filtering and differential taint paths to achieve precise clone vulnerability scanning at an extensive scale. In our evaluation across ten open-source software projects, FIRE demonstrates a precision of 90.0% in detecting 298 recurring vulnerabilities out of 385 ground truth instance. This surpasses the performance of existing advanced recurring vulnerability detection tools, detecting 31.4% more vulnerabilities than VUDDY and 47.0% more than MOVERY. When detecting vulnerabilities in large-scale software, FIRE outperforms MOVERY by saving about twice the time, enabling the scanning of recurring vulnerabilities on an ultra-large scale.

1 Introduction

Vulnerabilities refer to security issues such as errors, defects, bugs, and other flaws in software. They arise due to logical errors, non-compliance with coding standards, low code quality in the code-writing process, or the lack of security testing. Vulnerabilities expose software to threats such as information leakage, remote control, and denial of service attacks. Exploiting vulnerabilities, hackers can compromise the security of software systems and networks, leading to severe consequences. Therefore, timely vulnerability detection is crucial for enhancing software security. Various methods for vulnerability detection exist today, including traditional approaches such as manual inspection, static analysis [17, 35, 45, 46], fuzz testing [7, 12, 32, 42, 51], symbolic execution [5, 10, 22, 44]. Additionally, there are some methods based on deep learning [14, 15, 26, 40, 57].

However, traditional vulnerability detection methods typically require manually defined rules and heavily rely on expertise in vulnerability-related domains. Intelligent vulnerability detection methods based on deep learning demand substantial labeled datasets for training, depending on both the quantity and exactness of the labeled datasets. Therefore, it is difficult for them to achieve large-scale vulnerability detection. In fact, with the continuous development of software open-sourcing, reusing open-source software has become a common practice in software development. This trend results in an increasing number of recurring vulnerabilities. These vulnerabilities exhibit similar characteristics, share code logic, or may even be identical, hence they are also known as clone vulnerabilities. Conventional vulnerability detection techniques can only leverage the general behaviors of the majority of vulnerabilities, lacking precise identification for recurring vulnerabilities that reuse specific behavior vulnerabilities. Moreover, they are not suitable for detecting vulnerabilities in large-scale open-source software. Therefore, when certain vulnerabilities are known, it is necessary to design novel techniques to rapidly identify widely prevalent recurring vulnerabilities in real-world code environments on a large scale.

Existing Approaches. Existing methods [8, 27, 29, 52, 53, 54, 56] specifically designed for detecting recurring vulnerabilities extract lexical, syntactic, or semantic signatures rich in vulnerability information from known vulnerabilities. Clones matching these signatures are considered potential vulnerabilities. These methods rely on the exactness of the extracted vulnerability features. If the extracted features are too simple or cannot precisely capture the vulnerability behavior, it may lead to false positives or false negatives. In addition to considering vulnerability features, adding consideration for patch information is essential to distinguish whether the function is a patched vulnerability (false positive) or a genuine vulnerability. Moreover, as the size of open-source communities continues to grow, it requires a substantial time investment to handle the massive amount of code for detection. The time gap between the appearance and discovery of vulnerabilities

*Yueming Wu is the corresponding author.

may lead to further propagation of vulnerabilities. Therefore, there is a current need for a method that can: 1) Enable rapid detection of extremely large-scale recurring vulnerabilities. 2) Support for detecting vulnerabilities that make syntactically different but semantically identical changes. 3) Consider the differences between vulnerabilities and patches.

Among existing recurring vulnerability detection tools, VUDDY [29] supports large-scale vulnerability scanning through a length filtering technique that reduces the number of signature comparisons. However, it can not detect clone vulnerabilities with syntactic changes and does not consider patch information. To address this, MOVERY [53] detects syntactic clone vulnerabilities by adding the oldest version of vulnerability and patch functions. However, the introduction of control flow and data flow information and matching at line granularity leads to low efficiency of MOVERY. Even if MOVERY reduces the search scope by obtaining public functions with identical syntax through path information, it only reduces the number of candidate functions by half, and the time overhead remains high. Furthermore, while MOVERY considers the differences between vulnerabilities and patches in most cases, it does not consider the statement order when extracting vulnerability and patch features. This lack of consideration may result in false positives by not thoroughly analyzing the behavioral differences between vulnerabilities and patches.

Our Approach. To address the limitations of existing tools, we propose utilizing taint analysis to maximize the behavioral differences between vulnerabilities and patches. Taint analysis techniques have been widely employed in vulnerability detection, providing a precise characterization of the semantic behavior of software [21, 28, 37]. Specifically, we perform taint analysis on vulnerable functions, concurrently conducting data-flow analysis for taint propagation. This allows us to extract the propagation paths of tainted markers in the program, resulting in tainted paths. Furthermore, we not only extract tainted paths of vulnerabilities but also perform the same tainted path extraction operation for patches. Subsequently, we focus solely on the differential aspects between vulnerabilities and patches. We consider the differential paths as signatures of vulnerabilities and patches.

However, the use of taint analysis comes with a high cost. To make taint analysis feasible for ultra-large-scale vulnerability scanning, we propose a multi-stage filtering approach. This approach filters vulnerabilities at three levels: simple vulnerability features, lexical features, and syntactic features, enabling taint analysis to be applicable to ultra-large-scale vulnerability scanning. Firstly, we extract vector representations for each function based on their simple features. Then, we employ our innovative Shuffle Fuzzy Bloom Filter, which supports approximate membership queries, to filter at the level of simple vulnerability features. Secondly, we extract token sequences from functions and then utilize traditional token similarity to compute the similarity between target functions and vulnerabilities, achieving filtering at the lexical level. Finally, we incorporate patch functions, conduct static analysis on functions to extract *abstract syntax trees* (ASTs), and use improved AST similarity to calculate the similarity among target functions, vulnerable functions, and patch functions, achieving filtering at the syntactic level.

Evaluation. We implement a prototype system, FIRE, and conduct testing on ten open-source software projects, comparing its performance with two state-of-the-art recurring vulnerability detectors (*i.e.,* VUDDY [29] and MOVERY [53]). Specifically, FIRE identifies 298 instances of recurring vulnerabilities, which is 2.55 times the number discovered by MOVERY. Moreover, it achieves higher precision, with a 23.2% improvement compared to VUDDY, and a 40.4% improvement compared to MOVERY. The recall also increases by 31.4% and 47%, respectively. While maintaining good precision and recall, FIRE demonstrates high efficiency. Particularly in the case of large-scale projects, it can save half the time compared to MOVERY. Furthermore, we also compare FIRE with two advancing learning-based methods (*i.e.,* REVEAL [11] and VulBERTa [23]), two static analysis-based vulnerability detectors (*i.e.,* Checkmarx [1] and FlawFinder [2]), and two general code clone detection tools (*i.e.,* SourcererCC [43] and Lazar et al. [30]). The experimental results indicate that FIRE outperforms these approaches as well.

Contribution. In summary, our method makes the following contributions:

- We propose a scalable recurring vulnerability detection method based on multi-stage filtering that extracts semantic signatures of vulnerabilities and patches through differential tainted paths.
- We implement a prototype system called FIRE [1] for effective and scalable detection of recurring vulnerabilities in open-source software.
- We conduct an in-depth comparative evaluation of FIRE against state-of-the-art vulnerability detection methods. The results demonstrate that FIRE can achieve large-scale vulnerability scanning with superior precision and recall.

2 Related Work

This section introduces related works closely associated with recurring vulnerability detection, including code clone detection methods and recurring vulnerability detection methods.

Code Clone Detection Techniques. Many techniques have been proposed to detect code clones (*e.g.,* [18, 24, 25, 50, 55, 59]). Some are designed for high precision in detecting complex clones, while others focus on achieving high efficiency. However, these methods primarily aim at detecting general code clones rather than recurring vulnerabilities. There are two reasons why code clone detectors cannot be used directly for vulnerable code clone detection: Firstly, the code differ-

[1]https://github.com/CGCL-codes/FIRE.

ences between vulnerable functions and their patched versions are often small. Consequently, code clone detectors that focus solely on vulnerable functions may mistakenly identify patched functions as vulnerable, leading to a high rate of false positives. Secondly, vulnerabilities are often subtle and context-dependent. Code clone detectors typically analyze the entire vulnerable function, which can result in missing the vulnerability if there are significant changes in areas unrelated to the vulnerability. This may lead to a high rate of false negatives.

Recurring Vulnerability Detection Techniques. Several techniques have been proposed for detecting recurring vulnerabilities [8, 27, 29, 52, 53, 54, 56]. Jang et al. introduce ReDeBug [27], a fast recurring vulnerability discovery technique using a slicing window approach. However, the use of exact matching leads to numerous false negatives, and matching only the context information of vulnerability-modified lines introduces many false positives. Kim et al. present VUDDY [29], a scalable vulnerable code clone discovery technique for large-scale software. However, VUDDY relies on normalization and abstraction, and can only detect fully identical and renamed clone vulnerabilities, missing variations with slight modifications. Additionally, due to the lack of patch information, VUDDY might misidentify already patched secure functions as vulnerabilities. Bowman et al. propose VGRAPH [8], a graph-based recurring vulnerability discovery technique that is more robust to code modifications, especially for vulnerabilities with syntax changes. Xiao et al. introduce MVP [56], which discovers recurring vulnerabilities with syntactic similarity by considering code lines directly related to vulnerabilities. Woo et al. present MOVERY [53], which identifies recurring vulnerabilities induced by internal OSS modifications by adding information from the oldest version of vulnerabilities. However, these methods have limited efficiency (*i.e.,* they require more time overhead to detect vulnerabilities), making it challenging to support large-scale recurring vulnerability detection.

In this paper, we propose an accurate and scalable recurring vulnerability detection method, which is designed for large-scale security assessments of open-source software. This approach employs multi-stage filtering to address the efficiency issues present in current methods. Additionally, it enhances detection effectiveness by utilizing differential taint paths to extract semantic signatures for vulnerabilities and patches, thereby addressing the issues of current methods that do not consider the differences between vulnerabilities and patches, and neglect semantic information.

3 Filtering Phase

As shown in Figure 1, our proposed method (*i.e.,* FIRE) comprises two phases: filtering phase and vulnerability identification phase. The filtering phase improves efficiency by reducing the number of candidate functions. The vulnerability identification phase improves the effectiveness by distinguishing between vulnerabilities and patches through tainted paths enriched with semantic information.

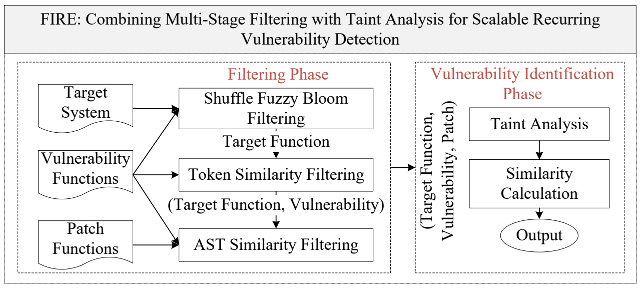

FIRE: Combining Multi-Stage Filtering with Taint Analysis for Scalable Recurring Vulnerability Detection
Filtering Phase
Target System
Vulnerability Functions
Patch Functions
Shuffle Fuzzy Bloom Filtering
Target Function
Token Similarity Filtering
(Target Function, Vulnerability)
AST Similarity Filtering
(Target Function, Vulnerability, Patch)
Vulnerability Identification Phase
Taint Analysis
Similarity Calculation
Output

Figure 1: The overview of our method

In this part, we focus on introducing the filtering phase of FIRE. Figure 1 gives the overview of our filtering phase, which consists of three stages: **Bloom Filter**, **Token Filter**, and **AST Filter**. The precision of the three-stage filtering increases incrementally, accompanied by a gradual increase in time overhead. However, each filtering stage in our method significantly reduces the number of functions to be inspected in the subsequent stage. This substantially improves the overall speed of our method. In the filtering phase, the source code of the target software is input, and the output is potentially vulnerable target functions with vulnerability and patch function pairs similar to them.

3.1 Bloom Filter

Bloom Filter is a concise data structure that uses a bit array to represent a set and efficiently determines whether an element belongs to that set. It is known for its high space and time efficiency and finds applications in various network-related tasks such as traffic identification [19], optimal replacement [39], longest prefix matching [16], route lookup [9], and packet classification [6]. If we abstract the recurring vulnerability detection as a set inclusion problem, determining whether a function is vulnerable is equivalent to checking if it belongs to the set of vulnerabilities. Despite the potential for false positives in Bloom Filter, our method leverages them as a filter to discard functions that clearly do not meet the conditions. For functions that may incur false positives, subsequent filtering stages are implemented to address them. Given the rapid speed of Bloom Filter, they are particularly suitable as the first filtering stage in our method to eliminate highly dissimilar functions. Therefore, we consider using Bloom Filter for the first stage of our filtering process.

3.1.1 Preprocessing

Before filtering with Bloom Filter, we preprocess all functions in the target software. Developers often add comments and empty lines to aid in code writing and understanding. These comments may introduce unnecessary noise for recurring vulnerability detection. To address this, we begin by removing comments and blank lines from each function before performing any operations. This normalization process ensures that our method can adapt to changes in code formatting or comments. In addition, similar to the previous approach [29, 53], we discard the functions with less than five *lines of code* (LOC) after normalization. Because the probability of these simple functions having vulnerabilities is low. This operation substantially reduces the number of functions to be inspected, thereby enhancing the efficiency of our method.

3.1.2 Standard Bloom Filter

Bloom Filter consists of H hash functions and an M-bit array used to represent the set S. Each element in the array can be either zero or one. The number of hash functions (*i.e.,* H) is between 1 and $M-1$. As shown in Figure 2, the steps for inserting and querying elements in Bloom Filter are as follows:

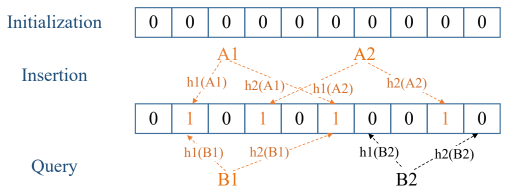


Figure 2: Inserting and querying elements in Bloom Filter

Initialization: The Bloom Filter is initialized as an M-bit array with all zeros.

Insertion: For element “A1” in the set S, H hash functions are applied to obtain H hash values, and the positions in the bit array corresponding to these H hash values are set to one.

Query: For querying an element “B1”, H hash functions are applied to obtain H hash values, and the positions in the bit array corresponding to these H hash values are checked to see if they are all set to one. If yes, it indicates that the queried element “B1” belongs to the set S within the range of false positive probability (*i.e.,* P); otherwise, the element “B1” does not exist in set S.

The formulas for calculating M, H, and P are as follows, where N represents the number of elements in the set represented by the Bloom Filter:

$$M = -\frac{N \times \ln(P)}{(\ln 2)^2} \tag{1}$$

$$H = -\frac{M \times \ln(2)}{N} \tag{2}$$

$$P = \left(1 - e^{-\frac{HN}{M}}\right)^H \tag{3}$$

Bloom Filter uses hash functions and bit arrays to represent data sets, and there may be hash conflicts leading to false positives. Increasing the length of the byte array and the number of hash functions reduces the false positive rate but increases the memory consumption (Equation 3). With a fixed acceptable false positive rate P, the appropriate M can be chosen by the number of elements N (Equation 1). The number of hash functions also needs to be weighed, the more hash functions then the Bloom Filter will be less efficient. However, if it is too few then the false positive rate will become higher. Therefore, the appropriate number of hash functions can also be calculated from the length of the bit array M and the number of elements N added to the Bloom Filter (Equation 2).

We apply the standard Bloom Filter for recurring vulnerability detection. First, we extract feature vectors for each vulnerability in the vulnerability dataset to form a set of vulnerability vectors. Then, each element (feature vector) in the set has been inserted into the Bloom Filter, including the vulnerability vector A_2 (1,2,3,4,**5**) in Figure 2. Two hash functions are applied to A_2 to get hash values 4 and 9, so the 4th and 9th position in the Bloom Filter's bit array are set to 1, as shown in Figure 2. When a vector B_2 (1,2,3,4,**6**) of target function is queried, since the last element of this vector is different from A_2, hashing B_2 will yield 7 and 10. Neither of these positions in the bit array is 1. Therefore, it is determined that the target function is not vulnerable. In fact, A_2 is very similar to B_2, differing in only one feature. As a result, standard Bloom Filter only supports exact membership queries, so functions with slight modifications cannot be queried. To enhance the variety of vulnerabilities we can detect, we design a new Bloom Filter to support approximate queries, enabling the detection of similar vulnerabilities.

3.1.3 Shuffle Fuzzy Bloom Filter

In this part, we introduce the design of our *shuffle fuzzy bloom filter* (SFBF). It is comprised of a certain number of standard Bloom Filters, including the initialization parts, insertion part, and query part.

The initialization part involves initializing a series of Bloom Filters (B_1, B_2, ..., $B_{maxTries}$) and a series of seeds (s_1, s_2, ..., $s_{maxTries}$). We use these seeds to construct these Bloom Filters during the insertion part.

The insertion part is where we construct the Bloom Filters containing the vulnerability dataset. For each feature vector v_0 of a vulnerable function, *maxTries* rounds of operations are needed to complete the insertion part. As illustrated in Figure 3, each round of the insertion stage consists of three steps. We use the operation in the j^{th} round as an example.

- **Step 1**: Shuffle v_{j-1} using seed s_j to generate v_j.
- **Step 2**: Discard the first $d\%$ positions of v_j, reducing the vector length to $1-d\%$ times of its original length, resulting in v_j.
- **Step 3**: Insert v_j into B_j.

After performing *maxTries* rounds of insertion operations for each function vector, we construct *maxTries* Bloom Filters, forming our SFBF.

The query part is similar to the insertion part as shown in Figure 3, but different in Step 3. For each function feature vector t_0, multiple rounds of operations are performed to complete the query part.

- **Step 1**: Shuffle t_{j-1} using seed s_j to generate t_j.
- **Step 2**: Discard the first $d\%$ positions of t_j, reducing the vector length to $1-d\%$ times of its original length, resulting in t_j.

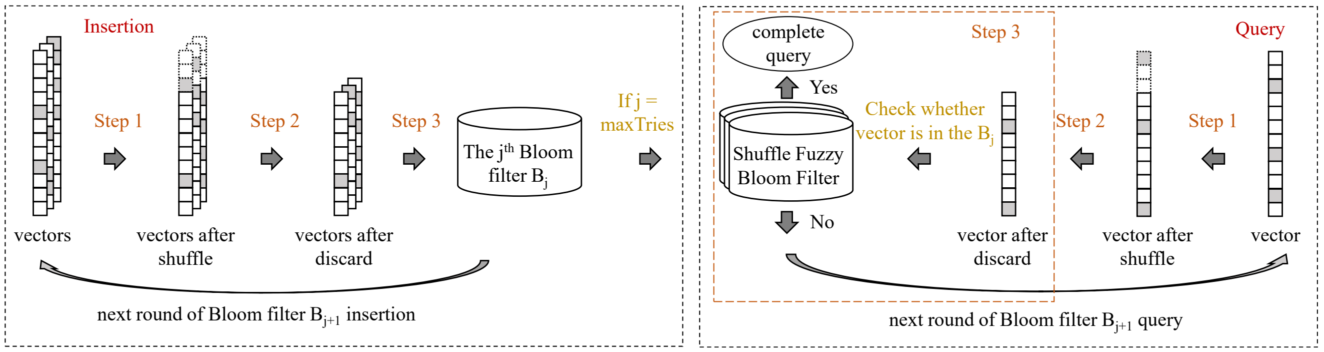


Figure 3: Each round of the insertion part and query part in Shuffle Fuzzy Bloom Filter

- **Step 3**: Check whether t_j is in the B_j. If it is, the query vector completes the query stage in the j^{th} round of operations; otherwise, new rounds of query operations continue until the query stage is completed or the maximum number of operations, $maxTries$, is reached.

We use the SFBF to query B_2 in Section 3.1.2. Since it has the operation of shuffle and discard, then the n$^{\text{th}}$ round of the insertion phase will shuffle A_2 (1,2,3,4,**5**) to (**5**,4,3,1,2) and discard the first element to get A_2' (4,3,1,2). B_2 (1,2,3,4,**6**) in the query phase of the n$^{\text{th}}$ round is also shuffled to (**6**,4,3,1,2), which yields B_2' (4,3,1,2) after discarding the first element. Since A_2 and B_2 discard the only different element, A_2' and B_2' are exactly the same. Therefore, the SFBF can query B_2, which is implemented to support the querying of slightly modified functions.

3.1.4 Crucial Features of Vulnerabilities

Second, in order to achieve the filtering of similar vulnerabilities, we extract crucial features to construct feature vectors, and hash the vectors instead of hashing the entire code information of a function. We introduce the crucial features of vulnerabilities that we select and construct feature vectors based on these features. Recent research [33] and [49] emphasize the significant correlation between source code vulnerabilities and specific syntactic features. For instance, syntax structures involving pointers and arrays in the source code are often vulnerable, as these operations frequently lead to out-of-bounds access or NULL pointer dereference. Additionally, specific arithmetic expressions may indicate potential improper operations, such as integer overflow.

Recent research GraphSPD [47] extracts features that reflect syntactic information about vulnerabilities. Therefore, we borrow the key features extracted in GraphSPD and make modifications and extensions based on them. Specifically, GraphSPD only considers identifiers and literal features (*e.g.,* variables, numbers, strings, pointers, arrays, null identifiers), and some keywords related to control flow. In order to capture more functional features and consider more types of vulnerabilities, we extend them to include all 73 C/C++ keywords, thus covering a wider range of features and structures in the code. GraphSPD contains only 34 regular operators, to which we add eight less common operators. GraphSPD does not contain format strings, but buffer overflows, format string vulnerabilities, and other types of vulnerabilities are closely related to format strings. Therefore, we add 20 format strings. Overall, given the syntactic features of vulnerabilities, we extract 177 features belonging to four groups from each code snippet. The specific feature and descriptions are shown in Table 4 in Appendix A.

These different crucial features correspond to different vulnerability types. The sensitive APIs, formatting strings, operators, or keywords used by different vulnerability types differ. For example, buffer overflow vulnerabilities are more related to formatted strings and memory allocation functions (*e.g.,* "*malloc*", "*alloc*"). Unauthenticated user input vulnerabilities are more relevant to formatted string functions (*e.g.,* "*printf*", "*sprintf*"). Out-of-bounds write and out-of-bounds read vulnerabilities are related to sensitive APIs of "*copy*" and "*sizeof*", and pointer-related operations. The null pointer dereference vulnerability is related to the use of the dereference operator * and NULL pointers.

We focus exclusively on these 177 crucial features and based on the presence of these features in each function, we generate the feature vector. First, we initialize a 177-dimensional feature vector with all zeros for each function. Then, if a feature exists, the value at the corresponding position in the vector is set to one. This process creates the feature vectors for each function, preparing for the efficient filtering of potential vulnerable functions in SFBF. Utilizing our SFBF for the initial stage of filtering helps reduce irrelevant functions in the target software by 80.63% (refer to Section 5.5). This substantial reduction of functions to be analyzed in subsequent steps significantly improves the overall speed.

3.2 Token Similarity Filter

The SFBF only filters out functions that do not contain similar features to vulnerable functions. However, there are cases where functions have the same vulnerability features but differ lexically from the vulnerable functions. The SFBF lacks the ability to discard such functions. Therefore, as the second filtering stage, we consider extracting the tokens of functions to increase consideration of lexical information. In this stage, we parse the functions and extract their token sets. For example, the token set of the code "*int a = b*c*" consists of "*int*", "*a*", "=", "*b*", "*", and "*c*". Then we calculate the similarity score between the token sets of the function to be examined and the vulnerable functions, and retain the functions with similarity scores above a threshold (*i.e.,* T_1) along with all corresponding similar vulnerabilities. We use a simple similarity calculation method (*i.e.,* Jaccard similarity) to compute the similarity between token sets, ensuring high efficiency.

Jaccard similarity is commonly used to compare the similarity between sets. Given two sets A and B, the Jaccard similarity is defined as the ratio of the size of the intersection of A and B to the size of the union set, it is calculated as:

$$Jaccard\,(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{4}$$

Notably, variable names are not symbolized because in the subsequent taint analysis phase, we need to extract the data flow based on the variable names to complete the paths collection. Generally, incorrectly excluding functions with different variable names but identical content does not occur frequently. This is because tokenization divides the entire function into small granularities, with variable names constituting only a very small part of the token set. If a function differs from a vulnerability function solely in the variable names but has the same content, the Jaccard similarity scores between the two would not be sufficiently low to result in incorrect exclusion.

After the filtering based on token similarity, not only are functions dissimilar lexically from the vulnerable functions filtered out, but each suspicious function is also provided with a list of potential reused vulnerabilities. Consequently, in the subsequent filtering and detection steps, there is no need to compare the suspicious function with all vulnerabilities. Instead, it only needs further validation against the potentially reused vulnerabilities. In summary, this filtering stage further improves the speed of our method by reducing the number of functions to be examined and narrowing down the matching candidates for each suspicious function. By using the second step of token similarity filtering, we are able to filter out 99.82% (refer to Section 5.5) of the irrelevant functions.

3.3 AST Similarity Filter

In the previous filtering, we only consider the lexical similarity between target functions and vulnerabilities. However, a patched secure function may have a very high lexical similarity with the vulnerability. Therefore, in the final filtering stage, we introduce patch functions to further calculate the syntactic similarity between target functions, vulnerable functions, and patch functions. This helps filter candidate functions based on syntactic similarity.

3.3.1 Delete Lines and Add Lines

Similar to the previous work [53, 56], before performing the syntactic similarity analysis, we first perform line-level filtering based on two conditions. We use F_v to represent the vulnerable function and F_p to represent the patched function after fixing the vulnerability. Deleted lines refer to the lines that appear in F_v but not in F_p, while added lines refer to the lines that do not appear in F_v but are present in F_p. Therefore, for a given pair of functions (F_v, F_p), we further define S_{del} as all deleted statements and S_{add} as all added statements. The target functions eligible for syntactic similarity analysis must meet the following two conditions:

- **C1**: The target function (F) must incorporate all deleted statements, *i.e.,* $\forall h \in S_{del}, h \in F$.
- **C2**: The target function must not include any of the added statements, *i.e.,* $\forall h \in S_{add}, h \notin F$.

C1 is to ensure that there are deleted statements in the target function that are directly related to how the vulnerability is created. C2 is to ensure that there are no added statements in the target function that are directly related to how the vulnerability is fixed.

3.3.2 Syntactic Similarity Analysis

We then employ AST generated from source code for similarity comparison. Specifically, we measure the similarity by calculating the number of nodes shared between two ASTs (*i.e.,* Jaccard similarity).

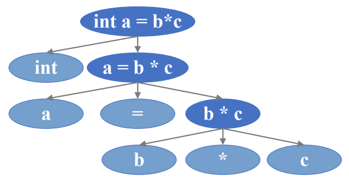


Figure 4: The AST of the code "int a = b*c"

We employ AST as the objects of comparison because compared to other graph representations, extracting AST of function does not require compilation and is very fast. For example, Figure 4 illustrates the AST of the code "*int a = b*c*". The node "*b*c*" has three child nodes: "*b*", "*", and "*c*". Nodes shaded in light blue in Figure 4 represent the leaf nodes (*i.e.,* nodes without child nodes). The leaf node sequence of each AST corresponds to the token sequence of the code line. The AST not only encompasses the token sequence of the code but also encodes syntactic information, representing the hierarchical structure of code decomposition. In this stage, we experiment with various algorithms to calculate the similarity between two ASTs, including 1) traditional hash-based comparison methods [60], 2) computing the similarity between sequences by deep traversal of AST nodes, 3) computing the similarity by calculating the number of edges shared between two ASTs, and 4) computing the similarity by directly calculating the number of nodes shared between two ASTs (*i.e.,* Jaccard similarity). Our experimental results show that calculating simple Jaccard similarity between nodes is sufficient for rapidly and accurately assessing the similarity between two ASTs. Therefore, we choose it to compute the AST similarity.

Moreover, we also consider the syntactic information of patch functions and simultaneously calculate the similarity between the ASTs of the target function, vulnerable function, and patch function. If the similarity with the vulnerability is higher and exceeds a threshold (*i.e.,* T_2), we retain the target function for further fine-grained semantic analysis, otherwise we discard the target function. In summary, the target function must satisfy the following conditions:

- **C3**: The similarity between target function and vulnerable function should surpass a predefined threshold, *i.e.,* $Sim(AST\_F, AST\_F_v) \geq T_2$.
- **C4**: The target function should have a higher syntactically similarity to the vulnerable function, *i.e.,* $Sim(AST\_F, AST\_F_v) \geq Sim(AST\_F, AST\_F_p)$.

This AST-based similarity filter evaluates the syntactic similarity between the target function and both the vulnerable and patched functions. It enhances precision while reducing the false positives associated with patch matching.

3.4 Summary to Filtering Phase

Using SFBF for the initial coarse-grained filtering significantly reduces the number of irrelevant functions in the target software. This reduction substantially decreases the number of functions that need to be analyzed in subsequent steps. The token filter not only filters out functions that are dissimilar to vulnerabilities lexically but also provides a list of potential vulnerabilities for each suspicious function. Consequently, in the subsequent AST filter, there is no need to compare suspicious functions with all vulnerable functions, but only with potentially reusable vulnerabilities. The AST filter compares the target function with the vulnerable and patch functions, and the target functions meeting the criteria are passed to the subsequent detection stages. This reduction in the number of functions to be analyzed further enhances the speed of FIRE.

4 Vulnerability Identification Phase

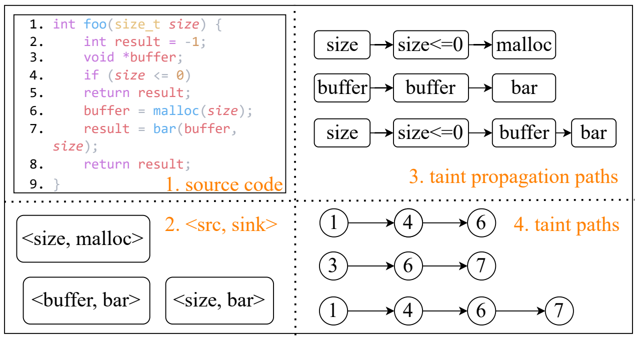
The vulnerability identification phase involves **signature extraction** and **vulnerability detection**. In the signature extraction phase, we perform taint analysis on target functions, vulnerable functions, and patch functions to extract taint propagation paths. In the vulnerability detection phase, the similarity between the tainted paths of the target function and the divergent parts of the taint paths between the vulnerability and patch is calculated. This is done to determine if the target function represents a vulnerability.

4.1 Signature Extraction

4.1.1 Extracting Function Signature

Taint analysis is a program analysis technique used to detect program vulnerabilities. Taint analysis focuses on sensitive data (*e.g.,* user input) in the program, accurately tracing the flow of data to pinpoint potential vulnerability points [36]. The analysis of taint propagation is closely related to the sensitivity of the information, and the flow of data reflects the data dependency relationships, carrying a significant amount of semantic information in the code [21, 28, 37].

Therefore, we use taint analysis to extract signatures, paying more attention to the sensitive information and semantic details within functions, allowing us to detect recurring vulnerabilities at a finer-grained semantic level. Specifically, we perform taint analysis on each function, extracting all <sources, sinks> points in the function. Unlike regular taint analysis, we do not consider sanitizers, meaning we do not check whether taint (*i.e.,* sensitive data) has been neutralized. The sanitization analysis requires a series of measures for verification and confirmation, which would introduce additional processing overhead. Hence, we only focus on the <sources, sinks> tuples. In response to the extracted tuple, we conduct taint propagation analysis by considering data dependency relationships. We extract the propagation paths of tainted data in the program, which we refer to as the taint paths. These paths constitute the signature of the function. We use Joern [58] for taint analysis. Joern generates *code property graph* (CPG) for function. CPG is a graphical representation that covers information about AST, *control flow graph* (CFG), and *program dependence graph* (PDG). A CPG consists of nodes and their types, labeled directed edges, and key-value pairs. We get parameters and variables by getting nodes with node type "*identifier*", which we consider as *sources*, and we get all function calls by getting nodes with node type "*call*", which we consider as *sinks*. Joern provides the "*reachableByFlows*" method to analyze the possible data flow between the specified nodes. By using this method, all the paths from sources to sinks can be obtained.



```
1. int foo(size_t size) {
2.     int result = -1;
3.     void *buffer;
4.     if (size <= 0)
5.     return result;
6.     buffer = malloc(size);
7.     result = bar(buffer,
   size);
8.     return result;
9. }
```

Figure 5: Extract taint paths from source code

For example, consider the function in Figure 5 that contains three variables: "*size*", "*result*", and "*buffer*". Two of these variables ("*size*" and "*buffer*") are used by two different functions, "*malloc*" and "*bar*". Since variable "*result*" is not used by any function, we do not analyze the taint propagation paths related to variable "*result*". Therefore, we can extract three <sources, sinks> tuples, namely <size, malloc>, <buffer, bar>, and <size, bar>. We analyze the propagation process of each tuple, *i.e.,* which operations each variable undergoes before ultimately propagating to the function use. For example, variable "*size*" appears in the conditional statement of an "*if*" statement and is then used by the "*malloc*" function. This completes the first taint propagation path for variable "*size*", *i.e.,* the first path in Figure 5. Then, variable "*size*" propagates to the variable "*buffer*" through the return value of the "*malloc*" function, and it is ultimately used by the "*bar*" function. This completes the second taint propagation path for the variable "*size*", *i.e.,* the third path in Figure 5.

After extracting all taint propagation paths, we replace each node in the paths with the corresponding complete code line. For example, we use the statement "*if (size <= 0)*" from line 4 to replace the node "*size <= 0*" in the first path of Figure 5. This results in the taint path, which is the signature of the function. These taint paths constitute the signature of the function.

4.1.2 Extracting Vulnerability and Patch Signatures

Similar to extracting taint paths for the target function, we also extract taint paths for both the vulnerabilities and patch functions. However, we do not utilize all paths, as most content in vulnerabilities and patch functions is similar, with only a few lines of code being different. Therefore, we extract the differing parts, considering paths that are unique to vulnerabilities as vulnerability signatures and paths that are unique to patches as patch signatures. The differential components represent critical features of vulnerabilities and precisely capture vulnerability elicitation and patching. Concentrating on the differential paths helps in distinguishing between vulnerabilities and patches, mitigating the impact of identical paths. The inputs to this phase are the potential vulnerabilities to be verified and the pairs of vulnerability patch functions that are similar to them, and the outputs are the target functions that are verified as vulnerabilities.

4.2 Vulnerability Detection

For the obtained function signatures, as well as the signatures of vulnerabilities and patches, we determine if the target function is vulnerable by comparing the similarity between signatures. Our signatures are composed of sets of paths, and each path is composed of code lines. Direct text comparison may lead to inaccurate similarity measurements due to minor changes causing significant variations in similarity. In contrast, using vector representations allows for a better capture of the semantic information in the text, understanding the meaning beyond mere reliance on vocabulary and surface structure. In our approach, we use CodeBERT for the vectorization of code lines. CodeBERT is pretrained on a large-scale code repository, providing a robust understanding of the semantics of the code and enhancing the capture of semantic relationships between code lines [20].

Therefore, we extract vector representations for all paths contained in the obtained signatures. Specifically, for each code line in a given path, we vectorize it to obtain a fixed-length vector. Subsequently, the vectors corresponding to all code lines in a path are averaged to reduce dimensionality while maintaining uniformity in signature vector dimensions. After such operation, each path is represented by a vector and each function is represented by a set of vectors. Next, we compute the similarity between the set of target function vectors and the set of vulnerable function and patch function vectors, respectively. Specifically, we compute the similarity of each path vector from the target function with all path vectors from the vulnerable function and keep the highest similarity score. For example, if the vulnerable function has m paths, then one path of target function will calculate the similarity with the m paths of vulnerable function, thus obtaining m similarity scores and retaining the maximum value s_1. If the target function has n paths, then there will be n similarity scores retained at the end, *i.e.,* s_1, s_2, s_3 ...... s_n, and averaging these n values gives the final similarity score S between the target function and the vulnerable function. Since all the scores are combined in an averaging operation, they are not affected by the order in which the paths are combined.

Given the signature vectors set S_f for each function in the target system, S_v for vulnerability signatures, and S_p for patch signatures, we determine the presence of a vulnerability in a target function based on the principle that its signature matches the vulnerability signature but not the patch signature. Specifically, if the target function satisfies the following condition, it possesses potential vulnerability:

- **C5**: The target function should have a higher semantic similarity to the vulnerable function, *i.e.,* $Sim(S_f, S_v) \geq Sim(S_f, S_p)$.

In this paper, we employ cosine similarity to calculate the similarity between signature vectors. Cosine similarity is a widely used algorithm for measuring similarity between vectors. Given two feature vectors A and B, their cosine similarity is defined as follow:

$$Cosine\_similarity(A,B) = \frac{A \cdot B}{\|A\| \cdot \|B\|} \quad (5)$$

Here, $A \cdot B$ represents the dot product of A and B, and $\|A\|$ and $\|B\|$ represent the Euclidean norms of A and B. C5 ensures the target function is more similar to the vulnerability compared to the patch function, reducing the false positives where patches are incorrectly detected as vulnerabilities.

5 Evaluation

In this section, we evaluate the effectiveness of FIRE and aim to address the following research questions:

- **RQ1:** How accurate is FIRE compared to other advanced recurring vulnerability detection methods?
- **RQ2:** How efficient is FIRE compared to other advanced recurring vulnerability detection methods?
- **RQ3:** How sensitive is FIRE in threshold selection?
- **RQ4:** How do the multi-stage filters contribute to FIRE?
- **RQ5:** How does the tainted path contribute to FIRE?
- **RQ6:** What is the performance of generic vulnerability detection tools in detecting recurring vulnerabilities?
- **RQ7:** What is the performance of the code clone detection approach in detecting recurring vulnerabilities?

5.1 Evaluation Setup

5.1.1 Dataset

We select target systems for detection from the open-source community, considering the following criteria during the selection process: 1) Programming Language: The target systems should be popular C/C++ open-source projects, as our method is primarily designed for vulnerability detection on C/C++; 2) Presence of Vulnerabilities: The target systems should contain a sufficient number of vulnerabilities to allow for evaluating the effectiveness of FIRE and baseline methods in detecting recurring vulnerabilities; 3) Diverse Application Domains: The selected systems should cover various application domains to demonstrate the generalizability of FIRE. Based on these criteria, we collect C/C++ repositories from GitHub with over 1,000 stars. From these projects, we select the top 10 software in terms of release time for detection. Two software have less than two vulnerabilities detected by FIRE and our comparative recurring vulnerability detection tools (*i.e.,* VUDDY [29] and MOVERY [53]). Therefore, we exclude them and add two frequently detected software in the previous work [33, 34], SeaMonkey and Xen. Finally, we choose ten open-source projects, and the details are presented in Table 5 in Appendix B in descending order of LOC. The lines of code range from 490,103 to 15,573,896, showcasing the scalability of FIRE. Application domains include internet app suite, machine learning, database, emulator, scripting language, multimedia processing, computer vision, virtualization, and image processing, which is diverse enough to show the generalizability of FIRE.

For the collection of our vulnerability dataset, we first use PatchDB [48] as our vulnerability dataset. PatchDB is the most widely used vulnerability patch dataset available, which includes 11,167 security patches. However, PatchDB does not cover complete vulnerability data and the latest vulnerability in PatchDB was discovered in 2019, so it does not include new vulnerabilities that have emerged in the last five years. Secondly, there are nearly six thousand security patches in PatchDB that do not have a vulnerability type. We need more complete and updated vulnerability data with information on vulnerability types. Therefore, we expand our vulnerability dataset by manually collecting further vulnerability data. Similar to previous methods [31, 41], we check if the CVE contains a Git commit URL from the *National Vulnerability Database* (NVD). Subsequently, we collect these URLs to crawl the secure patch submissions for CVE vulnerabilities from the respective Git repositories. Thus, we gather 3,316 C/C++ secure patches from the NVD. From these secure patches, we extract vulnerable functions and patch functions. Specifically, we focus on the header of the secure patch, which displays the file commits before and after the vulnerability is fixed [29, 56]. We extract all functions containing deleted code lines from the vulnerability file as vulnerable functions (F_v) and all functions containing added code lines from the patch file as patched functions (F_p). After manual collection, we collect over 10,874 pairs of vulnerability-patch function pairs (F_v, F_p). Thus our vulnerability dataset contains 22,041 (11,167 + 10,874 = 22,041) pairs of vulnerability-patch functions.

5.1.2 Comparative Tools

To evaluate the effectiveness of FIRE, we compare it with two recurring vulnerability detection tools (*i.e.,* VUDDY [29] and MOVERY [53]), four general-purpose vulnerability detection tools (*i.e.,* REVEAL [11], VulBERTa [23], Checkmarx [1], and FlawFinder [2]), and two general code clone detection tools (*i.e.,* SourcererCC [43] and Lazar et al. [30]). VUDDY is an accurate and scalable tool for detecting recurring vulnerabilities, utilizing a length-filtering technique to reduce the number of signatures for comparison. MOVERY is an accurate tool for detecting recurring vulnerabilities, considering the oldest vulnerable functions that are susceptible to attacks. REVEAL is a deep learning-based vulnerability detection tool by using graph neural network. VulBERTa is a method that utilizes deep knowledge representations to learn code syntax and semantics for detecting security vulnerabilities in the source code. Checkmarx and FlawFinder are two traditional static analysis-based vulnerability detection tools. SourcererCC is a token-based code clone detector by computing the overlapping similarities of two token sets. Lazar et al. design a novel code clone detector by analyzing the AST similarity of two functions.

5.1.3 Evaluation Metrics

We adopt five widely-used metrics, *true positive* (TP), *false positive* (FP), *false negative* (FN), *precision* ($P = TP/(TP+FP)$), and *recall* ($R = TP/(TP+FN)$) to evaluate the effectiveness of different methods. TPs and FPs are determined through manual inspection by three security analysts of all vulnerability detection results. To ensure the reliability of the vulnerability inspection results, we refer the source code of vulnerable functions and patch functions, NVD descriptions, and issue descriptions during the inspection process. Detecting all vulnerabilities in the target program is almost impractical, making it difficult to measure the FNs of each tool easily. Therefore, similar to the previous approaches [53, 56], we take the union of all TPs detected by three tools as the *ground truth* (GT), serving as a benchmark to measure the FNs of each tool. For example, the FNs in FIRE refer to the vulnerabilities detected by the other two methods but not discovered by FIRE. The specific evaluation environments are in Appendix C.

5.2 Effectiveness Evaluation (RQ1)

We run VUDDY, MOVERY, and FIRE to detect recurring vulnerabilities in the target projects. Among them, VUDDY uses our collected vulnerability dataset, while MOVERY uses its own vulnerability dataset since it only discloses the vulnerability signatures without the code for signature generation. Table 1 presents the effectiveness of these three methods, with the second and third columns displaying the project name and

Table 1: The True Positive, False Positive, False Negative, Precision, and Recall of VUDDY, MOVERY, and FIRE

IDX	Target System	GT	VUDDY					MOVERY					FIRE				
			TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
T1	FreeBSD	104	36	17	68	67.9%	34.6%	30	34	74	46.9%	28.8%	78	7	26	91.8%	75.0%
T2	SeaMonkey	23	11	14	12	44.0%	47.8%	3	7	20	30.0%	13.0%	16	1	7	94.1%	69.6%
T3	Turicreate	44	20	11	24	64.5%	45.5%	13	17	31	43.3%	29.5%	38	6	6	86.4%	86.4%
T4	MongoDB	10	6	2	4	75.0%	60.0%	6	7	4	46.2%	60.0%	7	0	3	100.0%	70.0%
T5	Xemu	7	4	21	3	16.0%	57.1%	0	2	7	0.0%	0.0%	4	1	3	80.0%	57.1%
T6	PHP	10	3	4	7	42.9%	30.0%	2	13	8	13.3%	20.0%	7	0	3	100.0%	70.0%
T7	OpenCV	127	74	11	53	87.1%	58.3%	49	29	78	62.8%	38.6%	101	3	26	97.1%	79.5%
T8	FFmpeg	9	3	4	6	42.9%	33.3%	1	4	8	20.0%	11.1%	6	7	3	46.2%	66.7%
T9	Xen	3	0	4	3	0.0%	0.0%	1	2	2	33.3%	33.3%	2	6	1	25.0%	66.7%
T10	OpenMVG	48	20	0	28	100.0%	41.7%	12	4	36	75.0%	25.0%	39	2	9	95.1%	81.3%
Total	-	385	177	88	208	66.8%	46.0%	117	119	268	49.6%	30.4%	**298**	**33**	**87**	**90.0%**	**77.4%**

the GT number of vulnerabilities, respectively. The remaining columns show the measurements for each tool.

Overall Results. FIRE achieves a detection precision of 90.0% for 298 recurring vulnerabilities with a recall of 77.4%, missing 87 vulnerabilities. In contrast, VUDDY and MOVERY detect only 177 and 117 recurring vulnerabilities, with recall of 46% and 30.4%, precision of 66.8% and 49.6%, respectively, both inferior to FIRE. Overall, FIRE outperforms VUDDY and MOVERY in detecting recurring vulnerabilities, achieving an average improvement of 31.8% in precision and 39.2% in recall.

Vulnerability Types. We analyze the 298 vulnerabilities detected by FIRE. Among them, 38 vulnerabilities are Buffer Overflow vulnerabilities (CWE-119), 37 vulnerabilities are Integer Overflow or Wraparound vulnerabilities (CWE-190), 28 vulnerabilities are Improper Input Validation vulnerabilities (CWE-20), 17 vulnerabilities are Out-of-bounds Write vulnerabilities (CWE-787), 15 vulnerabilities are Out-of-bounds Read vulnerabilities (CWE-125), and 11 vulnerabilities are Null Pointer Dereference vulnerabilities (CWE-476). This indicates that FIRE is more proficient in detecting these six types of vulnerabilities, while the number of other types of vulnerabilities detected by FIRE is relatively small. For example, CWE-399, CWE-834, CWE-434, CWE-362, and CWE-326, etc. Several factors contribute to this phenomenon. Firstly, the critical features selected for vulnerability detection may exhibit biases towards certain types of vulnerabilities, making FIRE more proficient in detecting them. For example, including formatting strings and memory allocation functions enhances the detection of buffer overflow vulnerabilities. Our use of sensitive APIs such as "*copy*", "*sizeof*", and our focus on pointers make FIRE proficient in detecting Out-of-bounds Write and Read vulnerabilities. Similarly, the lack of consideration for file reading-related functions such as "*CreateFile*" and "*WriteFile*" limits the detection of vulnerabilities of CWE-434. Additionally, FIRE also has limitations in detecting vulnerabilities that are difficult to detect through sensitive APIs or keywords, such as CWE-326. Secondly, some types of vulnerable functions have lower similarity to patch functions, thus can be better distinguished. Additionally, the composition of the dataset may influence the types of vulnerabilities detected by FIRE, indicating the need to expand the dataset to cover a broader range of vulnerability types. Lastly, the prevalence of certain vulnerability types in real-world scenarios, such as out-of-bounds write, may contribute to their higher detection numbers by FIRE.

```
    if (rc) {
+       if (s->ops->cleanup && s->ctx.private) {
+           s->ops->cleanup(&s->ctx);
+       }
        g_free(s->tag);
        g_free(s->ctx.fs_root);
        v9fs_path_free(&path);
    }
```

List 1: A patch snippet for CVE-2016-9914

```
    if (rc) {
*       v9fs_device_unrealize_common(s);
    }
    v9fs_path_free(&path);
    return rc;
```

List 2: Part of Function v9fs_device_realize_common

```
void v9fs_device_unrealize_common(V9fsState *s) {
    if (s->ops && s->ops->cleanup) {
        s->ops->cleanup(&s->ctx);
    }
    ...
    g_free(s->tag);
    ...
    g_free(s->ctx.fs_root);
}
```

List 3: Function v9fs_device_unrealize_common

False Positive Analysis for FIRE. We analyze all 33 false positives in the experimental results and identify two main reasons. The first one is patch fixes extending beyond the function granularity, which is also one of the reasons for false positives in VUDDY and MOVERY. For example, the patch for CVE-2016-9914 shown in List 1 is a denial of service vulnerability due to a missing cleanup operation. The patch adds the "*s->ops->cleanup()*" function for the cleanup operation.

FIRE determines the function in List 2 as a vulnerability due to the absence of the lines introduced by patch and its high similarity to the vulnerable function. However, the function "*v9fs_device_unrealize_common*" (shown in List 3) called in List 2 actually implements the patched functionality. In situations where the patch functionality is implemented outside the function, FIRE encounters false positives as it cannot conclusively determine whether a vulnerability has been fixed. While inter-procedural analysis could potentially alleviate this issue, it comes at the cost of increased computational overhead, thereby compromising the efficiency of FIRE.

The second reason is caused by the similarity of vulnerabilities, a phenomenon also observed in MOVERY, resulting in a substantial number of false positives. For instance, CVE-2016-8654 [2] encompasses three vulnerable functions (*i.e.,* "*jpc_qmfb_split_col*", "*jpc_qmfb_split_colgrp*", and "*jpc_qmfb_split_colres*") that are highly similar, all addressing heap buffer overflow vulnerabilities and fixing them by adjusting the allocated buffer size. When the target function is a clone of one vulnerability (*e.g.,* "*jpc_qmfb_split_colgrp*"), it exhibits high similarity with the other two vulnerabilities (*i.e.,* "*jpc_qmfb_split_col*" and "*jpc_qmfb_split_colres*"), leading to false positives. This situation can be mitigated by restricting the target function to match only the vulnerability with the highest similarity within the same CVE. This modification addresses the majority of false positives, but there is a specific scenario that remains unresolved. In cases where the target function introduces changes to deleted lines that do not impact the triggering of vulnerabilities, the conditions for deleted lines (C1) may not be satisfied. As a result, false negatives may occur with genuine vulnerabilities (*i.e.,* "*jpc_qmfb_split_colgrp*"), while false positives may arise with similar vulnerabilities (*i.e.,* "*jpc_qmfb_split_colres*").

False Negative Analysis for FIRE. We analyze the 87 false negatives in the experimental results and categorize their causes into three types. The first reason is as mentioned in the false positive analysis, where the target functions underwent changes on deleted lines that do not affect the vulnerability triggering. This leads to conditions on deleted lines (C1) not being satisfied, preventing them from entering the judgment of AST filtering and taint analysis, resulting in false negatives.

The second reason is that there are frequently occurring statements in S_{add}, such as return statements. These statements may often appear in the target functions, making them not satisfy our detection criteria for added lines (C2). As a result, they cannot proceed to AST filtering and taint analysis, leading to a significant number of false negatives. To address this issue, we count the occurrences of S_{add} in target functions, vulnerable functions, and patch functions. If the statement counts in the target function match those in the vulnerable function, the target function can proceed to the next stage, otherwise, it is filtered out. However, when the target function undergoes significant changes, this method may still result in some false negatives. For example, List 4 shows the added line statements in CVE-2018-14567, where the "*return -1*" statement appears eight times in the vulnerable function, nine times in the patch function, and nine times in a target function. In this scenario, even if the target function has not been fixed, it may still be filtered out due to the mismatch in added line counts, resulting in false negatives.

```
            return  −1;
        }
+       if  (( state −>how != GZIP) && (ret != LZMA_OK) && (ret
        != LZMA_STREAM_END)) {
+           xz_error ( state ,  ret ,  "lzma  error " ) ;
+           return  −1;
+       }
    } while  (strm−>avail_out  && ret != LZMA_STREAM_END);
```

List 4: A patch snippet for CVE-2018-14567

The third reason is that we apply LOC filtering to filter out all target functions with fewer than five lines of code, leading to some false negatives. However, the LOC filtering also resulted in the exclusion of numerous short functions, significantly enhancing our efficiency (refer to Section 5.5).

Due to space limitations, the false positive and false negative analysis for VUDDY and MOVERY are presented in Appendix D.

5.3 Efficiency Evaluation (RQ2)

To evaluate the efficiency of FIRE, we measure the time overhead of FIRE, VUDDY, and MOVERY in detecting vulnerabilities across ten target software projects. These tools share similar processes with FIRE, including *extracting vulnerability or patch signatures*, *extracting target function signatures*, and *matching*. As the codes for generating vulnerability and patch signatures of MOVERY are not open-source, we can not measure the time spent on this operation. Additionally, considering that the generation of vulnerability and patch signatures is an one-time operation, we pre-generate all the required vulnerability signatures and cache down them. Therefore, we only record the time for target function signature generation and matching, in other words, the time from the beginning to the end of the detection process.

The time overhead of each tool is recorded in Figure 6, where the x-axis is arranged from small to large according to the number of code lines in the target software, and the y-axis represents the time overhead in seconds. Overall, the time overhead of all tools except T5 (PHP) increases with the size of the project. This is because despite having more lines of code, PHP parses out fewer functions, and therefore its detection is relatively faster. Among the three tools, VUDDY has the least detection time because it simply extracts functions from files and performs normalization and abstraction operations. Since it does not consider any semantic information, its speed is faster.

[2] Due to the large amount of code, we provide the commit link for CVE-2016-8654: https://github.com/jasper-software/jasper/commit/4a59cfaf9ab3d48fca4a15c0d2674bf7138e3d1a

Table 2: The effectiveness and efficiency of FIRE as thresholds change

T_1	0.6			0.7			0.8		
T_2	0.6	0.7	0.8	0.6	0.7	0.8	0.6	0.7	0.8
Total F1 Score	82.46%	80.94%	76.26%	**82.62%**	81.69%	76.42%	80.38%	80.33%	76.26%
Speed (loc/s)	5,264.9	5,563.7	6,207.7	5,889.6	6,193.6	7,021.8	7,161.1	7,120.7	7,759.6

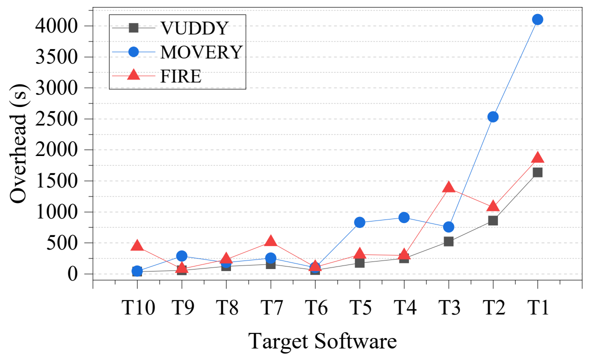


Figure 6: The time overhead of VUDDY, MOVERY, and FIRE

Compared to MOVERY, which also considers data flow and control flow information, our detection speed is faster. This is because MOVERY requires semantic analysis of functions to generate signatures by extracting data flow and control flow and matching them at line granularity. Although MOVERY also reduces the search scope by taking the measure of comparing path information, it can only reduce roughly half of the search space from the software codebase. FIRE can reduce 99.96% (discussed in RQ4) of the search space through the multiple filtering approach, and therefore achieves higher efficiency. The exception is T3 (Turicreate), which is due to the fact that T3 has more vulnerabilities, which means that more candidate functions enter the taint analysis phase. Taint analysis consumes more time, so T3 is slower to detect. When there are a large number of vulnerabilities in the software, the speed of detection decreases, which is the limitation of FIRE in terms of efficiency.

In addition, as shown in Figure 6, when the target program changes from a small program to a large program, the increase in our time overhead is more moderate and does not increase rapidly compared to MOVERY. For example, T10 (OpenMVG) is the smallest program among the ten target programs with 490,103 lines of code, and T1 (FreeBSD) is the largest program with 15,573,896 lines of code, which is 31.77 times the program size of T10. The detection time of FIRE for T1 is only 4.22 times longer than that of T10. As a comparison, the detection time of MOVERY for T1 is 93 times longer than that of T10, and VUDDY is also 45.51 times longer. These data clearly prove that FIRE can effectively reduce the detection time of large software, and this advantage is more obvious as the size of the program increases. Even with a project size of 38.6 M code lines, we can still complete the detection in less than two hours. This indicates that FIRE can scale to large software projects, meeting practical needs in real-world applications.

5.4 Threshold Sensitivity Analysis (RQ3)

In the filtering stage, we can achieve a balance between effectiveness and recall by configuring two thresholds. One is the threshold for token filtering (*i.e.,* T_1), and the other is the threshold for AST filtering (*i.e.,* T_2). Our default configuration is set to 0.7 for T_1 and 0.6 for T_2, and we use these settings for the effectiveness experiments in Section 5.2. In this part, we evaluate the sensitivity of FIRE to the two thresholds in terms of effectiveness and efficiency. For each threshold, we select three values: 0.6, 0.7, and 0.8, resulting in a total of nine threshold combinations. For each combination, we run FIRE on ten target software projects, recording the overall F1 score as a measure of effectiveness and the average lines of code processed per second as a measure of efficiency.

The experimental data in Table 2 shows that when both thresholds are set to 0.8, the F1 score is slightly lower. This is because the strict filtering conditions result in more false negatives, leading to a decrease in recall. Choosing thresholds of 0.7 or 0.6 has little impact on the F1 score, both achieving around 80%. The default configuration achieves the highest F1 scores. Compared to the 0.6 for both T_1 and T_2 configuration (0.6-0.6), which have almost the same F1 score, the default configuration is faster, processing an average of 5,889.6 lines of code per second. Therefore, we consider the default configuration to be a better choice.

5.5 Contribution of Multi-Stage Filter (RQ4)

In order to achieve high efficiency, we add multi-stage filter to FIRE. In this part, we explore the effectiveness of each filtering step by analyzing its speed, normal function filtering rate, and vulnerability detection recall, where the filtering speed is the average number of functions processed per second in each stage. The normal function filtering rate is calculated by dividing the number of remaining functions at the end of each stage by the overall number of functions. As for the vulnerability detection recall, we inspect 385 GT instances after each phase and derive the recall by dividing the number of remaining GT instances in each phase by the number of instances in the previous phase. The percentages represent the combined percentages across all projects.

First of all, only 6.63% of the functions in the vulnerability dataset have less than five lines of code. Among the software to be inspected, about 30.96% of the functions have less than 5 lines of code in PHP and 21.59% in FFmpeg. Therefore, it makes sense to improve efficiency by discarding functions with less than five LOC in the processing phase.

With the second row of Table 3, we can get that SFBF can filter 80.63% of target functions, the Token Filter further filters out 99.82% of functions, and after the AST Filter, 99.96% of functions are filtered. For instance, the number of functions extracted from files in FFmpeg is 23,315. After SFBF, only 7,737 functions remain, and after the Token Filter, only 308 functions are retained. The final AST Filter further reduces the number of functions to 15. It can be observed that the number of functions retained after each filtering layer significantly decreases for each software, indicating that each filtering layer we set up plays a role. The fact that only 0.04% of the final functions need to extract taint paths indicates that the multi-step filtering is able to massively reduce the number of functions to be detected, allowing us to keep the overall speed up despite the relatively slow extraction of taint paths (only 0.12 functions per second).

Table 3: The speed and proportion of functions that can be filtered at each filter

	Bloom	Token	AST	Taint Path
Filtering Rate	80.63%	99.82%	99.96%	99.97%
Recall	93.24%	99.27%	91.97%	99.99%
Speed (f/s)	167.71	54.31	1.43	0.12

By looking at the third row of Table 3, we can see that each stage produces a certain percentage of FNs, with the SFBF and the AST Filter producing more FNs. The reason for generating FNs in the SFBF stage is the incomplete and imprecise vulnerability features. As analyzed in RQ1, the vulnerability features have a mapping relationship with the vulnerability types. The lack of vulnerability features focusing on certain vulnerability types generates FNs. In addition, the vulnerability features contain the full range of C/C+ keywords, which differ in their ability to characterise whether a function is a vulnerability or not. The introduction of low-capability keywords can be disruptive to vulnerability judgment. In our future work, we will further investigate the vulnerability features to extract more comprehensive and precise vulnerability features to reduce FNs. The main reason for FN in the AST Filter phase is the checking of added and deleted lines, too strict judgments will bring FNs as analyzed in RQ1. Overall, the filtering phase does add some FNs, but brings more massive improvements in detection efficiency. The filtering phase makes sense from the perspective of balancing recall and time overhead.

Moreover, we also investigate the effectiveness of SFBF in improving the detection of vulnerabilities. Specifically, we replace the SFBF with a standard Bloom Filter in FIRE and then count the number of GT reported. The result reveals that only 142 out of 385 GT vulnerabilities are reported, resulting in a recall of only 36.9%. By employing the SFBF, we are able to detect an additional 156 recurring vulnerabilities, thus increasing the recall by 40.5%. This phenomenon demonstrates that our SFBF can tolerate certain modifications to functions through shuffling and discarding, thereby fulfilling the requirements for approximate member queries.

5.6 Contribution of Taint Analysis (RQ5)

In the Vulnerability Identification Phase, we perform vulnerability identification in three steps: extracting taint paths, embedding the paths as vectors using CodeBERT, and computing similarities. To assess the effectiveness of using taint paths, we conduct two ablation experiments. In the first experiment, we remove the initial step (*i.e.,* without extracting the taint paths) and directly extract vectors using CodeBERT. In the second experiment, we entirely remove the Vulnerability Identification Phase and label all the target functions filtered by the AST as vulnerabilities.

For the first ablation experiment, we select all functions (*i.e.,* 359 functions) that pass the AST Filter for CodeBERT to analyze. Out of the 298 TPs detected by FIRE, CodeBERT is able to detect only 257. This means there are 41 instances that CodeBERT misses, resulting in 41 FNs. Additionally, CodeBERT produces 14 more FPs than FIRE. These findings indicate that taint analysis provides benefits for FIRE, enabling it to detect more vulnerabilities with higher precision.

For the second ablation experiment, the experimental data indicates that by extracting signatures through taint paths, FIRE can eliminate 14 FPs. These eliminated FPs primarily include two types: 1) Target functions that added semantically equivalent statements to patch the vulnerabilities, and 2) Patches fix vulnerabilities by changing the order of statements, which AST cannot distinguish between vulnerability and patch functions.

For example, the patch in List 7 in Appendix E addresses a vulnerability related to unauthorized information disclosure. The target function in List 8 in Appendix E achieves semantically equivalent patching through two nested if statements. Through the extraction of data flow, the use of taint paths can effectively address such semantically equivalent alternatives, identifying semantic equivalence between the target function and patch function, thereby reducing FPs. The patch in List 9 in Appendix E solves the issue of mismatched assumptions by swapping the order of code blocks. This change is reflected as a positional shift of subtrees in the AST and does not alter the number or content of nodes. Therefore, using AST similarity filter is insufficient to distinguish differences between the vulnerable function and the patched function, leading to a misclassification of the target function as a vulnerability. Taint paths can highlight the order of variable occurrences, allowing them to identify differences between vulnerability and patch, thus eliminating FPs.

5.7 Performance of General-Purpose Vulnerability Detection (RQ6)

Compare with REVEAL and VulBERTa. We train the models of REVEAL and VulBERTa using the Diversevul dataset [13]. 385 GTs are used as the test set to assess their detection performance. We do not use all functions from the ten target software as the test dataset because analyzing all functions from predictions is a challenging and time-consuming task.

The test results reveal that, out of the 385 GTs, REVEAL only detects 134, with a recall of 34.8%, while VulBERTa can detect 158, with a recall of 41%. Our method achieves a 77.4% recall in detecting the 298 GTs, outperforming the two deep learning-based methods. This is because the performance of deep learning-based methods relies on the model architecture and the training dataset, especially on the quantity and correctness of the labeled dataset. However, the intelligent collection, labeling, and classification of vulnerability datasets still present challenges [38]. Therefore, our method proves to be more effective in detecting recurring vulnerabilities.

Compare with FlawFinder and Checkmarx. We use two static analysis tools to inspect vulnerabilities in ten software to see if they could detect the 385 GTs. Among them, FlawFinder only detects 44 vulnerabilities, while Checkmarx detects 50 vulnerabilities. With recall of only 11.4% and 13%, respectively, they lag far behind FIRE, VUDDY, and MOVERY. These results indicate that static analysis is inadequate for detecting recurring vulnerabilities, confirming the effectiveness of our method.

5.8 Performance of Clone Detection (RQ7)

In this part, we examine the ability of general code clone detection tools to detect recurring vulnerabilities. Specifically, we select two state-of-the-art code clone detectors, SourcererCC [43] and the tool developed by Lazar et al. [30], as comparative tools. We use the full vulnerability dataset as input, treating functions similar to these vulnerable functions in each project as potential vulnerabilities. SourcererCC and Lazar et al. report 35,127 and 17,839 potential vulnerabilities, respectively. We manually analyze 10% of the randomly selected results. The analysis shows that SourcererCC and Lazar et al. have a precision of 0.72% and 0.46%, respectively. Additionally, we use these two tools to detect 385 GTs to calculate recall. The results show that the recall of SourcererCC and Lazar et al. are 41.9% and 36.7%, respectively. These results indicate that general code clone detection methods are not suitable for detecting recurring vulnerabilities. In most cases, the code differences between vulnerable functions and patched functions are minimal. Therefore, code clone detection tools that use only vulnerable functions may incorrectly identify patched functions as vulnerable, resulting in high FPs.

6 Discussion

Vulnerability Disclosure. Of the 298 vulnerabilities we identified, 13 vulnerabilities are successfully replicated. We report these vulnerabilities to the respective software development teams. Among them, eight development teams confirm our findings, while we are still awaiting responses from the remaining. For the vulnerabilities that have not been confirmed, we will not disclose any information until patches are applied.

Limitations. Our method still has some limitations: Firstly, we detect vulnerabilities by computing the similarity between target functions and both vulnerability and patch functions. Only when the similarity between a target function and a vulnerability is higher and surpasses a threshold is the target function considered a potential vulnerability. Therefore, this approach is challenged in detecting extensive modifications in target functions that preserve the semantic equivalence with vulnerabilities, necessitating the assistance of dynamic analysis techniques. However, this could have a substantial impact on our efficiency. This reason also leads us to remain limited in dealing with similar vulnerabilities. We mitigate this issue by restricting target functions to match only the most similar vulnerabilities with the same CVE. However, there are still specific cases that remain unresolved as we discussed in Section 5.2.

Secondly, as mentioned in our FN analysis, we filter target functions by checking for the presence of deleted lines in the target function. This filtering approach may be too strict and could miss potential vulnerabilities where modifications are made to the deleted lines. In future work, we plan to adjust the criteria for judging added and deleted lines to tolerate a certain degree of modification.

Thirdly, our method is designed for vulnerabilities that occur at the function level. Therefore, any changes beyond the function level cannot be handled. This limitation leads to FPs and FNs. Implementing inter-procedural analysis could mitigate this issue. However, inter-procedural analysis comes with higher computational costs.

Lastly, there are limitations to the generalizability of FIRE to other programming languages and vulnerabilities. Since FIRE only targets C/C++ when selecting crucial features, this results in FIRE only being able to detect vulnerabilities in C/C++. In the future, we will extract corresponding keywords for other programming languages so that FIRE can be ported to other programming languages. The lack of vulnerability profiles for certain types of vulnerabilities may limit FIRE to extend to other vulnerability types. In the future, we will further investigate the vulnerability features to extract more comprehensive and precise features to reduce FNs.

7 Conclusion

In this paper, we propose and implement a novel method named FIRE. FIRE can 1) rapidly detect extensive recurring vulnerabilities through multi-stage filtering, 2) support for detecting complex recurring vulnerabilities with syntax changes, and 3) consider differences between vulnerabilities and patches by using differential taint paths. Our evaluation results demonstrate that FIRE significantly outperforms state-of-the-art methods for recurring vulnerability detection. It can detect 298 recurring vulnerabilities, achieving an average improvement of 31.8% in precision and 39.2% in recall.

Acknowledgements

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Key Program of the National Science Foundation of China under Grant No. U2336203.

References

[1] Checkmarx. https://checkmarx.com/, 2024.

[2] Flawfinder. https://dwheeler.com/flawfinder/, 2024.

[3] A parser generator tool and a incremental parsing library. https://tree-sitter.github.io/tree-sitter/, 2024.

[4] Universal ctags. https://github.com/universal-ctags/ctags/, 2024.

[5] Babić, D., Martignoni, L., McCamant, S., and Song, D. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2011), pp. 12–22.

[6] Baboescu, F., and Varghese, G. Scalable packet classification. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 199–210.

[7] Böhme, M., Pham, V., and Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1032–1043.

[8] Bowman, B., and Huang, H. H. Vgraph: A robust vulnerable code clone detection system using code property triplets. In *Proceedings of the 2020 IEEE European Symposium on Security and Privacy* (2020), pp. 53–69.

[9] Broder, A., and Mitzenmacher, M. Using multiple hash functions to improve ip lookups. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society* (2001), pp. 1454–1463.

[10] Cha, S. K., Woo, M., and Brumley, D. Program-adaptive mutational fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 725–741.

[11] Chakraborty, S., Krishna, R., Ding, Y., and Ray, B. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering 48*, 9 (2022), 3280–3296.

[12] Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., and Liu, Y. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 2095–2108.

[13] Chen, Y., Ding, Z., Alowain, L., Chen, X., and Wagner, D. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (2023), pp. 654–668.

[14] Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., and Wei, T. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 1253–1270.

[15] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2007), pp. 252–266.

[16] Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. E. Longest prefix matching using bloom filters. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003), pp. 201–212.

[17] Du, X., Chen, B., Li, Y., Guo, J., Zhou, Y., Liu, Y., and Jiang, Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering* (2019), pp. 60–71.

[18] Feng, S., Suo, W., Wu, Y., Zou, D., Liu, Y., and Jin, H. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (2024), pp. 1–13.

[19] Feng, W., Kandlur, D. D., Saha, D., and Shin, K. G. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society* (2001), pp. 1520–1529.

[20] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing* (2020), pp. 1536–1547.

[21] Feng, Z., Wang, Z., Dong, W., and Chang, R. Bintaint: a static taint analysis method for binary vulnerability mining. In *Proceedings of the 2018 International Conference on Cloud Computing, Big Data and Blockchain* (2018), pp. 1–8.

[22] Godefroid, P., Levin, M. Y., and Molnar, D. A. Automated whitebox fuzz testing. In *Proceedings of the 2008 Annual Network and Distributed System Security Symposium* (2008), pp. 151–166.

[23] Hanif, H., and Maffeis, S. Vulberta: Simplified source code pre-training for vulnerability detection. In *Proceedings of the 2022 International Joint Conference on Neural Networks* (2022), pp. 1–8.

[24] Hu, T., Xu, Z., Fang, Y., Wu, Y., Yuan, B., Zou, D., and Jin, H. Fine-grained code clone detection with block-based splitting of abstract syntax tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2023), pp. 89–100.

[25] Hu, Y., Zou, D., Peng, J., Wu, Y., Shan, J., and Jin, H. Treecen: Building tree graph for scalable semantic code clone detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022), pp. 1–12.

[26] Huang, Z., Dangelo, M., Miyani, D., and Lie, D. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (2016), pp. 618–635.

[27] Jang, J., Agrawal, A., and Brumley, D. Redebug: finding unpatched code clones in entire os distributions. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (2012), pp. 48–62.

[28] Kang, W., Son, B., and Heo, K. Tracer: signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 1695–1708.

[29] Kim, S., Woo, S., Lee, H., and Oh, H. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy* (2017), pp. 595–614.

[30] Lazar, F., and Banias, O. Clone detection algorithm based on the abstract syntax tree approach. In *Proceedings of the 9th IEEE International Symposium on Applied Computational Intelligence and Informatics* (2014), pp. 73–78.

[31] Li, F., and Paxson, V. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 2201–2215.

[32] Li, Y., Xue, Y., Chen, H., Wu, X., Zhang, C., Xie, X., Wang, H., and Liu, Y. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), pp. 533–544.

[33] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing 19*, 4 (2021), 2244–2258.

[34] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium* (2018), pp. 1–15.

[35] Livshits, V. B., and Lam, M. S. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 2005 USENIX Security Symposium* (2005), vol. 14, pp. 271–286.

[36] Newsome, J., and Song, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 2005 Annual Network and Distributed System Security Symposium* (2005), pp. 2–43.

[37] Niu, W., Zhang, X., Du, X., Zhao, L., Cao, R., and Guizani, M. A deep learning based static taint analysis approach for iot software vulnerability location. *Measurement 152* (2020), 1–12.

[38] Nong, Y., Ou, Y., Pradel, M., Chen, F., and Cai, H. Generating realistic vulnerabilities via neural code editing: an empirical study. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 1097–1109.

[39] Pagh, A., Pagh, R., and Rao, S. S. An optimal bloom filter replacement. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms* (2005), pp. 823–829.

[40] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W., Zibin, Y., Ernst, M. D., and Rinard, M. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), pp. 87–102.

[41] Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., and Acar, Y. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 426–437.

[42] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium* (2017), pp. 1–14.

[43] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering* (2016), pp. 1157–1168.

[44] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium* (2016), pp. 1–16.

[45] Vanegue, J., and Lahiri, S. K. Towards practical reactive security audit using extended static checkers. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), pp. 33–47.

[46] Viega, J., Bloch, J., Kohno, Y., and McGraw, G. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference* (2000), pp. 257–267.

[47] Wang, S., Wang, X., Sun, K., Jajodia, S., Wang, H., and Li, Q. Graphspd: Graph-based security patch detection with enriched code semantics. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy* (2023), pp. 2409–2426.

[48] Wang, X., Wang, S., Feng, P., Sun, K., and Jajodia, S. Patchdb: A large-scale security patch dataset. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2021), pp. 149–160.

[49] Wang, X., Wang, S., Sun, K., Batcheller, A., and Jajodia, S. A machine learning approach to classify security patches into vulnerability types. In *Proceedings of the 2020 IEEE Conference on Communications and Network Security* (2020), pp. 1–9.

[50] Wang, Y., Ye, Y., Wu, Y., Zhang, W., Xue, Y., and Liu, Y. Comparison and evaluation of clone detection techniques with different code representations. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering* (2023), pp. 332–344.

[51] Woo, M., Cha, S. K., Gottlieb, S., and Brumley, D. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (2013), pp. 511–522.

[52] Woo, S., Choi, E., Lee, H., and Oh, H. V1scan: Discovering 1-day vulnerabilities in reused c/c++ open-source software components using code classification techniques. In *Proceedings of the 32nd USENIX Security Symposium* (2023), pp. 6541–6556.

[53] Woo, S., Hong, H., Choi, E., and Lee, H. Movery: A precise approach for modified vulnerable code clone discovery from modified open-source software components. In *Proceedings of the 31st USENIX Security Symposium* (2022), pp. 3037–3053.

[54] Woo, S., Lee, D., Park, S., Lee, H., and Dietrich, S. V0finder: Discovering the correct origin of publicly reported software vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium* (2021), pp. 3041–3058.

[55] Wu, Y., Feng, S., Zou, D., and Jin, H. Detecting semantic code clones by building ast-based markov chains model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022), pp. 1–13.

[56] Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W., and Shi, W. Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *Proceedings of the 29th USENIX Security Symposium* (2020), pp. 1165–1182.

[57] Xu, Z., Zhang, Y., Zheng, L., Xia, L., Bao, C., Wang, Z., and Liu, Y. Automatic hot patch generation for android kernels. In *Proceedings of the 29th USENIX Security Symposium* (2020), pp. 2397–2414.

[58] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), pp. 590–604.

[59] Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering* (2019), pp. 783–794.

[60] Zhao, J., Xia, K., Fu, Y., and Cui, B. An ast-based code plagiarism detection algorithm. In *Proceedings of the 10th International Conference on Broadband and Wireless Computing, Communication and Applications* (2015), pp. 178–182.

A Simple Vulnerability Features

Given the syntactic features of vulnerabilities, we extracted 177 features belonging to the following four groups from each code snippet as shown in Table 4.

The first group comprises 42 sensitive APIs, which are related to memory management, string operations, locks and concurrency, and system-level functionalities. Improper use of these APIs for memory allocation, release, and manipulation may lead to issues such as memory leaks and buffer overflows. Using unsafe string manipulation functions may result in vulnerabilities like buffer overflows and format string vulnerabilities. Incorrect use of locks can lead to concurrency issues like deadlocks and race conditions. Inappropriate use of system-level operations may result in problems such as permission issues, data inconsistency, and denial-of-service attacks. Hence, our crucial features include APIs that are prone to unsafe behavior in the C/C++ programming languages.

The second group comprises 20 format strings, which are critical features in vulnerabilities. Improper input validation or incorrect use of format strings when using format string functions (*e.g.,* "*printf*", "*sprintf*", "*fprintf*", etc.) often leads to potential security vulnerabilities, such as code injection attacks. Therefore, our crucial features include format strings.

Table 4: Simple Vulnerability Features

sensitive APIs	alloc, free, mem, copy, new, open, close, delete, create, release, sizeof, remove, clear, dequene, enquene, detach, Attach, str, string, lock, mutex, spin, init, register, disable, enable, put, get, up, down, inc, dec, add, sub, set, map, stop, start, prepare, suspend, resume, connect
format strings	%d, %i, %o, %u, %x, %X, %f, %F, %e, %E, %g, %G, %a, %A, %c, %C, %s, %S, %p, %n
operators	bitand, bitor, xor, not, not_eq, or, or_eq, and, ++, –, +, -, *, /, %, =, +=, -=, *=, /=, %=, «=, »=, &=, ≙, |=, &&, ||, !, ==, !=, >=, <=, >, <, &, |, «, », , ; ->
C/C++ keywords	asm, auto, alignas, alignof, bool, break, case, catch, char, char16_t, char32_t, class, const, const_cast, constexpr, continue, decltype, default, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, noexcept, nullptr, operator, private, protected, public, reinterpret_cast, return, short, signed, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while, compl, override, final, assert

The third group comprises 42 operators, which are often directly related to the built-in semantics of programming languages. These operators can map directly to underlying computer instructions or the semantic rules of high-level languages without requiring additional function calls. Additionally, the use of operators may pose potential risks, such as integer overflow and bitwise operation errors. Therefore, our crucial features include operators.

The fourth group comprises 73 C/C++ keywords. In C/C++, keywords are identifiers with special meanings, holding significant positions in the syntax of programming languages. Compilers use keywords to represent the fundamental structures, control flow, data types, and other fundamental elements of the code. Thus, our crucial features include keywords.

B Target Software

The target systems are presented in Table 5.

Table 5: Target software

IDX	Name	Version	#Lines	Domain
T1	FreeBSD	12.2.0	15,573,896	Operating System
T2	SeaMonkey	2.53.18	8,370,870	Internet App Suite
T3	Turicreate	6.4.1	5,003,684	Machine Learning
T4	MongoDB	r4.2.11	3,295,598	Database
T5	Xemu	0.7.118	1,642,871	Emulator
T6	PHP	8.3.2	1,390,193	Scripting Language
T7	OpenCV	4.5.1	1,201,122	Computer Vision
T8	FFmpeg	n4.3.2	1,118,186	Multimedia Processing
T9	Xen	4.17.3	527,124	Virtualization
T10	OpenMVG	2.1	490,103	Image Processing
Total	-	-	38,613,647	-

C Evaluation Environment

We employ the precise and efficient open-source function parser Ctags [4] to extract functions from vulnerability files, patch files, and the target software. We use Tree-Sitter [3] to extract the AST of functions and Joern [58] to generate taint paths for functions. FIRE is implemented using 2,836 lines of Python code. The experiments are conducted on a machine with a 3.40 GHz Intel i7-13700k processor and 48 GB of RAM, running on ArchLinux with Linux Zen Kernel. The memory usage of FIRE is 20 GB. All the advanced methods compared in the experiments are configured with settings identical to those reported in their respective original papers.

D False Positive and False Negative Analysis for VUDDY and MOVERY

FP Analysis for VUDDY and MOVERY. In addition to false positives caused by exceeding function granularity, VUDDY generates significant false positives mainly due to abstraction. VUDDY detects renaming clones of vulnerability by replacing all formal parameters, local variables, data types, and function calls with specific symbols in a function. However, when a vulnerability is fixed by only changing these abstracted parts, the abstracted vulnerable function and the patched function will have identical hash values. As a result, VUDDY may incorrectly identify patched functions as vulnerabilities, leading to a large amount of false positives. Moreover, excessive abstraction can also cause hash collisions for relatively simple functions. For example, the patch in List 5 fixes a vulnerability by adding range checks for block device read or write requests. However, due to excessive abstraction, the target function in List 6 shares the same hash value as the vulnerable function in List 5, even though they implement entirely different functionalities and the target function is not vulnerable, leading to false positives.

Similar to the second false positive reason in our method, MOVERY generated a large number of false positives due to the similarity of vulnerabilities. As the absence of restrictions limits the target function to only match the most similar vulnerability within the same CVE, MOVERY generates significantly more false positives for this reason compared to FIRE. Additionally, similar to VUDDY, MOVERY also exhibits false positives caused by abstraction.

```
  BlockDriverAIOCB *bdrv_aio_readv(BlockDriverState *bs,
      int64_t  sector_num, QEMUIOVector *iov, int nb_sectors ,
      BlockDriverCompletionFunc *cb, void  *opaque) {
+     if  (bdrv_check_request(bs,  sector_num,  nb_sectors))
+         return  NULL;
+
      return  bdrv_aio_rw_vector(bs,  sector_num,  iov,  nb_sectors
      ,  cb,  opaque,  0);
  }
```

List 5: A patch snippet for CVE-2008-0928

```
  int  xmlParseBalancedChunkMemory(xmlDocPtr doc,
      xmlSAXHandlerPtr sax, void *user_data, int depth,  const
      xmlChar *string ,  xmlNodePtr *lst )  {
       return  xmlParseBalancedChunkMemoryRecover(doc, sax,
      user_data, depth,   string , lst ,0) ;
  }
```

List 6: Function xmlParseBalancedChunkMemory

FN Analysis for VUDDY and MOVERY. The false negatives in VUDDY arise from its use of exact matching. If a target function undergoes changes that do not affect the triggering of the vulnerability, VUDDY cannot detect such vulnerabilities. In other words, VUDDY can only detect exact clones and renamed clones of vulnerability. As for MOVERY, since it also uses the same deletion and addition line detection as ours, the reasons for its false negatives are similar to ours. Moreover, MOVERY does not count the occurrences of added lines, leading to more false negatives than FIRE.

E Code Snippets

List 7, List 8, and List 9 are the code snippets used in Section 5.6.

F Multi-Version Vulnerability

We use FIRE to investigate different versions of OpenMVG (*i.e.,* versions 0.1, 0.5, 0.7, 0.8, 1.0, 1.6, 2.0, and the latest version 2.1) to analyze vulnerability propagation between versions. We find that there is a vulnerability that propagates continuously from version 0.1 to version 2.1. Version 0.8 introduced a large number of vulnerabilities, of which only one was fixed in version 1.0, and the remaining 38 vulnerabilities propagated continuously to version 2.1, where a new vulnerability was introduced. This phenomenon illustrates that vulnerability propagation between different versions of the same software is serious. Vulnerabilities introduced by a certain version will have a higher probability of propagating to subsequent versions, which is more serious than the propagation between different software. Therefore, efficient vulnerability detection is needed to detect vulnerabilities on time and prevent further propagation.

```
      nl1e  =  l1e_from_intpte ( val ) ;
+     if  (  !( l1e_get_flags ( nl1e )  & _PAGE_PRESENT) &&
      pv_l1tf_check_l1e(d, nl1e) )
+         return  X86EMUL_RETRY;
      switch  (  ret  = get_page_from_l1e(nl1e ,  d,  d)  )
      {
          default :
```

List 7: A patch snippet for CVE-2018-3620

```
      nl1e  =  l1e_from_intpte ( val ) ;
*     if  (  !( l1e_get_flags ( nl1e )  & _PAGE_PRESENT) )
*     {
*         if  (  pv_l1tf_check_l1e ( d,  nl1e )  )
*             return  X86EMUL_RETRY;
*     }
      else
      {
          switch  (  ret  = get_page_from_l1e(nl1e ,  d,  d)  )
          {
              default :
```

List 8: A snippet for Function ptwr_emulated_update

```
+     if  (  buf−>have_grant  )
+     {
+         __release_grant_for_copy ( buf−>domain, buf−>ptr.u. ref ,
      buf−>read_only);
+         buf−>have_grant  =  0;
+     }
      if  (  buf−>have_type  )
      {
          put_page_type(buf−>page);
      .....
          put_page(buf−>page);
          buf−>page = NULL;
      }
−     if  (  buf−>have_grant  )
−     {
−         __release_grant_for_copy ( buf−>domain, buf−>ptr.u. ref ,
      buf−>read_only);
−         buf−>have_grant  =  0;
−     }
```

List 9: A patch snippet for CVE-2017-15597