TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities

Wooseok Kang
KAIST
Korea
kangwoosukeq@kaist.ac.kr

Byoungho Son*
POSTECH
Korea
byhoson@postech.ac.kr

Kihong Heo
KAIST
Korea
kihong.heo@kaist.ac.kr

ABSTRACT

Similar software vulnerabilities recur because developers reuse existing vulnerable code, or make similar mistakes when implementing the same logic. Recently, various analysis techniques have been proposed to find *syntactically* recurring vulnerabilities via code reuse. However, limited attention has been devoted to *semantically* recurring ones that share the same vulnerable behavior in different code structures. In this paper, we present a general analysis framework, called TRACER, for detecting such recurring vulnerabilities. The main idea is to represent vulnerability signatures as traces over interprocedural data dependencies. TRACER is based on a taint analysis that can detect various types of vulnerabilities. For a given set of *known* vulnerabilities, the taint analysis extracts vulnerable traces and establishes a signature database of them. When a *new unseen* program is analyzed, TRACER compares all potentially vulnerable traces reported by the analysis with the known vulnerability signatures. Then, TRACER reports a list of potential vulnerabilities ranked by the similarity score. We evaluate TRACER on 273 Debian packages in C/C++. Our experiment results demonstrate that TRACER is able to find 112 previously unknown vulnerabilities with 6 CVE identifiers assigned.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Software security engineering**.

KEYWORDS

software security, program analysis, software engineering

ACM Reference Format:
Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560664

*This work was done during internship at KAIST.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560664

1 INTRODUCTION

Similar software vulnerabilities recur over time even across programs. One of the well-known reasons is the prevalence of code reuse [23, 26, 33, 34, 47] that can lead to the spread of security vulnerabilities in the reused code. In addition to such *syntactic* recurrences, *semantically* similar vulnerabilities frequently recur in unrelated codebases that are independently developed. One of the reasons is that developers often make similar mistakes when implementing the same standard concepts such as mathematical formulas, laws of physics, protocols, or language interpreters [37, 42]. Another reason is common misconceptions due to complicated low-level semantics of programming languages such as undefined behaviors in C [14]. They can induce developers to write incorrect code with similar error patterns. According to a recent report from Google, 6 out of 24 0-day vulnerabilities in 2020 were actually variants of previously seen ones [42].

Although researchers have developed many successful techniques to detect recurring security vulnerabilities, existing approaches have limitations in several aspects. Approaches based on code similarity [12, 15, 23, 26, 29, 38, 47] aim at detecting recurring vulnerabilities via code reuse. They generate signatures of known vulnerabilities within a pre-defined boundary (e.g., file or function) and compare syntactic patterns in a new program with the signatures. These approaches are highly precise, scalable and general as their approaches are based on syntactic matching. However, they are usually unable to detect variants of known vulnerabilities with completely different syntactic structures but with the same root causes. On the other hand, pattern-based static analyses [2, 3, 18] estimate the semantics of target programs as well as consider their syntactic patterns. This in turn enables the analyzer to detect vulnerabilities that have similar syntactic and semantic characteristics of programs with known vulnerabilities. However, designing such analyses requires static analysis expertise and incurs a nontrivial engineering burden.

To address this problem, we set out to build an effective *software immune system* against recurring vulnerabilities. We identified the following criteria to be satisfied for such a system:

- *Accuracy*: Does the system accurately report potential vulnerabilities with a low false positive rate?
- *Robustness*: Is the system able to find variants of vulnerabilities that have the same root cause?
- *Generality*: Is the system applicable to a wide range of security bugs?
- *Scalability*: Is the system applicable to large programs?
- *Usability*: Does the system provide easily interpretable reports?

In this paper, we present a signature-based static analysis for detecting recurring vulnerabilities, TRACER, that is designed to satisfy

the above criteria. The key idea is to represent vulnerability signatures as traces over interprocedural data dependencies. Tracer is based on a general taint analysis that aims at a variety of security vulnerabilities such as integer overflow/underflow, format string, buffer overflow, command injection, etc. The analyzer detects potentially vulnerable data flows from untrusted inputs (so called, *source*) to security-sensitive functions (so called, *sink*). We run the static analyzer on a codebase with known vulnerabilities and identify the actual vulnerabilities in the analysis results. Next, Tracer extracts traces on the data dependency relations of the vulnerabilities from the source points to the sink points. The traces are encoded as feature vectors that form the signatures of the vulnerabilities. Once a new program is analyzed, Tracer extracts traces of all the reported alarms in the program, and derives their feature vectors in the same manner. Then, Tracer compares the feature vectors of the alarms with those of the known vulnerable traces using a typical similarity measure such as cosine similarity. Finally, Tracer provides a list of alarms sorted by similarity.

We implemented Tracer based on Facebook's Infer analyzer [5] and demonstrated the effectiveness on a suite of Debian packages written in C/C++. According to our experimental results on 273 Debian packages, Tracer discovered 112 recurring vulnerabilities that are similar to known CVEs, vulnerability examples in Juliet test suite [4], and sample code in online tutorials for secure coding.

This paper makes the following contributions:

- We propose a general analysis framework, Tracer, for detecting semantically recurring vulnerabilities. Tracer is applicable to a wide range of vulnerabilities.
- We present a trace-based method for computing the similarity of vulnerabilities. Our method is based on data dependencies of alarms reported by a general taint analysis.
- We evaluate the effectiveness of Tracer on 273 Debian packages. We found 112 vulnerabilities with 6 CVE identifiers assigned.

2 OVERVIEW

2.1 Motivating Examples

We illustrate our approach with the programs with security vulnerabilities in Figure 1. All three programs have similar issues related to a certain kind of security vulnerability: overflowed integers can be used as the size argument of memory allocation functions (e.g., `malloc`). Such integer overflows cause the program to unintentionally allocate small memory chunks, potentially leading to buffer overflows.

Figure 1(a) shows the vulnerability in an image processing tool gimp reported in 2009. The program reads a byte string from a given file (line 10), transforms the string into an integer (line 12). Since this value depends on the contents of the input file, the integer can be arbitrarily large. The integer value at line 13 can also become arbitrarily large because of the same reason. Then, the program multiplies the integers, leading to an integer overflow (line 14). Finally, the overflowed integer (`rowbytes`) is passed to the function `ReadImage` and used as an argument of `malloc` (line 21). Notice that the size of the allocated buffer can be much smaller than what the developer expected. Therefore, potential buffer overflows can happen when the buffer is used to store the data of the input file afterward.

```
1  gint32 ToL(guchar *puffer) {
2    return (puffer[0] | puffer[1] << 8 | puffer[2] << 16 | puffer[3] << 24);
3  }
4  gint16 ToS(guchar *puffer) { return (puffer[0] | puffer[1] << 8); }
5
6  gint32 ReadBMP(gchar *name) {
7    FILE *fd = fopen(name, "rb");
8    if (!fd) return -1;
9    // Read from a file
10   if (fread(buffer, Bitmap_File_Head.biSize - 4, fd) != 0)
11     return -1;
12   Bitmap_Head.biWidth = ToL(& buffer[0x00]);
13   Bitmap_Head.biBitCnt = ToS(& buffer[0x0A]);
14   rowbytes = ((Bitmap_Head.biWidth * Bitmap_Head.biBitCnt - 1) / 32) * 4 + 4;
15   image_ID = ReadImage(rowbytes);
16     ...
17 }
18
19 gint32 ReadImage(gint rowbytes) {
20     /* memory allocation with an overflowed size */
21     guchar *buffer = malloc(rowbytes);
22     /* uses of buffer */
23 }
```

(a) gimp-2.6.7 (CVE-2009-1570)

```
1  long ToL(unsigned char *puffer) {
2    return (puffer[0] | puffer[1] << 8 | puffer[2] << 16 | puffer[3] << 24);
3  }
4  short ToS(unsigned char *puffer) { return (puffer[0] | puffer[1] << 8); }
5
6  bitmap_type bmp_load_image(FILE *fd) {
7    if (fread(buffer, 18, fd) || (strncmp((const char *)buffer, "BM", 2)))
8      FATALP("BMP:_not_a_valid_BMP_file");
9    // Read from a file
10   if (fread(buffer, Bitmap_File_Head.biSize - 4, fd) != 0)
11     FATALP("BMP:_Error_reading_BMP_file_header_#3");
12   Bitmap_Head.biWidth = ToL(&buffer[0x00]);
13   Bitmap_Head.biBitCnt = ToS(&buffer[0x0A]);
14   rowbytes = ((Bitmap_Head.biWidth * Bitmap_Head.biBitCnt - 1) / 32) * 4 + 4;
15   image.bitmap = ReadImage(rowbytes);
16   ...
17 }
18
19 unsigned char *ReadImage(int rowbytes) {
20   /* memory allocation with an overflowed size */
21   unsigned char *buffer = (unsigned char *) new char[rowbytes];
22   /* uses of buffer */
23 }
```

(b) sam2p-0.49.4 (CVE-2017-16663)

```
1  XcursorBool _XcursorReadUInt(XcursorFile *file, XcursorUInt *u) {
2    unsigned char bytes[4];
3    if ((*file->read)(file, bytes, 4) != 4)   // Read from a file
4      return XcursorFalse;
5    *u = (bytes[0] | (bytes[1] << 8) | (bytes[2] << 16) | (bytes[3] << 24));
6    return XcursorTrue;
7  }
8
9  XcursorImage *_XcursorReadImage(XcursorFile *file) {
10   XcursorImage head;
11   XcursorImage *image;
12   if (!_XcursorReadUInt(file, &head.width)) return NULL;
13   if (!_XcursorReadUInt(file, &head.height)) return NULL;
14   image = XcursorImageCreate(head.width, head.height);
15   ...
16 }
17
18 XcursorImage *XcursorImageCreate(int width, int height) {
19   XcursorImage *image;
20   /* memory allocation with an overflowed size */
21   image = malloc(sizeof(XcursorImage) + width * height * sizeof(XcursorPixel));
22   /* initialize struct image */
23   return image;
24 }
```

(c) libXcursor-1.1.14 (CVE-2017-16612)

Figure 1: Examples code excerpted from similar vulnerabilities from different programs.

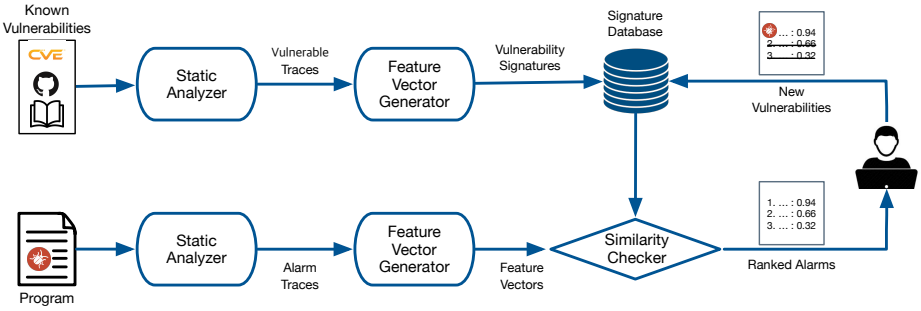


Figure 2: System overview of TRACER

After 8 years, a similar vulnerability was found in another program, sam2p depicted in Figure 1(b). sam2p is also an image processing tool, so it has a similar piece of code that reads a BMP file. Because of exactly the same reason as gimp, this program is also vulnerable. Notice that the code snippet is quite similar to that of gimp. Conceptually, existing methods based on code clone detection may help catch such recurring vulnerabilities given the vulnerability in gimp as a *signature*. However, it is sometimes challenging in practice. Clone-based approaches typically compare two pieces of code within a pre-defined syntactic boundary (e.g., functions or blocks). This in turn hinders vulnerability detection when vulnerable behavior involves multiple functions as in the examples. State-of-the-art tools [26, 47] heuristically choose a vulnerability signature function that contains the patches of the known vulnerability (`ReadBMP` in the gimp case). However, this is still fragile if the functions are large and contain considerable syntactic differences. For example, `ReadBMP` in gimp consists of 382 lines while `bmp_load_image` in samp2p has only 151 lines. Although the essence of the vulnerability is the same, they have many discrepancies in the other parts. For example, lines 7–8 in the two programs are completely different, and sam2p, which is a C++ program, uses `new` rather than `malloc`.

Moreover, recurring vulnerabilities are not always induced by code clones. Developers often make similar mistakes when they write programs that have typical or standard behavior both at a low level (e.g., reading data from files or allocating heap memory blocks) and at a high-level (e.g., calculating the area of a square or processing an image file). An example from libXcursor is shown in Figure 1(c). Similar to the previous examples, the program reads data from an input file (line 3), converts the input byte string to an integer (line 5), and computes the multiplication of two arbitrary large integers (line 21). The multiplication also leads to an integer overflow at the same line that can cause buffer overflows afterward. Notice that the root cause of the vulnerability is the same as the other examples. However, libXcursor has completely different syntactic structures. For example, libXcursor uses an indirect call to `fread` at line 3 while the other programs directly call the function.

Existing approaches are not appropriate to detect such *semantically* recurring vulnerabilities. Clone-based approaches [26, 47] are not effective to detect this vulnerability, given the vulnerability in gimp or sam2p as a signature. While the essence of the vulnerability is still the same, the different code structure of libXcursor fundamentally hinders the detectability of the tools. Static bug-finding tools that aim at general integer overflows may detect this vulnerability but also can incur many false positives. One can also design a specialized static analysis dedicated to each pattern. However, it would impose a high engineering burden while producing suboptimal solutions. For example, the TaintedAllocationSize checker from Github's CodeQL [8], which is a state-of-the-art pattern-based analyzer, does not detect the particular vulnerabilities in Figure 1.

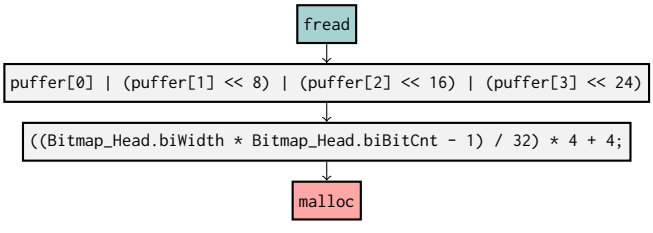
2.2 Our Approach

Now, we introduce how TRACER can detect recurring vulnerabilities. Our approach is shown in Figure 2. In the rest of this section, we explain the procedure of each component of TRACER and show the vulnerabilities in sam2p and libXcursor can be accurately detected by TRACER given the one in gimp as a signature.

2.2.1 Taint Analysis. TRACER is based on a generic taint analysis that can be instantiated to bug detectors for various types of security vulnerabilities. The analysis computes potential data flows from untrusted inputs (sources) to sensitive functions (sinks) with a simple abstract domain for tainted values: $\mathbb{T} = \{\bot_t, \top_t\}$ where each element denotes that the value is not tainted ($\bot_t$) and may be tainted ($\top_t$). For example, in Figure 1(a), the malicious data flow from `fread` to `malloc` is detected by the analyzer.

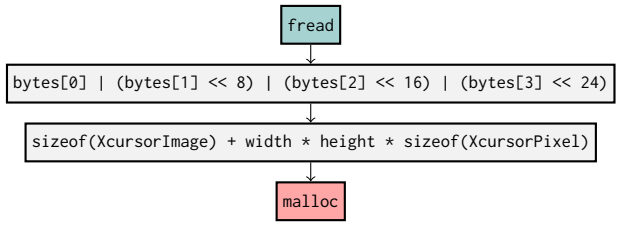
One may elaborate the analysis with other abstract domains along with the basic taint domain for a more accurate analysis. In our implementation, we have a simple abstract domain $\bar{\mathbb{I}} = \{\bot_o, \top_o\}$ for estimating whether an integer value is potentially overflowed ($\top_o$) or not ($\bot_o$). For example, an untrusted input value is initially tainted ($\top_t$) but not overflowed ($\bot_o$). Once the value is used as an operand of an operator that can potentially introduce integer overflow (e.g., +, <<), the result becomes tainted ($\top_t$) and overflowed ($\top_o$). For the `malloc` case, our analyzer raises an alarm only when the abstract value of the argument is both tainted ($\top_t$) and overflowed ($\top_o$). By doing so, we do not report trivial false alarms while efficiently computing malicious data flows. The details of our implementation are described in Section 4.

2.2.2 Traces on Data Dependency Graphs. We run the taint analysis on a given set of programs whose vulnerabilities are already known. For each known vulnerability, TRACER extracts vulnerable traces from the source and sink points based on the static analysis result. To filter out statements that are irrelevant to the vulnerability as much as possible, we derive vulnerable traces on data



⟨fread : 1, | : 3, << : 3, * : 2, + : 1, - : 1, / : 1, malloc : 1⟩

(a) gimp in Figure 1(a)



⟨fread : 1, | : 3, << : 3, * : 2, + : 1, - : 0, / : 0, malloc : 1⟩

(b) libXcursor in Figure 1(c)

Figure 3: Vulnerable traces and their feature vectors. The blue and red nodes represent the source and sink points, respectively.

dependency graphs rather than control-flow graphs. Once the taint analysis detects potentially malicious flows in gimp and libXcursor in Figure 1, Tracer derives data dependency graphs and extracts the vulnerable traces from the sources to sinks as shown in Figure 3. Such traces will be used as signatures of vulnerabilities.

The same procedure will be applied to new target programs. In particular, Tracer extracts all possible traces from sources to sinks of the reported alarms while unrolling each loop only once. These traces will be compared to the signature traces.

2.2.3 Feature Representation. Next, Tracer encodes each trace as an integer feature vector. We design a program-independent and common feature space that can represent transferable knowledge for vulnerabilities. Our feature vector consists of two parts: low-level and high-level features.

Low-level features represent the frequencies of primitive operators X (e.g., *, <<) and common APIs X (e.g., `strlen`) on the trace. Figure 3(a) shows the feature vector of the vulnerable trace in gimp. Likewise, the feature vector for libXcursor is shown in Figure 3(b).

On the other hand, high-level features describe detailed behavior of traces that are not noticeable using only the low-level ones. We manually designed 5 high-level features. In general, they characterize crucial behavior of programs that can affect our target vulnerabilities. For example, one of our features IfSmallerThanConst checks whether a trace has a conditional statement whose condition is of the form `x < c` where `x` is a variable and `c` is a constant. This pattern is common when programs prevent integer overflows. Suppose there exists such an expression in a trace of the target program, but not in the signature trace. Then, the target trace is deemed to be safe and the similarity score becomes lower.

Algorithm 1: Tracer($\Pi, \mathcal{A}, P$) where Π is a set of feature vectors of signature traces, $\mathcal{A}$ is a static analyzer, and P is the program to be analyzed.

1 $\Omega \leftarrow \mathcal{A}(P)$;
2 $G \leftarrow \text{build\_dfg}(P)$;
3 $R \leftarrow \emptyset$;
4 **for** $\omega \in \Omega$ **do**
5 $\quad \mathcal{T}_\omega \leftarrow \text{extract\_traces}(G, \omega)$;
6 $\quad \Pi_\omega \leftarrow \{\text{generate\_feature}(\tau) \mid \tau \in \mathcal{T}_\omega\}$;
7 $\quad s \leftarrow \max\{\text{Sim}(\pi_\omega, \pi) \mid \pi_\omega \in \Pi_\omega, \pi \in \Pi\}$;
8 $\quad R \leftarrow R\{\omega \mapsto s\}$;
9 **return** R;

$$\begin{array}{llll} \textit{Expression} & E & \rightarrow & n \mid x \mid E + E \mid E - E \mid \text{source}_l() \\ \textit{Command} & C & \rightarrow & x := E \mid \text{assume}(x < n) \mid \text{sink}(E) \end{array}$$

Figure 4: Language

2.2.4 Similarity Checking. Once a new program is analyzed, Tracer extracts all alarm traces and compares them against the known vulnerability signatures. Since all the traces are encoded as vectors, we can use any common similarity measures. In our implementation, we use cosine similarity, a well-known similarity measure for two vectors. For example, the cosine similarity of the two feature vectors in Figure 3 is computed as follows:

$$\frac{\langle 1,3,3,2,1,1,1,1\rangle \cdot \langle 1,3,3,3,2,1,0,0,1\rangle}{||\langle 1,3,3,2,1,1,1,1\rangle|| \; ||\langle 1,3,3,3,2,1,0,0,1\rangle||} = 0.96$$

Therefore, Tracer can precisely detect semantically recurring vulnerabilities with high similarity scores.

3 FRAMEWORK

In this section, we formalize our approach. The overall procedure of Tracer is described in Algorithm 1. Tracer first analyzes the target program and derives a set of alarms (line 1). Next, Tracer computes the data dependency graph of the program (line 2). For each alarm of the program, the algorithm extracts a set of traces (line 5) and encodes them as feature vectors (line 6). Finally, we compare each generated feature vector of the alarm ω with vulnerability signature traces. The score of the alarm is determined as the maximum similarity score of them (line 7). In the rest of this section, we formalize the details of each component of Tracer.

3.1 Program

A program is represented as a control flow graph $\langle \mathbb{C}, \rightarrow \rangle$ where $\mathbb{C}$ is the set of control points and $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is the control-flow relation. Each control point is associated with a command. We assume a simple imperative language defined in Figure 4[1]. An expression is an integer, variable, addition operation, subtraction operation, or call to a source function. A command is an assignment, assume, or call to a sink function. `source` and `sink` represent functions

[1]For brevity, we only consider this simple language but our implementation handles the full features of C/C++.

$$
\begin{array}{lrcl}
\text{(Abstract state)} & \mathbb{S} & = & \mathbb{C} \to \mathbb{M} \\
\text{(Abstract memory)} & \mathbb{M} & = & \mathbb{X} \to \mathbb{T} \times \mathbb{V} \\
\text{(Taint)} & \mathbb{T} & = & \wp(\mathbb{C})
\end{array}
$$

(a) Abstract domains

$$
\begin{array}{rcl}
[\![E]\!] & : & \mathbb{M} \to \mathbb{T} \times \mathbb{V} \\
[\![n]\!](m) & = & \langle \emptyset, \mathcal{V}(n)(m) \rangle \\
[\![x]\!](m) & = & m(x) \\
[\![E_1 + E_2]\!](m) & = & \langle T_1 \cup T_2, \mathcal{V}(E_1)(m) +_{\mathbb{V}} \mathcal{V}(E_2)(m) \rangle \\
[\![E_1 - E_2]\!](m) & = & \langle T_1 \cup T_2, \mathcal{V}(E_1)(m) -_{\mathbb{V}} \mathcal{V}(E_2)(m) \rangle \\
[\![\mathsf{source}_l()]\!](m) & = & \langle \{l\}, \mathcal{V}(\mathsf{source})(m) \rangle \\
\\
[\![C]\!] & : & \mathbb{M} \to \mathbb{M} \\
[\![x := E]\!](m) & = & m\{x \mapsto [\![E]\!](m)\} \\
[\![\mathtt{assume}(x < n)]\!](m) & = & m \\
[\![\mathsf{sink}(E)]\!](m) & = & m
\end{array}
$$

(b) Abstract semantics

Figure 5: Generic Taint Analysis

that read untrusted inputs (e.g., `fread`), and use the arguments in a sensitive context (e.g., `malloc`), respectively. We assume each source point is associated with a unique label l.

3.2 Generic Taint Analysis

We present a generic static analysis for taint tracking. The goal of the analysis is to estimate potential data flows from source points to sink points. The analysis can be instantiated to a family of taint analyses that are applicable to common types of vulnerabilities such as integer overflow, format string, or command injection [20, 21, 44]. We will present the detailed instantiation for our implementation in Section 4.

Abstract domains are shown in Figure 5(a). For a given program, our analyzer computes an abstract state ($\in \mathbb{S}$) that is a mapping from control points to the corresponding abstract memories. An abstract memory ($\in \mathbb{M}$) is a mapping from variables ($\in \mathbb{X}$) to their abstract values. An abstract value consists of two parts: the abstract domains for taint information ($\mathbb{T}$) and value information ($\mathbb{V}$). The taint domain is the power set of source labels. For taint checking, we collect all possible source points that lead to the value. The value domain represents general information of variables. For instance, one may define a simple abstract domain that only represents whether a value is overflowed, or a more sophisticated domain such as the interval domain. Our design choice will be explained in Section 4. Note that the value domain is not mandatory but is used to improve the precision of the analysis.

Abstract semantics is defined in Figure 5(b). The abstract semantics for expressions $[\![E]\!]$ computes the abstract value of an expression given an abstract memory. We assume that the value domain $\mathbb{V}$ is accompanied by an evaluation function $\mathcal{V} : E \to \mathbb{M} \to \mathbb{V}$ that computes the abstract value for an expression. Constant values (n) are not tainted and introduce an abstract value according to $\mathcal{V}$. For binary operations (+ and −), we join the taint information of two operands and compute the results of the corresponding abstract operator. For source points, the analyzer collects the labels, which will be used for taint checking, and computes its abstract value. The abstract semantics for commands $[\![C]\!]$ computes the abstract memory after the execution of C given an abstract memory.

TRACER derives a set of alarms from the analysis results. An alarm of the taint analysis $\omega = \langle c_1, c_2 \rangle$ is a pair of two control points where c_1 is a source point and c_2 is a sink point that uses the untrusted data from the source c_1. We assume that the analysis is accompanied by an alarm inspection function $\mathcal{Q} : \mathbb{C} \to \mathbb{M} \to \wp(\mathbb{C})$. Given a sink point c and an abstract memory m at c from the analysis result, $\mathcal{Q}(c)(m)$ is a set of source points from which vulnerable data-flows start to the sink point c. Once an analysis $\mathcal{A}(P)$ for program P is completed, a set of alarms Ω is derived using the alarm inspection function.

DEFINITION 1 (ALARM). *Let $\mathbb{C}_s$ be a set of all sink points of a program. A set of alarms Ω of the program is defined as follows:*

$$\Omega = \{\langle c_1, c_2 \rangle \mid c_2 \in \mathbb{C}_s, c_1 \in \mathcal{Q}(c_2)(m)\}$$

where m is the abstract memory at c_2 according to the analysis results.

3.3 Data Dependency Graph and Tainted Traces

Next, we build a data dependency graph for the input program. Given a control-flow graph $\langle \mathbb{C}, \to \rangle$ of the program, the data dependency graph is defined as a tuple $\langle \mathbb{C}, \rightsquigarrow \rangle$. The data dependency graph has the same set of nodes but is based on data dependency relations rather than control-flow relations. We follow the standard notion of data dependency:

$$c_1 \rightsquigarrow c_2 \iff c_1 \to^+ c_2 \ \wedge\ x \text{ is defined at } c_1 \ \wedge\ x \text{ is used at } c_2 \\ \wedge\ x \text{ is not re-defined in any points between } c_1 \text{ and } c_2.$$

where x is a program variable. Such data dependency relation can be computed during the static analysis by bookkeeping additional information about the definition and use points. While control dependencies are not considered, we capture crucial branch conditions by specially treating conditional expressions (`assume` in Figure 5(b)). We consider variables used in conditional expressions also as defined ones. With this heuristic choice, TRACER can capture important steps such as bound checking in practice.

Once a data dependency graph is established, we extract tainted traces of alarms. For each alarm, TRACER derives all paths from the source point to the sink point on the data dependency graph.

DEFINITION 2 (TAINTED TRACE). *Given an alarm $\omega = \langle c_0, c_n \rangle$, a set of tainted traces $\mathcal{T}_\omega \subseteq \mathbb{C}^+$ is defined as follows:*

$$\mathcal{T}_\omega = \{\langle c_0, \ldots, c_n \rangle \mid \forall i \in [0, n-1].\ c_i \rightsquigarrow c_{i+1}\}.$$

In the presence of loops, there can exist infinitely many traces of an alarm. In our implementation, TRACER unrolls each loop by once.

3.4 Feature Vector and Similarity Score

TRACER transforms each tainted trace to a feature vector that encodes the characteristics of the trace. We define a set of features to capture essential knowledge of vulnerable traces that are reusable across different programs. TRACER uses numerical features $f_i : \mathbb{C}^+ \to \mathbb{N}$. Given a set of n features $\{f_1, \ldots, f_n\}$, TRACER derives a feature vector of a trace τ: $\langle f_1(\tau), \ldots, f_n(\tau) \rangle$. Then, a set of feature vectors Π_ω of an alarm ω is defined as follows:

$$\Pi_\omega = \{\langle f_1(\tau), \ldots, f_n(\tau) \rangle \mid \tau \in \mathcal{T}_\omega\}$$

$$
\begin{array}{rcl}
\text{(Abstract value)} \quad \mathbb{V} & = & \bar{\mathbb{I}} \times \underline{\mathbb{I}} \\
\text{(Overflow)} \quad \bar{\mathbb{I}} & = & \{\bot_o, \top_o\} \\
\text{(Underflow)} \quad \underline{\mathbb{I}} & = & \{\bot_u, \top_u\} \\
\\
\mathcal{V}(n)(m) & = & \langle \bot_o, \bot_u \rangle \\
\mathcal{V}(E_1 + E_2)(m) & = & \langle \top_o, U_1 \sqcup U_2 \rangle \\
 & & \text{where } \mathcal{V}(E_1)(m) = \langle \_, U_1 \rangle \\
 & & \text{and } \mathcal{V}(E_2)(m) = \langle \_, U_2 \rangle \\
\mathcal{V}(E_1 - E_2)(m) & = & \langle O_1 \sqcup O_2, \top_u \rangle \\
 & & \text{where } \mathcal{V}(E_1)(m) = \langle O_1, \_ \rangle \\
 & & \text{and } \mathcal{V}(E_2)(m) = \langle O_2, \_ \rangle \\
\mathcal{V}(\text{source})(m) & = & \langle \bot_o, \bot_u \rangle
\end{array}
$$

Figure 6: Abstract domains

Finally, Tracer compares the feature vectors of alarms in program P to the set of all pre-computed feature vectors of known vulnerabilities, $\Pi_{\mathcal{S}}$. The set $\Pi_{\mathcal{S}}$ can be derived using the same steps described in the previous sections except that only known true alarms are considered. We also assume a function $\mathsf{Sim} : \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{R}$ that computes the similarity of two feature vectors. Using the similarity function, the score of an alarm is defined as the maximum similarity score between alarm traces and signature traces.

Definition 3 (Score of alarm). *Given an alarm ω and a set of feature vectors of signatures $\Pi_{\mathcal{S}}$, the score of the alarm is defined as follows:*

$$\max\{Sim(\pi_\omega, \pi) \mid \pi_\omega \in \Pi_\omega, \pi \in \Pi_{\mathcal{S}}\}$$

where Π_ω is a set of feature vectors of alarm ω.

4 INSTANTIATION

This section describes the details of our system. First, we instantiate the generic taint analysis to detect common types of vulnerabilities. Our implementation aims at detecting integer overflows, integer underflows, buffer overflows, format string bugs, or command injections. Then, we explain our feature design.

4.1 Abstract Domains and Semantics

We define the abstract domain $\mathbb{V}$ and function $\mathcal{V}$ that are used in our implementation in Figure 6. The abstract domain $\mathbb{V}$ constitutes two parts: the overflow domain and the underflow domain. The overflow domain $\bar{\mathbb{I}}$ (resp., underflow domain $\underline{\mathbb{I}}$) represents whether the value may be overflowed ($\top_o$) (resp., underflowed) or not ($\bot_o$). The function $\mathcal{V} : E \to \mathbb{M} \to \mathbb{V}$ approximates the chances of integer overflows and underflows for a given expression and an abstract memory. Constant values (n) are not overflowed and underflowed. For addition (resp., subtraction) operators, we conservatively approximate the value to be potentially overflowed (resp., underflowed).

For the five types of vulnerabilities, we use the following alarm inspection function Q:

$$Q(c)(m) = Q_T(c)(m) \cup Q_O(c)(m) \cup Q_U(c)(m).$$

Each sub-function is defined as follows:

$$
\begin{aligned}
Q_T(c)(m) &= \{c_0 \mid c_0 \in T, \langle T, \_, \_ \rangle = [\![E]\!](m)\} \\
Q_O(c)(m) &= \{c_0 \mid c_0 \in T, \langle T, \top_o, \_ \rangle = [\![E]\!](m)\} \\
Q_U(c)(m) &= \{c_0, \mid c_0 \in T, \langle T, \_, \top_u \rangle = [\![E]\!](m)\}
\end{aligned}
$$

where c is a sink point and the abstract memory at c is m. Function Q_T collects all the source points of a sink point if the argument of a sink function is tainted. Tracer uses Q_T to detect format string, command injection, and buffer overflow at `printf`-like functions, `exec`-like functions, and `memcpy`-like functions, respectively. Q_O and Q_U additionally check whether the argument can be potentially overflowed and underflowed, respectively. The functions are used to detect malicious uses of memory allocations (e.g., `malloc`) with an overflowed (i.e., unintentionally small) argument, and memory copies (e.g., `memset`) with an underflowed (i.e., unintentionally large) argument.

4.2 Features and Similarity Measure

We have designed a set of features for tainted alarm traces that are shown in Table 1. The set of features comprises two categories: low-level and high-level features.

The low-level features NumOfOpX and NumOfLibX describe the frequency of each primitive operator X (e.g., + and <<) and standard library call X (e.g., `strlen` and `strcmp`) on a trace. For example, Figure 3 shows the feature vectors of traces with the low-level features. The motivation behind this feature design is based on the observation of typical bug reports. Developers typically recognize and describe a vulnerability in terms of a sequence of operations[2]. In our experience, such a sequence is a reasonable signature to characterize standard concepts such as geometry formulas or protocols. Even though our features only consider frequencies of operators on traces, each trace is carefully derived using a sophisticated context-sensitive static analysis. We extract traces only when the analyzer raises alarms. Furthermore, the traces are based on data flows rather than control flows. These design choices improve the accuracy of the system based on the similarity comparison.

On the other hand, the high-level features are designed to capture deeper contexts of traces. Instead of counting individual occurrences of operators, the features describe relationships among expressions and operators. We separated the low-level features from the high-level ones to balance manual efforts and accuracy. The low-level features consist of general program components that are transferable across different programs and do not require manual efforts. Instead, we manually designed the high-level features by observing typical bug-fix patterns. For example, EqualToPercentage is inspired by usual conditional expressions to prevent format string bugs. A similar design choice is used in MVP [47]; they also handle format strings specially. Such high-level features help filter out typical false alarms. In our experiments, the accuracy of Tracer with only the low-level features is already reasonably high, but the high-level features can further improve the accuracy. The details will be discussed in Section 5.4.

Tracer uses cosine similarity, a well-known measure of similarity between two vectors. Given two feature vectors π_1 and π_2, the

[2]For example, https://cgit.freedesktop.org/xorg/lib/libXcursor/commit/?id=4794b5dd34688158fb51a2943032569d3780c4b8.

Table 1: Features of traces. E and K represent an arbitrary expression and a constant, respectively.

Name	Description
NumOfOpX	# primitive operator X on the trace
NumOfLibX	# calls to library X on the trace
LargerThanConst	# expressions of the form $E > K$ or $E \geq K$
SmallerThanConst	# expressions of the form $E < K$ or $E \leq K$
EqualToVar	# expressions of the form E == K
NotEqualToVar	# expressions of the form E != K
EqualToPercentage	# expressions of the form E == '%'

similarity is defined as follows:

$$\text{Sim}(\pi_1, \pi_2) = \frac{\pi_1 \cdot \pi_2}{||\pi_1|| \; ||\pi_2||}.$$

4.3 Application to Other Vulnerability Types

The general principle behind TRACER—computing similarity scores between traces—is applicable to other types of vulnerabilities if the underlying analyzer produces traces of alarms. For example, static analyzers for double-free or use-after-free bugs typically report potentially erroneous traces from a call to `free` to other calls to `free` or uses of the same pointer. We demonstrate the applicability to other vulnerability types in the next section.

5 EXPERIMENT

Our evaluation is designed to answer the following questions:

- **RQ1**: How effective is TRACER for finding unknown recurring vulnerabilities?
- **RQ2**: How accurate is TRACER compared with existing approaches?
- **RQ3**: How effective is the high-level features of TRACER?
- **RQ4**: How scalable is TRACER to large programs?

All experiments were conducted on Linux machines with Intel Xeon 2.90GHz. We set the timeout to one hour for running the static analysis for each package. Our code and data are publicly available anonymously at the URL https://zenodo.org/record/5927110.

5.1 Experimental Setup

5.1.1 Implementation. We implemented TRACER on top of Facebook's Infer analyzer [5]. The taint analyzer is designed as described in the previous sections. We use pointer information computed by Infer's buffer overrun checker. Following Infer's framework, our taint analysis is designed to be a modular interprocedural analysis (i.e., context-sensitive). For each benchmark, we run 20 tasks in parallel. Our taint analysis checks five common vulnerabilities described in Section 4: integer overflows, integer underflows, buffer overflows, command injections, and format string bugs. We used the Pulse engine of Infer for use-after-free and double-free bugs.

5.1.2 Signature programs. We collected signature programs from different sources of real-world and synthetic vulnerabilities:

(1) **Real-world vulnerabilities**: We collected 16 vulnerabilities that can be reproduced by our taint analysis from the CVE report [10] and prior work [20, 21].
(2) **Juliet test suite** [4]: Juliet Test Suite consists of a large set of small programs each of which has a common vulnerability. We used 5,383 programs that have the same types of vulnerabilities handled by our analysis.
(3) **Online tutorial**: We collected 5 examples from online tutorials on secure programming provided by OWASP [16].

5.1.3 Benchmarks. We evaluated TRACER using 273 Debian packages written in C/C++. We selected 16 common categories (web, sound, utils, etc) of Debian packages [13] that contain at least one package that Infer can analyze. This step excludes many categories consisting of unsupported languages (e.g., R, Haskell, etc) and packages having build issues in our environment. We randomly chose 20 packages that have at least one alarm raised by our analyzer in the selected categories. For categories that have less than 20 packages, we used all packages in the categories.

5.1.4 Baselines. We compare TRACER with state-of-the-art bug detection tools from three categories: *1)* clone-based approach *2)* learning-based approach *3)* pattern-based static analyzer. For each category, we chose tools that were recently proposed and are publicly available: VUDDY [26], CCAligner [43], Devign [48] and Github's CodeQL [2]. We ran the baselines for the same types of vulnerabilities handled by TRACER. For VUDDY, we selected the reported alarms based on their CWE ID [11] that matches the vulnerability types. For CodeQL, we ran all their security-related queries dedicated to the corresponding CWE IDs [7] of the types.

5.1.5 Metrics. Each analyzer reports alarms in different manners. For example, tools based on static analysis (e.g., CodeQL, Infer) typically report sink points as alarms. However, this may be an overestimation if there are multiple sink points from the same malicious input point. For a fair comparison, we used the following metrics:

- **Root causes**: To show the effectiveness (Section 5.2), we report the number of root causes of discovered vulnerabilities. We manually inspected the true alarms from the analyzers and counted the number of root causes.
- **Sink points**: To compare the precision and recall of the analyzers (Section 5.3), we use sink points as both TRACER and CodeQL provide them to the users as alarms. We counted the number of true alarms and false alarms reported by the analyzers. For the other tools (VUDDY, CCAligner, Devign) that report vulnerable functions, we counted the number of reported functions.

5.2 RQ1: Effectiveness

5.2.1 New Bugs. This section shows how effective TRACER is for detecting previously unknown vulnerabilities in the Debian packages. We manually inspected all the reported alarms whose similarity scores are larger than 0.85. In addition, we randomly selected 100 alarms below the threshold and manually inspected them. In total, 424 reports (i.e., sink points) were investigated.

TRACER found 112 new vulnerabilities in 67 packages. Among them, 30 vulnerabilities have been confirmed by the developers and 6 CVEs have been assigned as of writing this paper. For all the other reports, we have not received answers by the time of submission. Among the 112 bugs, only 10 can be found by the baseline tools (VUDDY_O and Devign_O).

```
bool ssgLoadTGA(...) {
    GLubyte header[18];
    fread(header, 18, 1, f);
    ...
    int xsize = get16u(header + 12);
    int ysize = get16u(header + 14);
    int bits = header[16];
    ...
    // potential integer overflow
    GLubyte *image = new GLubyte [ (bits / 8) * xsize * ysize ];
    ...
}

int get16u(const GLubyte *ptr) { return (ptr[0] | (ptr[1] << 8)); }
```

Figure 7: An integer overflow discovered in libplib1-1.8.5 that is similar to the one from sam2p-0.49.4 in Figure 1(b).

```
void CWE190_Integer_Overflow__int64_t_fscanf_square_01_bad() {
    int64_t data;
    data = 0LL;
    fscanf (stdin, "%" SCNd64, &data);
    // potential integer overflow
    int64_t result = data * data;
    char *p = malloc(result);
}
```

(a) Juliet test suite (CWE-190)

```
static DiaObject *fig_read_polyline(FILE *file, DiaContext *ctx) {
    fscanf(file, "%d_%d_%d_%d_%d_%d_%d_%d_%lf_%d_%d\n", ..., &npoints)
    newobj = create_standard_polyline(npoints, ...);
    ...
}

DiaObject *create_standard_polyline(int num_points, ...) {
    pcd.num_points = num_points;
    new_obj = otype->ops->create(NULL, &pcd, &h1, &h2);
    ...
}

static DiaObject *polyline_create(Point *startpoint, void *user_data,
    Handle **handle1, Handle **handle2) {
    MultipointCreateData *pcd = (MultipointCreateData *)user_data;
    polyconn_init(poly, pcd->num_points);
    ...
}

void polyconn_init(PolyConn *poly, int num_points) {
    // potential integer overflow
    poly->points = g_malloc(num_points * sizeof(Point));
    ...
}
```

(b) dia-0.97.3

Figure 8: An integer overflow bug in dia-0.97.3 and a signature vulnerability from Juliet test suite.

Table 2 shows the vulnerabilities discovered by Tracer. We observed that Tracer can detect various types of vulnerabilities using signatures from different sources including known CVEs or synthetic vulnerabilities. Most of the detected vulnerabilities have high similarity scores. We will discuss the detailed distribution of the scores in the next section.

Tracer is able to detect new vulnerabilities that are similar to known ones. Figure 7 shows a vulnerability found in libplib1. The signature that gives the highest score for the case is sam2p in Figure 1(b) which is itself a recurring vulnerability similar to

```
int main(void) {
    char *ptr_h;
    char h[64];
    ptr_h = getenv("HOME");
    if (ptr_h != NULL) {
        // potential buffer overflow
        sprintf(h, "Your_home_directory_is:_%s_!", ptr_h);
        printf("%s\n", h);
    }
    return 0;
}
```

(a) OWASP tutorial

```
XrmDatabase resource_buildDatabase(...) {
    char locale1[100];
    char loc_lang[100];
    char *locale = getenv("LC_ALL");
    String s = getenv("XUSERFILESEARCHPATH");
    char *cP = loc_lang;
    char *cL = locale;
    ...
    while (*cL) {
        ...
        *cP++ = *cL++;
    }
    *cP = 0;

    if (s == NULL || !strcasecmp(s, "False")) {
        // potential buffer overflow
        sprintf(locale1, "noint:%s%s", loc_lang, ...);
        ...
    }
}
```

(b) gv-3.7.4

Figure 9: A buffer overflow bug in gv-3.7.4 and a signature vulnerability from an OWASP tutorial.

```
generic *gp_alloc(size_t size, ...) { /* wrapper of malloc */ }

static int LUA_init_lua(void) {
    char *script_fqn;
    char *gp_lua_dir = getenv("GNUPLOT_LUA_DIR");
    ...
    // allocation with a large enough length
    script_fqn = gp_alloc(strlen(gp_lua_dir) + ... + 2, ...);
    // potential buffer overflow (false alarm)
    sprintf(script_fqn, "%s%c%s", gp_lua_dir, ...);
    ...
}
```

(a) gnuplot-5.2.8

```
SHPHandle SHPAPI_CALL SHPOpenLL(...) {
    SHPHandle psSHP;
    uchar *pabyBuf = (uchar *)malloc(100);
    fread(pabyBuf, 100, 1, psSHP->fpSHX);
    psSHP->nRecords = pabyBuf[27] + pabyBuf[26] * 256 + ...;
    ...
    // bound checking
    if (psSHP->nRecords > 256000000) {
        return (NULL);
    }
    ...
    // false alarm (integer overflow)
    int32 *panSHX = (int32 *)malloc(sizeof(int32) * 2 * psSHP->nRecords);
}
```

(b) grass-7.8.2

Figure 10: False alarms filtered by the similarity measure of Tracer

Table 2: List of new vulnerabilities detected by TRACER. #Bugs reports the number of bugs by root causes, Signature shows the sources of vulnerability signatures and Score represents the similarity scores between the bugs and the signatures. ✔indicates the vulnerabilities whose CVE IDs have been assigned. Type indicates the type of bugs (IO: integer overflow, IU: integer underflow, BO: buffer overflows, CI: command injection, FO: format string bug, UAF: use-after-free, DF: double-free).

Program	#Bugs	Type	Score	Signature	CVE
4ti2	1	IO	0.71-1.00	Juliet-CWE190	-
abyss	1	IO	0.82	Juliet-CWE190	-
alsa-utils	1	DF	1	Juliet-CWE415	-
bowtie2	1	IO	0.74	CVE-2017-9181	-
bsdutils	1	CI	0.86	CVE-2016-10729	-
bsdutils	1	IO	1	Juliet-CWE190	✔
bwbasic	1	BO	0.44	CVE-2018-1100	-
coinor-libcgl1	1	IO	0.87	Juliet-CWE190	-
coinor-libclp1	1	IO	1	Juliet-CWE190	-
crafty	1	IO	0.86	CVE-2017-1000229	-
cron	1	CI	0.68	OWASP tutorial	-
crrcsim	2	IO	0.85-0.90	CVE-2017-16663	-
darktable	3	IO	1	Juliet-CWE680	-
dcraw	1	IO	0.93-0.94	CVE-2017-9181	✔
dia	1	IO	1	Juliet-CWE190	✔
drawxtl	1	IO	0.79	CVE-2017-9181	-
dvbstreamer	1	BO	1	OWASP tutorial	-
elvis-tiny	2	BO	0.50-1.00	OWASP tutorial	-
gap-guava	1	IO	1	Juliet-CWE190	-
gnuplot	2	FS	0.82	Juliet-CWE134	-
grass	8	BO	0.41-1.00	OWASP tutorial	-
groff	1	IO	1	Juliet-CWE190	-
gv	1	BO	1	OWASP tutorial	-
htmldoc	2	IO	0.90-0.95	CVE-2017-9181	✔
hugin	2	IO	0.87-1.00	Juliet-CWE190	-
ispell	2	BO	1	OWASP tutorial	-
libaudio2	1	BO	1	OWASP tutorial	-
libfreeimage3	1	BO	0.83	CVE-2017-6313	-
libkrb5support0	1	IO	1	Juliet-CWE190	-
liblinear-tools	1	BO	0.3	CVE-2018-1100	-
liblinear-tools	1	IO	0.93-1.00	Juliet-CWE190	-
liblrs0	1	IO	0.91	Juliet-CWE191	-
libmjpegutils-2.1-0	1	BO	1	OWASP tutorial	-
libmount1	1	CI	0.72	CVE-2015-9059	-
libmount1	1	IO	1	Juliet-CWE190	-
libpano13-3	1	FS	0.59	mp3rename-0.6 [20]	✔
libpano13-3	1	IO	0.87	CVE-2017-16663	-
libplib1	5	IO	0.76-0.93	shntool-3.0.5 [20]	✔
libquicktime2	1	IO	0.85-0.95	CVE-2017-9181	-

Program	Bugs	Type	Score	Signature	CVE
lp-solve	1	IO	1	Juliet-CWE190	-
mailutils	1	UAF	1	Juliet-CWE415	-
mdadm	1	BO	0.16	CVE-2019-14523	-
minidlna	1	IO	0.94	Juliet-CWE190	-
nageru	1	IO	0.87	CVE-2017-16663	-
nedit	1	BO	1	OWASP tutorial	-
newmail	1	FS	0.82	Juliet-CWE134	-
nickle	1	BO	1	OWASP tutorial	-
nickle	1	CI	0.67	Juliet-CWE78	-
octave-nan	3	IO	0.87-1.00	Juliet-CWE190	-
printer-driver-foo2zjs	1	IU	0.94	Juliet-CWE191	-
r-cran-lpsolve	1	IO	1	Juliet-CWE190	-
rawtherapee	4	IO	0.86-1.00	Juliet-CWE680	-
rlwrap	1	CI	0.82	Juliet-CWE78	-
rtcw	1	BO	0.4	CVE-2018-1100	-
sa-exim	1	CI	1	Juliet-CWE78	
sane	1	IO	0.87	CVE-2017-9181	-
scheme48	1	IO	0.85	CVE-2009-1570	-
seaview	1	BO	0.56	CVE-2018-1100	-
siril	2	IO	0.87-1.00	Juliet-CWE680	-
siril	1	IU	0.82	Juliet-CWE191	-
snap	2	BO	1	OWASP tutorial	-
snap	1	IO	1	Juliet-CWE680	-
stk	1	IO	0.87	shntool-3.0.5 [20]	-
sweed	1	IO	1	Juliet-CWE190	-
tcliis	1	BO	1	OWASP tutorial	-
tome	1	FS	0.96	CVE-2015-8106	-
vacation	1	CI	0.67	Juliet-CWE78	-
w3m	1	FS	0.96	CVE-2015-8106	-
wily	5	BO	0.47-1.00	OWASP tutorial	-
xbuffy	1	BO	1	OWASP tutorial	-
xfig	2	BO	1	OWASP tutorial	-
xsane	1	IO	0.87-1.00	Juliet-CWE190	-
xwpe	1	BO	1	OWASP tutorial	-
xwpe	1	CI	0.87	Juliet-CWE78	-
xwpe	3	IO	0.87-0.89	Juliet-CWE190	-
zangband	1	BO	0.77	CVE-2017-6313	-
zangband	1	FS	0.97	CVE-2015-8106	-
zangband	1	IO	0.93	CVE-2017-9181	-

the one in Figure 1(a). We also notice that TRACER can effectively discover real-world security bugs using synthetically generated toy examples. Figure 8 shows an integer overflow vulnerability in dia detected by TRACER and a signature vulnerability from the Juliet test suite. Notice that they have completely different syntactic structures. For example, the vulnerability in dia involves three function calls including one indirect call, as well as complicated pointer dereferences. On the other hand, synthetic code has an extremely simple structure. However, they have the same root cause of the vulnerabilities. Both of the programs read an external input using `fscanf`, and cause an integer overflow by multiplying the input value with another integer value. TRACER exactly detects the same vulnerable behavior from the two programs and sets the similarity score to 1.0.

For other types of vulnerabilities, TRACER can also detect recurring vulnerabilities that are similar to existing ones. Figure 9 depicts a buffer overflow error in gv. This bug happens because the program reads an untrusted string using `getenv` that is used to construct a new string via `sprintf`. The vulnerable behavior is described in a tutorial by OWASP [36]. Likewise, TRACER detects one use-after-free and one double-free bug similar to examples in the Juliet test suite. While the example codes are simple, the real-world vulnerability involves complicated aliases, indirect assignments, and control flows. Nevertheless, TRACER can find that the essence

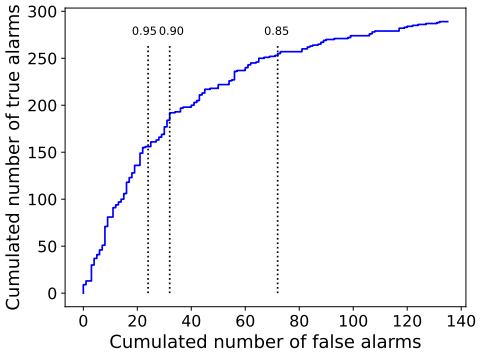


Figure 11: Ranking effectiveness. Each data point represents the number of true and false alarms that the user will obtain when enumerating the sorted alarms by ranking.

of the bug is actually the same as the tutorial examples as the static analyzer estimates the detailed semantics.

5.2.2 Ranking Effectiveness. Next, we evaluate the effectiveness of our ranking scheme. For all the manually inspected 424 alarms, we investigate where true and false positives are located in the bug reports ranked by their similarity scores.

Figure 11 and the top three rows in Table 3 show that most of the true positives are highly ranked in the report. While the underlying static analysis reports 1,975 alarms, only 176 of them have scores higher than 0.95 (Tracer$_{95}$). Among them, 154 (87.5%) are confirmed as true positives. The true positive ratio is still high (85.7% and 78.1%) if we set the threshold to 0.9 (Tracer$_{90}$) and 0.85 (Tracer$_{85}$), respectively. The similarity-based score of Tracer effectively filters out a large number of false positives while retaining many real bugs. These results indicate that Tracer can help developers effectively discover recurring vulnerabilities.

Tracer's similarity measure not only prioritizes similar bugs, but also effectively suppresses false alarms. Figure 10(a) shows an alarm that is falsely identified by our taint analysis in gnuplot. This alarm looks similar to the vulnerability in Figure 9(a). However, the call to `sprintf` is safe because the program always allocates enough memory blocks using `strlen` and multiple addition operations. These operators are captured by our features and differentiate this alarm from typical vulnerable patterns. Figure 10(b) shows another example of a false alarm in grass. The integer overflow will never occur since there is a bound checking for external user inputs. This is also captured by one of the high-level features LargerThanConst. Notice that these features only appear in the false alarm traces, not in the signatures. This in turn leads to a lower score for these alarms (0.77–0.88) and ranks them below many other true alarms.

Overall, the experimental results show that Tracer is effective to detect semantically recurring vulnerabilities. In particular, our trace-based similarity measure powered by static analysis is robust to syntactic variants. Thus, Tracer can report recurring vulnerabilities with high similarity scores even though two programs have significantly different syntactic characteristics.

Table 3: Comparison to state-of-the-art approaches. Column RC, TP, FP and FN report the number of vulnerabilities by root causes, true positives, false positives and false negatives reported by each analyzer, respectively.

Analyzer	RC	TP	FP	FN	Prec (%)	Recall (%)
Tracer$_{95}$	58	154	22	299	87.5	34.0
Tracer$_{90}$	69	192	32	261	85.7	42.4
Tracer$_{85}$	87	253	71	200	78.1	55.8
VUDDY$_O$	3	5	7	448	41.7	1.1
VUDDY$_S$	0	0	10	453	0.0	0.0
CCAligner	0	0	150	453	0.0	0.0
CodeQL	86	161	163	292	49.7	35.5
Devign$_O$	-	10	-	443	-	2.2
Devign$_S$	-	0	-	453	-	0.0

5.3 RQ2: Comparison

This section compares the accuracy of Tracer with the state-of-the-art tools. In the rest of this section, we use Tracer$_{85}$ for comparison because all its reports are manually inspected. Since there is no fully labeled data set for recurring vulnerabilities and it is hard to manually inspect all vulnerabilities in our benchmarks, we set up a set of ground truths as follows:

- We ran all the tools (Tracer$_{85}$, VUDDY, CCAligner, and CodeQL) and manually inspected the same number of alarms as Tracer$_{85}$ (i.e., 324). If a tool reports fewer than 324 alarms, we inspected all of them.
- We collected the true alarms detected by all the tools and included the true alarms from the random sampling in Section 5.2, which are not detected by Tracer$_{85}$.

In total, we collected 453 ground truths and compared the accuracy of Tracer$_{85}$ to the other baselines.

Table 3 shows the performance of each analyzer. Note that we could not use Devign [48], a learning-based approach, to collect ground truths. Unlike the other tools, Devign does not provide explainable reports, such as vulnerabilities they are similar to (e.g., Tracer, VUDDY, CCAligner) or description of vulnerabilities (e.g., CodeQL). Thus, we only investigated whether Devign can detect any of 453 ground truths.

5.3.1 Comparison to VUDDY and CCAligner. We compared Tracer against clone-based detectors, VUDDY and CCAligner. For VUDDY, we established two different settings in ways to collect the vulnerability database for clone detection. VUDDY$_O$ is based on the original database provided by the official web service that has 1,764 CVEs as signatures [22]. In order to discard the effect of the quality of the database per se, we also tried VUDDY$_S$ that uses our own signature database. Following the same methodology as VUDDY$_O$, we collected all the vulnerable functions that are patched in the later versions for the real-world benchmarks, or annotated in the source code for the synthetic benchmarks. For CCAligner, we followed the same setting as VUDDY$_S$. The threshold of CCAligner is set to 0.85 which is the same setting as used in Tracer.

VUDDY$_O$ reports 7 false positives out of 12 alarms. The reason for the false alarms is due to a practical issue regarding establishing their databases. VUDDY$_O$ collects all the modified functions in

```
1  unsigned sget4 (unsigned char *s) {
2      ...
3      return s[0] << 24 | s[1] << 16 | s[2] << 8 | s[3];
4  }
5
6  unsigned get4() {
7      unsigned char str[4] = { 0xff,0xff,0xff,0xff };
8      fread (str, 1, 4, ifp);
9      return sget4((unsigned char *)str);
10 }
11
12 void foveon_load_camf() {
13     unsigned wide = get4();
14     unsigned high = get4();
15     ...
16     // potential integer overflow
17     char *meta_data = (char *) malloc(wide * high * 3/2);
18     ...
19 }
```

Figure 12: A vulnerability found in dcraw-9.28 and rawtherapee-5.8.

```
void badVaSink(char *data, ...) {
    va_list args;
    va_start(args, data);
    vfprintf(stdout, data, args);
    va_end(args);
}
```

(a) Juliet test suite (CWE-134)

```
void lqt_dump(char * format, ...) {
    va_list argp;
    va_start(argp, format);
    vfprintf(stdout, format, argp);
    va_end(argp);
}
```

(b) libquicktime2-1.2.4

Figure 13: A code clone detected by VUDDY$_S$

patch commits of known CVEs as signatures. However, a single commit may contain numerous irrelevant modifications. This leads to spurious signatures that match non-vulnerable functions. In fact, all of the false positives from VUDDY$_O$ turned out to be the case.

VUDDY$_O$ detects 3 vulnerabilities by reporting 5 function clones. Among them, TRACER can detect two vulnerabilities but misses one because there is no similar trace in our signature database. The commonly detected vulnerabilities are found in dcraw and rawtherapee, both being exactly the same functions as shown in Figure 12. The function `foveon_load_camf` reads `wide` and `high` from an external file (line 13–14), and allocates memory after multiplication (line 17) that can cause a potential integer overflow. VUDDY$_O$ reports that this bug originated from the same vulnerable source (LibRaw-demosaic-pack-GPL2, CVE-2017-6889). TRACER detected these vulnerabilities with a high similarity score (0.92) even though the origin is not in our signature database. Instead, TRACER captures that the vulnerability is similar to the one in sam2p shown in Figure 1(b). Notice that the bug from sam2p has a totally different syntactic structure from the code in Figure 12. This example demonstrates that TRACER effectively generalizes known vulnerability patterns to detect unseen ones.

Even though the database is carefully established with only vulnerable functions, VUDDY$_S$ reports 10 false alarms and no true bugs. The behavior of a function often depends on the context in which it is used. For instance, the function in Figure 13(a) is a signature in our database. If the argument `data`, which is passed to the second argument of `vfprintf`, can be controlled by an attacker, this in turn causes format string vulnerability. With this signature,

```
1  int32_t endianSwapI32(int32_t i) {
2      int32_t tmp = 0;
3      tmp |= ((i >> 24) & (0xff << 0));
4      tmp |= ((i >> 8) & (0xff << 8));
5      tmp |= ((i << 8) & (0xff << 16));
6      tmp |= ((i << 24) & (0xff << 24));
7      return tmp;
8  }
9
10 T readI(FILE *in, ...) {
11     T x;
12     fread((void *)&x, 1, sizeof(T), in);
13     ...
14     if (sizeof(T) == 4) {
15         return endianSwapI32(x);
16     }
17     ...
18 }
19
20 void Ebwt::readIntoMemory(...) {
21     uint32_t _nPat;
22     FILE *_in1;
23     string _in1Str;
24     ...
25     _in1 = fopen(_in1Str.c_str(), "rb"));
26     _nPat = readI<uint32_t>(_in1, ...);
27     // false alarm (integer overflow)
28     _plen.init(new uint32_t[_nPat], _nPat, true);
29 }
```

Figure 14: A false positive reported by CodeQL recognized as a potential integer overflow bug in bowtie2-2.3.5.1.

VUDDY$_S$ detects function `lqt_dump` in Figure 13(b) as a recurring vulnerability. However, according to our manual investigation, all the calls to `lqt_dump` takes only safe format arguments. Therefore this function is not vulnerable in the context of libquicktime2.

CCAligner reports 150 function clones but does not detect any vulnerability. Since CCAligner is not designed to find vulnerabilities, functions that do not have security-sensitive calls are reported. Similar to VUDDY, CCAligner also detects function clones without considering their contexts as in Figure 13(b). These lead to a large number of false positives in vulnerability detection.

Both VUDDY and CCAligner cannot detect most of the recurring vulnerabilities detected by TRACER. This is mainly because they are based on syntactic similarity measures at the function-level granularity. However, most of the vulnerabilities detected by TRACER involve multiple functions and have significantly different syntactic structures from signatures. Such characteristics in real-world programs hinder those tools from detecting semantically recurring vulnerabilities.

5.3.2 Comparison to CodeQL. In this section, we compare TRACER to CodeQL, which is a static analysis with human-written bug patterns (i.e., queries). We applied CodeQL with the queries related to the same types of vulnerabilities as TRACER, that are available in their repository[3]. In total, CodeQL reports 3,488 alarms from the benchmark programs. Among them we inspected 324 alarms.

Our experiments show that TRACER effectively detects recurring vulnerabilities that are missed by CodeQL. CodeQL reports 161 true alarms (35.5%) out of 453 ground truths while TRACER$_{85}$ detects 253

[3]https://github.com/github/codeql/blob/main/cpp/ql/src/Security/CWE/

true alarms (55.8%). Interestingly, there are no common vulnerabilities detected by both of the analyzers. There are mainly three reasons:

- Bugs can be detected by both our underlying analysis and CodeQL. However, Tracer filters out them because there is no similar trace in our signature database.
- Bugs can be detected by only CodeQL but missed by our underlying analysis. This is due to the unsoundness of our implementation. In particular, we only implemented the semantics of external libraries (e.g., `fread`, `getenv`) that are observed in our signature database. However, CodeQL supports a wider range of library models.
- Bugs can be detected by only our analysis but missed by CodeQL. We conjecture that this is mainly because of their unsound design choices as usual static bug detectors.

We agree that it is hard to make an apple-to-apple comparison between Tracer and CodeQL because they are based on different analysis frameworks, and CodeQL does not use signatures. The main goal of this comparison is to show that Tracer can accurately discover nontrivial bugs that are not detected by state-of-the-art tools.

Tracer$_{85}$ also has a lower false positive ratio than CodeQL. Among 324 inspected alarms, CodeQL reports 163 false positives (50.3%) while Tracer$_{85}$ has 71 (21.9%). Figure 14 shows a false positive reported by CodeQL. Function `readI` reads a number from an external file, and changes the endianness of the number using function `endianSwapI32`. This function involves complicated bitwise operations but does not incur an integer overflow on variable `_nPat` in `Ebwt::readIntoMemory`. Tracer$_{85}$ filters out this alarm as its score is 0.72, which is lower than the threshold. This demonstrates that Tracer's similarity score can effectively suppress false alarms compared to manually designed heuristics used in CodeQL.

5.3.3 Comparison to Devign. This section compares Tracer against a learning-based vulnerability detector, Devign. Similar to VUDDY in Section 5.3.1, we instantiated Devign to two settings: *1)* Devign$_O$ is trained with the training data provided by the authors[4], *2)* Devign$_S$ is trained with functions in our signature database. We sampled 10,000 vulnerable and 10,000 non-vulnerable functions for the training of Devign$_S$.

According to our experiments, Devign$_O$ detects only 10 vulnerabilities from the ground truths while Devign$_S$ fails to report any of them. Among the 10 true alarms, 8 of them are also detected by Tracer$_{85}$. As in previous work [47], our experiments also show that the learning-based approach is not practical to find recurring vulnerabilities. In particular, in our case, many vulnerabilities involve multiple functions. However, Devign classifies vulnerabilities in function-level granularity, that can lead to a substantial number of false negatives. Moreover, the results from the learned models are not explainable. This can significantly degrade the usability of the vulnerability detection system.

[4]https://sites.google.com/view/devign

Table 4: The precision of Tracer with different sets of features.

Threshold	W/ High-level Features TP	FP	Prec	W/O High-level Features TP	FP	Prec
0.95	154	22	0.88	154	22	0.88
0.90	192	32	0.86	198	35	0.85
0.85	253	71	0.78	256	79	0.76

5.4 RQ3: Impact of High-level Features

This section conducts a sensitivity study with different sets of features. We instantiated Tracer with two settings: similarity measures with and without high-level features. Table 4 shows the impact of high-level features.

The results show that Tracer's performance is not fundamentally limited to the choice of high-level features. When high-level features are disabled, Tracer with threshold 0.95 still reports the same set of alarms. However, the high-level features often effectively filter out typical false alarms by Tracer with lower thresholds. Tracer with thresholds 0.85 and 0.90 report 10.1% and 8.6% fewer false positives while retaining all the true alarms, when the high-level features are used.

5.5 RQ4: Scalability

This section evaluates the scalability of Tracer to large programs. We measure the whole computation time of the static analysis and similarity checking for each benchmark. Then, we report the running time of Tracer according to the size of the programs in Figure 15.

The results indicate that Tracer is scalable to large programs. On average, the static analysis takes 140.42 seconds for each package. The time spent for the feature vector construction and similarity computation is at most 2.71 seconds which is a negligible cost compared to the overall procedure. Although the analysis finishes within 20 minutes for most of the packages, some packages take considerably more time than the average. For example, `hugin` takes about 53 minutes. This is mainly because of the imprecision of function pointer resolution that leads to analyzing too many functions via spurious indirect calls. Another exceptional example is `gettext` that takes only 91 seconds while it comprises 982K lines of code. Despite the huge code size, the program consists of a large number of small library functions. Thus, the modular analysis can be highly parallelized.

6 RELATED WORK

Our work is inspired by a large body of research on recurring vulnerability detection. All the existing work aims at discovering recurring vulnerabilities via code reuse [23, 26, 29, 38, 47]. These approaches transform buggy code fragments within a certain boundary (e.g., functions) into various forms of vulnerability signatures such as hashes [23, 26] or dependency graphs [38, 47]. Then they search for similar representations of code fragments in the programs under investigation. On the other hand, Tracer is designed to detect vulnerabilities that share the semantically same root cause. We use a sophisticated static analysis that captures vulnerable semantics along arbitrarily long paths.

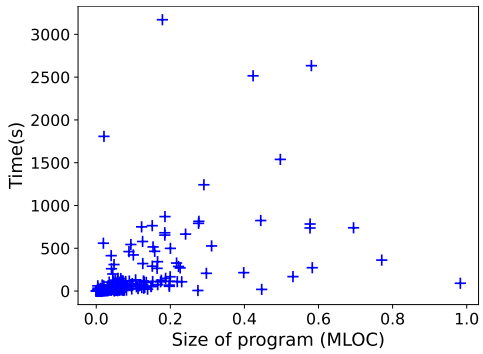


Figure 15: Running time of TRACER by program size.

Recently, several approaches have been proposed to detect vulnerabilities using learning methods [30, 31, 48]. They learn models to capture the characteristics of a variety of forms of code such as bug patches, program slices, or abstract syntax trees. Then, the learned models predict whether a slice or a function in a target program is vulnerable or not. Unlike our work, the learning-based approaches aim at general-purpose vulnerability detection. According to the literature, these approaches are not effective to detect recurring vulnerabilities [47]. This is also demonstrated by our experiments. The state-of-the-art learning-based approaches cannot detect all the recurring vulnerabilities that TRACER reports with high similarity scores.

Most of the existing static analyses that take into account code patterns highly rely on manual design [2, 3, 18]. FindBugs [3], SpotBugs [1], and ErrorProne [18] specify hundreds of human-written patterns each of which describes a specific buggy scenario. To reduce the engineering burden, CodeQL [2] introduces a query language to succinctly define bug patterns on top of their static analysis. However, it is still nontrivial for ordinary developers to write desired queries for their own purposes without static analysis expertises [32]. Instead of relying on manually written queries, TRACER automatically captures vulnerable patterns from data that provides an accessible framework for developers.

Researchers have proposed many techniques to detect code clones ranging from syntactic ones [9, 24, 40, 43, 46] to semantic ones [17, 25, 27, 41, 45]. Since their goal is to detect generally similar code fragments, they are not suitable for accurately finding recurring vulnerabilities even via code reuse [26, 47]. Instead, our work is designed to detect semantically similar vulnerabilities between two programs using a static analysis combined with a trace-based similarity measure.

Our similarity checking method can be understood as an alarm (or, analysis results, in general) ranking system for static analysis. There have been many alarm ranking methods proposed to lower the user's alarm inspection burdens. Existing approaches rank alarms by their confidence [6, 28, 35, 39], expected reactions from developers [19] or relevance to a specific commit [21]. Furthermore, recent approaches [6, 21, 39] incorporate user feedback on alarms and prioritize correlated alarms within programs. However, to our best knowledge, none of the existing work ranks alarms by similarity to a specific known vulnerability across programs.

7 CONCLUSION

We proposed TRACER, a framework for detecting semantically recurring vulnerabilities. TRACER is based on a static analysis that discovers potentially vulnerable traces in a target program. Each candidate trace is then compared with known vulnerabilities collected from various sources. Our empirical study shows that TRACER can accurately detect semantically similar vulnerabilities from a variety of open source programs. We anticipate that TRACER will allow developers to easily prevent recurring vulnerabilities without requiring static analysis expertise.

ACKNOWLEDGMENTS

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT)(No. 2021R1A5A1021944, 2021R1C1C1003876) and Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-00758, Development of Automated Program Repair Technology by Combining Code Analysis and Mining, and No.2022-0-01202, Regional strategic industry convergence security core talent training business).

REFERENCES

[1] Spotbugs. https://spotbugs.github.io, 2021.
[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *European Conference on Object-Oriented Programming (ECOOP 2016)*, 2016.
[3] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Softw.*, 25, 2008.
[4] Paul Black. Juliet 1.3 test suite: Changes from 1.2. *NIST Technical Note*, 8 2018.
[5] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods - Third International Symposium (NFM)*, volume 6617. Springer, 2011.
[6] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. In *29th ACM Joint European Software Engineering Conferenceand Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM.
[7] CodeQL. Codeql cwe queries. https://github.com/github/codeql/tree/main/cpp/ql/src/Security/CWE, 2021.
[8] CodeQL. TaintedAllocationSize.ql. https://github.com/github/codeql/blob/main/cpp/ql/src/Security/CWE/CWE-190/TaintedAllocationSize.ql, 2021.
[9] James R Cordy and Chanchal K Roy. The NiCad clone detector. In *The 19th IEEE International Conference on Program Comprehension (ICPC 2011)*. IEEE Computer Society, 2011.
[10] The MITRE Corporation. Common vulnerabilities and exposures, 2021.
[11] The MITRE Corporation. Common weakness enumeration, 2021.
[12] Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*. ACM, 2018.
[13] Debian. Debian packages. https://packages.debian.org/sid/, 2021.
[14] Will Dietz, Peng Li, John Regehr, and Vikram S Adve. Understanding integer overflow in c/c++. In *34th International Conference on Software Engineering (ICSE 2012)*. IEEE Computer Society, 2012.
[15] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019.
[16] The OWASP Foundation. Attacks. https://owasp.org/www-community/attacks/, 2021.
[17] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *30th International Conference on Software Engineering (ICSE 2008)*. ACM, 2008.
[18] Google. Error prone. https://errorprone.info, 2021.
[19] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: improving actionable alert ranking. In *11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, 2014.

[20] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. IEEE / ACM, 2017.

[21] Kihong Heo, Mukund Raghothaman, Xujie Si, and Mayur Naik. Continuously reasoning about programs using differential bayesian inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, 2019.

[22] IoTcube. Iotcube. https://iotcube.korea.ac.kr, 2021.

[23] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy (S&P 2012)*. IEEE Computer Society, 2012.

[24] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, 2007.

[25] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwangkeun Yi. MeCC: memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. ACM, 2011.

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *IEEE Symposium on Security and Privacy (S&P 2017)*. IEEE Computer Society, 2017.

[27] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of 8th International Static Analysis Symposium (SAS 2001)*, volume 2126. Springer, 2001.

[28] Ted Kremenek and Dawson R Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of 10th International Static Analysis Symposium (SAS 2003)*, volume 2694. Springer, 2003.

[29] Jingyue Li and Michael D Ernst. Cbcd: Cloned buggy code detector. In *34th International Conference on Software Engineering (ICSE 2012)*. IEEE Computer Society, 2012.

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC 2016)*. ACM, 2016.

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium (NDSS 2018)*. The Internet Society, 2018.

[32] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. Arbitrar : User-guided api misuse detection. *IEEE Symposium on Security and Privacy (S&P 2021)*, 2021.

[33] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proc. ACM Program. Lang.*, 1, 2017.

[34] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *IEEE Symposium on Security and Privacy (S&P 2015)*, 2015.

[35] Damien Octeau, Somesh Jha, Matthew Dering, Patrick D. McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016.

[36] OWASP. Buffer overflow via environment variables. https://owasp.org/www-community/attacks/Buffer_Overflow_via_Environment_Variables, 2021.

[37] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *IEEE Symposium on Security and Privacy (S&P 2020)*. IEEE, 2020.

[38] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Detection of recurring software vulnerabilities. In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*. ACM, 2010.

[39] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, 2018.

[40] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 2016.

[41] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *15th IEEE International Conference on Machine Learning and Applications (ICMLA 2016)*. IEEE Computer Society, 2016.

[42] Maddie Stone. Déjà vu-lnerability. https://googleprojectzero.blogspot.com/2021/02/deja-vu-lnerability.html, 2021.

[43] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. CCAligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.

[44] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012)*. USENIX Association, 2012.

[45] Huihui Wei and Ming Li. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*. ijcai.org, 2018.

[46] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on AutomatedSoftware Engineering (ASE 2016)*. ACM, 2016.

[47] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *29th USENIX Security Symposium (USENIX Security 2020)*. USENIX Association, 2020.

[48] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Neural Information Processing Systems 2019 (NeurIPS 2019)*, 2019.