# TRACER: Signature-based Static Analysis for Detecting Recurring Vulnerabilities

Anonymous Author(s)

## 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. 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 281 previously unknown vulnerabilities with 6 CVE identifiers assigned.

## 1 INTRODUCTION

Similar software vulnerabilities recur over time even across programs. One of the well-known reasons is the prevalence of code reuse [22, 27, 34, 35, 48] 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, 43]. Another reason is common misconceptions due to complicated low-level semantics of programming languages such as undefined behaviors in C [13]. 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 [43].

Although researchers have developed many successful techniques to detect recurring security vulnerabilities, existing approaches have limitations in several aspects. Approaches based on code similarity [22, 27, 32, 38, 48] 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, 16] 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 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. 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 the similarity score.

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 281 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.

*ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA*
. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

- We evaluate the effectiveness of Tracer on 273 Debian packages. We found 281 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 allocate unintentionally small size of memory chunks that potentially leads 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 that leads to an integer overflow (line 14). Finally, the overflowed integer (`rowbytes`) is passed to 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 afterwards.

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 that 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 the vulnerability detection when vulnerable behavior involves multiple functions as in the examples. State-of-the-art tools [27, 48] heuristically choose a vulnerability signature function that contains the patches of the known vulnerability (`ReadBMP` in the gimp case). However, this is still frangible 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 a high-level (e.g., calculating area of 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

```
1  long ToL(char *pbuffer) {
2      return (puffer[0] | puffer[1] << 8 | puffer[2] << 16 | puffer[3] << 24);
3  }
4  short ToS(char *pbuffer) { return ((short)(puffer[0] | puffer[1] << 8)); }
5
6  gint32 ReadBMP(gchar *name) {
7      FILE *fd = fopen(name, "rb");
8      if (!fd) return -1;
9
10     if (fread(buffer, Bitmap_File_Head.biSize - 4, fd) != 0)  // Read from a file
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(int rowbytes) {
20     /* memory allocation with an overflowed size */
21     char *buffer = malloc(rowbytes);
22     /* uses of buffer */
23 }
```

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

```
1  long ToL(char *pbuffer) {
2      return (puffer[0] | puffer[1] << 8 | puffer[2] << 16 | puffer[3] << 24);
3  }
4  short ToS(char *pbuffer) { return ((short)(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
10     if (fread(buffer, Bitmap_File_Head.biSize - 4, fd) != 0) // Read from a file
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.**

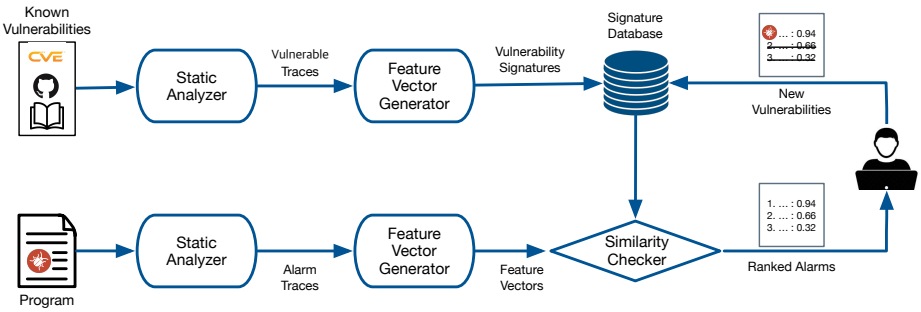



**Figure 2: System overview of TRACER**

(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 [27, 48] 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 is 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 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.

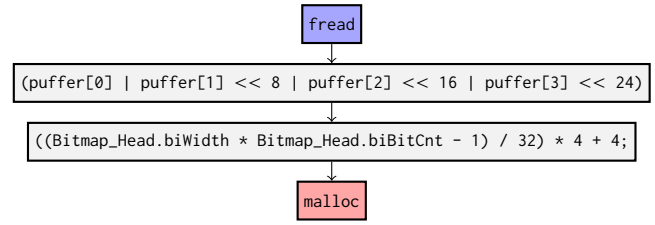
The same procedure will be applied for new target programs. Instead, TRACER extracts all possible traces from sources to sinks of the reported alarms while unrolling each loop only once. These traces will be compared to the signature traces.

*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 (e.g., *, <<) and common APIs (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).

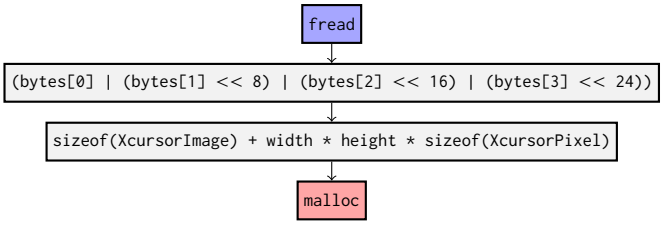
On the other hand, high-level features describe detailed behavior of traces that are not noticeable using only the low-level ones. We manually designed 5 high-level features. In general, they characterize crucial behavior of programs that can affect our target vulnerabilities. For example, one of our 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.

*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,




⟨fread : 1, | : 3, « : 3, * : 2, + : 1, − : 1, malloc : 1⟩

**(a) gimp in Figure 1(a)**




⟨fread : 1, | : 3, « : 3, * : 2, + : 1, − : 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.**

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

```
1 Ω ← 𝒜(P);
2 G ← build_dfg(P);
3 R ← ∅;
4 for ω ∈ Ω do
5     T_ω ← extract_traces(G, ω);
6     Π_ω ← {generate_feature(τ) | τ ∈ T_ω};
7     s ← max{Sim(π_ω, π) | π_ω ∈ Π_ω, π ∈ Π};
8     R ← R{ω ↦ s};
9 return R;
```

we can use any common similarity measures. In our implementation, we use cosine similarity which is 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:

$$\frac{\langle 1,3,3,2,1,1,1\rangle \cdot \langle 1,3,3,2,1,0,1\rangle}{||\langle 1,3,3,2,1,1,1\rangle|| \; ||\langle 1,3,3,2,1,0,1\rangle||} = 0.98$$

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

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

**Figure 4: Language**

(line 5) and encodes them as feature vectors (line 6). Finally, we compare each generated feature vector of the alarm $\omega$ 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 $\langle \mathbb{C}, \to \rangle$ where $\mathbb{C}$ is the set of control points and $(\to) \subseteq \mathbb{C}\times\mathbb{C}$ 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 that read untrusted inputs (e.g., `fread`), and use the arguments in a sensitive context (e.g., `malloc`), repectively. We assume that 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 [18, 19, 45]. 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 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.

[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**

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

**(b) Abstract semantics**

**Figure 5: Generic Taint Analysis**

The abstract semantics for commands $[\![C]\!]$ computes the abstract memory after the execution of $C$ given an abstract memory. Finally, the analyzer computes the abstract semantics of a program that is defined by the least fixed point of the following function as is standard:

$$F(X) = \lambda c \in \mathbb{C}.[\![c]\!](\bigsqcup_{c' \to c} X(c')).$$

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 $\Omega$ is derived using the alarm inspection function.

*Definition 3.1 (Alarm).* Let $\mathbb{C}_s$ is a set of all sink points of a program. A set of alarms $\Omega$ 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.

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 3.2 (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 is 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 $\tau$: $\langle f_1(\tau), \ldots, f_n(\tau) \rangle$. Then, a set of feature vectors $\Pi_\omega$ of an alarm $\omega$ is defined as follows:

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

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 $\text{Sim} : \mathbb{N}^n \times \mathbb{N}^n \to \mathbb{R}$ 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 3.3 (Score of alarm).* Given an alarm $\omega$ and a set of feature vectors of signatures $\Pi_{\mathcal{S}}$, the score of the alarm is defined as follows:

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

where $\Pi_\omega$ is a set of feature vectors of alarm $\omega$.

# 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

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

**Figure 6: Abstract domains**

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 $\mathcal{Q}$:

$$\mathcal{Q}(c)(m) = \mathcal{Q}_T(c)(m) \cup \mathcal{Q}_O(c)(m) \cup \mathcal{Q}_U(c)(m).$$

Each sub-function is defined as follows:

$$
\begin{aligned}
\mathcal{Q}_T(c)(m) &= \{c_0 \mid c_0 \in T, \langle T, \_, \_ \rangle = [\![E]\!](m)\} \\
\mathcal{Q}_O(c)(m) &= \{c_0 \mid c_0 \in T, \langle T, \top_o, \_ \rangle = [\![E]\!](m)\} \\
\mathcal{Q}_U(c)(m) &= \{c_0, \mid c_0 \in T, \langle T, \_, \top_u \rangle = [\![E]\!](m)\}
\end{aligned}
$$

where $c$ is a sink point and the abstract memory at $c$ is $m$. Function $\mathcal{Q}_T$ collects all the source points of a sink point if the argument of a sink function is tainted. Tracer uses $\mathcal{Q}_T$ to detect format string, command injection, and buffer overflow at `printf`-like functions, `exec`-like functions, and `memcpy`-like functions, respectively. $\mathcal{Q}_O$ and $\mathcal{Q}_U$ 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 describe the frequencies of the primitive operator (e.g., + and «) and the standard library calls (e.g., `strlen` and `strcmp`) on a trace. 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. Especially, we identify typical code patterns that appear in patches of common vulnerabilities.

Tracer uses cosine similarity which is a well-known measure of similarity between two vectors. Given two feature vectors $\pi_1$ and $\pi_2$, the similarity is defined as follows:

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

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

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

# 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 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.

## 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.

*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 [18, 19].
(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 4,437 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 [14].

*5.1.3 Benchmarks.* We evaluated Tracer using 273 Debian packages written in C/C++. The programs were collected from 16 common categories of Debian packages [12] such as web, sound, utils, etc. For each category, we selected 20 packages such that our taint analysis reports at least one alarm. 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 two categories: *1)* clone-based vulnerability detector *2)* pattern-based static analyzer. For each category, we chose one tool that was recently proposed and is publicly available: VUDDY [27] and Github's CodeQL [2]. We ran the baselines for the

```
1 bool ssgLoadTGA(...) {
2     GLubyte header[18];
3     fread(header, 18, 1, f);
4     ...
5     int xsize = get16u(header + 12);
6     int ysize = get16u(header + 14);
7     int bits = header[16];
8     ...
9     // potential integer overflow
10    GLubyte *image = new GLubyte [ (bits / 8) * xsize * ysize ];
11    ...
12 }
13
14 inline 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).**

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.

## 5.2 RQ1: Effectiveness

This section shows how effective TRACER is for detecting previously unknown vulnerabilities in the Debian packages. For a fair comparison, we count the number of sinks rather than source-sink pairs as shown in the previous section, because the other baselines only report sink points. We manually inspected all the reported alarms whose similarity scores are larger than 0.85. Furthermore, we randomly selected 100 alarms below the threshold and manually inspected them.

TRACER found 281 new vulnerabilities in 62 packages. Among them, 108 vulnerabilities have been confirmed by the developers and 6 CVEs have been assigned as of writing this paper. Table 2 shows the true alarms confirmed by the developers[2]. We observed that TRACER can detect various types of vulnerabilities using the signatures from different sources including known CVEs or synthetic vulnerabilities. Most of the true alarms 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 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 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, the 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 set the similarity score to 1.0.

[2]The full list of discovered vulnerabilities is available in the supplementary material.

```
1 void CWE190_Integer_Overflow__int64_t_fscanf_square_01_bad() {
2     int64_t data;
3     data = 0LL;
4     fscanf (stdin, "%" SCNd64, &data);
5     // potential integer overflow
6     int64_t result = data * data;
7     char *p = malloc(result);
8 }
```

**(a) Juliet test suite (CWE-190)**

```
1 static DiaObject *fig_read_polyline(FILE *file, DiaContext *ctx) {
2     fscanf(file, "%d_%d_%d_%d_%d_%d_%d_%d_%lf_%d_%d_%d_%d_%d_%d\n", ..., &npoints)
3     newobj = create_standard_polyline(npoints, ...);
4     ...
5 }
6
7 DiaObject *create_standard_polyline(int num_points, ...) {
8     pcd.num_points = num_points;
9     new_obj = otype->ops->create(NULL, &pcd, &h1, &h2);
10    ...
11 }
12
13 static DiaObject *polyline_create(Point *startpoint, void *user_data,
14     Handle **handle1, Handle **handle2) {
15     MultipointCreateData *pcd = (MultipointCreateData *)user_data;
16     polyconn_init(poly, pcd->num_points);
17     ...
18 }
19
20 void polyconn_init(PolyConn *poly, int num_points) {
21     // potential integer overflow
22     poly->points = g_malloc(num_points * sizeof(Point));
23     ...
24 }
```

**(b) dia-0.97.3**

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

Also, for other types of vulnerabilities, TRACER can 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]. While the example code is simple, the real-world vulnerability involves complicated aliases and indirect assignments. Nevertheless, TRACER can find that the essence of the bug is actually the same as the tutorial example as the static analyzer estimates the detailed semantics.

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 addition operations. This behavior is captured by our features. Figure 10(b) shows another example of a false alarm in grass. Since there is a bound checking for external user inputs, the integer overflow will never occur. 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 of these alarms (0.82–0.88) 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

**Table 2: List of new vulnerabilities detected by TRACER. Signature shows the sources of vulnerability signatures and Score represents the similarity scores between the true alarms and the signatures. This table reports only true alarms confirmed by the developers as of writing this paper and ✔ indicates the vulnerabilities whose CVE IDs are assigned. The information of the remaining vulnerabilities is available in the supplementary material.**

| Program | Bugs | Bug Type | Score | Signature | CVE Assigned |
|---|---|---|---|---|---|
| bsdutils | 1 | Integer Overflow | 1 | Juliet-CWE190 | ✔ |
| dia | 3 | Integer Overflow | 1 | Juliet-CWE190 | ✔ |
| htmldoc | 3 | Integer Overflow | 0.90-0.95 | CVE-2017-9181 | ✔ |
| dcraw | 3 | Integer Overflow | 0.93-0.94 | CVE-2017-9181 | ✔ |
| libplib1 | 15 | Integer Overflow | 0.76-0.93 | shntool-3.0.5 [18] | ✔ |
| libkrb5support0 | 2 | Integer Overflow | 1 | Juliet-CWE190 | - |
| groff | 1 | Integer Overflow | 1 | Juliet-CWE190 | - |
| xsane | 35 | Integer Overflow | 0.87-1.00 | Juliet-CWE190 | - |
| darktable | 5 | Integer Overflow | 1 | Juliet-CWE680 | - |
| siril | 20 | Integer Overflow | 0.87-1.00 | Juliet-CWE680 | - |
| nageru | 1 | Integer Overflow | 0.87 | CVE-2017-16663 | - |
| sane | 1 | Integer Overflow | 0.87 | CVE-2017-9181 | - |
| drawxtl | 1 | Integer Overflow | 0.79 | CVE-2017-9181 | - |
| libmjpegutils-2.1-0 | 2 | Buffer Overflow | 1 | OWASP tutorial | - |
| libaudio2 | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| xbuffy | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| xfig | 2 | Buffer Overflow | 1 | OWASP tutorial | - |
| gv | 4 | Buffer Overflow | 1 | OWASP tutorial | - |
| nedit | 5 | Buffer Overflow | 1 | OWASP tutorial | - |
| nickle | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| libpano13-3 | 1 | Format String | 0.59 | mp3rename-0.6 [18] | ✔ |

```c
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**

```c
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.**

```c
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**

```c
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 reported by TRACER**

trace-based similarity measure powered by the 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.

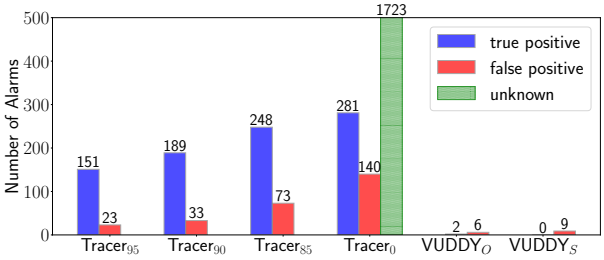



**Figure 11: Comparing the accuracy of TRACER and VUDDY.**

```
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.**

## 5.3 RQ2: Comparison with other approaches

This section compares the accuracy of TRACER with the state-of-the-art tools. Figure 11 shows the performance of each analyzer.

*5.3.1 Comparison within TRACER variants.* First, we instantiate TRACER with four variants with different thresholds of the similarity score: TRACER95, TRACER90, TRACER85, and TRACER0. Each TRACER$_n$ filters out all the alarms reported by the static analyzer whose similarity scores are less than the threshold. For example, TRACER95 reports all alarms whose scores are larger than 0.95 and TRACER0 does not suppress any alarms.

The similarity-based score of TRACER effectively filters out a large number of false positives while retaining many real bugs. The underlying taint analysis (TRACER0) is able to detect 281 vulnerabilities interspersed with many alarms (2,144). However, TRACER with a high threshold significantly suppresses a large number of false alarms. For example, TRACER95 reports only 23 false positives while detecting 151 vulnerabilities. If a lower threshold is chosen such as 0.85, the number of false positives increases compared to TRACER95 but the false positive rate is still significantly lower than TRACER0.

*5.3.2 Comparison to VUDDY.* 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 [21]. To throw away 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.

```
1 void badVaSink(char *data, ...) {
2     va_list args;
3     va_start(args, data);
4     vfprintf(stdout, data, args);
5     va_end(args);
6 }
```

**(a) Juliet test suite (CWE-134)**

```
1 void lqt_dump(char * format, ...) {
2     va_list argp;
3     va_start(argp, format);
4     vfprintf(stdout, format, argp);
5     va_end(argp);
6 }
```

**(b) libquicktime2-1.2.4**

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

VUDDY$_O$ reports 6 false positives out of 8 alarms. The reason of the false alarms is due to a practical issue regarding establishing their databases. VUDDY$_O$ collects all the modified functions in 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.

The remaining 2 true alarms of VUDDY$_O$ 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 is originated from the same vulnerable source (LibRaw-demosaic-pack-GPL2, CVE-2017-6889). Interestingly, the bug is also captured by TRACER with a high similarity score (0.92) even though TRACER does not have the origin in the 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.

VUDDY$_S$ reports 9 false alarms. This shows that VUDDY$_S$ may report false positives even if the database is carefully established. 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 function causes format string vulnerability. With this signature, 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.

VUDDY cannot detect most of the recurring vulnerabilities detected by TRACER. This is mainly because VUDDY is based on a syntactic matching algorithm 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 VUDDY from detecting semantically recurring vulnerabilities.

*5.3.3 Comparison to CodeQL.* In this section, we compare TRACER to CodeQL. Because CodeQL works differently from TRACER (i.e., the use of signatures) and there are no standard benchmarks for recurring vulnerabilities with labels, we believe that it is not fair to directly compare their results. Instead, we measure how many vulnerabilities detected by TRACER are also detected by CodeQL.

Here we intend not to compare their accuracy directly but to argue that Tracer can detect non-trivial vulnerabilities.

Our experiments show that Tracer can effectively detect recurring vulnerabilities that are not detected by CodeQL, which is based on human-written bug patterns. In total, CodeQL reports 3,557 alarms from the benchmark programs. Among all the 281 vulnerabilities detect by Tracer, CodeQL can only detect 97 bugs. This implies that it is challenging to manually strike a balance between false positives and negatives in practice. Static analysis designers typically introduce heuristics to filter out false positives. Such heuristics often make the analyzer miss real bugs when complicated program behavior is involved such as pointer dereferences, indirect function calls, or loops. On the other hand, Tracer does not use such hand-crafted heuristics but relies on the similarity measure that effectively prioritizes recurring vulnerabilities given signatures.

## 5.4 RQ3: 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 program in Figure 14.

The results indicate that Tracer is scalable to large programs. On average, the static analysis takes 115.86 seconds for each package. The time spent for the similarity checking 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 51 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 37 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 THREATS TO VALIDITY

The benchmarks and signature vulnerabilities used in our experiments may not be representative. We used open source programs written in C/C++. Thus, it may have different results for programs in other languages or from the industry. However, we collected the benchmarks from a wide range of categories, and signatures are also from diverse sources of vulnerability data.

We have restricted our attention to specific types of vulnerabilities that can be discovered by our taint analysis. We need to generalize Tracer to arbitrary types of vulnerabilities in future work. However, the types of vulnerabilities used in our experiments are common in practice and also targeted by other work for vulnerability detection [18, 19, 45].

# 7 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 [22, 27, 32, 38, 48]. These approaches transform buggy code fragments within a certain boundary (e.g., functions) into various forms of vulnerability signatures such as hashes [22, 27] or dependency graphs [38, 48]. 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.

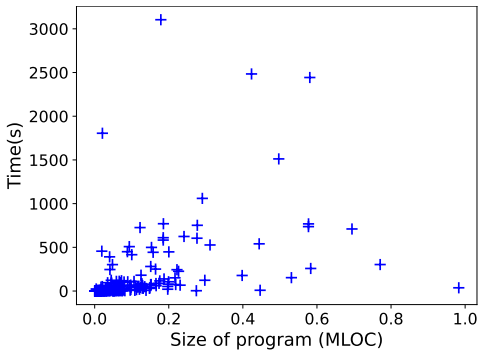



**Figure 14: Running time of Tracer by program size.**

Most of the existing static analyses that take into account code patterns highly rely on manual design [2, 3, 16]. FindBugs [3], SpotBugs [1], and ErrorProne [16] 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. However, it is still non-trivial for ordinary developers to write desired queries for their own purposes [33]. Instead, Tracer is based on a general static analysis designed by experts that provides an accessible framework for developers without static analysis expertises.

Researchers have proposed many techniques to detect code clones ranging from syntactic ones [6, 9, 20, 23, 25, 29, 30, 40, 41, 44, 47] to semantic ones [15, 24, 26, 28, 42, 46, 49]. Since their goal is to detect generally similar code fragments, they are not suitable to accurately find recurring vulnerabilities even via code reuse [27, 48]. 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 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 [31, 39], expected reactions from developers [17] or relevance to a specific commit [19]. To our best knowledge, none of the existing work ranks alarms by similarity to a specific known vulnerability.

# 8 CONCLUSION

We proposed Tracer, a framework for detecting 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.

# REFERENCES


[1] 2021. SpotBugs. https://spotbugs.github.io

[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. *European Conference on Object-Oriented Programming (ECOOP 2016)*.

[3] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Softw.* 25 (2008). Issue 5.

[4] Paul Black. 2018. Juliet 1.3 Test Suite: Changes From 1.2.

[5] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. *NASA Formal Methods - Third International Symposium (NFM)* 6617.

[6] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. *36th International Conference on Software Engineering (ICSE 2014)*.

[7] CodeQL. 2021. CodeQL CWE queries. https://github.com/github/codeql/tree/main/cpp/ql/src/Security/CWE

[8] CodeQL. 2021. TaintedAllocationSize.ql. https://github.com/github/codeql/blob/main/cpp/ql/src/Security/CWE/CWE-190/TaintedAllocationSize.ql

[9] James R Cordy and Chanchal K Roy. 2011. The NiCad Clone Detector. *The 19th IEEE International Conference on Program Comprehension (ICPC 2011)*.

[10] The MITRE Corporation. 2021. Common Vulnerabilities and Exposures.

[11] The MITRE Corporation. 2021. Common Weakness Enumeration.

[12] Debian. 2021. Debian Packages. https://packages.debian.org/sid/

[13] Will Dietz, Peng Li, John Regehr, and Vikram S Adve. 2012. Understanding integer overflow in C/C++. *34th International Conference on Software Engineering (ICSE 2012)*.

[14] The OWASP Foundation. 2021. Attacks.

[15] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. *30th International Conference on Software Engineering (ICSE 2008)*.

[16] Google. 2021. Error Prone. https://errorprone.info

[17] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding patterns in static analysis alerts: improving actionable alert ranking. *11th Working Conference on Mining Software Repositories (MSR 2014)*.

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

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

[20] Benjamin Hummel, Elmar Jürgens, Lars Heinemann, and Michael Conradt. 2010. Index-based code clone detection: incremental, distributed, scalable. *26th IEEE International Conference on Software Maintenance (ICSM 2010)*.

[21] IoTcube. 2021. IoTcube. https://iotcube.korea.ac.kr

[22] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. *IEEE Symposium on Security and Privacy (S&P 2012)*.

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. *29th International Conference on Software Engineering (ICSE 2007)*.

[24] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. *Proceedings of the 8th International Symposium on SoftwareTesting and Analysis (ISSTA 2009)*.

[25] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28 (2002). Issue 7.

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

[27] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. *IEEE Symposium on Security and Privacy (S&P 2017)*.

[28] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. *Proceedings of 8th International Static Analysis Symposium (SAS 2001)* 2126.

[29] Rainer Koschke. 2014. Large-scale inter-system clone detection using suffix trees and hashing. *J. Softw. Evol. Process.* 26 (2014). Issue 8.

[30] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. *13th Working Conference on Reverse Engineering (WCRE 2006)*.

[31] Ted Kremenek and Dawson R Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. *Proceedings of 10th International Static Analysis Symposium (SAS 2003)* 2694.

[32] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. *34th International Conference on Software Engineering (ICSE 2012)*.

[33] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. ARBITRAR : User-Guided API Misuse Detection. *IEEE Symposium on Security and Privacy (S&P 2021)* (2021).

[34] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proc. ACM Program. Lang.* 1 (2017). Issue OOPSLA.

[35] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. *IEEE Symposium on Security and Privacy (S&P 2015)*.

[36] OWASP. 2021. Buffer Overflow via Environment Variables. (2021). https://owasp.org/www-community/attacks/Buffer_Overflow_via_Environment_Variables

[37] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. *IEEE Symposium on Security and Privacy (S&P 2020)*.

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

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

[40] Hitesh Sajnani, Vaibhav Saini, and Cristina Videira Lopes. 2015. A parallel and efficient approach to large scale clone detection. *J. Softw. Evol. Process.* 27 (2015). Issue 6.

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

[42] Abdullah Sheneamer and Jugal Kalita. 2016. Semantic Clone Detection Using Machine Learning. *15th IEEE International Conference on Machine Learning and Applications (ICMLA 2016)*.

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

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

[45] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012)*.

[46] Huihui Wei and Ming Li. 2018. Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training. *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*.

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

[48] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. *29th USENIX Security Symposium (USENIX Security 2020)*.

[49] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. *Proceedings of the 27th International Conference on Program Comprehension (ICPC 2019)*.

**Table 3: List of new vulnerabilities detected by Tracer. Signature shows the sources of vulnerability signatures and Score represents the similarity scores between the true alarms and the signatures.**

| Program | Bugs | Bug Type | Score | Signature | CVE Assigned |
|---|---|---|---|---|---|
| 4ti2 | 3 | Integer Overflow | 0.71-1.00 | Juliet-CWE190 | - |
| bowtie2 | 1 | Integer Overflow | 0.74 | CVE-2017-9181 | - |
| bsdutils | 1 | Command Injection | 0.86 | CVE-2016-10729 | - |
| bsdutils | 1 | Integer Overflow | 1 | Juliet-CWE190 | ✔ |
| bwbasic | 1 | Buffer Overflow | 0.44 | CVE-2018-1100 | - |
| coinor-libclp1 | 6 | Integer Overflow | 1 | Juliet-CWE190 | - |
| crafty | 1 | Integer Overflow | 0.86 | CVE-2017-1000229 | - |
| cron | 1 | Command Injection | 0.68 | OWASP tutorial | - |
| crrcsim | 2 | Integer Overflow | 0.85-0.90 | CVE-2017-16663 | - |
| darktable | 5 | Integer Overflow | 1 | Juliet-CWE680 | - |
| dcraw | 3 | Integer Overflow | 0.93-0.94 | CVE-2017-9181 | ✔ |
| dia | 3 | Integer Overflow | 1 | Juliet-CWE190 | ✔ |
| drawxtl | 1 | Integer Overflow | 0.79 | CVE-2017-9181 | - |
| dvbstreamer | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| elvis-tiny | 3 | Buffer Overflow | 0.50-1.00 | OWASP tutorial | - |
| gap-guava | 2 | Integer Overflow | 1 | Juliet-CWE190 | - |
| gnuplot | 2 | Format String | 0.82 | Juliet-CWE134 | - |
| grass | 22 | Buffer Overflow | 0.41-1.00 | OWASP tutorial | - |
| groff | 1 | Integer Overflow | 1 | Juliet-CWE190 | - |
| gv | 4 | Buffer Overflow | 1 | OWASP tutorial | - |
| htmldoc | 3 | Integer Overflow | 0.90-0.95 | CVE-2017-9181 | ✔ |
| hugin | 9 | Integer Overflow | 0.87-1.00 | Juliet-CWE190 | - |
| ispell | 4 | Buffer Overflow | 1 | OWASP tutorial | - |
| libaudio2 | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| libfreeimage3 | 1 | Buffer Overflow | 0.83 | CVE-2017-6313 | - |
| libkrb5support0 | 2 | Integer Overflow | 1 | Juliet-CWE190 | - |
| liblinear-tools | 1 | Buffer Overflow | 0.3 | CVE-2018-1100 | - |
| liblinear-tools | 2 | Integer Overflow | 0.93-1.00 | Juliet-CWE190 | - |
| liblrs0 | 1 | Integer Overflow | 0.91 | Juliet-CWE191 | - |
| libmjpegutils-2.1-0 | 2 | Buffer Overflow | 1 | OWASP tutorial | - |
| libmount1 | 1 | Command Injection | 0.72 | CVE-2015-9059 | - |
| libmount1 | 1 | Integer Overflow | 1 | Juliet-CWE190 | - |
| libpano13-3 | 1 | Format String | 0.59 | mp3rename-0.6 [18] | ✔ |
| libpano13-3 | 3 | Integer Overflow | 0.87 | CVE-2017-16663 | - |
| libplib1 | 15 | Integer Overflow | 0.76-0.93 | shntool-3.0.5 [18] | ✔ |
| libquicktime2 | 22 | Integer Overflow | 0.85-0.95 | CVE-2017-9181 | - |
| lp-solve | 7 | Integer Overflow | 1 | Juliet-CWE190 | - |
| mdadm | 1 | Buffer Overflow | 0.16 | CVE-2019-14523 | - |
| minidlna | 1 | Integer Overflow | 0.94 | Juliet-CWE190 | - |
| nageru | 1 | Integer Overflow | 0.87 | CVE-2017-16663 | - |
| nedit | 5 | Buffer Overflow | 1 | OWASP tutorial | - |
| newmail | 1 | Format String | 0.82 | Juliet-CWE134 | - |
| nickle | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| nickle | 1 | Command Injection | 0.67 | Juliet-CWE78 | - |
| octave-nan | 11 | Integer Overflow | 0.87-1.00 | Juliet-CWE190 | - |
| printer-driver-foo2zjs | 1 | Integer Underflow | 0.94 | Juliet-CWE191 | - |
| r-cran-lpsolve | 7 | Integer Overflow | 1 | Juliet-CWE190 | - |

# A COMPLETE LIST OF VULNERABILITIES

Table 3 and Table 4 show all bugs that are found by Tracer.

**Table 4: List of new vulnerabilities detected by TRACER.**

| Program | Bugs | Bug Type | Score | Signature | CVE Assigned |
|---|---|---|---|---|---|
| rawtherapee | 5 | Integer Overflow | 0.86-1.00 | Juliet-CWE680 | - |
| rlwrap | 1 | Command Injection | 0.82 | Juliet-CWE78 | - |
| rtcw | 1 | Buffer Overflow | 0.4 | CVE-2018-1100 | - |
| sane | 1 | Integer Overflow | 0.87 | CVE-2017-9181 | - |
| scheme48 | 1 | Integer Overflow | 0.85 | CVE-2009-1570 | - |
| seaview | 1 | Buffer Overflow | 0.56 | CVE-2018-1100 | - |
| siril | 20 | Integer Overflow | 0.87-1.00 | Juliet-CWE680 | - |
| siril | 3 | Integer Underflow | 0.82 | Juliet-CWE191 | - |
| snap | 2 | Buffer Overflow | 1 | OWASP tutorial | - |
| snap | 1 | Integer Overflow | 1 | Juliet-CWE680 | - |
| stk | 1 | Integer Overflow | 0.87 | shntool-3.0.5 [18] | - |
| sweed | 4 | Integer Overflow | 1 | Juliet-CWE190 | - |
| tcliis | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| tome | 8 | Format String | 0.96 | CVE-2015-8106 | - |
| vacation | 1 | Command Injection | 0.67 | Juliet-CWE78 | - |
| w3m | 1 | Format String | 0.96 | CVE-2015-8106 | - |
| wily | 7 | Buffer Overflow | 0.47-1.00 | OWASP tutorial | - |
| xbuffy | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| xfig | 2 | Buffer Overflow | 1 | OWASP tutorial | - |
| xsane | 35 | Integer Overflow | 0.87-1.00 | Juliet-CWE190 | - |
| xwpe | 1 | Buffer Overflow | 1 | OWASP tutorial | - |
| xwpe | 1 | Command Injection | 0.87 | Juliet-CWE78 | - |
| xwpe | 3 | Integer Overflow | 0.87-0.89 | Juliet-CWE190 | - |
| zangband | 1 | Buffer Overflow | 0.77 | CVE-2017-6313 | - |
| zangband | 7 | Format String | 0.97 | CVE-2015-8106 | - |
| zangband | 1 | Integer Overflow | 0.93 | CVE-2017-9181 | - |