Uncovering Recurring Vulnerabilities through Taint-Extracted Operator Sequences

Chang-Ming Yang*, Che-Jui Hsu*, Tao Ban†, Takeshi Takahashi†, Hsu-Chun Hsiao*‡
*National Taiwan University, Taiwan
†National Institute of Information and Communications Technology, Japan
‡Academia Sinica, Taiwan

***Abstract*—Recurring vulnerabilities, caused by code reuse, spread into new software when developers copy flawed open-source code. Detecting these vulnerabilities is challenging due to limitations in accuracy, scalability, and vulnerability-type coverage. Recent advancements by Kang et al. [1] improved accuracy and scalability using taint analysis but had limited coverage. We propose OPSMATCHER (Operation Sequence Matcher), a comprehensive method leveraging taint analysis to extract sequences of operators and function calls as signatures, and employing string-matching algorithms and rule-based filters for accurate signature matching. Experiments show that OPSMATCHER supports 24 common vulnerability types with 0.768 precision and 0.721 recall, whereas a state-of-the-art tool supports only seven types with about 0.3 lower accuracy. Additionally, OPSMATCHER identified eight previously unknown recurring vulnerabilities in Debian packages, showing superior coverage and effectiveness over existing methods.**

***Index Terms*—recurring vulnerability detection, taint analysis, string matching**

I. INTRODUCTION

Open-source software (OSS) and libraries have become ubiquitous across various industries. The 2023 Open Source Security and Risk Analysis report [2] indicates that 96% of codebases contained open-source components across 17 industries, including software, automotive, AI, FinTech, and IoT. Developers often leverage these ready-made functions to save time and effort. Additionally, generative AI tools frequently utilize open-source code to generate the required functionalities based on developer commands. While open-source code provides significant convenience, it also introduces unique cybersecurity challenges. Vulnerabilities present in open-source functions can proliferate across diverse and unrelated projects due to code reuse, leading to what is known as recurring vulnerabilities [3]. Despite being similar to the original vulnerabilities, these recurring vulnerabilities are difficult to detect because their relationships to the original vulnerabilities are often unclear to developers.

To effectively detect recurring vulnerabilities, a detection tool must meet three key criteria: accuracy, scalability, and generality. First, the tool must accurately identify vulnerabilities with low false positive and false negative rates to avoid unnecessary verification of safe code or missing actual vulnerabilities. Second, it should support a wide range of vulnerability types or be easily adaptable to various types. Third, it must be scalable to handle large codebases efficiently. However, existing approaches typically fall short of meeting all these objectives simultaneously.

Clone-based approaches [4], [5], [6], [7], [8] focus on detecting code similarity, offering good scalability and independence from specific vulnerability types. However, these approaches tend to have high false positive and false negative rates because they focus on function or file similarity rather than the distinctive characteristics of vulnerabilities. On the other hand, signature-based approaches [9], [10], [11], [12] extract vulnerability characteristics, such as execution paths, and use these signatures to detect vulnerabilities in target code, thus improving accuracy compared to clone-based ones.

A state-of-the-art signature-based tool, TRACER, proposed by Kang et al. [1], uses taint analysis to extract vulnerability signatures accurately. Using a taint analyzer with built-in detection capabilities for seven types of vulnerabilities [13], TRACER accurately identifies vulnerability traces and detects potentially vulnerable traces in programs. To uncover semantically recurring vulnerabilities, it extracts the frequency of operators and function calls from a trace as signatures, thereby mitigating the influence of syntax changes. Despite TRACER's advances in accuracy and scalability, several limitations remain. Primarily, it relies solely on the frequency of operators and function calls, which may not adequately capture the semantic and structural information of the trace for certain vulnerability types. In addition, extending the method to support other types of vulnerabilities requires extensive efforts to develop specialized taint analyzers for each type.

To overcome these challenges, we propose OPSMATCHER (Operation Sequence Matcher), a comprehensive approach that leverages a taint-analysis-based framework to enhance vulnerability detection. OPSMATCHER operates by utilizing a generic taint-analysis engine to extract traces based on the data dependency between data initialization points (DIPs) and data egress points (DEPs). These traces are then used to derive sequences of operators and function calls, which serve as signatures encapsulating significant semantic and structural information. By representing these signatures as sequences, OPSMATCHER applies string-matching algorithms to compute their similarity. Moreover, to minimize false positives, heuristic rules are employed to filter out signatures indicative of patched vulnerabilities. Ultimately, signatures that meet the similarity threshold and pass patch filtering are reported as recurring vulnerabilities.

Our implementation of OPSMATCHER on the open source static taint analysis platform Joern [14] demonstrates its robust taint analysis capabilities for extracting detailed semantic and structural information. For sequence similarity computation, we used Gestalt pattern matching [15] with the longest common subsequence (LCS) algorithm to balance the embodiment of structural information and computational speed.

The major contributions of the paper are:

- Enhanced Vulnerability Detection: OPSMATCHER extends support to 24 common vulnerability types, surpassing the capabilities of many existing tools.
- Precision and Recall: In our evaluations using the Juliet test suite and Debian packages, OPSMATCHER achieved a precision of 0.768 and a recall of 0.721, demonstrating its effectiveness on large-scale ground truth samples.
- Detection of Unknown Vulnerabilities: OPSMATCHER identified eight previously unknown recurring vulnerabilities in Debian packages, highlighting its practicality.
- Superior Performance: OPSMATCHER showed significantly better performance, with a precision of 0.829 and a recall of 0.745, compared to the existing tool TRACER's 0.744 precision and 0.321 recall.

The source code, datasets, and recurring vulnerability reports of OPSMATCHER are publicly available online [16].

II. BACKGROUND AND RELATED WORK

Recurring vulnerability detection involves comparing target code to a database of known vulnerabilities. High similarity indicates the presence of a recurring vulnerability. This field has recently gained attention, especially on clone-based and signature-based methods. Clone-based methods identify similar code segments to detect vulnerabilities, while signature-based use predefined patterns. This section provides a brief overview of these approaches, highlighting their limitations.

A. Related Work

1) Clone-Based Approaches: Clone-based approaches extend from code similarity and code clone detection. They compare each function or file in the target code with those containing known vulnerabilities. If the two are similar, the function or file is judged to have a recurring vulnerability. For example, Kim et al. [6] proposed a scalable approach at the function granularity level. They applied multilevel abstraction and normalization on functions to generate fingerprints, overcoming the influence of renaming. These fingerprints are stored in a dictionary for efficient lookup, where the key is the fingerprint length and the value is the list of fingerprint hashes. When a target code is checked for vulnerability, it is preprocessed, and its fingerprints are looked up in the dictionary to identify recurring vulnerabilities.

Clone-based approaches are scalable to large programs and agnostic to vulnerability types. However, they face the challenge of high false positives. Vulnerability-related code typically accounts for a small proportion of the function or file, and using functions or files as comparison units often includes irrelevant code. As a result, even if a vulnerability is patched, the overall similarity may still lead to a false positive.

2) Signature-Based Approaches: To improve accuracy, signature-based approaches focus on the characteristics of vulnerabilities, locating vulnerabilities and extracting features as signatures. For example, Xiao et al. [11] proposed an approach that leverages program-slicing techniques to extract vulnerability and patch signatures based on differences between vulnerable and patched functions. A target function is considered to have recurring vulnerabilities if it matches the vulnerability signature but not the patch signature.

Signature-based approaches offer better accuracy than clone-based ones but suffer from scalability issues and can still produce false positives and false negatives. This is because techniques to locate known vulnerabilities are imprecise, and patches may include changes irrelevant to the vulnerabilities.

3) Taint Analysis-Based Approaches: To address the limitations of signature-based approaches, Kang et al. [1] introduced TRACER, which leverages a specialized taint analysis engine, Facebook's Infer [13]. In taint analysis, a source is where data is received from external users, and a sink is a security-sensitive function. TRACER identifies vulnerable source-sink pairs in programs and extracts data dependency traces from source to sink. They represent features as frequencies of operators and function calls along the trace and use cosine similarity to compare feature vectors.

In their experiments, TRACER successfully discovered 112 recurring vulnerabilities across 273 Debian packages in C/C++, using a database containing known CVEs and samples from the Juliet test suite [17] for seven vulnerability types. This demonstrates the effectiveness of incorporating taint analysis in detecting recurring vulnerabilities.

B. Motivation

While TRACER effectively detects recurring vulnerabilities, it has several limitations that motivate the development of our proposed method. First, TRACER represents features using frequencies of operators and function calls, capturing only limited semantic information from traces and losing valuable structural details. This limited feature representation hinders its ability to fully characterize semantically recurring vulnerabilities. Second, TRACER's lack of generality limits its support for diverse vulnerability types. It relies on a specialized taint analyzer, originally designed to detect potential bugs, to identify potentially vulnerable data flows. However, extending TRACER to support new vulnerability types requires defining specific abstract domains and designing tailored engines using abstract interpretation techniques. This process requires significant time and effort for each new vulnerability type.

For instance, Figure 1a depicts an improper resource shutdown vulnerability, a type that TRACER cannot handle. This type of vulnerability, characterized by the failure to release or incorrectly releasing resources, requires intricate handling. To support detection of this vulnerability type, a detector must be capable of tracking file descriptor operations to ensure proper resource shutdown. Despite the apparent simplicity of this

vulnerability type, its detection requires scrutinizing various conditions, including all possible combinations of open and close function calls. Thus, extending TRACER to support the detection of improper resource shutdown would consume substantial effort and time, as it focuses on the occurrences rather than the ordering of the function calls.

```
1 void Improper_Resource_Shutdown(){
2     int data = -1;
3     data = open(...);
4     if (data != -1)
5         fclose((FILE *)data);
6 }
```

(a) source code

```
[3 - open, 4 - data,
5 - (File *),5 - fclose]
        ↓
[open, notEquals, fclose]
```

(b) vulnerable trace and signature

Fig. 1: CWE-404 (Improper resource shutdown) example

III. Design and Implementation

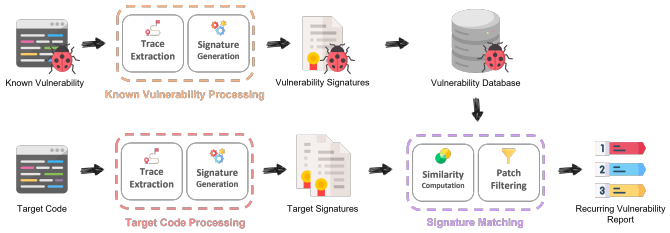

Fig. 2: Architecture of OpSMatcher

In order to enhance the generality of our recurring vulnerability detection framework, we devised solutions to address the above challenges. These solutions are presented as a tool named OpSMatcher. To ensure broad applicability, we utilize a generic taint-analysis engine capable of statically examining how relevant data propagates within the code, regardless of the vulnerability type, thereby facilitating the extraction of relevant traces.

Upon considering the most suitable features to encapsulate vulnerabilities, we recognize the diverse nature of vulnerability types. Relying on a single type of feature to encompass all vulnerabilities is impractical. Certain vulnerabilities arise from specific patterns of operation sequencing, while others hinge on the manipulation of data values, and still others have various root causes. To effectively address the most commonly seen vulnerability types, we narrow our focus to vulnerabilities associated with operation sequencing. This category includes common vulnerabilities such as command injection, format string, and memory leaks. As a result, we choose to extract sequences of operators and function calls from the traces. This method preserves rich semantic and structural details by capturing the logical order of operations within the code.

Figure 2 provides an overview of OpSMatcher. It comprises three main components: Known Vulnerability Processing, Target Code Processing, and Signature Matching. Known Vulnerability Processing examines code known to contain vulnerabilities and generates signatures to represent them. Target Code Processing analyzes the target code and generates signatures for any suspicious traces it identifies. Finally, Signature Matching compares the target signature with the vulnerability signature to identify recurring vulnerabilities.

Algorithm 1: Extracting Vulnerable Traces

```
/* DIP_i&DEP_i are from CVE reports. */
Input: {DIP_i, DEP_i | i = 1,...,n}, SourceCode
Output: Traces
Traces ← φ
CPG ← Joern.BuildCPG(SourceCode)
for i = 1 : n do
    t ← TraceQuery(CPG, DIP_i, DEP_i)
    Traces ← Traces ∪ t
```

Algorithm 2: Extracting Target Traces

```
/* DIP_i&DEP_j are predefined. */
Input: {DIP_i | i = 1,...,m}, {DEP_j | j = 1,...,n},
       SourceCode
Output: Traces
Traces ← φ
CPG ← Joern.BuildCPG(SourceCode)
for i = 1 : m do
    for j = 1 : n do
        t ← TraceQuery(CPG, DIP_i, DEP_j)
        Traces ← Traces ∪ t
```

Algorithm 3: Generating Signatures

```
/* traces are from Alg. 1 or 2. */
trace: [cpgNode_0, cpgNode_1, ..., cpgNode_n]
signature ← []
for item in trace do
    if typeof(item) == ID then
        /* Collect ops using only the ID
           but no propagation */
        op ← queryJoernforOp(item)
    else
        op ← item.op
    if isSemantic(op) then
        signature.append(op)
```

A. Known Vulnerability Processing

OpSMatcher extensively leverages a generic and well-developed taint analysis engine, Joern [14], for trace extrac-

tion. Joern is known for constructing a *code property graph* (CPG) directly from source code, upon which it conducts trace analysis. To capture the logic behind data propagation, Joern extracts traces from a data initialization point (DIP) to a data egress point (DEP) using taint analysis. A DIP is defined as a line of code responsible for receiving data from external sources or initializing a variable, while a DEP refers to a line of code associated with a security-sensitive function or the final operation involving a variable. The inclusion of variable operations in the signatures is critical for OPSMATCHER, as many patches involve checking variable values.

In the Known Vulnerability Processing phase, OPSMATCHER's workflow comprises two key steps: (1) extracting vulnerable traces and (2) generating vulnerability signatures.

1) Extracting Vulnerable Traces: DIPs and DEPs are identified manually from the Common Vulnerabilities and Exposures (CVE) database [18]. Algorithm 1 depicted OPSMATCHER's procedure to utilize these DIPs and DEPs. OPSMATCHER uses Joern to trace from DIP to DEP, such that these traces represent the root cause of vulnerabilities. However, the extracted traces include details such as assignments and callsites of tainted data, which are not yet suitable for our purposes. Subsequent processing is thus imperative, guiding OPSMATCHER to the next step.

2) Generating Vulnerability Signatures: As mentioned before, our objective is to characterize the root causes of vulnerabilities through a sequence of actions and functions involved. As Algorithm 3 shows, in the signature generation process, OPSMATCHER utilizes traces obtained from Joern, iterates through each node in the trace to extract representing information. If a node solely records the callsite of tainted data, additional extraction is performed to obtain associated operations from the node using Joern. These operations are not explicitly included in the traces by Joern as they solely manipulate the vulnerable data without propagating it further. Nonetheless, these actions play a pivotal role in representing the functionality of the code. They may prove instrumental in rectifying vulnerabilities by facilitating value checks or serving as integral components of critical functions.

For steps where operations can be directly returned by Joern, OPSMATCHER extracts them directly. Before incorporating them into the signature, we will check if they are semantic. Operations like assignment, access, and casting operators would be ignored because they are less relevant to the semantics of traces. TRACER also does not consider them features. This approach allows OPSMATCHER to adapt Joern's traces to the required format for representing a signature.

As an illustration, Figure 1b shows the vulnerable trace and signature of improper resource shutdown by OPSMATCHER. To derive the vulnerability signature, operators and function calls are sequentially extracted from the trace. In non-propagating operations such as `notEquals`, the associated nodes only record the tainted data. Consequently, it is necessary to enrich the trace by adding the detailed operations on the corresponding nodes to generate the signature, that is, `data` to `notEquals`. In contrast, non-semantic operations such as `(File *)` would be ignored.

B. Target Code Processing

Given a repository of known vulnerability signatures, OPSMATCHER in the Target Code Processing phase similarly examines the target code and produces signatures for any potentially suspicious traces within it.

1) Extracting Target Traces: OPSMATCHER meticulously identifies potential vulnerable traces within source code by employing a set of predefined DIPs and DEPs. Using Algorithm 2, it explores all possible combinations of DIPs and DEPs to extract candidate vulnerable traces as target traces. However, not all DIP-DEP pairings yield valid candidates, particularly if the DEP precedes the DIP. In such instances, OPSMATCHER relies on Joern's implementation to handle these complexities.

```
1 void Double_Free(){
2     char* ptr = malloc(0x100);
3     if (ptr == NULL){ exit(-1); }
4     free(ptr);
5     free(ptr);
6 }
```

(a) source code

```
[2 - malloc, 3 - ptr, 4 - free]
                ↓
      [malloc, equals, free]
```

(b) target trace and signature 1

```
[2 - malloc, 3 - ptr, 4 - free, 5 - free]
                ↓
      [malloc, equals, free, free]
```

(c) target trace and signature 2

Fig. 3: Juliet test suite (CWE-415: Double free)

In the scenario depicted in Figure 3, the functions *malloc()* and *free()* are included in the predefined list of DIPs and DEPs, respectively. OPSMATCHER identifies occurrences of these function calls and employs Joern to retrieve traces spanning from the DIP to the DEP. Consequently, OPSMATCHER produces two distinct traces as illustrated in Figures 3b and 3c. These traces depict the sequence of operations related to memory allocation and deallocation in the source code, serving as candidate vulnerable traces.

2) Generating Target Signatures: To facilitate the comparison between the target signatures and the vulnerability signatures, it is necessary to enrich the target traces obtained from Joern with additional operation details. To accomplish this, OPSMATCHER employs Algorithm 3 to process these candidate traces and generate target signatures. Subsequently, these target signatures are compared with the vulnerability signatures stored in the database. For example, Figures 3b and

3c depict the corresponding target signatures extracted from the traces after the enrichment process.

C. Signature Matching

After obtaining all the target signatures, OPSMATCHER attempts to match them against the vulnerability signatures of known CVEs. OPSMATCHER employs the Gestalt Pattern Matching algorithm [15], with the longest common subsequence (LCS) as a similarity measure. However, challenges arise when the analyzed code includes patches intended to fix previously reported vulnerabilities. These patches may change only a few lines of code, resulting in nearly identical code before and after the changes. Consequently, the corresponding signatures exhibit high similarity. The minimal changes introduced by patches do not significantly affect the similarity scores, potentially leading to misleading matches. To address this issue and prevent reporting patched sections as vulnerabilities, we have integrated a patch filtering phase into the process.

1) Similarity Computation: Similarity Computation involves assessing the similarity between a target signature and a vulnerability signature. Given that our signatures are represented as sequences, we utilize GPM based on the LCS as an effective similarity measure. Using LCS, rather than substrings as in the original GPM, mitigates the impact of intervening elements and captures all ordered common elements between two sequences. This method allows us to identify the ordered common operations between two traces while ignoring operations irrelevant to the tainted data.

To implement this, we first determine the LCS between the target signature and the vulnerability signature. We then compute the similarity using GPM. Let t and v denote the target signature and vulnerability signature, respectively, and let $LCS()$ be the function that obtains the LCS of two sequences. The similarity via GPM is calculated as follows:

$$Sim(t, v) = \frac{2 \times |LCS(t, v)|}{|t| + |v|}, \tag{1}$$

where $|\cdot|$ indicates the length of the sub-sequence or string.

This ensures a balanced comparison by considering the LCS length relative to the lengths of the original sequences.

After computing similarity, it is necessary to evaluate whether two signatures are similar enough to identify a recurring vulnerability that shares the same logic as the original one. This requires setting a threshold to effectively distinguish the degree of likeness between the two signatures, even when they are short. This is important because, with short signatures, small changes in the length of their LCS can significantly impact the similarity measure.

Consequently, we conducted a small-scale experiment (§IV-A3), whose result suggests a threshold between 0.85 and 0.95 for effectively distinguishing similar signatures, particularly for short ones. If the similarity score falls below the threshold, we conclude that the target signature does not represent a recurrence of the vulnerability, while scores above prompt further verification in the Patch Filtering phase.

Take the target trace from Figure 3c as an example, its signature ($[malloc, equals, free, free]$) is compared with the signatures of double-free ($[malloc, equals, free, free]$) and use-after-free ($[malloc, equals, free, printf]$) in the vulnerability database. Using GPM, we compute their corresponding similarity as follows:

$$Sim(t, df) = \frac{2 \times |[malloc, equals, free, free]|}{4 + 4} = 1.000 \tag{2}$$

$$Sim(t, uaf) = \frac{2 \times |[malloc, equals, free]|}{4 + 4} = 0.750 \tag{3}$$

By above, we determine that this target signature is closer to the vulnerability signature of double-free, despite the high similarity between the two vulnerability signatures. Based on the established threshold, we can conclude that it is more likely to be a recurring vulnerability of double-free.

2) Patch Filtering: Patch Filtering involves identifying and removing target signatures that appear to have been patched. Some patches might only make small changes to the original code, so the vulnerable code and patched code, along with their traces, remain very similar. For example, some patches might add a check to assert the variables, include a necessary operation for safety, or remove a problematic operation. There are different ways to patch different vulnerabilities, so we create filtering rules for each type of vulnerability. To create these rules, we look at both the patch methods provided by the Juliet test suite and previous research.

We employ simple, vulnerability-specific filtering rules to identify recurring vulnerabilities efficiently. If a target signature resembles a known vulnerability but fails to meet the corresponding filtering criteria, we assume it has been patched and disregard it. Conversely, signatures that pass these filters are flagged as potential recurring vulnerabilities for further analysis. For instance, the filtering rule for use-after-free vulnerabilities requires operations on deallocated memory. Thus, a signature resembling a use-after-free vulnerability without such operations is considered patched and excluded from further analysis.

For example, despite the high similarity score of 0.857 observed between the target signature in Figure 3b and the vulnerability signature ($[malloc, equals, free, free]$) of double-free, the former should be excluded by a filtering rule specific to double-free vulnerabilities, which requires more than one deallocating operation for the tainted data. This exclusion is warranted because the target signature contains only a single deallocating operation, *free*. In contrast, the target signature in Figure 3c should be considered indicative of a recurring double-free vulnerability, as it satisfies the filtering rule by including two deallocating operations, *free*. Consider another scenario exemplifying the efficacy of patch filtering, involving a use-after-free vulnerability, as depicted in Figure 4. Despite a high similarity score exceeding the threshold (0.857) between the trace from *malloc* to *free*, patch filtering incorporates a

rule mandating the presence of a *printf* call subsequent to the vulnerable code sequence.

```
int main() {
    int * x = NULL;
    x = (int *)malloc(5*sizeof(int));
    if (x == NULL){ exit(-1); }
    for(size_t i=0; i < 5; i++)
        x[i] = 5;
    free(x);
}
```

Fig. 4: A demo c program without Use After Free

IV. Experimental Results

This section presents the results of our comprehensive evaluation designed to address four key research questions. First, we investigate the effectiveness of OpSMatcher in detecting recurring vulnerabilities across various types (RQ1). Second, we compare OpSMatcher with the state-of-the-art tool, TRACER, to assess its relative efficacy (RQ2). Third, we examine the scalability of OpSMatcher when applied to large-scale projects, evaluating its execution time in more extensive environments (RQ3). Finally, we evaluate the effectiveness of OpSMatcher's Patch Filtering method, determining its impact on the overall performance and accuracy of the vulnerability detection process (RQ4).

A. Experimental Settings

We implemented OpSMatcher in Python and utilized Joern version 1.1.1641 for our analysis. TRACER was downloaded from its GitHub repository and configured without any structural modifications, aside from a minor adjustment to its signature database. All experiments were conducted on a machine running Ubuntu 20.04.6 LTS, equipped with an AMD Ryzen™ 9 5950X processor.

1) Dataset: Our experiment uses two collections of source code files written in C/C++. The first collection is the Juliet test suite, a comprehensive set of samples that includes both vulnerable functions and their corresponding patched versions, providing a reliable ground truth for evaluation. We focused on 24 types of vulnerabilities, including 7 types natively supported by TRACER and 17 additional types that we extended support for. Each vulnerability type in the Juliet test suite is represented by multiple variants exhibiting the same underlying issue. From each vulnerability type, we selected a pivot example to extract vulnerability signatures, which served as the known vulnerability signatures. The remaining 5,598 samples, covering all 24 types, are referred to as the Juliet test suite for performance evaluation. Among 24 types, TRACER natively supports the following vulnerabilities: integer overflow, integer underflow, buffer overflow, command injection, format string, use-after-free, and double-free.

We also evaluate 233 open-source Debian packages (Debian suite). For RQ2, packages were selected if they contained TRACER-detected vulnerabilities and excluded if they caused build errors in TRACER, resulting in a final selection of 24 Debian packages. Unlike the Juliet test suite, the Debian suite lacks ground truth (e.g., vulnerable functions, patches), limiting comprehensive evaluation. However, it offers insights into TRACER's real-world capabilities.

2) Evaluation Criteria: In the Juliet test suite, the ground truth is established using traces extracted from given packages. Let's denote the DIP-DEP pairs associated with these traces as $G = \{(DIP_i^g, DEP_i^g) | i = 1, \ldots, n\}$, where n is the total number of ground truth traces. For the evaluation package, a successful match is identified by finding a signature that matches a known vulnerable signature, surpassing a predetermined threshold. Let's denote the DIP-DEP pairs associated with these detected vulnerabilities as $D = \{(DIP_j^d, DEP_j^d) | j = 1, \ldots, m\}$, where m is the total number of detected vulnerabilities.

A DIP-DEP pair (DIP_j^d, DEP_j^d) is considered a true positive (TP) if and only if there exists a DIP-DEP pair (DIP_i^g, DEP_i^g) in G such that: $DIP_i^g == DIP_j^d$ and $DEP_i^g == DEP_j^d$. Conversely, a DIP-DEP pair (DIP_j^d, DEP_j^d) is considered a false positive (FP) if it does not match any pairs in G. Additionally, a DIP-DEP pair (DIP_j^g, DEP_j^g) is considered a false negative (FN) if there is no corresponding DIP-DEP pair (DIP_i^d, DEP_i^d) in D such that: $DIP_i^g == DIP_j^d$ and $DEP_i^g == DEP_j^d$.

These definitions can be extended to the source-sink pairs as defined in TRACER. In this context, sources are a subset of DIPs that exclude variable declarations, and sinks are a subset of DEPs that exclude variable destructions. For the sake of brevity, the detailed source-sink-based definitions of true positives, false positives, and false negatives are omitted.

In this study, we utilize precision and recall as the primary metrics for evaluating the performance of our detection system. These metrics are essential for determining the system's effectiveness in correctly identifying true vulnerabilities while minimizing false detections.

Precision assesses the accuracy of the detection system. It is defined as the ratio of TP to the total number of positive detections (the sum of TP and FP). Mathematically, precision is represented as: $Precision = \frac{TP}{TP+FP}$.

Recall evaluates the detection system's ability to identify all actual vulnerabilities. It is defined as the ratio of TP to the total number of actual vulnerabilities (the sum of TP and FN). Mathematically, recall is represented as: $Recall = \frac{TP}{TP+FN}$.

3) Parameter Tuning: As OpSMatcher's vulnerability detection is determined by a predefined similarity threshold, the choice of this threshold would impact OpSMatcher's performance. So we carefully selected a threshold value that maximizes the F1 score, the harmonic mean of precision and recall, on a sampled dataset. Mathematically, the F1 score is represented as: F1-score $= \frac{precision \times recall}{precision + recall}$.

Our experiment on the Juliet test suite revealed OpSMatcher's performance across different thresholds (Fig-

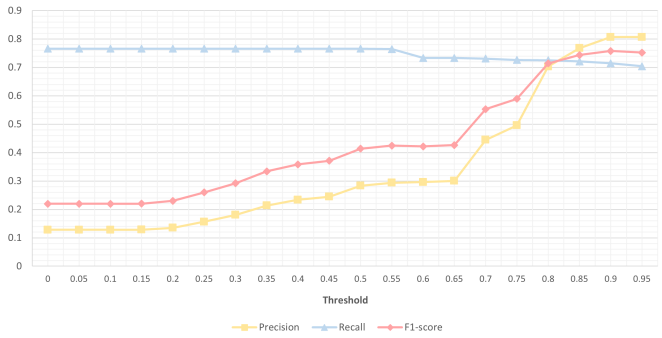

Fig. 5: Performance under different thresholds

ure 5). OPSMATCHER achieves near-optimal F1 scores with thresholds between 0.85 and 0.95, confirming its effectiveness in detecting recurring vulnerabilities by focusing on highly similar signatures. We recommend this range for balancing precision and recall. For a fair comparison with TRACER, we evaluate OPSMATCHER at a threshold of 0.85.

B. RQ1: Effectiveness

To address RQ1, we evaluated OPSMATCHER in detecting recurring vulnerabilities across various types. We considered all 24 vulnerability types (referred to as "All") and specifically examined OPSMATCHER's performance on types not supported by prior work. We labeled the vulnerability types supported by TRACER as "original" and those exclusively supported by OPSMATCHER as "extended." We applied OPSMATCHER to detect the recurring vulnerabilities in both the Juliet test suite and the Debian suite.

For the Juliet test suite, the performance evaluation is based on the precision and recall (§IV-A2). For the Debian suite, as there is no ground truth available, we manually inspected the reported traces to determine if they were indeed vulnerable. Consequently, the number of false negatives is not available for the Debian suite; therefore, we only report the precision of the detection.

Target	Type	TP	FP	FN	Prec	Recall	Time
Juliet	All	4035	1222	1563	0.768	0.721	52 s
	Orig.	1661	342	570	0.829	0.745	-
	Ext.	2374	880	993	0.730	0.705	-
Debian	All	14	5	-	0.667	-	5819 s
	Orig.	6	4	-	0.429	-	-
	Ext.	8	1	-	0.875	-	-

TABLE I: Effectiveness of OpSMatcher

Table I summarizes the results, including the number of true positives, false positives, false negatives, precision, recall, and average execution time. On the Juliet test suite, which provides ground truth information for assessing false negatives, OPSMATCHER achieved a precision of 0.768 and a recall of 0.721. For the original vulnerability types, OPSMATCHER attained a precision of 0.829 and a recall of 0.745. For the extended vulnerability types, OPSMATCHER achieved a precision of 0.730 and a recall of 0.705. These results show that OPSMATCHER can effectively detect recurring vulnerabilities across various types with high precision and recall.

Package	Num	Type
abyss 2.2.4	2	Memory Leak
anthy 0.4	1	Double Free
antimony 0.9.3	1	Mismatched Memory Management Routines
ascd 0.13.2	2	Integer Overflow to Buffer Overflow
gap-guava 3.15	2	Integer Overflow to Buffer Overflow
	1	Memory Leak
htmldoc 1.9.7	2	Duplicate Operations on Resource
rlwrap 0.43	2	Memory Leak
sweed 3.2.1	1	Integer Overflow to Buffer Overflow

TABLE II: List of vulnerabilities detected by OPSMATCHER

In the Debian suite, which resembles a real-world scenario, OPSMATCHER detected 14 recurring vulnerabilities, as shown in Table II, with an overall precision of 0.667 across all 24 vulnerability types. Notably, eight of these recurring vulnerabilities belong to the extended type, which TRACER cannot detect. Furthermore, to our knowledge, all of these newly detected vulnerabilities had never been reported before. These findings highlight OPSMATCHER's capability to detect a wide range of vulnerabilities with notable precision and recall, particularly extending the detection coverage beyond what previous tools like TRACER could achieve.

Vulnerability Type	Prec	Recall
Relative Path Traversal	1.000	0.517
Absolute Path Traversal	1.000	0.504
OS Command Injection	1.000	0.621
LDAP Injection	1.000	0.658
Process Control	1.000	0.836
Uncontrolled Format String	0.615	0.868
Integer Overflow	0.847	0.883
Integer Underflow	1.000	0.880
Unexpected Sign Extension	1.000	0.867
Signed to Unsigned Conversion Error	1.000	0.845
Heap Inspection	0.162	1.000
Plaintext Storage of Password	0.405	0.882
Divide by Zero	0.619	0.871
Resource Exhaustion	1.000	0.896
Memory Leak	0.375	0.723
Improper Resource Shutdown	0.528	0.571
Double Free	1.000	0.533
Use After Free	0.371	0.214
Uncontrolled Search Path Element	1.000	0.808
Free Memory Not on Heap	0.631	0.820
Reachable Assertion	1.000	0.865
Duplicate Operations on Resource	0.529	0.789
Integer Overflow to Buffer Overflow	1.000	0.793
Mismatched Memory Management Routines	0.944	0.666
Total	0.768	0.721

TABLE III: Performance statistics for each vulnerability type

Additionally, Table III illustrates the precision and recall for each vulnerability type. These results demonstrate that OPSMATCHER can effectively detect vulnerable functions across a majority of the vulnerability types. This comprehensive coverage underscores OPSMATCHER's robustness and reliability in identifying vulnerabilities, thereby offering a significant advantage over existing tools.

While OPSMATCHER effectively detects recurring vulnerabilities, it is not immune to FPs. Upon analyzing detection results for real-world projects, we identified that many FPs stem from illogical traces extracted by Joern. For instance, Figure 6 shows a code segment from gap-guava-3.15+ds,

where Joern detects illogical traces. Specifically, Joern's taint analysis engine extracts traces from `fscanf` in line 7 to `malloc` in both line 2 and 4. However, these traces are deemed unreasonable as they assign values to memory before allocation, contradicting expected program behavior. Moreover, there is no feasible execution path to justify such traces.

```
1 int generator_matrix(..., MATRIX *M){
2     M->m = (int **)malloc(...);
3     for (i=0; i<M->rows; i++)
4         M->m[i] = (int *)malloc(...);
5     for (i=0; i<M->rows; i++)
6         for (j=0; j<M->cols; j++)
7             fscanf(..., &M->m[i][j]);
8 }
```

Fig. 6: Illogical traces in gap-guava-3.15+ds

C. RQ2: Comparison with TRACER

To ensure fair comparison between OPSMATCHER and TRACER, we evaluate on the 7 vulnerability types supported by TRACER, a subset of OPSMATCHER's vulnerability database. Additionally, we employ the same base samples from the Juliet test suite to extract signatures in both OPSMATCHER and TRACER. Target codes are taken from the Juliet test suite, serving as the ground truth for detecting and focusing on the correctness of reported DIPs and DEPs.

Table IV reveals that OPSMATCHER exhibits superior precision and recall compared to TRACER, albeit with lower efficiency. For ground truth, OPSMATCHER achieves a precision of 0.829 and a recall of 0.745. In contrast, TRACER achieves a precision of 0.744 and a recall of 0.321, performing approximately 50 times faster than OPSMATCHER.

Further analysis by vulnerability type indicates that OPSMATCHER achieves higher precision in almost all vulnerability types except for format string, while exhibiting lower recall only in command injection and format string. Notably, OPSMATCHER achieves a precision of 1.000 in integer underflow, buffer overflow, command injection, and double free, whereas TRACER achieves over 0.9 precision and recall only in command injection and format string. For real-world projects, OPSMATCHER reports 10 true positives and 5 false positives, with an average execution time of 5.8 hours. In contrast, TRACER reports 13 true positives and 4 false positives, with an average execution time of 6.7 minutes.

Our observations reveal several factors contributing to these differences. Firstly, the Debian packages used in this experiment have been previously analyzed by TRACER, potentially biasing the selection towards packages where TRACER performs well. Secondly, Infer, the tool utilized by TRACER, boasts powerful engines for detecting potential vulnerabilities, enhancing its efficacy in identifying traces flagged by Infer as potential vulnerabilities. On the other hand, OPSMATCHER extracts all traces given common DIPs and DEPs in vulnerabilities, without employing specialized detection engines for each vulnerability type. This approach results in longer extraction times and higher false positives and false negatives for OPSMATCHER, despite incorporating more semantic and structural features. Furthermore, false negatives observed in OPSMATCHER can be attributed to limitations in Joern, the taint analysis engine used. Notably, Joern struggles with analyzing pointers successfully, leading to the omission of certain traces found by TRACER.

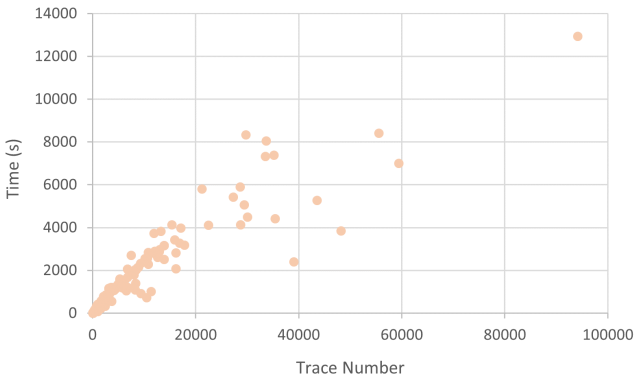

Fig. 7: Distribution of Execution Time of OPSMATCHER

We also conduct experiments comparing feature selection in OPSMATCHER and TRACER. The results reveal that OPSMATCHER's operation sequence outperforms TRACER's operation frequency. With a threshold of 0.85, OPSMATCHER achieves a precision of 0.768, while TRACER's feature yields only 0.295 (Table V). This highlights the operation sequence's superiority in representing trace semantics and structure.

D. RQ3: Scalability

We evaluate OPSMATCHER's scalability, expanding on the effectiveness assessment of Debian packages in Section IV-B. OPSMATCHER's average execution time of 5819 seconds per Debian package (Table I) indicates significant time investment for large-scale projects.

To ascertain the reasons behind the prolonged execution time, we delve into the time distribution of each step. Notably, we discover that Feature Extraction consumes 98% of the average time, equivalent to 5696 seconds. During Feature Extraction, the process of collecting operations solely utilizing taint data without propagating it necessitates continuous engagement with Joern to retrieve pertinent information. However, leveraging Joern entails the overhead of building the Joern server, sending requests, and awaiting responses, leading to considerable time consumption, particularly when processing features for numerous traces in larger projects.

Figure 7 provides insight into the relationship between execution time and the number of traces for Debian packages. Evidently, as OPSMATCHER extracts more traces, the total execution time proportionally increases. Consequently, our

TABLE IV: Comparison of OPSMATCHER and TRACER

Target	Vulnerability Type	OpSMatcher						TRACER					
		TP	FP	FN	Prec	Recall	Time	TP	FP	FN	Prec	Recall	Time
Juliet	OS Command Injection	187	0	114	1.000	0.621	-	301	11	0	0.965	1.000	-
	Uncontrolled Format String	310	194	47	0.615	0.868	-	357	24	0	0.937	1.000	-
	Integer Overflow	511	92	68	0.847	0.883	-	0	0	579	0.000	0.000	-
	Integer Underflow	383	0	52	1.000	0.880	-	0	0	435	0.000	0.000	-
	Double Free	172	0	151	1.000	0.533	-	59	97	264	0.378	0.183	-
	Use After Free	33	56	121	0.371	0.214	-	0	30	154	0.000	0.000	-
	Integer Overflow to Buffer Overflow	65	0	17	1.000	0.793	-	0	85	82	0.000	0.000	-
	Total	1661	342	570	**0.829**	**0.745**	54 s	717	247	1514	0.744	0.321	**0.94 s**
Debian	Original	3	4	-	0.429	-	-	13	4	-	0.765	-	-
	Extended	7	1	-	0.875	-	-	-	-	-	-	-	-
	Total	10	5	-	0.667	-	21043 s	13	4	-	0.765	-	404 s

Feature	PF	Type	TP	FP	FN	Prec	Recall
Sequence	w/	All	4035	1222	1563	0.768	0.721
		Orig.	1661	342	570	0.829	0.745
		Ext.	2374	880	993	0.730	0.705
	w/o	All	4059	4776	1539	0.459	0.725
		Orig.	1679	2372	552	0.414	0.753
		Ext.	2380	2404	987	0.497	0.707
Frequency	-	All	4060	9700	1538	0.295	0.725
		Orig.	1664	4327	567	0.278	0.746
		Ext.	2396	5373	971	0.308	0.712

TABLE V: Effect of Feature Selection & Patch Filtering (PF)

future focus lies in enhancing the scalability of OPSMATCHER by mitigating the execution time of feature extraction.

E. RQ4: Effectiveness of Patch Filtering

Experiments comparing OPSMATCHER's performance with and without patch filtering show its efficacy in reducing false positives. With patch filtering, it reported 1,222 false positives (precision: 0.768); without it, 4,776 false positives (precision: 0.459). This represents a substantial precision improvement of about 0.3, confirming patch filtering's effectiveness in discerning and eliminating patched target signatures that would otherwise be misidentified as vulnerabilities (Table V).

V. CONCLUSION

This paper introduces OPSMATCHER, a versatile taint-analysis-based framework designed for recurring vulnerability detection. By leveraging a robust taint analysis engine, OPSMATCHER extracts a comprehensive set of traces, enabling the detection of a wide range of vulnerability types. Moreover, OPSMATCHER excels in capturing semantic and structural features from the code, enhancing its effectiveness in vulnerability detection. Our empirical study underscores OPSMATCHER's efficacy, demonstrating its ability to achieve high precision and recall rates. OPSMATCHER successfully identifies previously unknown recurring vulnerabilities across various real-world projects, showcasing its practical applicability and effectiveness. We envision OPSMATCHER as a valuable tool for developers, aiding in proactive vulnerability detection during development and reducing the risk of security breaches and financial losses.

ACKNOWLEDGEMENT

This research was supported in part by the National Science and Technology Council (NSTC) of Taiwan under grants 112-2223-E-002-010-MY4, 113-2634-F-002-001-MBK, National Taiwan University under grant NTU-113L7871, and the Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Scientific Research (C) 22K12038, JAPAN.

REFERENCES

[1] W. Kang, B. Son, and K. Heo, "Tracer: Signature-based static analysis for detecting recurring vulnerabilities," in *ACM CCS*, 2022.

[2] "Open source security and risk analysis report," 2023. [Online]. Available: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html

[3] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *IEEE/ACM ASE*, 2010.

[4] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *IEEE S&P*, 2015.

[5] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in *ACSAC*, 2016.

[6] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *IEEE S&P*, 2017.

[7] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *ASPLOS*, 2018.

[8] H. Jang, K. Yang, G. Lee, Y. Na, J. D. Seideman, S. Luo, H. Lee, and S. Dietrich, "Quickbcc: Quick and scalable binary vulnerable code clone detection," in *ICT Systems Security and Privacy Protection*, 2021.

[9] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *ACSAC*, 2014.

[10] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla *et al.*, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[11] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures," in *USENIX*, 2020.

[12] Y. Xiao, Z. Xu, W. Zhang, C. Yu, L. Liu, W. Zou, Z. Yuan, Y. Liu, A. Piao, and W. Huo, "Viva: Binary level vulnerability identification via partial signature," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021.

[13] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465.

[14] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *IEEE S&P*, 2014.

[15] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.

[16] "Opsmatcher," 2024. [Online]. Available: https://github.com/csienslab/OpSMatcher

[17] P. E. Black and P. E. Black, *Juliet 1.3 test suite: Changes from 1.2*. US Department of Commerce, National Institute of Standards and Technology, 2018.

[18] "Common vulnerabilities and exposures," 2023. [Online]. Available: https://cve.mitre.org/index.html