VGRAPH: A Robust Vulnerable Code Clone Detection System Using Code Property Triplets

Benjamin Bowman H. Howie Huang
Graph Computing Lab
George Washington University
{*bbowman410, howie*}*@gwu.edu*

***Abstract*—Software vulnerabilities are a common attack vector for cyber adversaries. This problem has been exacerbated by the wealth of open-source software projects, as code is often copy-pasted to new locations. This causes a serious problem when a new security vulnerability is discovered in a particular software project, as it may potentially affect many others. Discovering vulnerable code reuse in source code is known as vulnerable code clone detection. This is a very challenging problem as the cloned code has the potential to be modified, sometimes significantly, from the original code, while still retaining the underlying vulnerability. Existing vulnerable clone detection techniques are either too strict, missing vulnerabilities when they have subtle modifications, or are too narrow, applicable only to a small number of vulnerability types. In this work we present VGRAPH, a technique for identifying vulnerable code clones, which is more robust to code modification, while still remaining generic to all vulnerability types. VGRAPHs are representations of vulnerable source code comprising three graph-based components representing code property relationships extracted from the *contextual code*, the *vulnerable code*, and the *patched code*. We develop a matching algorithm utilizing these three graph-based components which is able to identify vulnerable code clones with a precision of 98% and recall of 97%. Even for highly modified code clones, we are able to identify over 100 more vulnerable clones than the best performing comparison work ReDeBug. When we apply our technique to several versions of popular software packages (e.g., FFMpeg, OpenSSL), we are able to identify 10 vulnerabilities which were silently patched and are not listed in the National Vulnerability Database.**

***Index Terms*—vulnerability, code clone, source code, static analysis, software security**

1. Introduction

In 2019 there were over 17,000 new vulnerabilities published in the National Vulnerability Database (NVD) [4]. In the last five years alone, there have been over 60,000 new additions into the NVD, including the vulnerability exploited in the high profile Equifax hack of 2017, which exposed personal data of over 145 million Americans [20]. In a time when our personal and professional lives, as well as critical infrastructure, all rely on computer systems and consequently computer software, it is imperative that we find a way to identify vulnerabilities in source code before they are exploited by cyber adversaries.

Exacerbating the problem of vulnerable code is the growth of popular open-source software packages distributed freely on the Internet. At the time of writing this, the most popular source code repository GitHub [2] has over 20 million public source code projects, and 36 million registered users. The purpose of open-source code is to allow for the open distribution and reuse of computer software. Unfortunately, this leads to an increase in vulnerable code clones, which occur when unknowingly vulnerable code is copy-pasted from one location to another. When the vulnerability is discovered and patched, there is no guarantee that all occurrences of that vulnerability in all other locations within and across various projects and versions are patched as well. This means the source code with the vulnerable code clones will likely go unpatched, leaving them at risk for malicious exploitation.

Existing techniques for vulnerable code clone detection fall into two main categories: *code similarity* [16] [14] [12] [17] [19] [9], and *functional similarity* [18] [25] [23] [27] [28] [5] [24]. In code similarity approaches, target source code is compared against a set of known vulnerable code samples, and determined to be vulnerable if a threshold of similarity is met. Code similarity approaches are typically classified based on four types of detection coverage [21]: type-1 (identical), type-2 (syntactically equivalent), type-3 (syntactically similar), and type-4 (semantically similar). Existing code similarity techniques perform well when detecting identical (type-1) or syntactically equivalent (type-2) code clones, but suffer when the code has increased modification, such as the addition and deletion of lines of code (type-3 and type-4).

On the other hand, functional similarity approaches seek to generate abstract functional patterns of code which model vulnerable behavior. If the functional patterns are simple, the techniques suffer from low accuracy as they generate many false positives. Conversely, if the functional patterns are complex, they have the capability to identify vulnerable code clones with significant modifications, up to and including type-4 code clones. However, due to the complexity of building such a pattern, these techniques are typically specialized to only a small class of vulnerabilities, or to a particular source code project, rendering them ineffective as general-purpose vulnerable code clone detection techniques.

A recent study [10] of 35 software projects found that 50-60% of all vulnerable code clones were the result of type-3 syntactically similar code clones. However, existing techniques suffer to detect type-3 code clones, as they are

DOI 10.1109/EuroSP48549.2020.00012

either too strict, covering only identical or near-identical code clones, or too narrow, spanning only a few vulnerability classes or source code projects.

In this work we introduce VGRAPH, a code-similarity style technique which is capable of identifying highly modified vulnerable code clones, while remaining generic to all vulnerability types. VGRAPH abstracts vulnerabilities in source code to the graph domain, allowing for the ability to identify key relationships between textual elements that are not directly discernible from the text alone. Additionally, we utilize not only the vulnerable code, but also the patched code, to identify specific relationships in the graph that are tied directly to the vulnerable code segment, the patched code segment, and the contextual code of a particular vulnerability. By separating the vulnerability representation into these three components, we are able to develop our matching algorithm to tolerate modifications at each level independently, providing more robust detection of modified vulnerable code clones.

We build a database of VGRAPHs by mining vulnerable and patched source code for 8 popular open source projects from GitHub [2], resulting in VGRAPHs for 711 vulnerabilities (CVEs) spanning 51 vulnerability types (CWEs). We download an additional 5,566 vulnerable and patched code clones from different versions of the source code as our test dataset. Our evaluation shows that VGRAPH is able to accurately identify vulnerabilities in the test dataset with an F1 score of 97%. When detecting highly modified vulnerable code clones, VGRAPH is able to achieve an F1 score of 85% compared to 74% and 50% by state-of-art vulnerable code clone detection systems ReDeBug [11] and VUDDY [14] respectively.

To evaluate real-world applicability, we utilize the VGRAPH system to identify previously unknown vulnerable code clones. We apply our method on several versions of popular software packages FFmpeg and OpenSSL, and identify 10 vulnerable code clones that were silently patched and are not listed in the NVD.

In summary, we make the following contributions:

- A novel graph-based source code vulnerability representation containing key relationships between the vulnerable code, the patched code, and the vulnerability context, which is generic to all vulnerability types.
- A matching algorithm capable of identifying highly modified vulnerable code clones while still retaining the ability to differentiate between vulnerable and patched code samples.
- A framework for generating VGRAPHs in a data-driven and automated way allowing for the methods to scale to newly published vulnerabilities with ease.

The remainder of this paper is organized as follows. Section 2 will discuss the background and related work. Section 3 will provide a high level overview of the VGRAPH system. Section 4 will discuss our approach to modeling source code and our novel VGRAPH representation. Section 5 will discuss our technique for detecting vulnerable code clones. Section 6 will explain how we acquire our vulnerable code samples in an automated way. Section 7 will discuss our evaluation and experimental results. Section 8 will discuss some limitations and future work and Section 9 will conclude the work.

```
1  void foo() {
2      int x = intput();
3      if (x > MIN) {
4          int y = x * 10;
5          output(y);
6      }
7  }
```

(a) Vulnerable code as the call to $output(y)$ has an unchecked upper bound on the variable y.

```
1  void foo() {
2      int x = intput();
3      if (x > MIN) {
4          int y = x * 10;
5          if (y < MAX)
6              output(y);
7      }
8  }
```

(b) Patched code as the variable y is checked against the upper bound MAX prior to the call $output(y)$.

Figure 1: An example of a source code function in both the vulnerable state and the patched state.

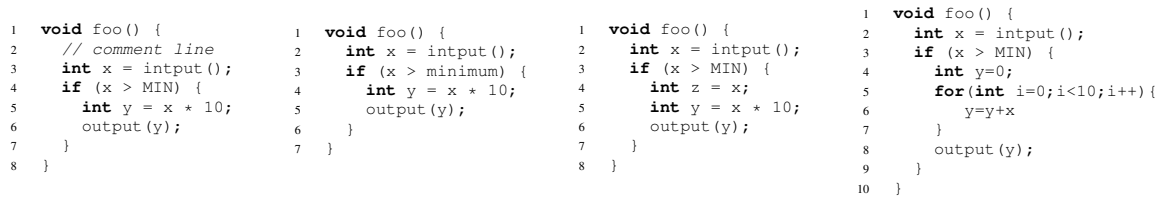
2. Background and Related Work

A vulnerability in source code can be defined as any weakness of the code which can be exploited to perform unauthorized actions. For example, Figure 1 shows a synthetic function foo both before (Figure 1a) and after (Figure 1b) a vulnerability was discovered and patched. Both versions of the function read some input into a variable x on line 2. Then, both compare that input value against some variable MIN, and if x is larger then they will proceed inside the conditional statement. Both versions then perform some transformation of x into the variable y. Next, in the vulnerable version of foo, the value of y is simply passed to the $output$ function. Differently, in the patched version of foo, the value of y is first compared against some variable MAX, and is only passed to the $output$ function provided that y is less than MAX. Based on both the vulnerable version and the patch version of the function, we can infer that the function $output$ is only defined on values less than MAX, and is not safe to use with values above that limit. Thus, the vulnerability in this case was the omitted upper-bounds check on the value passed to the function $output$.

When vulnerabilities are discovered in software, they go through a process where they are assigned a Common Vulnerability Enumeration (CVE) identifier. This uniquely identifies the instance of a particular vulnerability, and is tied to specific versions of a software product. Additionally, CVEs are associated with Common Weakness Enumeration (CWE) identifiers, which represent different classes of vulnerabilities, such as improper input validation (CWE-200), out-of-bounds read (CWE-125), and use-after-free (CWE-416).

2.1. Clone Type Taxonomy

To compare the coverage of code clone detection techniques, we use the standard clone type taxonomy as



```
1  void foo() {
2    // comment line
3    int x = intput();
4    if (x > MIN) {
5      int y = x * 10;
6      output(y);
7    }
8  }
```

(a) Type-1

```
1  void foo() {
2    int x = intput();
3    if (x > minimum) {
4      int y = x * 10;
5      output(y);
6    }
7  }
```

(b) Type-2

```
1  void foo() {
2    int x = intput();
3    if (x > MIN) {
4      int z = x;
5      int y = x * 10;
6      output(y);
7    }
8  }
```

(c) Type-3

```
1  void foo() {
2    int x = intput();
3    if (x > MIN) {
4      int y=0;
5      for(int i=0;i<10;i++){
6        y=y+x
7      }
8      output(y);
9    }
10 }
```

(d) Type-4

Figure 2: Example of type-1 through type-4 code clones of the source code introduced in Figure 1.

introduced in [21]:

Type-1: Identical code except changes to whitespace and comment lines.

Type-2: Syntactically identical code with modifications to identifiers, literals, types, whitespace, and comments.

Type-3: Syntactically similar code with addition and/or deletion of lines, as well as modification to identifiers, literals, types, whitespace, and comments.

Type-4: Syntactically different code with the same functionality (i.e., semantically similar)

Figure 2 shows an example of each type of code clone for the vulnerable function introduced in Figure 1a. The type-1 code clone has a single comment line added on line 2. The type-2 clone has the bounds check variable MIN renamed to $minimum$ on line 3. The type-3 code clone defines an additional variable z and initializes it to the value of x on line 4. The Type-4 code clone has replaced the $y = x * 10$ multiplication statement instead with a series of 10 addition operations on lines 5-7. Note that each example represents a pure clone, meaning it only has the modification most associated with each type. However, based on the definitions, each clone type can also include the modifications associated with the types below it. For example, in the type-3 code clone, we could also rename the variable MIN to $minimum$, and it would still be considered a type-3 clone.

Type 1-3 clones can be thought of as clones which are textually similar, while type-4 clones are functionally similar. The related works can be broadly categorized based on these two similarity measures.

2.2. Textual Similarity Techniques

The textual similarity techniques generally involve dividing a program into individual units (e.g., files, functions, tokens, etc), and performing a similarity measurement against a set of known vulnerable code samples. In general, these approaches perform well at detecting type-1 and type-2 code clones, but fail to detect type-3 and type-4 code clones. For example, VUDDY [14] is a technique which relies on hashing vulnerable functions to allow for a quick table lookup to determine if a target function is vulnerable. VUDDY is able to detect type-1 and type-2 vulnerable code clones, but it will fail to detect type-3 and type-4 clones.

As textual similarity techniques grow more abstract in order to detect type-3 vulnerable code clones, they often have a severe decrease in accuracy. This is due to the fact that a patched function is often itself a type-3 code clone of the vulnerable function. Therefore, any techniques that do not take into account information from the patch will suffer from the inability to differentiate between vulnerable code clones and patched code clones.

SourcerCC [22], CPMiner [16], and CCFinder [13] are all code clone detection techniques which tokenize source code and identify clones based on some measure of token overlap. In each case, however, their applicability to vulnerable code clone detection is inhibited since they do not take into account information from the patch. Similarly, DECKARD [12] builds abstract syntax trees (AST) from the code and identifies similar subtree structures as clones, but again suffers from the inability to differentiate between vulnerable and patched code.

On the other hand, a recent work ReDeBug [11] is a technique which does use the information in both the vulnerable code and the patched code. ReDeBug performs sequence based matching utilizing the *diff* files associated with a particular vulnerability. A *diff* file contains the lines that were explicitly modified during the transition of the code from vulnerable to patched, as well as some context code within close textual proximity. This allows ReDeBug to detect some type-3 clones, however if the code modification is near the location of the lines modified during the patch process, this technique will fail to detect the vulnerable clone.

Because there are so many different textual similarity techniques, each with their own strengths and weaknesses, VulPecker [17] developed a technique which identifies a vulnerability-to-similarity-algorithm mapping. This way each algorithm can be applied to the vulnerabilities to which they are best suited. However, this approach is still limited by the underlying accuracy of the similarity algorithms, and only achieves a recall score of 60%, meaning many vulnerable clones were left undetected.

2.3. Functional Similarity Techniques

The functional similarity techniques are markedly different from the textual similarity techniques discussed previously as they attempt to model functional patterns indicative of vulnerable code, rather than using the textual contents of the code directly. This means these techniques are better suited to identifying type-3 and type-4 vulnerable code clones. However, in general, these techniques are either exceedingly noisy with many false positives and false negatives, or very narrow in scope as they apply to only specific vulnerability types, or are tied to particular source code projects.

The simplest functional similarity approaches are based on manually defined patterns of functionality which have been deemed vulnerable or unsafe based on software security experts. These patterns exist in many open-source vulnerability discovery tools such as FlawFinder [24] and the Rough Auditing Tool For Security (RATS) [5]. Due to the simplicity of the functionality defined in the patterns, these techniques often have many false positives and false negatives. In addition, they require an expert to manually define the functional patterns of vulnerable code.

Other techniques resort to identifying anomalous functionality as a proxy for vulnerable functionality. Yamaguchi et al. [29] [26] identifies anomalous input validation routines according to other functions in the same code repository. Chang et al. [8] identifies similar missing conditions by mining program dependence graphs (PDG) and identifying outliers. Not only are these techniques tied to a specific class of vulnerability, but their results are based on individual code repositories, and information gleaned from one project is likely not applicable to another.

In other related works, Yamaguchi et al. [28], [27] [25] extrapolates known vulnerabilities by mining information from Abstract Syntax Trees (AST) and Code Property Graphs (CPG). Based on a seed vulnerability, they use graph traversals to extrapolate to new vulnerable code clones. While these techniques are capable of identifying type-4 code clones, they are highly coupled to the type of vulnerabilities being extrapolated, as well as the code repositories on which the analysis is run.

VulDeePecker [18] introduced a deep learning framework for learning the features necessary to identify vulnerable source code functionality. However this requires a robust dataset so that the machine learning algorithm can accurately learn the vulnerable functional pattern. Because of this requirement, their method was only evaluated on two vulnerability classes, and would require significant manual effort to extend to additional types.

2.4. VGRAPH Comparison

Comparatively, VGRAPH is a textual similarity vulnerable code clone detection system, with emphasis placed on accurately detecting type-3 code clones, while remaining generic to all vulnerability types. Our technique is most similar to ReDeBug, as we focus on the lines of source code that are modified during the patching process. However, unlike ReDeBug, we utilize a graph representation of the code rather than direct sequences of text, which allows for more robust detection of type-3 clones as the underlying text can change while the resulting graph structure remains the same. Our technique is generic to all vulnerability types and can be applied across many programs, contrasting the functional similarity approaches which often only target a few vulnerability classes.

3. VGRAPH System

Prior to discussing any individual component, we will first provide an overview of the VGRAPH vulnerable code clone detection system. A high level system architecture diagram is shown in Figure 3. The VGRAPH system consists of two distinct phases: a *Generation phase*, and a *Detection Phase*. During the *Generation phase*, source

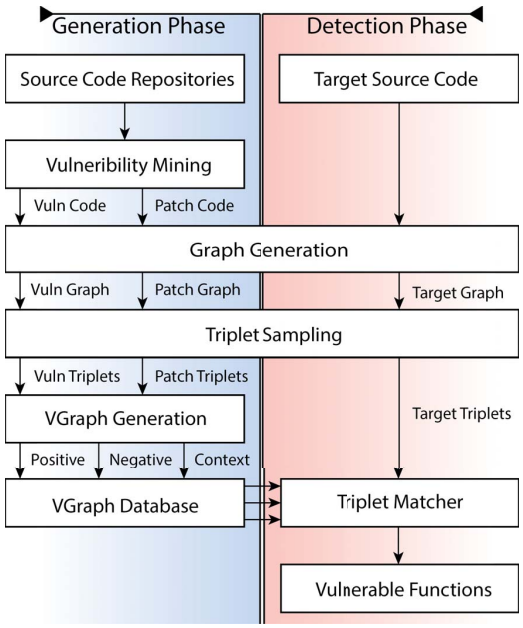


Figure 3: VGRAPH Vulnerability Detection System.

code repositories are mined in an automated way to identify references to known vulnerabilities, and the relevant source code is downloaded and cataloged. We download both the vulnerable source code, as well as the patched source code, as both are needed to accurately differentiate between vulnerable code clones and their patched counterparts. These pairs of vulnerable and patched code samples are then sent to the *Graph Generator*, which converts the raw source code into the highly expressive Code Property Graph (CPG) representation [25]. We utilize the open source utility *Joern* [3] to accomplish this. In order to avoid expensive graph matching, the CPGs of the vulnerable and patched code are then sampled via the *Triplet Sampler*, which converts the graphs into a set of code property triplets. These triplets are then sent to the VGRAPH *Generator*, which is responsible for generating the positive, negative, and context triplets of each vulnerability. This process is discussed at length in Section 4. These triplets and their corresponding vulnerability CVE identifier are then stored in the VGRAPH *Database*.

During the *Detection Phase*, target source code that is to be evaluated is first converted into CPGs via the *Graph Generator* and then sampled via the *Triplet Sampler* using the same pipeline as during the *Generation Phase*. The target triplets are then sent to the *Triplet Matcher*, which performs the matching algorithm described in Section 5. This component will detect if any of the target functions are vulnerable code clones of any of the functions with VGRAPHs in the VGRAPH *Database*.

4. Vulnerability Representation

Determining a representation for vulnerable source code generally requires two main design choices. One is the level of granularity (e.g., program level, file level, function level, line level, token level), and the other is the

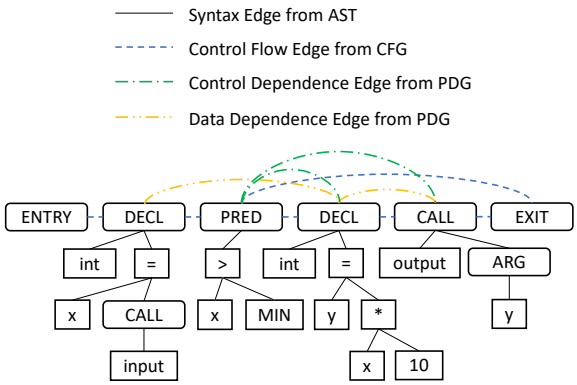


Figure 4: Example Code Property Graph for the source code function in Figure 1a.

source code representation (e.g., text, metric, tree, graph). We discuss our choices below.

Granularity Level. We choose to operate at the line level of granularity, meaning a vulnerability will be defined as a series of lines of source code. When vulnerable code is patched, the act of patching involves the addition, subtraction, or modification of specific lines of code. Thus, we believe operating at the line level of granularity is the most logical way to represent a vulnerability if we hope to be able to differentiate between vulnerable and patched functions, while also being able to detect highly modified code clones. Additionally, popular source code repositories such as GitHub [2] make it possible to acquire the source code for both vulnerable and patched code samples for a wide variety of programs. This allows us to acquire the specific source code lines which are added, removed, or modified for a wide variety of vulnerabilities in a reliable and automated way.

Source Code Representation. We choose to represent source code as a graph. Specifically, we choose to utilize the Code Property Graph (CPG) [25] which is a multigraph containing the representative nodes and edges from the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). Because vulnerabilities are highly complex, capable of manifesting in many different ways, it is important that we have a representation of source code that is capable of modeling a wide variety of complex relationships. The CPG is one of the most complex and expressive representations of source code available, as it has the ability to simultaneously model syntactic structure, control and data flow, and control and data dependence. An example CPG for the source code in Figure 1a is shown in Figure 4. We can see how the AST provides the general syntactic structure of the source code, tokenizing each statement and categorizing them as declaration statements (DECL), call statements (CALL), predicate statements (PRED), etc. The CFG then provides an ordering to the AST elements, identifying all the possible logic traversal paths, such as the path from the predicate statement to the variable declaration, or the function exit. Finally, the PDG provides information on control and data dependence between elements, such as the data dependence between the call to the *output* function and the previous variable declaration of y. This source code representation will allow us to extract relationships between source code elements that would not be directly discernible based on the textual contents alone. Further discussion of the CPG including a full list of node and edge types can be found in Appendix A.

4.1. VGRAPH Structure

Conceptually, the VGRAPH structure is the combination of elements extracted from the graph representation of both the vulnerable function as well as the patched function for a particular vulnerability. Each graph element is attributed with identifiers indicating if it was found in exclusively the vulnerable function, exclusively the patched function, or both. This way, the VGRAPH structure captures key relationships indicative of the vulnerability, the patch, and the necessary function context.

The first step in generating a VGRAPH for a particular vulnerability is to generate the Code Property Graph of both the vulnerable function as well as the patched function. A CPG is a directed, edge-labeled, attributed multigraph of the form $G = (V, E, \lambda, \mu)$ where V is a set of nodes, E is a set of directed edges, λ is an edge labeling function, and μ is a node property labeling function. We utilize the open source tool *Joern* [3] to accomplish this task.

At this point, we need to identify the overlapping graph elements so that we can extract the key relationships related to explicitly the vulnerable and patched code. As subgraph isomorphism is a computationally expensive NP-complete problem, we utilize a sampling technique where we convert the graphs into triplets of the form $(Source, Relationship, Destination)$, where $Source$ is a source node property, $Destination$ is a destination node property, and $Relationship$ is the type of edge between these two nodes as found in the CPG. For each node in the graph, we extract the code property triplets as described in Algorithm 1. We typically generate four separate triplets as seen in lines 6 through 9 of the algorithm for each edge in the CPG. Line 6 generates a triplet containing the textual source code contents. We could stop here, but this representation would not lend itself to type-2 and beyond code clones, so we add additional triplets with varying levels of abstraction. In Line 7 and 8, we abstract the source node and destination node respectively to their node types, rather than the textual contents. Finally, in line 9, we abstract both nodes to their type representation.

By generating the code property triplets in this manner, we are able to not only capture the relationship between

Algorithm 1 Triplet Sampler

```
1:  procedure TRIPLET_SAMPLER(G)
2:      triplets = []
3:      for n1 ∈ G.nodes do
4:          for n2 ∈ G.neighbors(n1) do
5:              for e ∈ G.edges(n1, n2) do
6:                  triplets.append(n1.code, e, n2.code)
7:                  triplets.append(n1.type, e, n2.code)
8:                  triplets.append(n1.code, e, n2.type)
9:                  triplets.append(n1.type, e, n2.type)
10:     return triplets
```

TABLE 1: A sample of the positive, negative, and context triplets generated for the vulnerability in Figure 1.

Positive Triplets (PT)	Negative Triplets (NT)	Context Triplets (CT)
$(x > MIN, CONTROLS, output(y))$	$(y < MAX, CONTROLS, output(y))$	$(x = input(), DEF, x)$
$(y = x * 10, FLOWS\_TO, output(y))$	$(y = x * 10, FLOWS\_TO, y < MAX)$	$(y = x * 10, IS\_AST\_PARENT, x * 10)$
$(y = x * 10, FLOWS\_TO, Expression)$	$(y = x * 10, FLOWS\_TO, Condition)$	$(x = intput(), REACHES, y = x * 10)$

the textual source code contents, but also the more abstract relationships between types of source code statements. This way, even if a piece of source code has textual modification, there will still be triplets containing relevant information in our VGRAPH structure.

We can now generate a VGRAPH based on the set of vulnerable code property triplets V extracted from the vulnerable function, and the patched code property triplets P extracted from the patched function. We define the three components of a VGRAPH as follows:

Positive Triplets (PT): This is the set of triplets from the vulnerable graph which are not found in the patched graph. Intuitively, this can be thought of as the specific relationships in the graph which contributed to it being vulnerable. Note that this is not strictly textual modifications, as textual modification will result in additional changes to the graph structure, which is explicitly captured by this approach. Formally, *PT* can be defined as:

$$PT = V \setminus P$$

Negative Triplets (NT): This is the set of triplets from the patched graph which are not found in the vulnerable graph. Intuitively, this can be thought of as the specific relationships of the graph which contribute to it being patched to a particular vulnerability. Formally, *NT* can be defined as:

$$NT = P \setminus V$$

Context Triplets (CT): This is the set of triplets that are shared by both the vulnerable and the patched graph. Intuitively, these are the contextual relationships in the function that were not modified during the transition of the function from vulnerable to patched. As vulnerabilities are highly context dependent, this component is very important to represent the required context for the vulnerability to be present. Formally, *CT* can be defined as:

$$CT = V \cap P$$

These three structures combined represent a VGRAPH for a particular vulnerability. Table 1 provides a sample of the VGRAPH triplets generated for the example vulnerable and patched code in Figure 1. From the table we can see that the PT and NT accurately capture key information as to what relationships between source code elements are related to the function being identified as vulnerable vs. patched. In the first row we can see that the PT and NT capture the different control dependence relationships on the $output(y)$ call, with the source code $x > MIN$ controlling the $output(y)$ call in the vulnerable function, and $y < MAX$ controlling the $output(y)$ call in the patched function. Similarly, in the second row we can see that the PT and NT capture the different control flow relationships which occur after the initialization of the y variable, with control flowing directly to the $output(y)$ call in the vulnerable function, and to the bounds check condition $y < MAX$ in the patched function. In the third row, we see a similar relationship to the second row for the PT and NT, however this time more abstract. The PT here represents control flow from the initialization of the variable y to any expression statement, and the NT to any condition statement. This is an accurate, yet much more abstract representation of a key relationship that contributes to the vulnerability determination.

Differently, in all rows of the CT column we can see various general contextual information of the function. The first row provides some information on how the variable x is defined based on the call to the $input$ function. The second row provides some syntax-related context between the declaration for the variable y and the $x * 10$ expression. The third row provides additional data dependence context between the declaration for y and the initialization of the variable x.

5. Vulnerability Detection

The goal of our detection algorithm is to provide an efficient way to utilize our VGRAPH representation to accurately identify vulnerable code clones ranging from exact clones to highly modified clones. This means we need an approximate matching algorithm which will not overwhelm our results with false positives. As our core VGRAPH representation is based on sets of graph triplets, we are able to use highly efficient set overlap operations to perform the bulk of the matching. Thus we develop a *Triplet Match* algorithm which we discuss below.

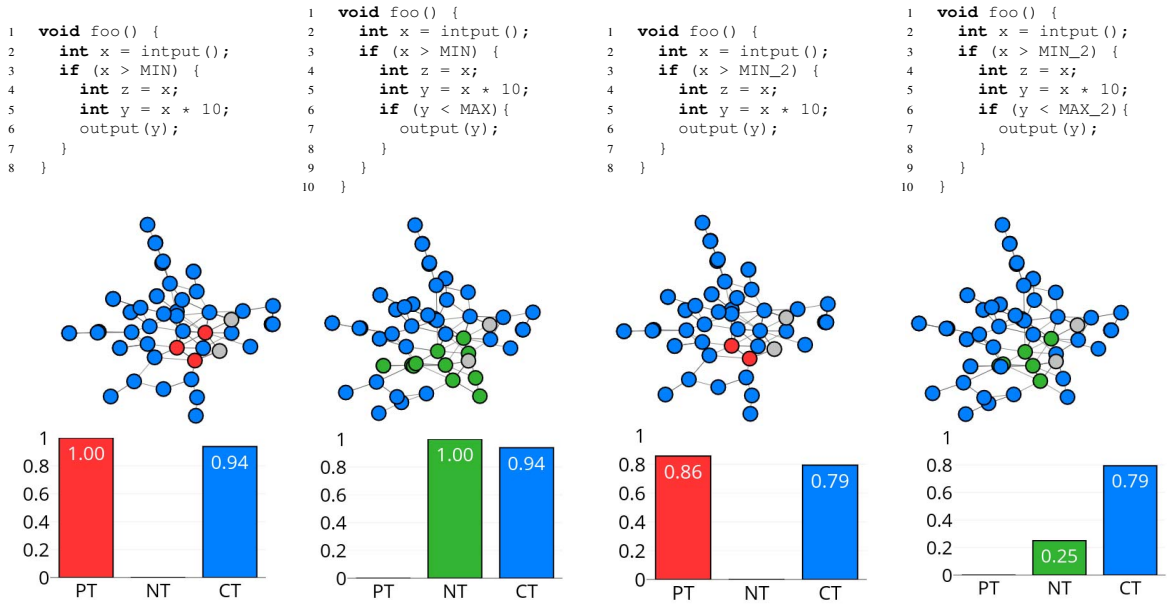
5.1. Triplet Match

The intuition behind our triplet matching algorithm is relatively straightforward. We expect vulnerable code clones of a particular vulnerability represented by a VGRAPH VG to have the following characteristics: (1) share many context triplets with VG, (2) share many positive triplets with VG, and (3) share few negative triplets with VG. To improve the ability to identify vulnerable code clones in the type-2 to type-4 range, we match triplets at each level independently (positive, negative, and context), and allow for some level of mismatch at each stage.

Algorithm 2 VGRAPH Vulnerability Detection

```
1: procedure ISVULNERABLE(VGraph, target)
2:     score_C = overlap(VGraph.CT, target)
3:     if score_C > thresh_C then
4:         score_P = overlap(VGraph.PT, target)
5:         if score_P > thresh_P then
6:             score_N = overlap(VGraph.NT, target)
7:             if score_N < score_P then
8:                 return True
9:     return False
```

Algorithm 2 provides the pseudocode for our triplet matching algorithm. This algorithm takes as input a



(a) Vulnerable clone with single line added.

(b) Patched clone with single line added.

(c) Vulnerable clone with a variable renamed and line added.

(d) Patched clone with a variable renamed and line added.

Figure 5: Using a VGRAPH to detect type-3 vulnerable code clones and differentiate from their patched counterparts based on clones of the source code introduced in Figure 1.

VGRAPH as well as graph triplets of an unknown target function, and produces a binary result indicating if the particular target function is detected as a vulnerable code clone of the vulnerability represented by the VGRAPH. The *overlap* function in the algorithm is a simple set overlap routine which returns the ratio of the query triplets found in the target triplets. There are two thresholds in the algorithm, $thresh_C$ and $thresh_P$. These thresholds dictate the amount of mismatch permitted, and hence the approximation ability of the matching algorithm. These thresholds are determined experimentally based on our VGRAPH dataset in Section 7. Notice that there is no threshold for the negative triplets, as we found that simply comparing the negative triplet score against the positive triplet score provided the best performance.

We can see in Algorithm 2 that our matching routine proceeds in a very hierarchical nature. This is by design, as real world vulnerabilities are highly context dependent. Thus, we first match against the context triplets (line 2), and only continue pursuing a match if the CT score exceeds $thresh_C$ (line 3). Next we match against the positive triplets (line 4) and only continue if the PT score exceeds $thresh_P$ (line 5). The last step is to perform the negative triplet matching (line 6). If the NT score is less than the PT score (line 7) then a true result will be returned (line 8), indicating that the target function is a vulnerable clone of the VGRAPH. In all other cases the function will return false, indicating that the target function is not a vulnerable clone of the VGRAPH.

To illustrate the ability of this algorithm to detect vulnerable code clones, we return to the example source code provided in Figure 1, and the associated VGRAPH as discussed in section 4. Based on this code, we generated 4 code clones, two of the vulnerable function, and two of the patched function. Figure 5 shows the cloned source code, the CPG of the source code with nodes highlighted according to which elements of the VGRAPH they matched with, and the overlap score for each VGRAPH triplet component. Red corresponds to positive triplet matches, green to negative triplet matches, blue to context triplet matches, and gray to no match.

Figure 5a and Figure 5b show type-3 code clones of the original vulnerable and patched code respectively. In both cases, a new variable declaration is made on line 4. We can see from the highlighted graph structure and the overlap scores, that the VGRAPH for the original vulnerability matches significantly differently against these two very similar functions. Despite the code modifications, all positive triplets and none of the negative triplets matched in the vulnerable function, and all of the negative triplets and none of the positive triplets matched in the patched function. This means, not only would our detection algorithm be able to accurately detect these type-3 vulnerable and patched code clones, but also, there is significant room for additional modification to the function while still maintaining the ability to detect this vulnerability.

Figure 5c and Figure 5d show another more complex type-3 code clone pair, this time with an additional type-2 style modification to the variables used in the critical bounds checks. The variable *MIN* has been replaced with *MIN_2* in both the vulnerable and patched functions, and *MAX* with *MAX_2* in the patched function. Despite this increase in modification, we can see that the VGRAPH was again able to identify many of the critical elements of the vulnerability in the vulnerable clone, and none of them in the patched clone. There was significantly less

negative triplet matching in this patched function, however the NT score was still higher than the PT score, and the PT score was nearly 0%, indicating that our detection algorithm would have properly labeled this function as not-vulnerable.

It is also important to notice that in all four of these examples the CT scores remained very high, indicating that the required context was present for these vulnerabilities to occur. This example shows that, with appropriate thresholding on the positive, negative, and context triplet overlap scores, the VGRAPH structure and triplet matching algorithm is able to accurately identify the vulnerable code clones, and, importantly, differentiate from their highly similar patched counterparts.

6. Vulnerability Mining

Another important consideration is how to acquire samples of vulnerable and patched source code. Many related works manually generate a dataset of vulnerable code samples, and, because of this, often only cover a small number of programs and/or vulnerability types. In addition, there is likely some bias introduced on behalf of the researcher as to what samples are added to the vulnerable code dataset.

We believe it is important to have an automated way to generate vulnerable source code samples for a wide range of programs and vulnerability types. Only when this is the case will a code-similarity-based technique be able to keep up with the continuous flow of new and diverse vulnerabilities. Therefore, we utilize an approach similar to [14] and mine content from the popular open-source code repository GitHub [2]. Note that any version control repository could be used, provided there is the ability to download specific versions of files, and there exist meaningful comments associated with the code modifications.

Version control software is widely utilized by developers of software projects both large and small as a way to manage and track changes to source code. They provide fine-grained and detailed information regarding what changed in the code, when, and why. We leverage this information to identify specific changes to source code that are related to security vulnerabilities. We developed a GitHub mining utility which downloads source code from before and after a change was made associated with a particular vulnerability identified by the CVE number. The samples from before the security-relevant code modification are labeled as vulnerable, and after, as patched. Due to space limitations the details of the GitHub mining process are presented in Appendix B.

It should be noted here that we make a fundamental assumption that modifications to source code files which reference CVEs are related to the process of patching the vulnerability. This is consistent with a popular related work VUDDY [14]. In addition, during our manual evaluation of several hundred commits related to the patching of vulnerabilities we found this assumption to hold.

7. Evaluation

We utilize the typical metrics for accuracy comparison. True positives (TP) represent the number of functions which are identified as vulnerable, and are truly vulnerable. False positives (FP) represent the number of function which are identified as vulnerable, and are not vulnerable. True negatives (TN) are the number of results classified as not vulnerable and are truly not vulnerable. False negatives (FN) are those functions that are labeled as not vulnerable, but are truly vulnerable. Precision (P) is the ratio of the true positives to all classified positive functions: $P = TP/(TP+FP)$. Recall (R) is the ratio of true positives to all labeled positive functions: $R = TP/(TP + FN)$. F1 is an overall performance metric including both precision and recall: $F1 = 2/((1/P) + (1/R))$.

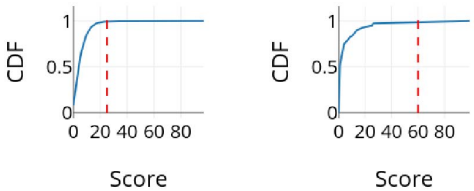
7.1. VGRAPH Database

We generated a VGRAPH Database based on 8 popular software packages that maintain source code on GitHub. Table 2 provides details of our VGRAPH database. In total we generated 1031 VGRAPHs for 711 unique vulnerabilities identified by their CVE number. For each CVE we determined the CWE, or vulnerability class, and found that our VGRAPH database covered 51 unique CWEs.

TABLE 2: VGRAPH Database Details

Repository	CVEs	Functions
Linux Kernel	197	269
OpenSSL	82	105
tcpdump	89	166
libtiff	31	40
FFMpeg	66	80
LibAV	58	72
QEMU	94	166
Xen	94	133
Total	711	1031

In order to determine the thresholds for our detection algorithm, we evaluated the matching characteristics of each VGRAPH against all functions in our VGRAPH database, with the assumption that each VGRAPH should not detect vulnerable code clones in the vast majority of other functions. We want to set our thresholds to the minimum value to allow for the best ability to identify modified code clones, but not so low as to introduce a significant number of false positives. To that end, we compute the Cumulative Distribution Function (CDF) of the context triplet scores, and the positive triplet scores, for each VGRAPH against all other functions in the database. Figure 6 shows both of the computed CDFs. We set our context triplet threshold, $thresh_C$, at the minimum value where at least 99% of the true negatives fall below. This was a score of 25% as marked in the figure with the



(a) Context Triplets (b) Positive Triplets

Figure 6: The overlap thresholds marked via the dashed red line for the context triples and the positive triples.

red dashed line. Similarly, for the positive triplets, at the given context triplet threshold of 25%, we again compute the CDF, and set our positive triplet threshold, $thresh_P$, at the minimum value where at least 99% of the true negatives fall below. This occurred at a score of 60% as marked in the figure by the red dashed line. This difference in threshold values makes sense, as the set of context triplets typically contains many more entries than the set of positive triplets for a particular vulnerability. This means we can afford a less strict threshold for the context matching compared to the positive matching. The referenced thresholds of $thresh_C = .25$ and $thresh_P = .60$ were used in the remaining experiments. In Section 7.7, we present a sensitivity analysis of these two parameters.

7.2. Test Dataset

In order to generate our test dataset, we require code clones of the functions in our VGRAPH database. According to a recent study [7], the average lifespan of a vulnerability in source code is 6.9 years. Thus, we assume that samples of the same function at different versions prior to the patch will still be vulnerable, provided they are well within the 6.9 year period. Similarly, we assume that versions of a function subsequent to the patching of a particular vulnerability will remain patched to that vulnerability. Therefore, we again utilize GitHub, and download samples of the functions in our VGRAPH database at six different time intervals relative to the patch date: immediately before, 1 month before, 1 year before, and immediately after, 6 months after, 1 year after. We temporally spaced our code samples in this manner in order to extract code clones with both small modifications (type-2 and below) to large modifications (type-3 and above), with the assumption being that temporally distant versions of the code are more likely to have a higher level of modification. We then label the functions as a vulnerable code clones if they were samples from before the original vulnerability-patching commit, or patched code clones if from after.

The details of the test dataset are listed in Table 3. Our test dataset includes 2,840 vulnerable code clones and 2,726 patched code clones, for a total of 5,566 code clones. In some cases the vulnerabilities used to build our VGRAPHs did not have commits to the same file at the various time intervals, which is the reason why these two numbers are not exactly equal.

TABLE 3: Test Dataset Details

Repository	Vulnerable Clones	Patched Clones
Linux Kernel	762	743
OpenSSL	310	288
tcpdump	370	349
libtiff	120	98
FFMpeg	220	233
LibAV	199	190
QEMU	483	463
Xen	376	362
Total	2840	2726

7.3. Scoring Criteria

The results generated against our test dataset can be scored in two ways. In the simpler case, our dataset can be labeled with a binary output of either vulnerable or patched, in which case scoring the results is trivial. In the more complex case we can label each sample in our dataset on a per-CVE basis. However, this introduces a challenge, as each sample in our dataset is labeled only with the CVE with which it was generated. We found that, in many cases, a function associated with a particular CVE is also associated with other CVEs. It is also possible that some functions may in fact be unknown vulnerable code clones, and thus not be labeled in our test dataset.

To remedy this, we score CVE-aware techniques as follows. For every positive result generated, it is considered a true positive under the following conditions. **Condition 1**: The function is a labeled vulnerable code clone and its CVE matches that of the source vulnerability. This is the base case where the original vulnerable function was used to identify a vulnerable code clone generated from the same originating CVE. **Condition 2**: The code repository, file name, and function name match that of the source vulnerability, and the source vulnerability is *newer* than the target function. In this case, a CVE from a particular function identified the vulnerability in an older version of the same function which happened to be associated with a different CVE. **Condition 3**: Manual inspection was performed and it was determined that the result is in fact a vulnerable code clone of the source vulnerability. This accounts for true vulnerable code clones which can span functions and even repositories which we did not have a label for in our test dataset. We refer to matches associated with conditions 2 and 3 as *cross-CVE* clones as one CVE is used to identify a vulnerability in a function associated with a different CVE.

For the negative results, a false negative is scored for every known vulnerable code clone that failed to generate a result, and a true negative for every known patched code clone that did not generate a result.

7.4. Detection Comparison

We compare with four state-of-art vulnerability detection techniques: FlawFinder [24], RATS [5], VUDDY [14], and ReDeBug [11]. FlawFinder and RATS are both open-source tools used in industry for identifying bugs in source code, and are both based on manually generated vulnerable functional code patterns. VUDDY is a vulnerable code clone detection technique which is based on the hashing of source code functions and a subsequent lookup of known vulnerable hashes. ReDeBug is a detection technique based on identifying sequences of known vulnerable and patched code harvested from the *diff* files associated with the patching process.

To compare with FlawFinder and RATS, we were able to download the respective tools, and run them directly on our test dataset. As these techniques do not contain any internal notion of CVEs, we can utilize the simple scoring criteria based on the binary label of the code samples.

To compare with VUDDY, we utilized their open web service [6] to identify all vulnerable code clones in our test dataset. In order to compare techniques fairly, we only calculate scores for CVEs that were shared between VUDDY and VGRAPH. This way we are comparing detection techniques rather than database generation techniques. To compare with ReDeBug, we were able to download the

TABLE 4: Vulnerable Clone Detection Comparison

System	TP	FP	FN	P	R	F1
RATS	33	**23**	2807	59	1	2
FlawFinder	712	663	2128	52	25	34
VUDDY	2021	117	371	95	84	89
VUDDY*	2021	27	371	**99**	84	91
ReDeBug	3401	94	329	97	92	94
VGRAPH	**3824**	63	**147**	98	**96**	**97**

source code, generate the required *diff* files using the same functions used to generate the VGRAPH database, and apply the detection algorithm to the test dataset. As these three techniques are CVE-aware, we scored results of each technique independently based on the conditions stated in Section 7.3.

Our results from these experiments are listed in Table 4. We can immediately see that both of the functional pattern based techniques, RATS and FlawFinder, have fairly poor results, with F1 scores below 40%. This is due to the fact that these systems are based on manually generated, predefined patterns of functionality which fail to accurately reflect most real world vulnerabilities.

Compared to RATS and FlawFinder, we can see that VUDDY, ReDeBug, and VGRAPH are all able to identify significantly more vulnerable code clones. Surprisingly, VUDDY did not immediately reveal itself as the most precise of the techniques. Upon investigation, we found that a small number of CVEs were alerting on a large number of functions, causing many false positives. For example, CVE-2017-11108, a vulnerability found in a single function of a single file in the *tcpdump* program, generated 342 results across 93 different functions. VUDDY uses a similar GitHub mining technique to generate their vulnerability hashes, and it is likely they mistakenly associated this particular CVE with an incorrect commit (likely a merge commit) covering many files unrelated to the actual vulnerability. If we disregard only two CVEs (CVE-2017-11108 and CVE-2017-5202), VUDDYs precision is over 99%, which is the level of precision we expected out of this hash-based technique. This test corresponds to the VUDDY* row in Table 4. In both scenarios, however, we can see that VUDDY performs the worst of the code similarity style techniques, with a recall score of only 84%. Also notice that VUDDY returns significantly less true positives than ReDeBug and VGRAPH, so the true recall is likely much less if we were to consider the cross-CVE results generated by the other techniques in the recall calculation.

ReDeBug, although performing with a slight reduction in precision compared to VUDDY, is able to identify over 1000 more vulnerable code clones than VUDDY, achieving 92% recall. ReDeBug is able to detect more vulnerable code clones because it only considers a localized context window around the location where the vulnerability was patched. This allows ReDeBug to detect vulnerable code clones with a significant amount of modifications provided that they occur in areas outside of the context window.

The best performing vulnerable code clone detector was VGRAPH, with an F1 score of 97%. Notably, VGRAPH is able to achieve this high score while also returning the most true positives out of all of the techniques, returning over 400 more vulnerable clones than ReDeBug, and over 2500 more vulnerable clones than VUDDY. Because of this, VGRAPH achieves the highest recall, detecting 96% of the ground truth vulnerable code clones. This improved recall is due to VGRAPHs increased ability to accurately detect type-3 and type-4 vulnerable code clones. Because VGRAPH captures key relationships associated with the contextual code, vulnerable code, and patched code, and subsequently matches on each independently, VGRAPH is able to tolerate more modification than the comparison works.

7.5. Code Sensitivity Analysis

In order to understand the sensitivity of these techniques to modifications in both vulnerable and patched code clones, we compute accuracy metrics specifically for those samples in our dataset which we know have type-2 through type-4 modifications from the original vulnerable and patched functions. Thus, we only evaluate the results from our test set which were scored according to Condition 1 in the scoring criteria mentioned in Section 7.3, and do not consider the cross-CVE clones. This was in order to isolate the results to only those clones that we knew came from the exact same function, but at different versions in that function's lifetime. This is why there are significantly less results in this experiment.

TABLE 5: Modified Code Only Detection Comparison

System	TP	FP	FN	P	R	F1
VUDDY	254	**8**	496	**97**	34	50
ReDeBug	464	43	286	91	62	74
VGRAPH	**579**	40	**147**	94	**77**	**85**

Table 5 shows the results for this subset of the test data. We can now see where the additional performance of VGRAPH is coming from. VGRAPH was able to identify twice as many modified vulnerable code clones as VUDDY, and over 100 more than ReDeBug. Note that VGRAPH is able to achieve this improved recall while

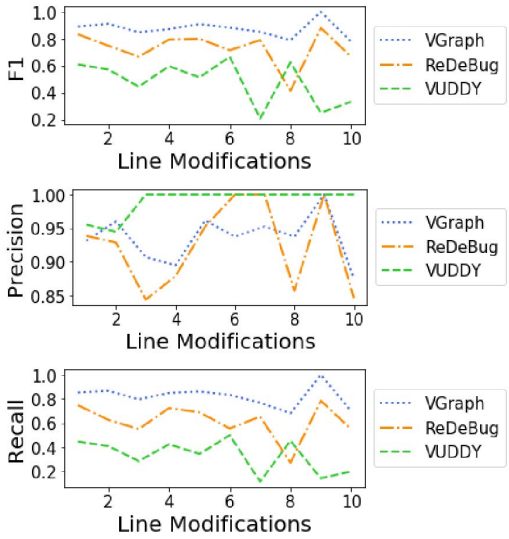


Figure 7: Accuracy comparison at different levels of modification in the code clones.

still remaining very precise at 94%, only slightly less than VUDDY at 97%, and surpassing ReDeBug at 91%.

To further understand how modifications to the code affect detection accuracy, we compute accuracy metrics at explicit levels of modification to the original vulnerable and patched functions. We use the Linux utility *diff* to identify the number of line modifications between the test function and the original vulnerable or patched functions. Figure 7 plots the accuracy metrics for each of the techniques ranging from one line modification up to 10 line modifications.

We can see from these results again that VUDDY is the most precise of the techniques, however at the expense of recall. We can see that VGRAPH and ReDeBug are similar in terms of precision, with VGRAPH only showing a slight advantage across all levels of modification. However, VGRAPH is consistently the most performant with regards to the recall rate, having the highest score at each level of modification. This results in the F1 score for VGRAPH to outperform both techniques at all levels.

Next, we score the techniques based on code clone type. We use the following heuristics to label the clone type of each pair of functions. As all pairs of functions are the same function, but from different versions of the code, we consider each pair to be at least a type-4 code clone, as ultimately the function is the same in the greater application codebase, and thus likely providing similar functionality across versions. We differentiate between type-3 and type-4 clones by thresholding the number of line modifications. When the number of line modifications is greater than half of the overall function size, we consider this to be a type-4 code clone. To differentiate between type-2 and type-3 code clones, we consider any function pair where there exists a 1-to-1 mapping of lines that were modified to be a type-2 clone.

Figure 8 shows the results from this experiment. We can see that, as expected, type-4 clones are much harder to detect than type-2 clones. Across all clone types, VGRAPH again achieves the best F1 score due to an improved recall rate while maintaining a reasonably high precision.

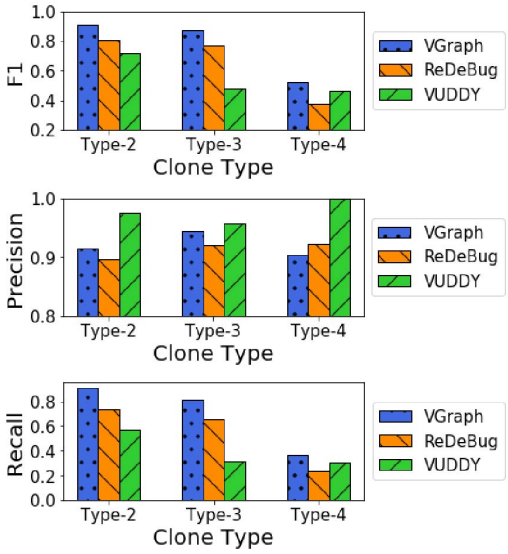


Figure 8: Accuracy comparison across code clone types.

7.6. Deep Analysis of Code Clones

In this subsection, we will analyze specific cases where VGRAPH and the comparison works diverge. Specifically, we found and evaluated three such cases where VGRAPH generated a true positive, false positive, and false negative, which were not shared by the comparison works. Due to space constraints, we discuss the first two cases below, and the false negative analysis can be found in Appendix C.

First we will look at a case where VGRAPH accurately detected a vulnerable code clone that was missed by both VUDDY and ReDeBug. The vulnerability CVE-2017-13012 is a buffer over-read vulnerability in a packet processing routine of the common network utlity *tcpdump*. This vulnerability was given a score of 9.8 out of a possible 10 points, indicating that it is a very serious vulnerability. According to the NVD it affects all versions of *tcpdump* prior to version 4.9.2. This vulnerability allows an attacker to read raw memory content from the victim machine which could contain, among other things, usernames and plaintext passwords.

Figure 9 shows the same code segment from three different versions of the affected function. Figure 9a contains the source code as it existed when the vulnerability was discovered. The key vulnerable code is related to the call to *ip_print* and *EXTRACT_16BITS(&ip->ip_len)*. As the attacker controls the value of *ip->ip_len*, it is necessary to check the value of this variable prior to its usage. The patch to this vulnerability shown in Figure 9b involved a single line addition in which a call to *ND_TCHECK_16BITS(&ip->ip_len)* was added. Our test dataset contained a third version of this function, shown in Figure 9c, which existed 6 months prior to the discovery of the vulnerability. In the code clone we can see that the call to *ND_TCHECK_16BITS* function is omitted, indicating that this code is indeed vulnerable to CVE-2017-13012. Additionally, we see a single line addition *ndo->ndo_snaplen=ndo->ndo_snapend-bp* which does not affect the core vulnerable code, as it does not prevent the buffer over-read caused by the call to *EXTRACT_16BITS(&ip->ip_len)* and *ip_print*.

VGRAPH correctly identified the code clone in Figure 9c because it was able to detect key vulnerable conditions by matching positive triplets such as:

Source: *snapend_save=ndo->ndo_snapend*
Relationship: FLOWS_TO (1)
Destination: *ip_print(ndo,bp,EXTRACT_16BITS(&ip->ip_len))*

This triplet represents the condition of the vulnerable code where the declaration of the *snapend_save* variable flows directly to the call to the *ip_print* function.

Additionally, key negative triplets indicative of the patch failed to match, such as:

Source: *ND_TCHECK_16BITS(&ip->ip_len)*
Relationship: DOM (2)
Destination: *ip_print(ndo,bp,EXTRACT_16BITS(&ip->ip_len))*

This triplet represents the fact that a call to *ND_TCHECK_16BITS* must come before the call to

```
if (ndo->ndo_vflag >= 1 &&
     ICMP_ERRTYPE(dp->icmp_type)){
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  snapend_save = ndo->ndo_snapend;
  ip_print(ndo, bp, EXTRACT_16BITS
       (&ip->ip_len));
  ndo->ndo_snapend = snapend_save;
}
```

(a) Vulnerable code with no check on attacker controlled variable *ip->ip_len*.

```
if (ndo->ndo_vflag >= 1 &&
     ICMP_ERRTYPE(dp->icmp_type))
     {
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  snapend_save = ndo->ndo_snapend;
  ND_TCHECK_16BITS(&ip->ip_len);
  ip_print(ndo, bp, EXTRACT_16BITS
       (&ip->ip_len));
  ndo->ndo_snapend = snapend_save;
}
```

(b) Patched code with a single additional line calling *ND_TCHECK_16BITS*.

```
if (ndo->ndo_vflag >= 1 &&
     ICMP_ERRTYPE(dp->icmp_type))
     {
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  ndo->ndo_snaplen = ndo->
       ndo_snapend - bp;
  snapend_save = ndo->ndo_snapend;
  ip_print(ndo, bp, EXTRACT_16BITS
       (&ip->ip_len));
  ndo->ndo_snapend = snapend_save;
}
```

(c) Vulnerable clone with a single line addition declaring *ndo->ndo_snaplen*.

Figure 9: Three versions of a code segment from the tcpdump function $icmp\_print$ associated with CVE-2017-13012. Only VGRAPH detected the vulnerable code clone.

ip_print in the patched version. The combination of high positive triplet score and low negative triplet allowed VGRAPH to properly label this function as a vulnerable code clone.

Because of the 8 line-level modifications, the hash generated by VUDDY for this function would not match the hash of the original function, which is the reason why VUDDY did not detect this vulnerable code clone. ReDeBug would have been able to tolerate most of the modifications as they existed far away from the specific code associated with the vulnerability. However, the single line addition directly around the vulnerable code caused the ReDeBug sequence-based matching algorithm to fail and thus not detect this vulnerable code clone.

Next we will look at a case where VGRAPH generated a false positive not shared by ReDeBug and VUDDY. Figure 10 shows three versions of the same code segment from the Qemu program associated with CVE-2016-4952. The vulnerable code shown in 10a allowed an attacker to cause denial-of-service due to an out-of-bounds array access caused by improper error checking. The patched code shown in 10b involved adding a conditional statement checking the return value of the call to *pvsci_ring_init_data*, and returning a failed state when appropriate. Our test dataset contained a third version of the function from several months after the vulnerability was patched, shown in Figure 10c. We can see that the code was updated to instead check for proper conditions prior to the function call to *pvcci_ring_init_data*. This meant that it was no longer necessary to check the return value of the call to *pvsci_ring_init_data*, and thus the conditional statement added in the original patch was removed.

Although the version of the code in Figure 10c is patched to the original vulnerability, VGRAPH incorrectly classified it as vulnerable to CVE-2016-4952. We can see that a large part of the code from the original vulnerable function exists in this patched version. This caused many positive triplets to match, such as:

Source: *pvscsi_ring_init_data(&s->rings, rc)*
Relationship: FLOWS_TO
Destination: *s->rings_info_valid=TRUE* (3)

This triplet represents the direct control flow from *pvscsi_ring_init_data* to the *rings_info_valid=TRUE*, and ultimately the successful return fo the function. Also, many of the negative triplets which would have indicated that this function was patched failed to match as well, such as:

Source: *pvscsi_ring_init_data(&s->rings,rc)<0*
Relationship: CONTROLS
Destination: *return PROCESSING_FAILED* (4)

This triplet describes the key control dependence in the patched code between the *return PROCESSING_FAILED* statement and the evaluation of the return value from the call to *pvscsi_ring_init_data*. This failed to match in the clone because the conditional statement no longer calls the *pvscsi_ring_init_data* function.

There were several key negative triplets that did in fact match, such as:

Source: *Condition*
Relationship: CONTROLS
Destination: *return PROCESSING_FAILED* (5)

This triplet represents a more generic relationship of the patch where there is a control dependence between the *return PROCESSING_FAILED* statement and a conditional statement. Several triplets like this one matched in the patched code clone, however they were too few to score higher than the positive triplets. This caused VGRAPH to incorrectly label this function as vulnerable.

Because the patched code clone involves relatively significant changes to the code close to the location of the original patching process, both VUDDY and ReDeBug properly classify this function as not-vulnerable. It should be noted here that we could adjust the thresholds for the matching algorithm such that VGRAPH does properly classify this function as not vulnerable, although this would have an overall negative effect on the total performance of the algorithm, as we would generate many more false negatives.

7.7. Parameter Sensitivity Analysis

In addition to external factors such as code modification, we were also interested in understanding how sensitive our technique was to internal factors such as our two algorithm hyperparamaters, identified as $thresh_C$ and $thresh_P$ in Algorithm 2. To that end, we used the same subset of our data as the previous sensitivity experiment, but this time varied these two hyperparameters from a threshold of 0 to a threshold of 100, and computed accuracy metrics for each configuration. Figure 11 shows

```
pvscsi_dbg_dump_tx_rings_config(
     rc);
pvscsi_ring_init_data(&s->rings,
     rc);
s->rings_info_valid = TRUE;
return PROCESSING_SUCCEEDED;
```

(a) Vulnerable code with improper error checking on *pvscsi_ring_init_data*.

```
pvscsi_dbg_dump_tx_rings_config(
     rc);
if (pvscsi_ring_init_data(&s->
     rings, rc) < 0) {
  return PROCESSING_FAILED;
}
s->rings_info_valid = TRUE;
return PROCESSING_SUCCEEDED;
```

(b) Patched code with added conditional check on *pvscsi_ring_init_data*

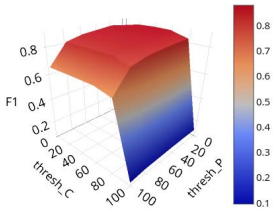
```
if (!rc->reqRingNumPages
    || rc->reqRingNumPages >
         RINGS_MAX_NUM_PAGES
    || !rc->cmpRingNumPages
    || rc->cmpRingNumPages >
         RINGS_MAX_NUM_PAGES) {
  return PROCESSING_FAILED;
}
pvscsi_dbg_dump_tx_rings_config(
     rc);
pvscsi_ring_init_data(&s->rings,
     rc);
s->rings_info_valid = TRUE;
return PROCESSING_SUCCEEDED;
```

(c) Patched clone with a different check not based on *pvscsi_ring_init_data*.

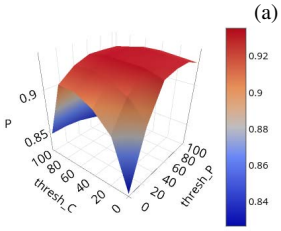
Figure 10: Three versions of a code segment from the Qemu function *pvscsi_on_cmd_setup_rings* associated with CVE-2016-4952. VGRAPH generated a false positive on the patched clone.

surface plots for Precision, Recall, and F1 at each of the different threshold configurations.

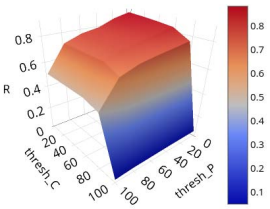
In these plots we can see the obvious performance drop-offs at the extreme values of the thresholds. At threshold values of 100% for both PT and CT, our F1 score is near zero. Conversely, at thresholds near 0%, our Recall is near 100%. However, other than these extreme drop-offs near the boundary conditions, we can see in all three plots there is a large flat plane at a high level of accuracy. This means that there is in fact a wide range of threshold values that will perform well at this task. In other words, our algorithm is not overly sensitive to these tuning parameters.



(a) F1



(b) Precision



(c) Recall

Figure 11: Accuracy sensitivity analysis at varying detection threshold values.

7.8. Runtime Performance Comparison

In this subsection, we evaluate the runtime performance of each technique. We compute the amount of time required, starting from the raw source code, to generate match results against the full test set. Each process is run in a single threaded non-parallel implementation. Note that this is an embarrassingly parallel problem, and significant speedup for all techniques could be achieved through parallel implementations.

TABLE 6: Run-time Comparison

System	Runtime (s)
RATS	1
FlawFinder	74
VUDDY	1091
ReDeBug	235
VGRAPH	1918

Table 6 shows the results for each technique. We can see that the pattern based approaches are among the fastest, yet as seen previously, least accurate techniques. They require only a single scan of the test set, and implement relatively simple matching algorithms. ReDeBug is the next fastest, taking a little under 5 minutes to process the full test set. ReDeBug also has minimal preprocessing required, and a simple token-based matching algorithm. VUDDY takes a little under 20 minutes. The majority of this time is generating the hash representation of the functions. After the hash is generated, the actual matching takes only a few seconds through their web service, which likely involves a simple database lookup. Finally, VGRAPH is the slowest of the approaches, taking over 30 minutes to generate the results on the test set. The graph generation process using the *Joern* tool takes roughly half of the total runtime, and the matching process the remainder. This is not unexpected, as the robust source code representation and approximate matching algorithm each come at the cost of increased preprocessing and matching times. However, we believe this to be a reasonable trade-off considering the improved ability to detect highly modified vulnerable code clones.

7.9. Using VGRAPH in Practice

In order to show the ability of VGRAPH to work in a real-world setting, we utilized the VGRAPH system to identify previously unknown vulnerable code clones in several popular open source programs. Undocumented vulnerabilities which are patched in the most up-to-date version of the code, but affect older versions and are undocumented as doing so, still pose a significant security risk. Many vulnerability scanners incorporate vulnerability

TABLE 7: Detected Vulnerabilities in versions of FFMpeg and OpenSSL. All versions listed are not reported by the NVD as being vulnerable to the specified CVE.

Target Product	CVE	VGRAPH Repository	VGRAPH Function	Target Function	Modifications
ffmpeg 0.11.5	CVE-2012-2776	libav	decode_cell_data	decode_cell_data	1
ffmpeg 0.11.5	CVE-2012-2791	ffmpeg	ivi_decode_blocks	ff_ivi_decode_blocks	6
ffmpeg 0.11.5	CVE-2012-2800	ffmpeg	decode_band	decode_band	11
ffmpeg 1.0.10	CVE-2012-2776	libav	decode_cell_data	decode_cell_data	1
ffmpeg 1.0.10	CVE-2012-2800	ffmpeg	decode_band	decode_band	8
ffmpeg 2.3.6	CVE-2015-3395	ffmpeg	msrle_decode_pa4	msrle_decode_pal4	0
openssl 1.0.0	CVE-2018-0739	openssl	asn1_item_embed_d2i	ASN1_item_ex_d2i	103
openssl 1.0.0s	CVE-2016-7053	openssl	asn1_template_ex_d2i	asn1_template_ex_d2i	0
openssl 1.0.1a	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0
openssl 1.0.1	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0
openssl 1.0.2s	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0
openssl 1.0.1u	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0

reporting information from sources such as the NVD. If a version of a particular software package is not listed as vulnerable to a particular CVE, vulnerability scanning software will likely not alert on those versions, despite them being vulnerable.

We utilized our VGRAPH database and hunted for vulnerabilities in several versions of FFMpeg and OpenSSL. We identified 10 vulnerabilities that were silently patched in various versions of the software which were not previously documented in the NVD. Table 7 lists each vulnerability we discovered along with the CVE identified, and the repository and function name of both the VGRAPH and the target graph. We also include the number of source code line differences that were identified in the vulnerable code clone compared to the original vulnerable function associated with the CVE.

Here we again see VGRAPHs ability to identify several identical code clones, as well as many type-3 code clones, some with significant modification. We see that VGRAPH was able to identify bugs across different products, in some cases using VGRAPHs from LibAV to identify vulnerable code clones in FFmpeg, and vice versa.

8. Limitations and Future Work

VGRAPH was designed to be a more effective code similarity based vulnerability detection system. In particular, our goal was to build a system which was more robust to modifications in the vulnerable code clones representative of type-3 clones. However, as this is a code similarity based technique, VGRAPH still struggles to accurately detect type-4 code clones. Type-4 code clones could share little to no code with the original vulnerability, and thus would be challenging for any code similarity technique to detect. We believe type-4 code clone detection is better suited to more intensive analysis based on symbolic execution and program testing [15].

This technique is also limited by the number of VGRAPHs available in the database. We developed a technique where we mine vulnerable and patched code from GitHub in order to build our VGRAPH database, however not all vulnerabilities have a representative sample in GitHub or equivalent repository. In addition, the current techniques rely on accurate documentation on behalf of the code maintainer as to what commits are related to particular vulnerabilities, which is not always the case. In the future we plan to expand our automated VGRAPH generation routines to also mine other data sources such as the NVD and Bugtraq [1].

The focus of this work was not on scalability or performance, and there are likely many areas for improvement. This problem is embarrassingly parallel, and a simple multi-threaded implementation could generate significant speedup. We leave the full scalability analysis and performance improvements to future work.

9. Conclusion

In this work we introduced a new representation of vulnerabilities in source code based on code property triplets of the vulnerable code, the patched code, and the contextual code. We designed an accurate approximate matching algorithm which is capable of detecting modified vulnerable code clones, and differentiating them from their patched counterparts. We developed the VGRAPH detection system which mines vulnerable and patched source code from GitHub and generates our VGRAPH database which we then use to identify vulnerable code clones. We built a test dataset containing vulnerable code clones ranging from identical clones to clones with many modifications. Compared with state-of-art vulnerable clone detection techniques we are able to identify significantly more vulnerable code clones, particularly for those clones with significant levels of modification, while generating as few or fewer false positives.

Acknowledgment

The authors thank the anonymous shepherd and reviewers for their help on improving this paper. This work was supported in part by DARPA under agreement number N66001-18-C-4033 and National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774, as well as support from the ARCS Foundation. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense, National Science Foundation, ARCS, or the U.S. Government.

References

[1] Bugtraq. https://securityfocus.com.

[2] Github. https://github.com.

[3] Joern. http://www.mlsec.org/joern/.

[4] National vulnerability database. https://nvd.nist.gov.

[5] Rough audit tool for security. https://github.com/andrew-d/rough-auditing-tool-for-security.

[6] Vuddy web service. https://iotcube.korea.ac.kr/.

[7] Lillian Ablon and Andy Bogart. *Zero days, thousands of nights: The life and times of zero-day vulnerabilities and their exploits*. Rand Corporation, 2017.

[8] R. Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, Sept 2008.

[9] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 60–71, Piscataway, NJ, USA, 2019. IEEE Press.

[10] Md Rakibul Islam, Minhaz F. Zibran, and Aayush Nagpal. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, pages 20–29, Piscataway, NJ, USA, 2017. IEEE Press.

[11] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 48–62, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[14] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.

[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[17] 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*, pages 201–213. ACM, 2016.

[18] 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 *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[19] Zhen Liu, Qiang Wei, and Yan Cao. Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint. In *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 548–553. IEEE, 2017.

[20] United States Government Accountability Office. Actions taken by equifax and federal agencies in response to the 2017 breach. 2018.

[21] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[22] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157–1168. IEEE, 2016.

[23] Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on Dependable and Secure Computing*, 12(6):688–707, 2015.

[24] David A Wheeler. Flawfinder, 2011.

[25] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[26] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812, May 2015.

[27] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[28] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.

[29] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.

Appendix A. Code Property Graph

The Code Property Graph (CPG) is a multigraph containing the representative nodes and edges from the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). In this work we use the open source tool *Joern* to generate the CPG for source code. Table 8 and Table 9 show the edge types and node types, respectively, that are generated by *joern* and supported by VGRAPH.

While readers are encouraged to refer to the original work [25] for the details, here we provide a brief explanation of what information is contained in the CPG and how it is generated.

The AST serves as the foundation of the CPG, decomposing the source code into various language constructs such as *ForStatement*, *Symbol*, *CallExpression*, etc. These different types of constructs define the node types of the CPG and are listed in Table 9. Edges from the AST are added to the CPG as either an *IS_AST_PARENT* edge, which provides the structure of the various language elements, or a *DECLARES* edge, which connects declaration statements to the declarations they contain.

TABLE 8: Edge Types

IS_AST_PARENT	FLOWS_TO	DECLARES
DEF	USE	REACHES
POST_DOM	DOM	CONTROLS

Next, the CFG is used to add control flow information to the various nodes of the CPG. This comes in the form of the *FLOWS_TO* edge which connects statement nodes

TABLE 9: Node types

IdentifierDeclStatement	ReturnType
EqualityExpression	CompoundStatement
DeclStmt	PostIncDecOperationExpression
IfStatement	IdentifierDecl
Parameter	Symbol
ClassDefStatement	ReturnStatement
CastTarget	CallExpression
BitAndExpression	IncDec
AssignmentExpression	ExpressionStatement
PrimaryExpression	InclusiveOrExpression
WhileStatement	IdentifierDeclType
UnaryOperator	ForInit
CFGExitNode	CFGEntryNode
PtrMemberAccess	ConditionalExpression
GotoStatement	BreakStatement
ArgumentList	MemberAccess
UnaryExpression	DoStatement
Callee	CastExpression
ParameterType	SizeofExpression
Sizeof	ShiftExpression
ForStatement	ArrayIndexing
ElseStatement	UnaryOperationExpression
Expression	InitializerList
MultiplicativeExpression	ContinueStatement
Statement	Argument
OrExpression	AndExpression
Identifier	CFGErrorNode
FunctionDef	SizeofOperand
AdditiveExpression	SwitchStatement
Decl	Label
Condition	InfiniteForNode
ClassDef	ExclusiveOrExpression
RelationalExpression	ParameterList

to their successors, providing an overall ordering and flow to the nodes of the graph.

Finally, the PDG is used to provide a significant amount of information pertaining to control dependence and data dependence between the elements in the CPG. Control dependence is a relationship between two statements whereb one statement directly affects whether or not the other will be executed. Data dependence is a relationship between two statements whereby one statement has a dependence on a data element defined by another. These two relationships are characterized by several different edge types in the CPG. The *DEF* and *USE* edges connect statements to the nodes which they define and use respectively. This allows for a reachability analysis to be performed, which results in the *REACHES* edge type which connects the statements that are reached by data flow. In other words, this edge will connect the statements where the definition of a particular data element reaches another statement, and thus the latter is data dependent on the former. Similarly, the *CONTROLS* edge connects the statements which are control dependent on one another, meaning one statement directly controls whether or not the other will be executed. In both of the previous edge types, it is important to be able to determine all statements that must occur prior to a particular statement being reached. This is determined by building dominator and post-dominator trees inside the CPG, represented by the edges *DOM* and *POST_DOM*. These edges describe the dominance relationships between nodes of the graph which provide insight into which nodes must occur prior or subsequent to others.

Appendix B. GitHub Mining

We adopt a method very similar to [14] and mine samples of source code from the popular open-source software repository GitHub [2]. We identify code changes related to security vulnerabilities and download the code from the functions in their vulnerable state, as well as their patched state. Each step in the process is outlined below, accompanied by the *git* commands utilized to perform the actions.

Commit Log Parsing . In order to identify source code relevant to a specific vulnerability, we utilize the log messages associated with the commits to a GitHub repository. A commit log is the message associated with some modification to the source code. It is intended to contain information relevant to the purpose for the code change. We identify the commits containing any reference to the string "CVE-20" as being commits related to the referenced vulnerability. This is accomplished with the following git command:

```
git log --grep="CVE-20"
```

File and Function Parsing. Each commit identified in the previous step uses a unique hash value as a commit identifier. For each commit, we use a second git command to show the details of that commit, which will include the files, functions, and locations of source code additions, deletions, and modifications. If the modifications happen inside a function of a C/C++ source code file, the hash value for both the original file and the modified file are identified so that we can use them to download the files in the next step. In addition, we parse out the modified function names so that we know what functions inside the source code files are of interest. The command below provides such details:

```
git show <commit_ID>
```

Source Code Download. At this point we have the hash ID for a source code file which contains our functions of interest for both the original vulnerable version, as well as the patched version. We can now use a final git command to download the source code files for both the vulnerable and patched version. We are then able to parse out the specific functions of interest which were explicitly modified during the transition from vulnerable to patched. The command below will show the contents of the source code file at a specific version of the code.

```
git show <file_ID>
```

The result is a set of source code function pairs, vulnerable and patched, each associated with a particular CVE which we can use to build our VGRAPH Database.

Appendix C. False Negative Evaluation

Figure 12 shows a case where VGRAPH generated a false negative not shared by ReDeBug. The Figure shows three versions of the *x509_decode_time* function found in the Linux Kernel. A vulnerability was found in the decode logic that allowed for a buffer over-read in the

```
if (*p != 'Z')
  goto unsupported_time;
mon_len = month_lengths[mon];
...
if (year < 1970 ||
    mon < 1 ||
    mon > 12 ||
    day < 1 ||
    day>mon_len ||
    hour > 23 ||
    min > 59 ||
    sec > 59)
  goto invalid_time;
```

(a) Vulnerable code with an unsafe access to the *month_lengths* array.

```
if (*p != 'Z')
  goto unsupported_time;
if (year < 1970 ||
    mon < 1 || mon > 12)
  goto invalid_time;
mon_len = month_lengths[mon - 1];
...
if (day < 1 ||
    day > mon_len ||
    hour > 23 ||
    min > 59 ||
    sec > 59)
  goto invalid_time;
```

(b) Patched code with conditions reordered to check value of *mon* prior to usage.

```
if (*p != 'Z')
  goto unsupported_time;
mon_len = month_lengths[mon];
...
if (year < 1970 ||
    mon < 1 ||
    mon > 12 ||
    day < 1 ||
    day > mon_len ||
    hour < 0 ||
    hour > 23 ||
    min < 0 ||
    min > 59 ||
    sec < 0 ||
    sec > 59)
  goto invalid_time;
```

(c) Vulnerable clone with a different conditional statement.

Figure 12: Three versions of a code segment from the Linux Kernel function $x509\_decode\_time$ associated with CVE-2015-5327.

month_lengths array. The patch is shown in Figure 12b, which was to move some of the validation logic prior to the array access, as well as offsetting the access by 1. Figure 12c shows a code clone of the vulnerable code from 4 months prior to the vulnerability being discovered. We can see in this version there are a few modifications to the lower conditional statement, however the key unchecked array access to *months_lengths[mon]* was performed, indicating this function is still vulnerable.

The main reason VGRAPH failed to detect this vulnerable code clone has to do with the fact that during the original patch process, code was removed from the large compound conditional statement at the bottom of the code excerpt, and moved to before the access of the month_lengths array. Because code was modified in the lower conditional statement, positive triplets were associated with the changes in the large conditional statement. Additionally, the way the CPG is generated for this function, the large compound conditional statement was treated as a single entity, rather than breaking it up into it's many constituent parts. This means there existed positive triplets such as:

Source: *mon_len=month_lengths[mon]*
Relationship: REACHES
Destination: *year<1970||mon<1||mon>12||day<1||day>mon_len||hour>23||min>59||sec>59* (6)

Source: *year<1970||mon<1||mon>12||day<1||day>mon_len||hour>23||min>59||sec>59*
Relationship: CONTROLS
Destination: *goto invalid_time* (7)

The first triplet describes the data dependence between the compound conditional statement and the declaration of mon_len, and the second describes the control dependence between the conditional statement and the *goto invalid_time* statement. Both of these triplets failed to match in the vulnerable code clone, due to the fact that there was a modification to the lower conditional statement. If the compound conditional statement were to be expanded into it's constituent parts, VGRAPH would have likely been able to detect this as a vulnerable code clone, as many of the components are the same in the vulnerable code clone.

On the other hand, ReDeBug was able to properly classify this vulnerable code clone since the modifications to the function were far enough away from the original vulnerable code. VUDDY, however, also failed to detect this function as vulnerable, as the function had changes from the original vulnerable code and the hashes would not match.