BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network

Yuede Ji
Graph Computing Lab
George Washington University
yuedeji@gwu.edu

Lei Cui
Graph Computing Lab
George Washington University
yuedeji@gwu.edu

H. Howie Huang
Graph Computing Lab
George Washington University
howie@gwu.edu

ABSTRACT

Binary code similarity detection, which answers whether two pieces of binary code are similar, has been used in a number of applications, such as vulnerability detection and automatic patching. Existing approaches face two hurdles in their efforts to achieve high accuracy and coverage: (1) the problem of *source-binary code similarity detection*, where the target code to be analyzed is in the binary format while the comparing code (with ground truth) is in source code format. Meanwhile, the source code is compiled to the comparing binary code with either a random or fixed configuration (e.g., architecture, compiler family, compiler version, and optimization level), which significantly increases the difficulty of code similarity detection; and (2) the existence of *different degrees of code similarity*. Less similar code is known to be more, if not equally, important in various applications such as binary vulnerability study. To address these challenges, we design BugGraph, which performs source-binary code similarity detection in two steps. First, BugGraph identifies the compilation provenance of the target binary and compiles the comparing source code to a binary with the same provenance. Second, BugGraph utilizes a new graph triplet-loss network on the attributed control flow graph to produce a similarity ranking. The experiments on four real-world datasets show that BugGraph achieves **90%** and **75%** TPR (true positive rate) for syntax equivalent and similar code, respectively, an improvement of **16%** and **24%** over state-of-the-art methods. Moreover, BugGraph is able to identify 140 vulnerabilities in six commercial firmware.

KEYWORDS

Code Similarity, Graph, Vulnerability

ACM Reference Format:
Yuede Ji, Lei Cui, and H. Howie Huang . 2020. BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network. In *Proceedings of Proceedings of The 16th ACM ASIA Conference on Computer and Communications Security (ASIACCS '21).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ASIACCS '21, June 7–11, 2021, Hong Kong, China

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

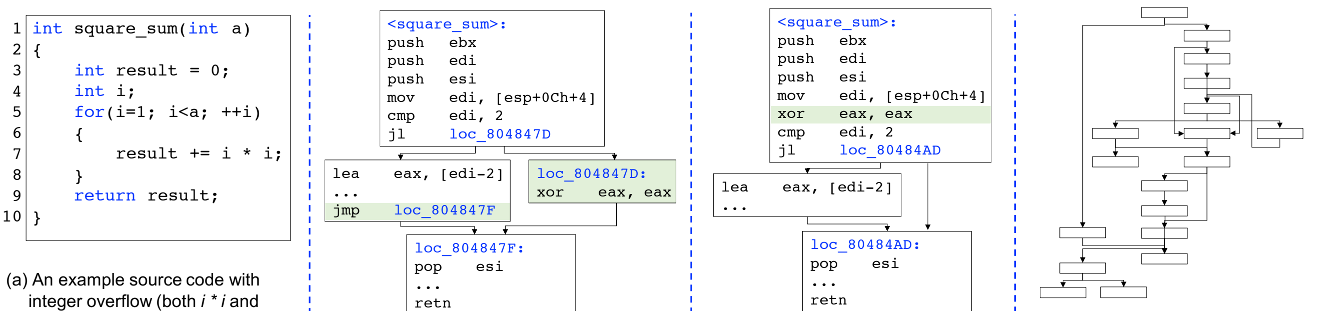
The binary code runs in billions of servers, personal computers, and wireless devices. Binary code similarity detection has a wide range of applications, such as vulnerability detection [14, 20, 21, 38], malware analysis [35], plagiarism detection [33], and security patch analysis [55]. The traditional approach for binary code similarity detection takes two different binary codes as the inputs (e.g., the whole binary [18], functions [14, 21, 51], or basic blocks [57]), and computes a measurement of similarity between them. The assumption is that if two binaries were compiled from the same or similar source code, this approach would produce a high similarity score.

In contrast to the aforementioned *binary-binary code similarity*, this work highlights a key aspect of the problem, that is, *source-binary code similarity detection*, where the code to be analyzed is in the binary format while the one for comparison is in the source code format. For example, as many open-source libraries are widely used, the vulnerabilities, such as those in OpenSSL and FFmpeg, are also inherited by closed-source applications (binaries) [20, 21, 38, 51]. In this scenario, although the source code of the application is unavailable, one can still leverage the availability of the open-source libraries to detect the existence of a similarity. Recent research has identified a similar problem but limited for Android binary patching [17] and Android firmware analysis [56].

For this type of problems, traditional binary-binary code similarity detection methods would first compile the source code with a particular configuration, and then compare the resultant binary against the other target binary. Unfortunately, such an approach faces two major challenges that prevent them from achieving high accuracy and coverage:

Challenge #1: canonicalize the source and binary code. In the problem of source-binary code similarity, because the two inputs are in different formats, one needs to canonicalize them into the same representation for comparison. Clearly, there are a large number of different compilation configurations that can be used, differing in terms of the compiler (e.g., gcc and llvm), version number (e.g., gcc and llvm each have tens to hundreds of versions), parameters (e.g., at least four optimization levels for gcc and llvm), and the target architecture (e.g., x86 and arm). In this paper, we use the term of *provenance* to represent the configuration used for compiling a binary code.

Figure 1 shows an example of three different binary codes compiled from the same source code (a). In this example, the binary code in Figure 1 (b) and (c) are similar as they share the same compiler family (llvm), optimization level (O1), and target architecture (x86), with the only difference in compiler version (version 3.3 vs. 3.5). In contrast, the code in Figure 1 (d) is drastically different, due to its choice of compiling configuration (gcc version 4.8.5 with O3 for



(a) An example source code with integer overflow (both *i* * *i* and *result* can exceed the maximum integer value)

(b) Assembly code of x86-llvm-3.3-O1

(c) Assembly code of x86-llvm-3.5-O1

(d) Assembly code of x64-gcc-4.8.5-O3 (codes are ignored due to the large amount)

Figure 1: Code example, (a) is the source code, (b) (c) (d) show the assembly code with control flow of the binary compiled with x86-llvm-3.3-O1, x85-llvm-3.5-O1, and x64-gcc-4.8.5-O3, respectively (the difference between (b) and (c) are shaded).

the x64 architecture). In this case, both the code size and control flow are greatly changed, mainly because of loop related optimization techniques, e.g., tree vectorization and loop unrolling. Thus, the binary-binary methods which rely on a single, binary level model for similarity analysis, would undoubtedly have difficulty in fully capturing code difference without taking into account the compiling provenance.

Challenge #2: different degrees of code similarity. Generally speaking, there are three types of syntax similarity, from type-1 (literally same), type-2 (syntactically equivalent), to type-3 (syntactically similar) [42]. Note that there is another semantic code similarity (type-4) which we leave as part of future work. Existing methods [20, 21, 38, 51, 57] have been shown to work well for the type-1 code, but less desirable for other types, especially type-3. Our own evaluation shows that when applied to the binaries compiled by 24 different compilation provenances from Binutils {2.25, 2.30} and Coreutils {8.21, 8.29}, prior work Gemini [51] can only achieve 55% true positive rate for type-3 similar code, which is significantly lower compared to its 77% for type-1 similar code.

On the other hand, the type-3 code is known to have significant importance in various applications. A recent study [26] finds that type-3 syntactically similar code can contribute to 50-60% of all vulnerabilities (discovered by two tools, Cppcheck [3] and Flawfinder [4]).

Our solution. To address both challenges, we have designed and implemented a new system called BugGraph, which detects the source-binary code similarity in two steps. In the first step, we canonicalize the source code and binary code with the help of compilation provenance identification. Specifically, we identify the compilation provenance of the target binary input, that is, figuring out which compiler family, compiler version, optimization level, as well as target architecture, have been used. This way, instead of comparing with a randomly compiled binary as done before, our method can produce a binary from the given source code with the same provenance, thereby greatly reducing the negative impact from the compiling process. To the best of our knowledge, we are the first to propose this approach which starts by identifying the provenance of the binary code and taking full advantage of the availability of the source code.

In the second step, we design a new graph triplet-loss network (GTN) to learn the similarity ranking in order to provide the high coverage of code similarity. Specifically, BugGraph uses an attributed control flow graph (ACFG) to capture the features of a binary function. The model takes a triplet of ACFGs, which represent the *anchor*, *positive* and *negative* functions, as the input. The learning goal is to ensure that the similarity between the *anchor* and *positive* functions is higher than that of the *anchor* and *negative*. Thus, our GTN approach can produce a ranking of code similarity, discerning the difference among similar codes, while prior work [20, 21, 38, 51] simply yields a binary (similar or not) decision.

We have also conducted extensive evaluations on a large number of representative datasets: (1) We perform a validation test on an existing dataset (type-1 similarity). In this test, we are able to not only reproduce the results reported in [51], but more importantly show the effectiveness and benefit of the provenance identification. (2) We compare with three recent methods of binary code similarity detection on a syntax similar dataset covering type-1/2/3 code. Considering the top-5 similar code as positive, BugGraph achieves 93% true positive rate (TPR) for type-1 similarity, which significantly outperforms 77%, 69%, and 45% of Gemini, Genius, and BGM, respectively. For type-2 and type-3 similar code, BugGraph achieves 90% and 75% TPR, respectively, again much higher than the best TPRs (74% and 51%) from other methods. (3) We further apply BugGraph to the binaries from six commercial firmware, and are able to identify 140 vulnerabilities in this case study.

In summary, we make the following contributions:

- **New insight and method.** This work focuses on a special problem of the source-binary code similarity detection. We develop a two-step approach of first identifying the provenance of the target binary code and compiling the comparing source code accordingly, coupled with a new graph triplet-loss network to rank the code similarity.
- **Extensive evaluation.** We implement a prototype BugGraph and evaluate on various real-world datasets. BugGraph outperforms previous works for the same (type-1) code, as well as less similar (type-2/3) code.

2 PROBLEM STATEMENT

2.1 Problem Definition

In this work, the problem of ***source-binary code similarity detection*** is referred to as computing the similarity between *each*

```
static void do_free_upto(BIO *f,
    BIO *upto) {
    if (upto) {
        BIO *tbio;
        do {
            tbio = BIO_pop(f);
            BIO_free(f);
            f = tbio;
        }
        while (f != upto)
    } else
        BIO_free_all(f);
}
```

(a) Original code of CVE-2015-1792

```
static void do_free_upto(BIO *f,
    BIO *upto) {
    if (upto) {
        BIO *tbio;
        do {
            tbio = BIO_pop(f);
            BIO_free(f);
            f = tbio;
        }
        while (f != upto)
    } else
        // free BIO *f
        BIO_free_all(f);
}
```

(b) Type-1 similar (literally same)

```
static void do_free_upto(BIO *f,
    BIO *upto) {
    if (upto) {
        BIO *temp_bio;
        do {
            temp_bio = BIO_pop(f);
            BIO_free(f);
            f = temp_bio;
        }
        while (f != upto)
    } else
        BIO_free_all(f);
}
```

(c) Type-2 similar (syntactically equivalent)

```
static void do_free_upto(BIO *f,
    BIO *upto) {
    if (upto) {
        BIO *tbio;
        do {
            tbio = BIO_pop(f);
            if (tbio == NULL)
                continue;
            BIO_free(f);
            f = tbio;
        }
        while (f != upto)
    } else
        BIO_free_all(f);
}
```

(d) Type-3 similar (syntactically similar)

Figure 2: Source code syntax similarity types (the different parts are shaded).

function to be analyzed from the target binary code, and *the* function to be compared from the source code. It can be formally defined as follows:

Definition 1. *Given two inputs, B and s, where B is the target binary, s is the comparing function with the source code, the problem is to compute the similarity between function s and every function b in the binary B, i.e., sim(∀b ∈ B, s).*

The similarity score is expected to be higher if the source code of b is similar to s, otherwise lower. The source code similarity types are defined in Definition 2.

Definition 2. *Let $U(\cdot)$ be the normalization operation, which normalizes the source code to a unified coding style, e.g., the Clang-Format. This would eliminate the difference brought by the white space, blank line, layout, and comment. Let operation $D(\cdot)$ show the different content between two codes. Given the source codes of two functions, a and b, we can get $a' = U(a)$, and $b' = U(b)$,*

- *Type-1 $\iff D(a', b') = \emptyset$.*
- *Type-2 $\iff D(a', b') \in \{I, L, T\}$, where $I, L,$ and T represent identifiers, literals, and data types, respectively.*
- *Type-3 $\iff D(a', b') \in \{I, L, T, S\}$ and $share(a', b') > t$, where S represents difference in statements, including changed, added, or deleted statements.*

The $share(\cdot)$ function is calculated as in Equation (1), where the operator $|\cdot|$ denotes the lines of code (LOC), $|a \cap b|$ the shared LOC, and t is a predefined threshold (0.5 by default).

$$share(a, b) = \frac{2 * |a \cap b|}{|a| + |b|} \tag{1}$$

Figure 2 presents the examples of similarity types. The original source code shown in Figure 2(a) has a vulnerability of denial of service (infinite loop) when the attacker controls the inputs to trigger a NULL value of a BIO data structure. Figure 2(b)(c)(d) show the code with type-1/2/3 similarity, respectively, where the vulnerability exists in all of them.

2.2 Assumptions

Since a function (procedure) usually serves as a standalone module, we compute the code similarity in the function level granularity, like many existing works [14, 21, 51]. We assume the input binary code is completely stripped, that is, no compilation or symbol information. Also, we assume the code in this binary shares the same compilation provenance, which often is the case for easy maintenance and usability [7]. Last, we assume the input source code is compilable.

3 OVERVIEW

BugGraph calculates the source-binary code similarity in two steps: source binary canonicalization and code similarity computation. The architecture of BugGraph is shown in Figure 3. The inputs are the unknown *target binary* to be investigated, and a source function with the ground truth, e.g., vulnerability. The output is the similarity score between every function from the binary code to the input source function.

In the first step, BugGraph canonicalizes the input source code and target binary code by converting the source code to a *comparing binary* with the compilation provenance identified based on the target binary code. Here the provenance is represented as a 4-tuple ($architecture, compiler\ family, compiler\ version, optimization\ level$), as they are the major factors that account for the variance in the binary. An example is ($x86, gcc, 4.8.4, O2$). In this work, we prepare a binary database offline with various compilation provenances, as compiling source code online would be slow. As a result, BugGraph can quickly obtain the comparing binary with a specific compilation provenance. It is possible that a particular provenance may not exist in the database, but one can always compile the source code as needed.

In contrast, prior works such as Gemini [51], Genius [21], and bipartite graph matching (BGM) [21, 51] would simply compile the source code with a predefined configuration, hoping to accounting for the impact of the compiling process at the later stage. We will show in Section 4.4 that while everyone performs reasonably well dealing with a handful of provenances, BugGraph is much more robust, with only 3% drop in TPR, as the number of provenances increases. In contrast, the three prior works experience a much larger TPR drop of 18%-28%.

In the second step, BugGraph computes the similarity between the target binary and the comparing binary code. To do so, we first disassemble both binary codes to assembly codes, and for each function in the binary, construct the attributed control flow graph (ACFG), which is demonstrated to be an effective representation for binary function [21, 51, 53]. Now the problem is transformed to the graph similarity computation, more accurately, one to many attributed graph comparison. In this work, we leverage graph neural network (GNN) to generate a representative embedding for each

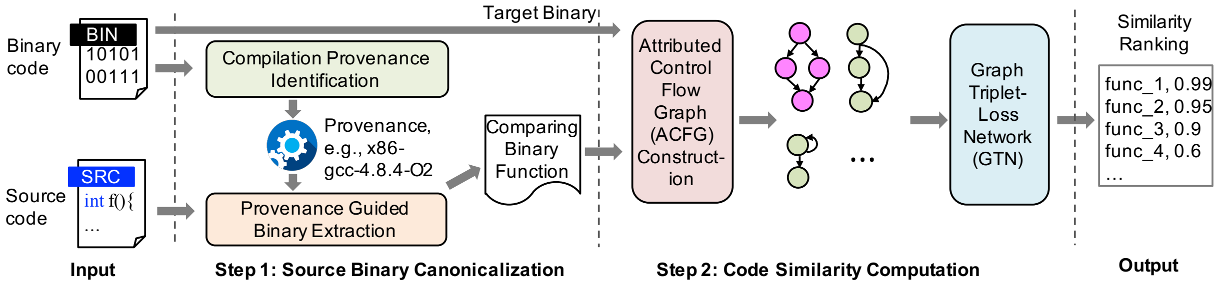


Figure 3: Architecture of BugGraph.

attributed graph, taking advantage of the recent development of machine learning techniques on graph data [29, 46, 50].

As the GNN model can not be directly used for learning the similarity, we add the triplet loss to the output of GNN model so that the GNN model can be supervised to learn to generate representative embeddings. The triplet loss takes a triplet, e.g., $\{a, b, c\}$, as the input and learns to rank the similarity so that the similarity between the first two is higher than that between the first and third, that is, $sim(a, b) > sim(a, c)$. Thanks to the ranking mechanism, our triplet loss is able to generate a fine-grained similarity value space, providing the desired coverage of less similar code, i.e., type-2 and type-3. Specifically, as we will show in Section 5.4, BugGraph outperforms the aforementioned works for both types, especially for type-3, 81% vs. up to 71% true positive rate.

4 PROVENANCE GUIDED SOURCE BINARY CANONICALIZATION

4.1 Compilation Challenge

In a compilation toolchain, there are three main stages: compilation (front end), optimization (middle end), and machine dependent code generation (back end) as shown in Figure 4. Given the source code, the compilation stage builds its intermediate representation (IR), where different compilers would apply different rules. The optimization stage applies various techniques on the IR aiming at improving the performance and code quality. The specified optimization level mainly affects this stage, with minor impact from the compiler. Note that the same optimization levels from different compiler families are different. For example, the optimization level O3 in gcc is different from the O3 in llvm. Lastly, the code generation stage further optimizes the code with architecture specific optimizations, and converts the optimized IR to the machine code. In short, the compiler, i.e., compiler family and version, affects all the three stages, the optimization level affects the optimization and code generation stage, and the architecture affects the code generation stage.

To show the impact of compilation variance, we conduct an experiment on a commonly used binary, where we compile OpenSSL (version 1.0.1f) with various provenances and measure the difference in control flow graph (CFG) to represent the code similarity. To calculate graph difference, one could use existing methods such as graph edit distance [5] and maximum common subgraph [25], all of which are NP-complete problems and could take hours to days in our tests to converge for two small graphs with only tens of nodes and edges. For the scalability reason, we calculate graph difference as defined in Equation 2, where $|V_i|$, $|E_i|$ denote the vertex and edge count for graph g_i, and $diff(\cdot)$ value is in range $[0, 1]$. A higher score shows the two graphs are more different from each other.

$$diff(g_1, g_2) = 1 - \frac{\min(|V_1|, |V_2|) + \min(|E_1|, |E_2|)}{\max(|V_1|, |V_2|) + \max(|E_1|, |E_2|)} \qquad (2)$$

In this experiment, we use the binary with the provenance ($x86$, $gcc4$, $4.8.4$, $O3$) as the baseline, and compare with others with a single change in the 4-tuple provenance, using arm, llvm-3.5, gcc-4.6.4, and O0 for architecture, compiler family, compiler version, and optimization level.

From Figure 5, one can see that the optimization level has the largest effect, where there are 12% functions completely different (difference score equals to 1) because the binary functions compiled from the same source code do not exist in the other. Also, up to 32% functions have difference scores greater than 0.5, and 69% of the functions are different (with a score greater than 0). Further, the other three configurations bring additional challenges. There are 65%, 57%, and 35% functions that are different as a result of different architecture, compiler family, and compiler version, respectively.

4.2 Compilation Provenance Identification

Identifying the compilation provenance of a binary is possible as the rules used during compilation process will be reflected on the binary code, such as instructions, instruction order, control flow structure, and function dependencies [40, 41]. In this paper, we use the standard *file* program in Unix-like operating systems to tell the correct architecture of a binary, and leverage a tool customized from Origin [40] for the purpose of identifying the compiler family, compiler version, and optimization level.

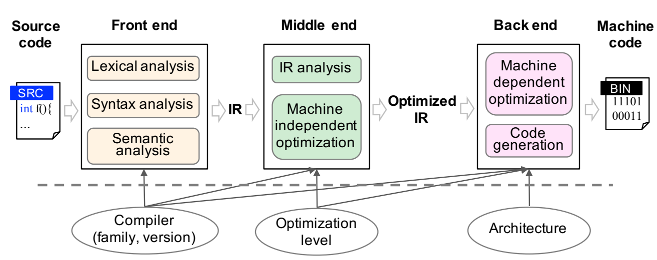


Figure 4: A typical compilation process with the impacted stages of each element from the 4-tuple compilation provenance.

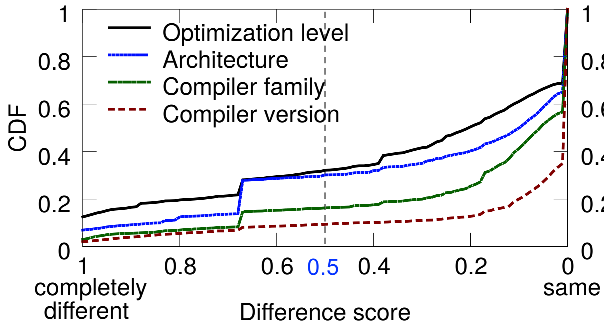


Figure 5: CDF of difference scores from the control flow graphs of OpenSSL with different compilation provenances.

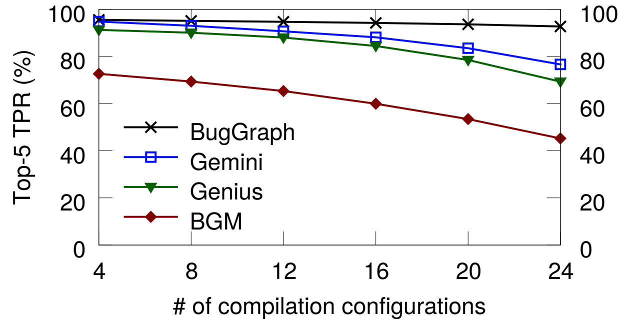


Figure 6: The top-5 TPRs of type-1 similar code when scaling to more compilation configurations w.r.t. compiler and optimization.

Below we will briefly introduce how Origin works. It extracts the code features from instruction and control flow graph. The instruction features are called idioms, which are short sequences of instructions with wildcards. A typical idiom feature is (*push ebp* | * | *mov esp, ebp*), which represents a common stack frame set-up operation with a wildcard in-between. On the other hand, the control flow graph features are small, non-isomorphic subgraphs of the control flow graph. Origin generates a large number of these two features and extracts the top-k (in thousands) representative features over the total (up to millions) with mutual information. In the end, Origin trains a provenance identifier with the conditional random field method.

4.3 Provenance Guided Binary Generation

Once we get the compilation provenance of the target binary code, we can compile the source code to the binary format with the same provenance. Thus, the two inputs are canonicalized to the same format.

As we have discussed earlier, online compilation would incur undesired overhead, thus we turn to offline compilation. That is, for one open-source software, we will compile it with various configurations, and extract the binaries to the database. Later, we can easily get the desired comparing function as we keep the symbol names during compilation. With offline compilation, one can easily add new binaries to the database. Another advantage of offline compilation is reliability because online compilation could fail due to unpredicted reasons, such as missed library or dependency, incompatible environment, or incorrect configuration.

4.4 Benefit of Provenance Identification

We evaluate the benefit of using provenance guided source binary canonicalization by comparing BugGraph and related works. In this test, we use the same source code, i.e., type-1, but with different compilation configurations in terms of compiler family, compiler version, and optimization level. Starting from one compiler (gcc-4.6.4) and all of its optimization levels (O0-O3), we add other compilers in the order of gcc-{4.8.4, 5.4.1} and llvm-{3.3, 3.5, 5.0}. Both the training and testing use same compilation configurations. We report the average results of the 1,000 source functions from a syntax similar dataset (Dataset II, which will be discussed in Section 7).

We compare BugGraph with two recent works, Gemini, Genius, and a baseline method, bipartite graph matching (BGM). For Gemini, we use the open source code[1], and for Genius, we obtain the source code on ACFG extraction and codebook generation from the authors, implementing other components ourselves. BGM measures the similarity of two ACFGs with the Hungarian assignment algorithm, where we reuse the implementation from Genius. The three methods will compile the source code with a random provenance, and we tune the parameters for best performance.

From Figure 6, one can observe that BugGraph is able to scale to more compilation configurations with consistent accuracy, while others can not. Particularly, with four compilation configurations, BugGraph, Gemini, Genius, and BGM achieve 96%, 95%, 91%, and 73% true positive rate (TPR), respectively, when we take the top-5 candidates as positives. With 24 compilation configurations, BugGraph only drops 3% TPR, while Gemini, Genius, and BGM drop 18%, 22%, and 28%, respectively. The scalability of BugGraph is benefited from the provenance guided canonicalization because we always compare a target binary with the comparing binary function sharing the (predicted) compilation provenance. Furthermore, it is important to note that this result is consistent with the reported numbers in both Gemini and Genius as they show good accuracy with less compilation configurations in terms of compiler and optimization level.

5 CODE SIMILARITY COMPUTATION

5.1 Binary Code to Attributed Graph

Existing works would first disassemble the binary code to assembly code, in which the statement is combined by operation code (opcode) and operand. Further, the control flow operations, e.g., branch statement, would split the assembly code to multiple basic blocks, where either all the statements inside one basic block will execute together, or none of them execute. Taken each basic block as node and the control flow relationship as edge, one can get the control flow graph (CFG). As CFG maintains the code structure, it is an essential representation for code analysis [48, 52]. However, only using the CFG without the specific assembly code ignores the syntax features.

In this work, we employ the attributed control flow graph (ACFG) by attributing each node as a syntax feature vector. The ACFG is

[1] https://github.com/xiaojunxu/dnn-binary-code-similarity

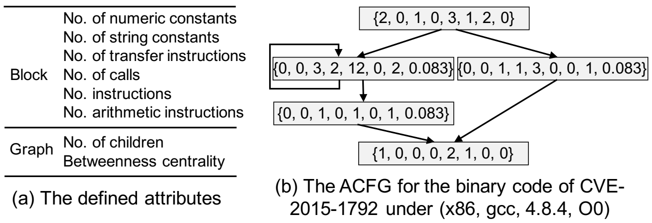


Figure 7: The attributed control flow graph, (a) the defined attributes, (b) an example ACFG.

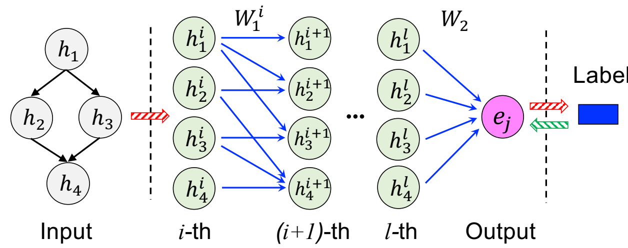


Figure 8: The architecture of graph neural network.

shown to be an efficient representation for binary code [21, 51, 53]. Particularly, the attributes are extracted from both the basic block and CFG level, shown in Figure 7(a). There are six basic block features, i.e., number of numeric constants, string constants, transfer instructions, calls, instructions, and arithmetic instruction, as well as two attributes calculated on the CFG, i.e., number of children, and betweenness centrality, which measures the node importance based on the passed shortest paths [11]. The ACFG for CVE-2015-1792 in Figure 1 under $(x86, gcc, 4.8.4, O0)$ is shown in Figure 7(b).

5.2 Attributed Graph Embedding

Once ACFGs are constructed, the similarity of two binary codes is transformed to the similarity of two attributed graphs. A good algorithm to calculate graph similarity needs to be not only accurate but also scalable. The latter is important due to the need for computing a large number of pairs of attributed graphs. For example, there are 6,441 functions in the OpenSSL binary (version 1.0.1f) if compiled with $(x86, gcc, 4.8.4, O0)$. If more than 100 vulnerable functions were to be studied as in prior work [21, 51], this would easily mean that one needs to compare millions of graph pairs for only one binary. To address this problem, we leverage the recent advances in graph neural network (GNN) to learn a representative embedding for each attributed graph, which can then be used for accurate similarity computation [29, 31, 46, 47, 50].

In BugGraph, the architecture of the GNN model is shown in Figure 8. There are three types of layers: input layer, hidden layers, and output layer. In the GNN, the input layer takes an attributed graph, in our case ACFG, where h_v denotes the original embedding for node v.

The most important layers of the GNN are the hidden layers that iteratively update the embedding of a node by accumulating the embeddings of its neighbors and itself from the previous iteration. The common graph convolution method learns a function f to generate a node v's embedding at the $(i+1)$-th layer with its embeddings h_v^i and all of its neighbors' embeddings $\mathcal{N}(v)$ from the i-th layer [29]. In general, $h_v^{i+1} = f\left(h_v^i, \mathcal{N}(v)\right)$. Thus, in the example, the embedding of node 2 at the $(i+1)$-th layer h_2^{i+1} is generated from h_1^i and h_2^i.

In this work, we use an attention mechanism to capture the more important nodes, assigning larger weights when generating the embedding [46]. Formaly, the attention-based function is defined in Equation 3, where $\sigma(\cdot)$ is a activation function, W_1^i the trainable weight matrix at i-th layer, and α_{uv} the learned attention coefficient for each edge.

$$h_v^{i+1} = \sigma\left(\alpha_{vv} h_v^i + \sum_{u \in N(v)} \alpha_{vu} W_1^i h_u^i\right) \tag{3}$$

After a total of l iterations, one accumulates all the node embeddings to produce the final embedding as $e_j = W_2 \sum_{v \in V} h_v^l$ for the j-th ACFG, where W_2 is another trainable weight matrix. Thus, the graph embedding of the j-th ACFG, e_j, equals to $gnn(g_j)$, which represents the computation process of a GNN model. In the end, based on the label from the downstream task, e.g., graph classification, the model will compute the loss value for that task, e.g., softmax function for classification [9], back propagate to the hidden layers, and tune the trainable parameters.

In the following, we will use an example to illustrate the use of the ACFG and graph embeddings. Figure 9 shows three functions from SNNS and PostgreSQL from Dataset II (discussed in Section 7): the functions `or`, `less`, and `gistwritebuffer` are compiled with (x86, llvm, 3.3, O2), (x86, llvm, 5.0, O2), and (x86, llvm, 3.3, O2), respectively. The first two functions share type-3 similarity and the third one is a different function. In this example, the ACFGs on the top row of Figure 9 are the inputs to the GNN, which in turn produces the graph embeddings for each ACFG. The outputs of the GNN are three graph embeddings, e.g., e_a, e_p, and e_n.

5.3 Graph Triplet-Loss Network for Similarity Ranking

The goal of BugGraph is to accurately capture the subtle difference among these ACFGs, and functions. In this work, the similarity is measured by the cosine similarity, which has been shown to be effective for the embeddings in high dimensional space [51]. For any two vectors, i.e., $\vec{A}$ and $\vec{B}$, it is formally defined as:

$$sim(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}} \tag{4}$$

The similarity score is in the range [-1, 1], where the higher the value is, the more similar the embeddings are. From Figure 9, one can see that the generated embeddings of the first two functions, `or` and `less`, show a high (0.95) cosine similarity score, while the first and third functions, `or` and `gistwritebuffer`, show a low (0.45) score.

The GNN model is not sufficient by itself to model the similarity, as it needs a proper loss function to supervise the learning process. In the context of code similarity computation, the loss function should address the following two challenges. First, it should be

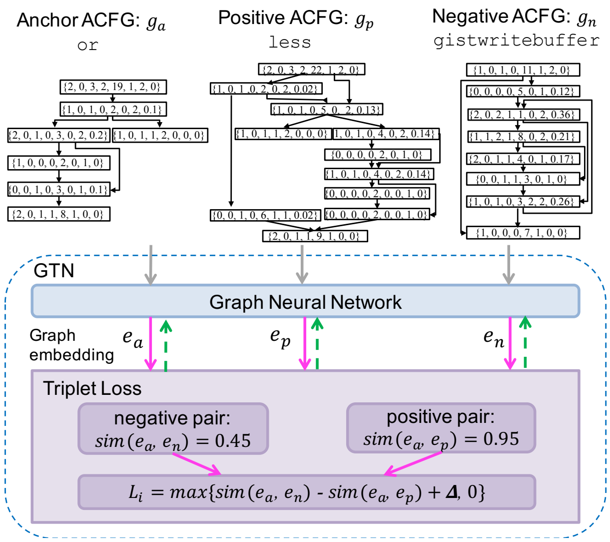


Figure 9: Graph triplet-loss network architecture. Solid and dashed arrow denote forward and backward propagation, respectively.

able to generate loss values based on the similarity, that is, the loss value should be small if two similar code have similar embedding. Second, the various code similarity types require the learned model to be able to detect subtle difference in codes. In other words, the model should be able to learn that the type-1 is more similar than type-2 and type-3, type-2 more similar than type-3, and type-3 more similar than completely different code. Therefore, the similarity ranking can be represented as type-1 > type-2 > type-3 > different.

To address both challenges, BugGraph builds a graph triplet-loss network (GTN) which relies on the triplet loss [43] to supervise the learning of the GNN model. Figure 9 shows the workflow of the GTN model. The input to our GTN is a triplet of ACFGs (binary functions), which consists of the anchor graph (g_a), positive graph (g_p), and negative graph (g_n), i.e., $\{g_a, g_p, g_n\}$. The idea is to compute the ranking of similarity where g_a and g_p are more similar than g_a and g_n.

At the core of graph triplet-loss network is the triplet loss computation for the similarity of two pairs, that is, the positive pair $\{e_a, e_p\}$ and negative pair $\{e_a, e_n\}$. Formally, the loss value $\mathcal{L}_i$ for the i-th triplet is defined as

$$\mathcal{L}_i = \max\{sim(e_a^i, e_n^i) - sim(e_a^i, e_p^i) + \Delta, 0\} \tag{5}$$

which is greater than or equal to 0. Here Δ denotes the margin to enhance the distance between positive and negative pairs so that the model can put the similar pair closer and the different pair further in the high dimensional space. For the example in Figure 9, the loss value would be $\max\{\Delta - 0.5, 0\}$. The margin value Δ plays an important role on the accuracy of similarity computation. A larger margin can better stretch the distance between positive and negative samples but requires more training time to reach a smaller loss value, while a smaller margin can reduce the training time at the loss of accuracy (will be evaluated in Section A.2).

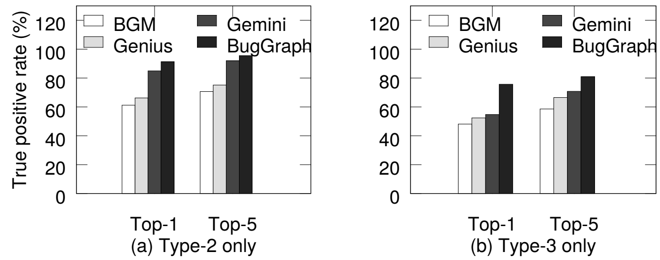


Figure 10: Benefits of triplet loss. (a) shows the TPR against type-2 similar code, (b) shows the TPR against type-3 similar code.

As the loss value is back propagated to the GNN model, one can use an optimizer, e.g., gradient optimization, to tune the trainable parameters in order to minimize the loss value. Formally, for the training triplet set $\mathcal{T}$, we will tune the GNN model based on Equation 6.

$$\min_{W_1^1,\ldots,W_1^l,W_2,\alpha} \sum_{i}^{|\mathcal{T}|} \mathcal{L}_i \tag{6}$$

As a result, the GNN model can be supervised to generate representative embeddings for the purpose of similarity ranking. To this end, our GTN model is end-to-end trainable.

It is important to note that the triplet loss also introduces another benefit that the similarity relationship can be transitive. That is, if the triplets $\{a, b, c\}$ and $\{a, c, d\}$ exist, that means $sim(a, b) > sim(a, c)$ and $sim(a, c) > sim(a, d)$, then $sim(a, b) > sim(a, d)$, which means the triplet $\{a, b, d\}$ inherently exists. Exploiting the transitivity among a large set of triplets, we can learn a more accurate model to map a broader similarity space, which enables highly similar code to be ranked higher at the inference stage.

5.4 Benefit of Triplet Loss

To evaluate the benefits of triplet loss towards covering the syntax similar code, we conduct an experiment for testing the syntax similarity types, i.e., type-2 and type-3. In particular, we use the binaries from a syntax similar dataset (Dataset II, discussed in Section 7). We make sure the code is compiled under the same compilation configuration, i.e., $(x86, gcc, 4.6.4, O2)$ in this experiment. Later, we use $2,000$ type-2 and type-3 source functions $(1,000$ for each) to search the target binaries.

The accuracies of type-2 and type-3 code similarity types are shown in Figure 10. One can see that BugGraph outperforms the compared works for both similarity types when measuring top-1 and top-5 TPR. For the difficult type-3 similar code, BugGraph achieves 81% of top-5 TPR compared to 71% for Gemini and 66% for Genius. For type-2, BugGraph also achieves the best accuracy. Specifically, for the top-5 TPR, BugGraph achieves 96%, and Gemini gets 92%, both outperform Genius by a large margin. The understanding of similarity ranking is discussed in Appendix A.1.

6 IMPLEMENTATION

ACFG construction is implemented with a commercial off-the-shelf tool, IDA-Pro [1]. We start by building the control flow graph

(CFG) and traverse each basic block to get the six basic block features. As for the CFG features, we count the number of children for each node, and run betweenness centrality algorithm on the CFG.

GTN is implemented on top of TensorFlow (version 1.3.0). We use the graph attention network as our GNN [46]. We set the intermediate and final embedding size to be 512, the number of epochs 100, the number of iterations 5, the margin value Δ 0.5, and the batch size 10.

Triplet generation. The triplets are generated in the following way. First, for each type-2 and type-3 similar function, we create two triplets $\{g_1, g_2, g_{r1}\}$ and $\{g_1, g_3, g_{r2}\}$, where g_1 and g_2 are type-2 similar to each other, g_1 and g_3 are type-3 similar, g_{r1} and g_{r2} are two randomly selected different functions. Second, for each binary function g_i, we randomly select one g_i' from its other compilation provenances and a different one g_r to generate a triplet $\{g_i, g_i', g_r\}$. This is to ensure that our model can handle the compilation variance, even when a wrong provenance was identified (18% of the time in our tests). Note that we can split a triplet to obtain the pairs for comparison with related works. For example, $\{g_1, g_2, g_{r1}\}$ can be divided into $\{g_1, g_2\}$ (of similar) and $\{g_1, g_{r1}\}$ (of different).

For a new target binary, BugGraph creates the triplets in the format $\{g_1, b_1, b_2\}$, where g_1 denotes the known comparing function (e.g., vulnerability), and b_i denotes the i-th function in the target binary. For every comparing function, we will create $(n + 1)/2$ triplets, where n denotes the number of functions in the target binary.

7 EXPERIMENT

This section presents the experiment, while we present the parameter sensitivity study and runtime analysis in Appendix A.2 A.3.

7.1 Experimental Setting

We run the experiments on a server with two Intel Xeon E5-2683 v3 (2.00 GHz) CPUs, each of which has 14 cores and 28 threads, 512 GB of memory, and a Tesla K40c GPU. As we have discussed in Section 4.4, we compare BugGraph against Gemini, Genius, and bipartite graph matching (BGM).

Evaluation metrics. We evaluate BugGraph and related works with the following metrics, i.e., true positive rate (TPR), false positive rate (FPR), true negative rate (TNR), false negative rate (FNR), and accuracy. Given a binary and a query function, there should be m matchings among a total of n functions. Assume the top-k extracted similar functions are positives, if there are p correctly matched functions, TPR = $\frac{p}{m}$, FPR = $\frac{k-p}{n-m}$, TNR = 1− FPR, and FNR = 1− TPR. The accuracy is defined as the sum of true positives and true negatives over all. We also use the receiver operating characteristic (ROC) curve to show the change of TPR against FPR, and the area under a ROC curve (AUC) to illustrate the effectiveness of a model, where the closer to 1 the better.

The four datasets used in our experiments (and will be released upon the publication of this paper) consist of:

Dataset I: Validation dataset[2] is a dataset used in Gemini [51] (also referred to as Dataset I), which is used to validate our results and ensure a fair comparison against related projects. This dataset extracts the ACFGs from the binaries of OpenSSL (version 1.0.1f and 1.0.1u) with compiler gcc-5.4, optimization level O0-O3, and architecture x86, ARM, and MIPS. In total, there are 129,365 ACFGs from 7,305 different functions.

[2]https://github.com/xiaojunxu/dnn-binary-code-similarity/blob/master/data.zip

Table 1: Specifications of syntax similar dataset (Dataset II).

	Software	# of Type-2	# of Type-3	# of binaries	# of functions
Train	SNNS-4.2, PostgreSQL-7.2	152	524	600	493,841
Test	Binutils-{2.25, 2.30}, Coreutils-{8.21, 8.29}	1,436	3,811	5,568	2,648,627
Total	-	1,588	4,335	6,168	3,142,468

Dataset II: Syntax similar dataset, including a publicly available source code similarity dataset from SNNS-4.2 and PostgreSQL-7.2 with 152 and 524 pairs of type-2 and type-3 code from prior work [8, 36], as well as our manually labelled dataset from Binutils-{2.25, 2.30} and Coreutils-{8.21, 8.29} with 1,436 and 3,811 pairs of type-2 and type-3 code. All the software is compiled with six different compilers, i.e., gcc-{4.6.4, 4.8.4, 5.4.1} and llvm-{3.3, 3.5, 5.0}, and four optimization levels (O0-O3) under the same architecture (x86). There are 24 compilation provenances for each binary. In total, there are 6,168 binaries and 3,142,468 functions.

It is important to note that different binaries from the same software may share the same functions, which can lead to biased testing results if the dataset were divided in the binary level for training and testing [6]. We avoid this issue by splitting in the software level as shown in Table 1. Particularly, the training dataset includes the binaries from SNNS-4.2 and PostgreSQL-7.2 dataset, while the testing dataset includes the binaries from Binutils-{2.25, 2.30} and Coreutils-{8.21, 8.29}. This way, there is no overlapping between the two dataset, which is able to provide reliable results and demonstrate the generalizability of the learned model.

Dataset III: ARM binary dataset includes the binaries for ARM architecture from SNNS-4.2, PostgreSQL-7.2, and OpenSSL. For OpenSSL, we get the source code of three versions (0.9.7f, 1.0.1f, and 1.0.1n). We cross compile them using four compilers gcc-{4.6.4, 4.8.4} and llvm-{3.3, 3.5} with four optimization levels, i.e., O0-O3. In the end, this dataset consists of 544 binaries.

Dataset IV: Firmware image dataset consists of six firmware images from TP-Link routers (model ArcherC9, AD7200, TouchP5, ArcherC2600, ArcherC5200, and ArcherC5400).

7.2 Validation Test

The Dataset I considers the binary functions compiled from the same source code as similar, a.k.a., type-1, and the ones from different source code as different. This dataset does not include type-2 or type-3. As in the experiment of Gemini, the whole dataset is divided into training, validation, and testing with ratio [0.8, 0.1, 0.1]. No two binary functions compiled from the same source code are assigned to the same group. The specifications of the dataset are shown in Table 2.

Gemini generates two pairs for each binary function as follows. Given a binary function g_1, a binary function g_1' compiled from the same source code with a different provenance is randomly selected as similar, and another binary function g_2 compiled from different

Table 2: Specifications of the validation dataset (Dataset I).

	Train	Validate	Test	Total
# of ACFGs	103,732	12,726	12,907	129,365
# of unique functions	5,844	730	731	7,305

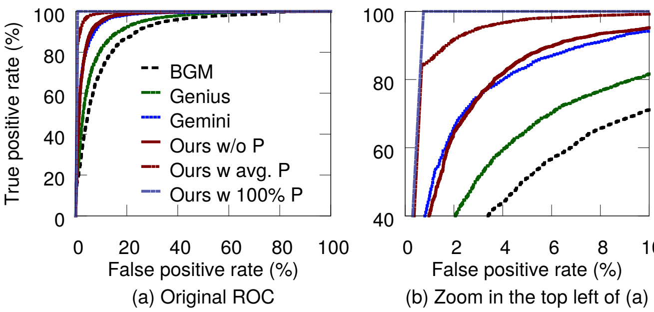


Figure 11: ROC curves for different methods on the validation dataset, (a) shows the original ROC, (b) zooms in on the top left of the original ROC (P denotes provenance).

source code is randomly selected as different. That is, $\{g_1, g_1'\}$ is similar, $\{g_1, g_2\}$ is different.

BugGraph generates the triplets by merging the two pairs from the same binary function. For example, for the two pairs, $\{g_1, g_1'\}$ and $\{g_1, g_2\}$, we generate one triplet, $\{g_1, g_1', g_2\}$. As the binaries are not shared in this dataset, we cannot directly apply our provenance identification. In this case, we provide three varieties of BugGraph, i.e., without provenance, average reported accuracy (82%) of provenance identification, and with an oracle (100% accuracy).

Figure 11 illustrates the ROC curves of different methods on the same testing dataset. We make three observations here. (1) We are able to reproduce the results of previous works (e.g., refer to Figure 5 in [51] for the ROC curves of Gemini, Genius, and BGM). (2) For type-1 code similarity detection, BugGraph without provenance is as good as Gemini, and they are significantly better than other works. Particularly, BugGraph without provenance achieves 0.973 AUC value, which is close to Gemini's 0.97. Both are higher than Genius' 0.936 and BGM's 0.905. It is interesting to point out that graph embedding methods (BugGraph, Gemini, and Genius) are all effective for binary code similarity detection. (3) With provenance identification, BugGraph is able to further improve the performance of type-1 code similarity detection. With the average accuracy (82%), BugGraph is able to achieve 0.991 AUC value. Further, BugGraph with the 100% correct provenance is able to achieve 0.996 AUC value. This clearly shows the importance of identifying compilation provenance for code similarity detection.

7.3 Accuracy of Source-Binary Code Similarity Detection

This experiment evaluates the accuracy of BugGraph and compared works on detecting source-binary code similarity. We use the syntax similar dataset (Dataset II), where the source functions are from Binutils-2.25 and Coreutils-8.21, and the target binaries are from Binutils-2.30 and Coreutils-8.29. The target binaries are compiled with 24 compilation provenances varying from compiler (family and version) and optimization level. We randomly select 1,000 type-1, type-2, and type-3 (3,000 in total) as the source functions. For each source function, we search all the binaries in the target dataset and report the average on each binary of all the queries. All the compared works are evaluated in the same setting.

Table 3: The false positive rate (FPR) in percentage (%) against different top-k values for source-binary code similarity detection (with the lowest FPR value highlighted).

	Method	Top-1	Top-5	Top-10	Top-15	Top-20
Type-1	BGM	0.07	0.46	0.95	1.45	1.94
	Genius	0.05	0.43	0.93	1.42	1.92
	Gemini	0.03	0.42	0.92	1.42	1.91
	BugGraph	**0.01**	**0.41**	**0.9**	**1.4**	**1.9**
Type-2	BGM	0.08	0.51	1.06	1.61	2.16
	Genius	0.07	0.5	1.04	1.59	2.15
	Gemini	0.05	0.47	1.02	1.58	2.13
	BugGraph	**0.02**	**0.46**	**1.01**	**1.56**	**2.12**
Type-3	BGM	0.1	0.55	1.13	1.71	2.29
	Genius	0.09	0.54	1.12	1.7	2.28
	Gemini	0.08	0.53	1.1	1.69	2.27
	BugGraph	**0.05**	**0.5**	**1.08**	**1.67**	**2.26**

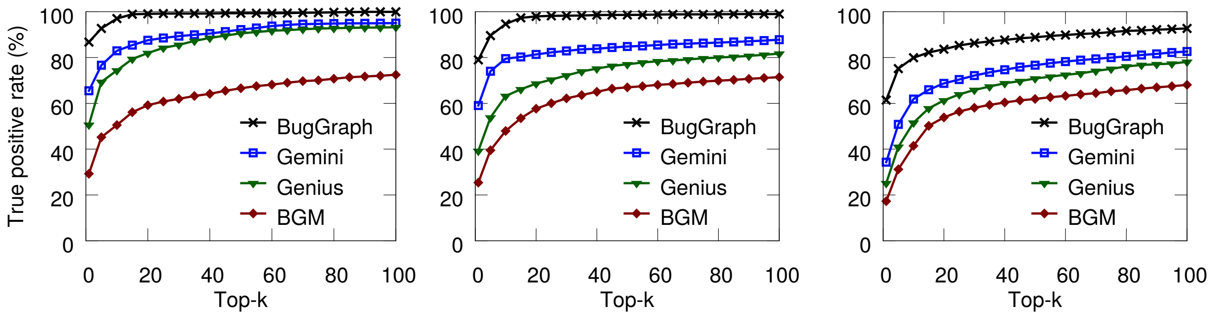
BugGraph leverages the provenance identification to canonicalize the source and binary code, where it identifies the compilation provenance with an overall accuracy 82% in this experiment. Specifically, the provenance identifier achieves 100%, 100%, 96% and 84% accuracy for architecture, compiler family, compiler version, and optimization level, respectively.

BugGraph compiles the source function to the comparing binary with the predicted provenance. In contrast, other projects use a binary with random provenance. The evaluation results against different top-k values are shown in Figure 12. One can see that, BugGraph is able to outperform the recent works with a large margin under different top-k values, especially for the smaller k values. Taking top-5 TPR as example, for the identical type-1 code, BugGraph is able to achieve 93% TPR, which is at least 16% better than others. In this case, Gemini, Genius, and BGM obtain 77%, 69%, and 45%, respectively. Again, this shows the importance of our provenance identification as only compiler induced variance exists in type-1 similar code.

For the syntax identical type-2 code, BugGraph achieves 90% TPR, while Gemini, Genius, and BGM obtain 74%, 54%, and 40%, respectively. For the most difficult syntax similar type-3 code, BugGraph is able to get 75% TPR, which is 24%-44% higher than others. In particular, Gemini, Genius, and BGM get 51%, 41%, and 31%, respectively. On can get similar observations from top-1 TPR.

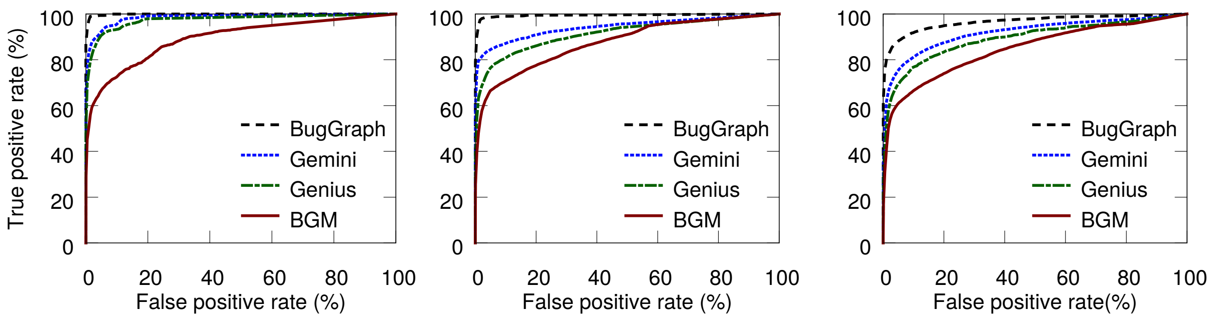
We present the ROC in Figure 13 to show the change of TPR against FPR, where (a) (b) (c) present type-1, 2, 3 similar code. One can see that BugGraph achieves better AUC values than compared works for all three code similarity types, especially for type-2 and type-3 code. Specifically, BugGraph achieves 0.998, 0.994, and 0.965 AUC values for type-1, 2, 3 similar code, respectively, while Gemini achieves 0.985, 0.943, and 0.919. Note that the ROC results in this paper also align with the reported results in previous works.

Further, we study the false positive rate (FPR) of BugGraph and compared works as shown in Table 3. Although all the works show close FPR values for different top-k, BugGraph gets the lowest under different settings. They are consistent for different types of



(a) Type-1 (b) Type-2 (c) Type-3

Figure 12: Accuracy of detecting source binary code similarity against different top-*k* values for (a) type-1, (b) type-2, and (c) type-3.



(a) Type-1 (b) Type-2 (c) Type-3

Figure 13: ROC of detecting source binary code similarity for (a) type-1, (b) type-2, and (c) type-3.

similar code. In addition, the FPR values for all the methods are small. When k is smaller than 20, all the methods get less than 3% FPR for all the three types of similar code. The small values are due to the high true negative (TN) as there are up to several thousands of different functions (true negatives) inside a binary.

7.4 Firmware Vulnerability Detection

In this test, we apply BugGraph to identify vulnerabilities from the real-world firmware, which is known to have numerous binaries [12, 24, 30]. To achieve that, we extend BugGraph to support the commonly used architecture of firmware, i.e., ARM. Particularly, we retrain BugGraph with the binaries under ARM architecture (Dataset III). Here we build a vulnerability database with 218 known vulnerable functions, where 126 of them are obtained from Genius [21], as well as additional 92 vulnerable functions are manually collected by ourselves. The vulnerabilities are from several versions of three open source projects, i.e., OpenSSL, Binutils, and Coreutils, and compiled to various provenances offline.

For each binary in the firmware image dataset (Dataset IV), BugGraph predicts its compilation provenance, builds the ACFGs for all the functions inside, and computes their similarities to the vulnerable functions in our database. For the compilation provenance, the binaries from ArcherC9, AD7200, TouchP5, ArcherC2600, ArcherC5200, ArcherC5400 are mostly predicted as gcc-4.6.4-O0, gcc-4.6.4-O2, gcc-4.6.4-O0, gcc-4.6.4-O2, gcc-4.8.4-O0, and gcc-4.8.4-O0, respectively. We search each firmware image for the 218 vulnerable functions. For each image, we get top-10 candidates for each vulnerable function, filtering out the candidates with the similarity score of less than 0.9. The remaining candidates are manually investigated and we are able to identify 140 OpenSSL vulnerable functions from 42 unique CVEs. The existence of these vulnerable functions is further confirmed by checking the binary versions in the image. The found CVEs and their appearances (number of firmware images that have such CVEs) are summarized in Table 4.

It is important to note that some severe vulnerabilities have not been patched. For example, the CVE-2016-2842, which allows the attacker to cause denial-of-service with the highest severity score 10 [2], appears in five of the investigated firmware except TouchP5. Also, some old vulnerabilities, e.g., CVE-2013-6449, still exist in the current firmware.

8 RELATED WORK

Binary features-based approaches. The early works of binary code similarity detection mainly use the instruction features, e.g., n-gram [37] and n-perms [27]. Beyond instruction features, Zynamics BinDiff [19] and BinSlayer [10] are two representative works of using structure features. Recent research has used both instruction and structural features. For example, Rendezvous [28] decomposes the CFG into subgraphs and uses size-k subgraph as graph feature, coupled with instruction features represented as n-grams and n-perms on instruction mnemonics. David et al. [13] decompose the original CFG into tracelet, which is a continuous, short, partial traces of an execution. The code similarity is measured as the similarity of tracelet. TEDEM [39] captures the instruction features using the expression tree for a basic block and computes code similarity with tree edit distance. BinHunt [22] uses symbolic execution

Table 4: Discovered vulnerabilities from six recent firmware.

CVE	# Appear	Vulnerability type	CVE	# Appear	Vulnerability type	CVE	# Appear	Vulnerability type
2016-6303	5	Out-of-bounds write	2016-0702	5	Side-channel attack	2015-0206	4	Allow DoS attack
2016-6302	5	Remote DoS attack	2016-0701	2	Miss required crypto	2015-0205	4	Allow remote access
2016-2842	5	Out-of-bounds write	2015-3197	3	Man-in-the-middle	2015-0204	4	Downgrade attack
2016-2182	5	Out-of-bounds write	2015-1794	2	Segmentation fault	2014-8176	4	DoS overflow
2016-2180	5	Out-of-bounds read	2015-1792	4	Allow DoS attack	2014-5139	3	Null pointer derefer
2016-2178	5	Side-channel attack	2015-1791	4	Double free	2014-3572	4	Downgrade attack
2016-2176	2	Buffer over-read	2015-1790	4	Null pointer derefer	2014-3567	4	Remote DoS attack
2016-2109	2	Allow DoS attack	2015-1789	4	Out-of-bounds read	2014-3511	1	Man-in-the-middle
2016-2105	2	Memory corruption	2015-1788	4	Allow DoS attack	2014-3508	1	Information leakage
2016-0799	5	Out-of-bounds read	2015-0292	4	Integer underflow	2014-3470	1	Null pointer derefer
2016-0797	3	Integer overflow	2015-0288	4	Null pointer derefer	2014-0221	1	Remote DoS attack
2016-0705	5	Double free	2015-0287	4	Invalid write	2014-0198	1	Null pointer derefer
2016-0704	4	Information leakage	2015-0286	4	Invalid read	2014-0195	1	Buffer overflow
2016-0703	1	Man-in-the-middle	2015-0209	4	Use-after-free	2013-6449	1	Daemon crash

and a theorem prover to check all possible pairs of equations between basic blocks. iBinHunt [34] extends the comparison to interprocedural CFGs and further prunes the candidates. Pewny *et al.* extend code similarity detection to cross architecture binaries[38], which uses I/O patterns as instruction features, represents each function as CFG, and computes the similarity of two CFGs with graph isomorphism.

Machine learning-based approaches. DiscovRe [20] extracts nine statistic features to represent each basic block. It firstly filters out the obviously different functions with a machine learning algorithm, i.e., k nearest neighbor. Later, it computes code similarity with maximum common subgraph matching on CFGs. On the other hand, Asm2Vec [14] decomposes the CFG of a function into multiple sequences and generates a numeric vector for each function based on the PV-DM model. The function similarity will be calculated as the corresponding vector similarity. Similarly, INNEREYE [57] decomposes the CFG into a number of sequences, and leverages Word2vec to generate instruction embedding and the long short term memory model to generate sequence embedding.

Graph embedding methods use various graph-based machine learning techniques. The early work Genius [21] combines the syntax features and CFGs into an attributed CFG, and calculates the function similarity with graph edit distance. In addition, Genius uses a "bag-of-words" idea to create the high-level embedding for each graph. To provide fast query scalability, Genius uses semantic hashing on high-level embeddings to fast get the candidates. To further improve the performance, Gemini designs the first work of using graph neural network (GNN) [51] where the Siamese Network is used to compute function similarity. DeepBinDiff [18] designs an unsupervised deep neural network based method, which applies the Word2vec model to extract semantic information for tokens and thus generates basic block feature vector. Later, it uses random walk-based unsupervised graph embedding method to learn basic block embedding from the inter-procedural CFG. In the end, they are able to differentiate two binaries from basic block level.

Comparison. BugGraph is different from prior works in three aspects. First, this work is motivated by the real-world use cases of binary code similarity detection, where the source code of the comparing binary is usually available. As a result, BugGraph focuses on the problem of source-binary code similarity detection. As it is challenging to capture consistent features between source and binary code, we devise a new canonicalization method with provenance identification. Second, for the comparing source code, the prior works usually compile it to a binary code with random configuration. Instead, BugGraph would identify the compilation provenance of the target binary code and then compile the source code to the same provenance. This would greatly ease the following code similarity detection task. Third, compared to prior works that are mostly focusing on the type-1 similar code, we introduce ranking based triplet loss to supervise the learned model. Thanks to the ranking mechanism, the model is able to stretch the similarity space, which is thus able to cover the less similar code (type-2/3).

9 DISCUSSION

Currently, BugGraph is designed for binary code which is not obfuscated or maliciously modified. The recent advances on binary deobfuscation have achieved high accuracy [15, 49], which we hope to utilize in the future development of BugGraph. Meanwhile, it is possible to maliciously modify the binary to fool the *file* software, which we used to identify the architecture. We would like to explore more robust architecture identification techniques in the future.

The offline compilation needs to prepare a variant with every known compilation provenance for the source code. It is possible that the source code and its dependent libraries may not be compilable under some compilation configurations. For such cases, we will find a secondary candidate in the order of architecture, compiler version, compilation family, and optimization level. Nevertheless, it is worthy noting that even with partial provenance information, say compiler family or optimization level, the code similarity comparison accuracy can still be improved as we have shown previously.

While BugGraph is designed for source-binary code similarity detection, our graph triple-loss based network can also be used for the traditional binary-binary code similarity detection when the source code is unavailable or uncompilable, which has been demonstrated in Section 7.2.

10 CONCLUSION

In this work, we have designed BugGraph, a new system that identifies source-binary code similarity. BugGraph achieves that with two steps. First, we identify the compilation provenance of the binary code and compile the comparing source code with the same provenance. Second, we utilize ranking-based graph triplet-loss network to cover the less similar code. The experiments on four datasets show that BugGraph can significantly improve the performance. Further, we use BugGraph to identify 140 vulnerabilities in six commercial firmware.

REFERENCES

[1] [n.d.]. IDA Pro - Interatctive Disassembler. https://www.hex-rays.com/products/ida/.

[2] [n.d.]. Vulnerability Details: CVE-2016-2842. https://www.cvedetails.com/cve/CVE-2016-2842/.

[3] Last accessed, May 2020. Cppcheck - A tool for static C/C++ code analysis. http://cppcheck.sourceforge.net/.

[4] Last accessed, May 2020. Flawfinder - C/C++ Source Code Analyzer. https://dwheeler.com/flawfinder/.

[5] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An exact graph edit distance algorithm for solving pattern recognition problems.

[6] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.

[7] Joy Batchelor and HENRIK REIF Andersen. 2012. Bridging the product configuration gap between PLM and ERPâĂŤan automotive case study. In *19th International product development management Conference, Manchester, UK*. 17–19.

[8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007).

[9] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. springer.

[10] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 4.

[11] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[12] Z Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick McDaniel. 2019. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–30.

[13] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[14] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 18.

[15] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. 2018. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In *International Conference on Security and Privacy in Communication Systems*. Springer, 172–192.

[16] Mengnan Du, Ninghao Liu, Qingquan Song, and Xia Hu. 2018. Towards explanation of dnn-based prediction with guided feature inversion. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[17] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries.. In *NDSS*.

[18] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. In *NDSS*.

[19] Thomas Dullien and Rolf Rolles. [n.d.]. Graph-based comparison of Executable Objects (English Version). ([n. d.]).

[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23th Symposium on Network and Distributed System Security (NDSS)*.

[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[22] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*. Springer, 238–255.

[23] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. Lemna: Explaining deep learning based security applications. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*.

[24] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 135–150.

[25] Maarten Houbraken, Sofie Demeyer, Tom Michoel, Pieter Audenaert, Didier Colle, and Mario Pickavet. 2014. The Index-based Subgraph Matching Algorithm with General Symmetries (ISMAGS): exploiting symmetry for faster subgraph enumeration. *PloS one* 9, 5 (2014).

[26] Md Rakibul Islam, Minhaz F Zibran, and Aayush Nagpal. 2017. 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*. IEEE Press, 20–29.

[27] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. 2005. Malware phylogeny generation using permutations of code. *Journal in Computer Virology* 1, 1-2 (2005), 13–23.

[28] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 329–338.

[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[30] William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. 2017. Semi-automated discovery of server-based information oversharing vulnerabilities in Android applications. In *Proceedings of ISSTA*.

[31] Anusha Lalitha, Osman Cihan Kilinc, Tara Javidi, and Farinaz Koushanfar. 2019. Peer-to-peer federated learning on graphs. *arXiv preprint arXiv:1901.11173* (2019).

[32] Klas Leino, Shayak Sen, Anupam Datta, Matt Fredrikson, and Linyi Li. [n.d.]. Influence-directed explanations for deep convolutional networks. In *2018 IEEE International Test Conference (ITC)*. IEEE, 1–8.

[33] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[34] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *International Conference on Information Security and Cryptology*. Springer, 92–109.

[35] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security and Privacy Conference*. Springer, 416–430.

[36] Hiroaki Murakami, Yoshiki Higo, and Shinji Kusumoto. 2014. A dataset of clone references with gaps. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 412–415.

[37] Ginger Myles and Christian Collberg. 2005. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, 314–318.

[38] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.

[39] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*.

[40] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. 2011. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 100–110.

[41] Nathan E Rosenblum, Barton P Miller, and Xiaojin Zhu. 2010. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*.

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

[43] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[44] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[45] Laurens Van Der Maaten. 2014. Accelerating t-SNE using tree-based algorithms. *Journal of machine learning research* 15, 1 (2014), 3221–3245.

[46] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. *arXiv preprint arXiv:1710.10903* (2017).

[47] Binghui Wang, Jinyuan Jia, and Neil Zhenqiang Gong. 2018. Graph-based security and privacy analytics via collective classification with joint weight learning and propagation. *arXiv preprint arXiv:1812.01661* (2018).

[48] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[49] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A large scale investigation of obfuscation use in

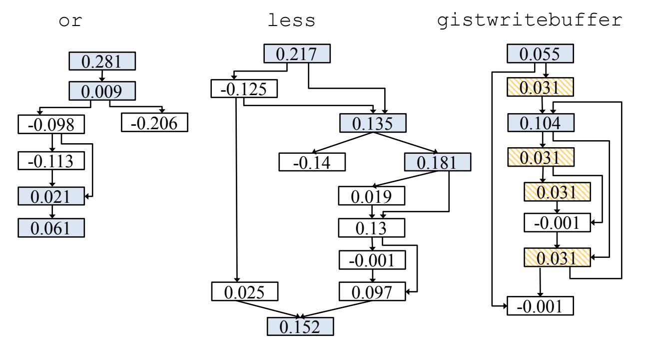


Figure 14: Node embedding similarity to its graph embedding (important nodes are colored).

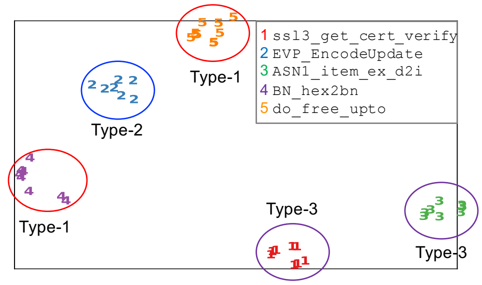


Figure 15: Visualization of ACFG embeddings generated from BugGraph using t-SNE. Each number represents a function, and each point represents a specifically compiled binary of that function.

google play. In *Proceedings of Annual Computer Security Applications Conference.*

[50] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. 2019. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596* (2019).

[51] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of CCS.*

[52] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 590–604.

[53] Jiaqi Yan, Guanhua Yan, and Dong Jin. 2019. Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network. In *Proceedings of DSN.*

[54] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNN Explainer: A Tool for Post-hoc Explanation of Graph Neural Networks. *arXiv preprint arXiv:1903.03894* (2019).

[55] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *27th {USENIX} Security Symposium ({USENIX} Security 18).* 887–902.

[56] Min Zheng, Mingshen Sun, and John CS Lui. 2014. DroidRay: a security evaluation system for customized android firmwares. In *Proceedings of the 9th ACM symposium on Information, computer and communications security.* 471–482.

[57] Fei Zuo, Xiaopeng Li, Zhexin Zhang, Patrick Young, Lannan Luo, and Qiang Zeng. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *NDSS.*

A APPENDIX

In this Appendix, we understand the similarity ranking of BugGraph, perform the parameter sensitivity study, and analyze the runtime.

A.1 Understanding Similarity Ranking

In this subsection, we will use two examples to explain our graph models. In the first example in Figure 14, we show the node importance for the ACFGs in Figure 9. The node importance is represented by the similarity between the node embedding and the graph embedding, which is also used in a prior work on GNN explanation [54]. As the embeddings for both graph and node are high dimensional which are difficult to visualize and interpret, we choose to present the cosine similarity of each node embedding to the graph embedding. In Figure 14, we highlight the top-4 similar nodes in the functions. Clearly, for the first two functions, the starting node, ending node, and two middle nodes have high similarity. In contrast, a number of nodes in the third function share the equal similarity and the ending node is negative. In summary, the learned graph embeddings are able to capture the essential nodes, attributes, and topological structure of ACFGs.

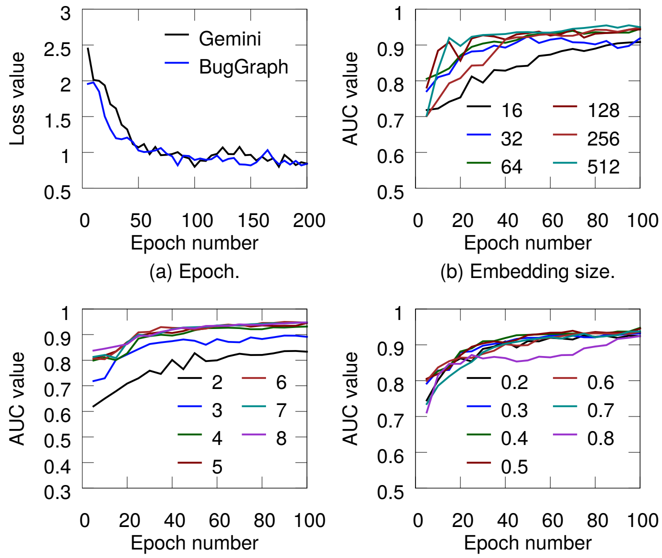
In the second example, we use five vulnerable functions from OpenSSL binaries. Particularly, the function `EVP_EncodeUpdate` has type-2 similar code, `ssl3_get_cert_verify` and `ASN1_item_ex_d2i` have type-3 similar code, and the rest two have type-1 similar code. For each function, we get multiple ACFGs under different compilation configurations. For this demonstration, because we intend to show the effectiveness of our triplet loss-based graph neural network, we do not use the provenance identification. Given the embedding generated by our GTN for each binary ACFG, t-SNE [45] is used to visualize the high-dimension embeddings into two dimensional space as shown in Figure 15. One can see that with the help of GTN, BugGraph is able to put the similar ACFGs closer no matter they are sharing type-2 or type-3 code similarity or compiled with different provenances. Note that there has been a lot of interests in explainable machine learning techniques [16, 23, 32, 44, 54], which we plan to explore as part of future work.

A.2 Parameter Sensitivity Study

In this test, we use the 600 binaries from SNNS-4.2 and PostgreSQL-7.2 (Dataset II), which has 493, 841 functions. We split this dataset by selecting 80% functions for training, 10% for validation, and the remaining 10% for testing. We guarantee that no two binary ACFGs compiled from the same source code exist in the same group. We use the default parameter values as discussed in Section 6 and test the validation dataset every 5 epochs. In the following, we will tune one parameter at a time and keep the others as default.

Number of epochs shows the convergence rate of the learning process. We use the average loss value on the validation dataset to study the impact of different epochs as shown in Figure 16(a). We test 200 epochs and show the results for both BugGraph and Gemini. One can see that, at around 50 epochs, both models start to converge. After 80 epochs, both are reaching the relatively stable status.

Embedding size represents the size of the intermediate node embedding and the final graph embedding. We test various embedding sizes starting from 16 to 512. The larger the embedding size, the stronger the model's learning ability, while the computation cost will increase accordingly. Figure 16(b) presents the AUC values on the validation dataset of various embedding sizes. The smaller embeddings (16, 32) take longer time to converge and get low AUC value even at epoch 100. The larger embeddings (256, 512) are able to converge with fewer epochs and achieve better accuracy. One



(a) Epoch. (b) Embedding size. (c) Iteration number. (d) Margin value.

Figure 16: Parameter sensitivity study.

can see the default size of 512 gets the best AUC value for most epochs.

Number of iterations in generating graph embedding also affects the convergence, learning ability, and computation cost. We study various iterations starting from 2 to 8 as shown in Figure 16(c). With 2 and 3 iterations, the AUC values are rather low compared with the other settings. The models with larger number of iterations (7, 8) converge quickly but the AUC values are close with the iteration number 4, 5, 6. Thus, we select 5 as the default iteration number.

Margin value Δ denotes the distance between the positive and negative pairs in the triplet loss. Figure 16(d) shows that the larger margin (0.7, 0.8) requires more time to converge because they are tuning the learned parameters more frequently than others. The small ones (0.2, 0.3) produce smaller AUC values. As a result, the default margin value is set to 0.5.

A.3 Runtime Analysis

In this subsection, we report the runtime of BugGraph in terms of training and inference. We omit the binary disassembly time as all the methods share the same process which is done by the third-party tools. The training is an offline process which only needs to be done once. During training, the triplet loss-based GNN costs 485 seconds per epoch, comparing to 450 seconds per epoch for Gemini. The longer time comes from our GNN which needs to learn the attention coefficient on each edge, taking more time than the graph embedding method used in Gemini. The training for the provenance identifier takes 130 seconds per epoch. In total, the training time of BugGraph is similar to Gemini, which is up to 300× faster than Genius because the latter needs to compute the graph similarity of any two graphs in the codebook.

During inference, BugGraph again has a similar runtime in disassembly and ACFG construction as Gemini, adding small overhead (5%) from the provenance identification. The code similarity computation is relatively close, with about 2% overhead brought by attention computation.