VESTIGE: Identifying Binary Code Provenance for Vulnerability Detection

Yuede Ji(✉), Lei Cui, and H. Howie Huang

Graph Computing Lab, George Washington University, Washington, D.C., USA
{yuedeji,leicui,howie}@gwu.edu

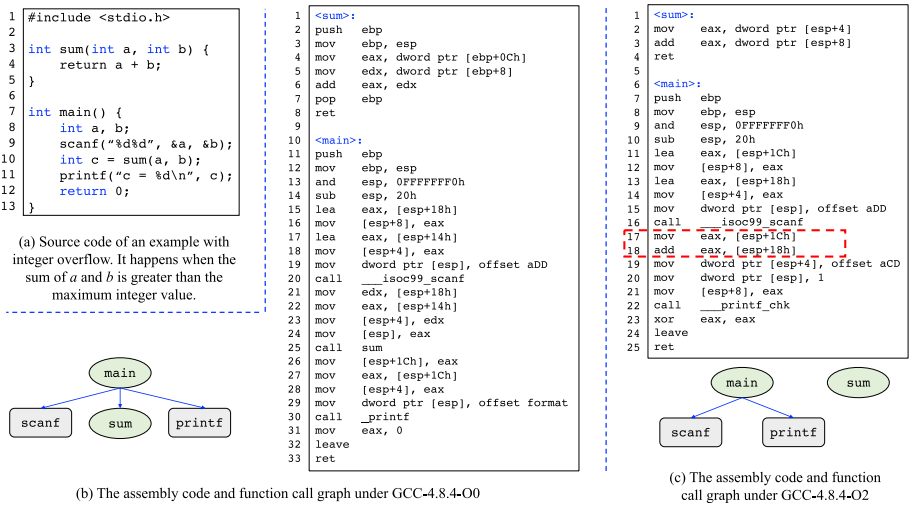
Abstract. Identifying the compilation provenance of a binary code helps to pinpoint the specific compilation tools and configurations that were used to produce the executable. Unfortunately, existing techniques are not able to accurately differentiate among closely related executables, especially those generated with minor different compiling configurations. To address this problem, we have designed a new provenance identification system, VESTIGE. We build a new representation of the binary code, i.e., attributed function call graph (AFCG), that covers three types of features: idiom features at the instruction level, graphlet features at the function level, and function call graph at the binary level. VESTIGE applies a graph neural network model on the AFCG and generates representative embeddings for provenance identification. The experiment shows that VESTIGE achieves 96% accuracy on the publicly available datasets of more than 6,000 binaries, which is significantly better than previous works. When applied for binary code vulnerability detection, VESTIGE can help to improve the top-1 hit rate of three recent code vulnerability detection methods by up to 27%.

Keywords: Compilation provenance · Code similarity · Vulnerability · Binary code · Graph neural network

1 Introduction

A binary code is generated from source code through the compilation process. The source code can be compiled to completely different binary codes, when different compilers, coupled with different configuration settings, are used. The process of identifying the compiler and configuration is referred to as the compilation provenance identification [35]. Knowing the compilation provenance is very helpful for binary code analysis, especially for malware analysis [16,21,41], code vulnerability detection [14,17,40], and code authorship identification [29,30]. In this context, compilation provenance identification aims to find out the used compiler family, compiler version, and optimization level. Note that in this paper we do not take into account the computer architecture for which the code is compiled, because it can be accurately identified by existing tools, e.g., the *file* software.

K. Sako and N. O. Tippenhauer (Eds.): ACNS 2021, LNCS 12727, pp. 287–310, 2021.
https://doi.org/10.1007/978-3-030-78375-4_12



(a) Source code of an example with integer overflow. It happens when the sum of a and b is greater than the maximum integer value.

(b) The assembly code and function call graph under GCC-4.8.4-O0

(c) The assembly code and function call graph under GCC-4.8.4-O2

Fig. 1. Code example. (a) shows an example source code with integer overflow, (b) (c) show the assembly code and function call graph by compiling the source code with compiler GCC-4.8.4 but different optimization levels O0 vs. O2.

Prior works transform this problem to a machine learning-based classification problem [35,36]. That is, they regard the compilation provenance as the label, extract features from the binary code, and leverage machine learning methods, e.g., conditional random field, to predict (classify) the provenance. The key component in this design is the feature, as one needs to mine the useful features that are able to show the difference between various compilation provenance. Two types of features have been used in prior works, that is, the normalized instruction patterns, and the control flow graph [35,36].

1.1 Motivation

However, we observe that these features focus only on the instruction and function levels, and as a result, are unable to differentiate closely related provenances. For the example source code with integer overflow (happens when the sum of a and b exceeds the maximum value of *int*) shown in Fig. 1(a), one may compile with the same compiler but with different optimization levels (O0 vs. O2). The assembly codes (disassembled with IDA-Pro [4]) are presented in Fig. 1(b) and (c), respectively. For the control flow graph (CFG) features, as there are no branch instructions in both *sum* and *main* functions, their CFGs remain the same (one node) in both cases. Similarly, for the instruction features, after normalization, e.g., unifying the register and memory address, the patterns will be the same with the minor difference in the occurred frequency. As a result, using the aforementioned features alone will unlikely to produce the correct provenance for these two binary codes.

In this paper, we have observed that a new feature, i.e., the function call graph at the binary code level, can be used to significantly improve the accuracy of provenance identification, especially for those binary codes generated with different optimization levels. As we will show in Sect. 2, the optimization level has the largest impact in binary code similarity detection, compared with compiler family and version. Figure 1 also shows the function call graphs of the binaries. With optimization level O0, the *main* function calls three functions, i.e., the *sum* function and two library functions (*scanf* and *printf*). In contrast, with optimization level O2, the *main* function only calls the two library functions as the *sum* function is inlined shown in lines 17 and 18 in Fig. 1(c). Clearly, the function call graphs will be able to help differentiate these two cases.

1.2 Contribution

To take advantage of this observation, we have designed a new code provenance identification method, VESTIGE. Given a binary, VESTIGE transforms it to a new representation, i.e., attributed function call graph, that covers code features from three levels, instruction, function, and binary. Later, VESTIGE applies an attention-based graph neural network to generate a representative embedding to predict the compilation provenance.

In summary, we make the following contributions.

- **New representation and method.** We design a new representation for binary code, i.e., attributed function call graph (AFCG). The AFCG takes the function call graph from the binary level as the graph structure. Later, we attribute each node in the AFCG as a vector with the features from both instructions and functions. Further, VESTIGE applies the attention-based graph neural network (GAT) [39] to generate more accurate embeddings by directly learning from the attributed graph. GAT can learn a representative embedding by automatically highlighting the important nodes, which are, in this case, the more representative functions for the correct compilation provenance.
- **Implementation and evaluation.** We have implemented a prototype of VESTIGE and tested it on several publicly available datasets with more than six thousand binaries. For provenance identification, VESTIGE achieves 96% accuracy for overall provenance, which significantly outperforms previous work's 90% accuracy. Particularly for the optimization level, VESTIGE achieves 99% accuracy over previous work's 92%.
- **Applying to code vulnerability detection.** We successfully apply VESTIGE as a pre-processing step to binary code similarity detection and vulnerability detection. In both cases, given an unknown binary, prior works would compare it with the *known* vulnerable code from a pre-built database [11,13,14,40,43]. Such a strategy often leads to comparing two binary codes compiled with different provenances. Instead, with VESTIGE, one can first identify the compilation provenance of the unknown binary. Then, one only needs to compare it with the known vulnerable code compiled with the same provenance. In this way, one avoids the blind comparison

and thus can improve accuracy. Particularly, for code similarity detection, we apply VESTIGE to three recent works, BGM [14], Genius [14], and Gemini [40]. VESTIGE can improve the top-1 hit rate by 27%, 13%, and 22% for BGM, Genius, and Gemini, respectively. On detecting 20 OpenSSL vulnerabilities, VESTIGE helps improve the top-1 hit rate by 16%, 19%, and 26% for BGM, Genius, and Gemini, respectively.

Paper Organization. Section 2 explains the use case of VESTIGE on binary code similarity detection. Section 3 presents the design of VESTIGE, and Sect. 4 shows experimental results. Section 5 summarizes related work. Section 6 discusses and concludes the paper.

2 Use Case: Binary Code Similarity Detection

This section studies the use case of VESTIGE in binary code similarity detection. In the following, we discuss the background, challenge of code difference, and VESTIGE solution.

In this work, we focus on the static code similarity detection methods. The dynamic methods that leverage the dynamic execution behaviors, but face the scalability challenge [12] are left for future work. It is also worthy to point out that in this work we assume a binary is compiled with one compilation provenance. Although it is possible to compile a binary with different settings, the real-world software usually uses one configuration for easy maintenance and usability [7]. Further, the binary code is assumed to be stripped, which means one can not get helper information, such as section names, debugging symbols and sections, symbol and relocation information, and compiler-generated symbols.

2.1 Background

A lot of executable binary code performing different functionalities run in the servers, mobile devices, and Internet-of-Thing (IoT) devices [9,27,33]. Unfortunately, a large number of vulnerabilities exist in these binaries and have become the major attacking vectors [23]. For example, the researchers from Independent Security Evaluators (ISE) find 124 vulnerabilities from 13 routers and network-attached storage (NAS) devices in 2019 [5]. Also, in February 2020, Cisco confirms the existence of five critical vulnerabilities that have affected tens of millions of network devices [1].

Binary code similarity detection is a commonly used method to detect vulnerabilities [11,13,14,17,25,40,43]. Given an unknown binary, such a method would compare it with the vulnerable code from a pre-built vulnerability database. If the unknown code were similar to one vulnerable code, it would be considered as a positive which might share the same vulnerability. The identified similar code will be further investigated either manually or by automatic verification methods to confirm the vulnerability. The detection of an unknown binary is regarded as the online phase. Meanwhile, these methods usually need an offline phase, which

includes preparing the known vulnerable code database and building the model, e.g., training for machine learning-based methods.

Here we illustrate the details of binary code similarity detection with a recent work Gemini [40]. First, for any function in the binary code, Gemini represents it as an attributed control flow graph (ACFG), where each node in the control flow graph is attributed as an eight-dimension feature vector, including betweenness centrality value and the number of string constants, numeric constants, transfer instructions, calls, instructions, arithmetic instructions, and offspring. Second, Gemini collects n (by default 154) vulnerable functions and extracts their ACFGs to build the known vulnerability database. Third, for an unknown binary, Gemini extracts the ACFG of each function, creates query pairs with the vulnerable functions in the vulnerability database, and computes their similarities. Assume there are m functions in the unknown binary, Gemini would create $m*n$ query pairs. Fourth, Gemini computes the similarity score of each function pair with a neural network-based embedding generation method. Particularly, Gemini applies a graph embedding network to firstly generate embeddings for the ACFGs. Later, Gemini uses the Siamese network to compute the similarity score. For each vulnerability function, Gemini would extract top-k (e.g., k equals 50) similar functions as positive. Finally, the security experts would verify the positive code to confirm the vulnerability.

2.2 Code Variance from Compilation

The same source code can be compiled to completely different binary codes, as various compilation toolchains can be used. Therefore, binary code similarity detection methods face the challenge of code variance brought by the compilation process. To figure out the impacts, we have studied a commonly used binary code representation, i.e., control flow graph (CFG), under different compilation provenances. The node in CFG denotes the basic block and the edge denotes the control flow. It is directly or indirectly used as the key data representation in the code similarity detection methods [11,14,40,43].

In this study, we compile the OpenSSL software (version 1.0.1f) with different compilation configurations. The default provenance is GCC-4.8.4 with optimization level O0 on architecture x86. Further, we compare with a higher optimization level O3, and a different compiler Clang-3.5. Figure 2 shows the cumulative distribution function of the similarity score between two CFGs of the same function but from differently compiled binaries. The similarity score is measured by the DSC similarity defined in Eq. (1), where $|A|, |B|$ denote the vertex count of CFG A and B, $|A \cap B|$ denotes the minimum vertex count between the two CFGs.

$$DSC(A, B) = \frac{2 * |A \cap B|}{(|A| + |B|)} \tag{1}$$

From Fig. 2, one can observe that both the optimization level and compiler affect the CFGs. In particular, the optimization level shows a bigger impact,

only 42% CFGs share the same size and 20% CFGs show a similarity score of less than 0.53, in which the CFGs are already quite different. The compiler also affects the CFGs but with a smaller effect. That is, 71% CFGs share the same size.

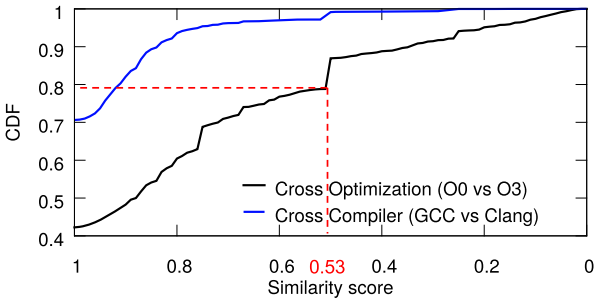


Fig. 2. The cumulative distributed function (CDF) of CFG size similarity for different optimizations and compilers.

2.3 Solution with VESTIGE

To tackle this challenge, existing binary code similarity detection methods focus on developing algorithms to eliminate these variances. For example, Genius uses clustering and locality sensitive hashing [14], Gemini uses graph embedding network and Siamese network [40], InnerEye uses recurrent neural networks [43], and Asm2Vec uses PV-DM model [11]. These methods have been shown to work well in their experiments, but the number of tested compilation provenances is limited. For example, excluding architecture variance, they tested 12, 4, 4, and 8 for Genius, Gemini, InnerEye, and Asm2vec, respectively.

Clearly, the complete coverage of compilation provenance poses a significant challenge. As of this writing, GCC has released 202 versions [3] and LLVM has 53 versions [2]. Each version has at least 4 optimization levels, which makes the number of compilation provenances for these two compilers more than one thousand, not to mention other factors that can also affect compilation provenance.

Besides capturing the code variance from different compilation provenances, the all-in-one models in these methods also need to identify the difference between the binary code compiled from different source code, which is also challenging. When the compilation provenance scales up, the performance of existing works will drop further as a result.

The key tenet of this work is that *identifying compilation provenance accurately will help produce better detection results of binary code similarity.* As shown in Fig. 3, one can leverage VESTIGE to first identify the compilation provenance of the unknown binary. With the provenance information, the following code similarity detector only needs to compute the similarity between the unknown

binary code and the known vulnerable code (from the vulnerability database) having the same compilation provenance. That means, existing code similarity detection methods only need to worry about the challenge from different source codes. Further, such a solution can scale up as VESTIGE takes over the burden of handling a large number of compilation provenances. With VESTIGE, the performance of code similarity and vulnerability detection can be significantly improved as we will show in Sect. 4.

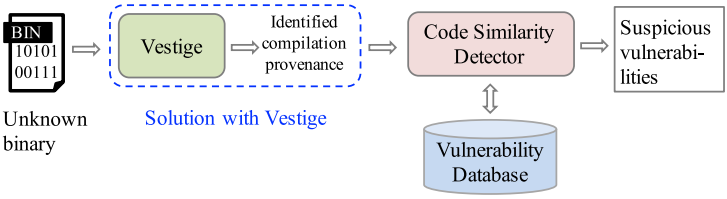


Fig. 3. Applying VESTIGE to binary code similarity detection. The dotted rectangle shows the preprocessing step with provenance identification.

3 VESTIGE Design

This section presents the design of VESTIGE, including attributed function call graph construction, provenance identification with graph neural network, and implementation.

3.1 Overview

The architecture of VESTIGE is shown in Fig. 4. Given a binary, VESTIGE first disassembles it to assembly code. Later, VESTIGE extracts three types of features, i.e., the idiom features from the instruction level, the graphlet features on the control flow graph from the function level, and the function call graph from the binary level. With all these features, we build a new representation, named attributed function call graph (AFCG). Next, VESTIGE generates the graph embedding for AFCG with an attention-based graph neural network.

During the training stage, VESTIGE tunes the graph neural network model by the downstream task, i.e., multi-graph classification. In this case, the label is the compilation provenance combined by compiler family, compiler version, and optimization level.

During the inference stage, VESTIGE will go through the same process of disassembling binary code and constructing AFCG, but output the predicted compilation provenance.

3.2 Representing Binary Code as Attributed Function Call Graph

The key to provenance identification is to find the appropriate features that can show the difference between various compilation provenances. We find there are three levels of features that can be used in combination to identify provenance. Together, we construct a new representation for the binary code as attributed function call graph (AFCG). Below, we will discuss why the features are useful for provenance identification and how we extract them.

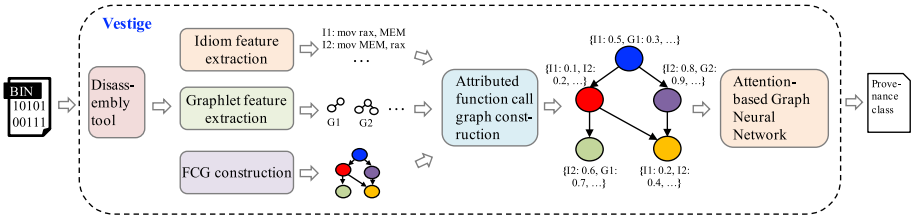


Fig. 4. The architecture of VESTIGE. Given a binary code, VESTIGE first builds the attributed function call graph by disassembling it and extracting three types of features, i.e., idiom, graphlet, and function call graph. Later, VESTIGE applies the attention-based graph neural network to predict the provenance.

1) ***Instruction level features*** are used because different compilers and configurations usually have different approaches in terms of instruction usage, register usage, instruction ordering, etc. For example, for the source code *"tbio = BIO_pop(f)"* in line 3 of Fig. 5(a), GCC-4.8.4-O0 would use the accumulator register *eax* and two *mov* instructions before calling the *BIO_pop* function. On the other hand, GCC-4.8.4-O2 would use the base address register *ebx* and just one *mov* instruction.

To identify the differences, we take the instruction patterns, known as idioms, as the instruction features for provenance identification [35,37]. These features are generated in two steps, instruction normalization, and feature extraction.

Instruction normalization will keep the essential opcode and normalize the operands to a general shape. Particularly, we will normalize the register, memory address, and other user-controlled operands, such as constant and function name. For the assembly code in Fig. 5(b), we will normalize them to the code shown in Fig. 6(a).

In the second step, we extract the unique instruction patterns and their combinations as the feature whose size is the number of covered instructions. To improve the representativeness of the patterns, we add the wildcard to represent any instruction. For the example code, the extracted features are shown in Fig. 6(a) with '|' as the instruction split symbol.

2) ***Function level features.*** Similarly, different compilation process will affect how the basic blocks form the control flow graph (CFG). As a CFG is extracted from a function, we consider such features at the function level. For the

example code in Fig. 5(a), it is one node with GCC-4.8.4-O0 shown in Fig. 5(b), while it is split into two nodes with GCC-4.8.4-O2 shown in Fig. 5(c).

Again, we extract the function features in two steps, CFG normalization and feature extraction. First, we normalize the CFG by assigning a type value to each node and edge. As each node is a basic block, its type value will be decided by the category of contained instructions, e.g., string, branch, and logic operation [24]. We classify the instructions into 14 categories and use a 14-bit integer to represent the type, where each bit denotes whether the specific instruction category exists or not. For the edges initiated by branch operations, we label them based on the different types of branch operations, e.g., *jnz, jge*. The normalized CFG of the example function is shown in Fig. 6(b).

Second, we extract different subgraphs from the normalized CFG as features. A subgraph is regarded as a subset of the connected nodes with the corresponding edges. For the example CFG in Fig. 6(b), its subgraphs included G1, G2, G3, and others. As the goal here is to mine useful subgraph patterns that can represent the compilation provenance, we set a threshold to the interested subgraph size (number of nodes) to avoid mining all the possible subgraphs, which is not scalable as it is an NP problem [15].

3) ***Binary level features.*** In this work, we especially focus on the fact that the compilers will optimize the program from the *binary level* to achieve the optimal global performance. Many compiler optimizations work on the binary level, such as, function inlining, interprocedural dead code elimination, interprocedural constant propagation, and procedure reordering [6]. Taking the *function inlining* (usually enabled in *O2* and *O3*) as an example, it heuristically selects

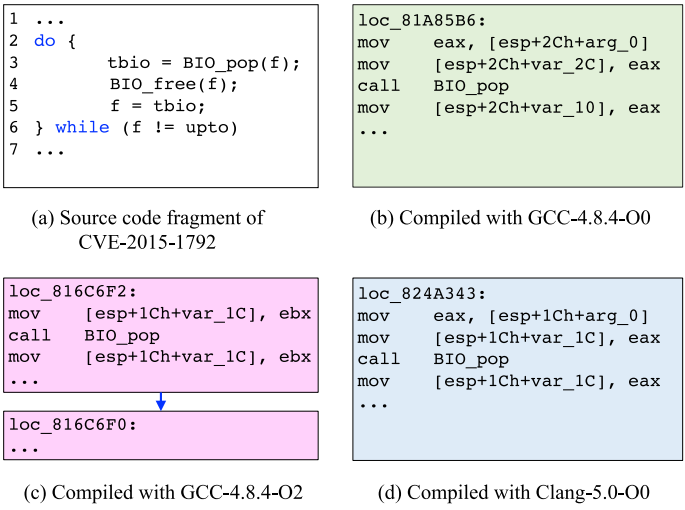


Fig. 5. A running example code and its assembly code with different compilation provenance. (a) shows the source code fragment of CVE-2015-1792, (b) (c) (d) show its assembly code with GCC-4.8.4-O0, GCC-4.8.4-O2, and Clang-5.0-O0, respectively.

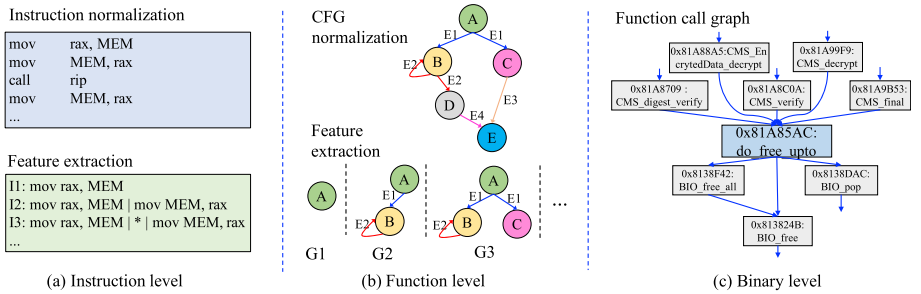


Fig. 6. The features in three levels for provenance identification, using the assembly code from Fig. 5(b) as an example.

the functions worth inlining. From the binary level, one can clearly see the difference from a feature like the call relationships between functions.

We find that the function call graph (FCG), generated in the binary level, is an effective representation to show the changes brought by different compilation provenances. In an FCG, the node denotes a function, and the edge denotes the call relationship. It is able to capture the difference from function changes in terms of number, call relationship, etc. Thus, we construct the FCG as the binary level feature.

4) ***Attributed function call graph (AFCG).*** To combine the features from three different levels together, we newly design a representation, namely attributed function call graph (AFCG). Taking the function call graph (FCG) as the core structure, it attributes each node (function in this case) as an initial feature vector.

At the training phase, we need to extract the features from a number of binaries. Since we are extracting the patterns from both instruction and CFG, the resulted number of features is massive. One can get an impression of the instruction features in Fig. 6(a). For the first two instructions, we construct 4 features in total, 2 for single instruction, 1 for the two instructions, and 1 for the two instructions with wildcard in-between. In our experiment with only 600 binaries, the number of extracted features is up to millions. This is known as the feature explosion challenge, which would cause the machine learning algorithms to take an incredibly long time to converge [26].

To solve that, we employ the feature selection technique. Particularly, we use the mutual information method to select a reasonable number of *good* features. In this case, a feature is *good* if it is important to classify different classes, which can be quantified by the mutual information between the feature and class. In the end, we select the top-k highly ranked features. Further, for the feature value, which is initialized as the frequency, we will normalize it to [0: 1] to avoid feature bias. Particularly, we divide each feature frequency to the maximum frequency value among all the binaries. To this end, we build the AFCG with a reasonable number of attributes. An example of the final AFCG is shown in Fig. 4.

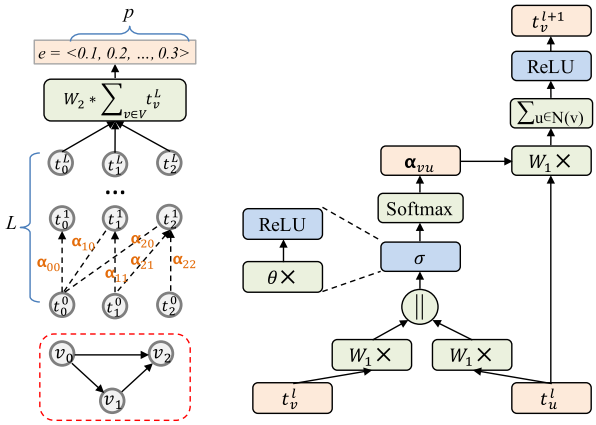


Fig. 7. Graph attention network (GAT). (a) shows the workflow, (b) explains one layer.

3.3 Identifying Provenance with Graph Neural Network

After we generate the AFCG for each binary, the problem is transformed into a multi-class graph classification problem. Such a problem is a perfect fit for the graph neural network (GNN) [8,10,22,39], which is able to learn an embedding for a graph and further tune the model based on the downstream task, i.e., multi-graph classification.

For provenance identification, we apply a recently developed graph neural network, i.e., graph attention network (GAT) [39]. Conventional graph neural networks, e.g., GCN [22] and structure2vec [10], iteratively learn a model by accumulating the neighbor embeddings based on the fixed graph structure, i.e., equally or degree-based. However, in this application, the neighbor nodes or edges on the AFCG have different impacts on the final embedding. For example, when generating the embedding of a node in the AFCG, the function with critical compilation features that can be used to identify the provenance should be more representative, and thus should be weighted more for embedding generation. Fortunately, the graph attention network (GAT) with the attention mechanism satisfies our requirement. GAT is able to automatically identify the important nodes and edges and will assign larger weights to the more important ones and smaller weights to the less important ones. We will elaborate on the details below.

GAT takes a graph g as input, iteratively computes the node embedding by attention on its neighbor nodes, and outputs a learned embedding e for the whole graph as shown in Fig. 7(a). GAT is stacked with L layers. Each layer (except the input layer) takes the node embeddings from the previous layer as input and outputs the computed node embeddings from this layer. Below, we will discuss the details of GAT.

Attention Mechanism. For the $(l+1)$-th layer, the node embedding computation for node v is shown in Fig. 7(b). For every neighbor node of v (including itself), GAT first learns an attention coefficient, and later computes the embedding for node v. We use t_v^l to represent the embedding for node v at the l-th layer which has d-dimension, and t_v^{l+1} to represent the embedding at the $(l+1)$-th layer which has d'-dimension. For every edge connecting u and v, we use α_{vu} to denote the attention coefficient, which is computed from a single-layer feedforward neural network. The formalized equation is shown in Eq. (2),

$$\alpha_{vu} = \text{softmax}\left(\sigma\left(\theta\left[W_1 t_v^l \parallel W_1 t_u^l\right]\right)\right) \tag{2}$$

where softmax$(\cdot)$ represents the standard softmax function which normalizes the input vector into a probability distribution, $\sigma(\cdot)$ represents the activation function which is the ReLU function in our setting, θ is a weight vector with $2d'$ dimensions, W_1 is a shared weight matrix with $d' \times d$ dimensions, and $\parallel$ is the concatenation operation.

Graph Convolution. After obtaining the attention coefficients from the neighbors of node v, GAT will accumulate the neighbor embedding, which is the graph convolution operation [22]. The formalized equation is shown in Eq. (3).

$$t_v^{l+1} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} W_1 t_u^l\right) \tag{3}$$

Here for each edge connecting u and v, its accumulated value will be the multiplication of the attention coefficient α_{vu}, weight matrix W_1, and embedding t_u^l of node u. Followed by another activation function, one will get the node embedding t_v^{l+1} with d'-dimension.

Graph Embedding. At the output layer, we will accumulate all the node embeddings in this graph to one embedding as in Eq. (4),

$$e = W_2\left(\sum_{v \in V} t_v^L\right) \tag{4}$$

where W_2 is a weight matrix with dimension $p \times p$ and p equals to d' of the previous layer, e is a p dimension vector. We use the cross-entropy loss function to compute the loss value between graph embedding and the provenance class. Later, it backward propagates the loss value to the previous layers and optimizes the learned model with Adam optimizer aiming at minimizing the loss value.

3.4 Implementation

VESTIGE includes two major components, AFCG constructor and graph attention network. The AFCG constructor is implemented on top of a binary analysis platform, Dyninst [38]. We set the pattern size of instruction and function level

features to be 3 since larger features are slow to generate and usually have low importance ranks. This setting already yields a million scale feature count for the evaluated dataset (discussed in Sect. 4). We set the selected number of features for instruction and function to be 1,024 (studied in Sect. 4.3).

We implement the graph attention network with TensorFlow (v1.3.0) and set the intermediate and final embedding size as 128, the number of epochs 100, the number of iterations 4, the number of heads 2, the learning rate 0.0001. The parameters are selected based on both accuracy and runtime (Sect. 4.3).

4 Experiment

In this section, we conduct extensive experiments to answer the following research questions:

- (**RQ1**) How does VESTIGE compare with other works on provenance identification?
- (**RQ2**) How do the two key designs in VESTIGE, i.e., attributed function call graph (AFCG) and graph attention network (GAT), affect the performance?
- (**RQ3**) How do various parameters impact the performance of VESTIGE, including the number of features for constructing AFCG and the hyper-parameters in graph attention network?
- (**RQ4**) How can we apply VESTIGE to binary code similarity detection as well as vulnerability detection?

4.1 Experiment Setting and Dataset

We run the experiments on an internal server, which has two Intel Xeon E5-2683 (2.00 GHz) CPUs. Each CPU has 14 cores and enables hyper-threading. It is also equipped with four Nvidia K40 GPUs, while only one is used for each run.

We use the following three datasets for the experiment.

Dataset I: Baseline dataset. We build a baseline dataset with five standard software, i.e., GNU Bash (v4.3), Diffutils (v3.3), Grep (v2.16), Tar (v1.27.1), and Wget (v1.15). We compiled them with 24 different compilation provenances, including six compilers, i.e., GCC-{4.6.4, 4.8.4, 5.4.1} and Clang-{3.3, 3.5, 5.0}, and four optimization levels (O0-O3) on x86 architecture. In the end, we are able to collect 336 binaries as some software may generate multiple binaries, e.g., 4 for Diffutils. We use this dataset for evaluating provenance identification.

Dataset II: Code similarity dataset. We build a code similarity dataset with six software, SNNS-4.2, PostgreSQL-7.2, Binutils-{2.25, 2.30}, and Coreutils-{8.21, 8.29}. They are compiled with the same 24 compilation provenances. In total, we get 6,168 binaries. This dataset is used for both provenance identification and code similarity detection.

Dataset III: Vulnerability dataset. We build a vulnerability dataset with three versions of OpenSSL (0.9.7f, 1.0.1f, and 1.0.1n) by collecting 20 CVEs.

They are also compiled with the same 24 provenances. This dataset is used for vulnerability detection.

The **evaluation metric** used for provenance identification is accuracy, which is defined as the number of correctly classified samples over the total. As the label distribution is balanced, the accuracy is a valid metric. For the experiments of code similarity and vulnerability detection, the used metric is hit rate. Among the top-k similar code, if the targeted code is included, it is a hit, otherwise, it is a miss. Since the top-k similar codes are usually manually investigated by the security analysts, a higher hit rate with a smaller k would be valued.

4.2 Accuracy of Provenance Identification

This section studies the accuracy of VESTIGE and related works on provenance identification, which answers research questions **RQ1** and **RQ2**. This experiment uses the 6,504 binaries from the baseline (dataset I) and code similarity dataset (dataset II). We perform 10-fold cross-validation. For each binary, we guarantee that all of its 24 provenance varieties are split into the same group so that we can justify the generalization of the trained model. In this experiment, all the methods use 1,024 instruction, and function level features.

We compare with two implementations, i.e., a recent work Origin [35] and a baseline of VESTIGE with a different graph neural network model, structure2vec (S2V) [10]. We get the source code of Origin from the Dyninst group. Both implementations are configured with the parameters leading to their best performance.

Table 1. Accuracy of binary code provenance identification (The best are highlighted).

	Origin	VESTIGE-S2V	VESTIGE-GAT
Optimization level	92.2%	98.7%	**99.0%**
Compiler version	96.8%	95.5%	**97.9%**
Compiler family	**99.5%**	**99.5%**	**99.5%**
Overall accuracy	90.2%	93.3%	**96.1%**

Table 1 presents the accuracy for overall provenance, and specific compilation configurations, i.e., optimization level, compiler version, and compiler family. Each case is studied independently while keeping the other two unchanged. We can get three consistent observations. First, VESTIGE outperforms other works on provenance identification for the overall and specific provenance (**RQ1**). For the overall provenance, VESTIGE with GAT is able to achieve 96.1% accuracy over Origin's 90.2%. For the specific provenances, VESTIGE achieves 99% accuracy over Origin's 92.2% for optimization levels. For compiler versions, VESTIGE achieves 97.9% accuracy over Origin's 96.8%. For compiler family, both methods achieve a rather high accuracy at 99.5%.

Second, GAT is effective for provenance identification (**RQ2**). With the same input, VESTIGE with GAT outperforms VESTIGE with S2V for both overall and specific compilation provenances as shown in Table 1. The improvement demonstrates the effectiveness of the attention mechanism on capturing the important nodes and features towards the correct compilation provenance.

Third, AFCG is an efficient representation for the binary code towards compilation provenance identification, especially for optimization level (**RQ2**). We intend to compare two implementations with the only difference in AFCG, while the machine learning method in Origin can not take AFCG as input and the graph neural network can not sustain the input of Origin. From both Table 1 and Fig. 8, we can see the effectiveness of AFCG. From Table 1, one can see VESTIGE-S2V gets 98.7% accuracy over Origin's 92.2% for optimization level.

Further, Fig. 8 shows the accuracy changes for overall and specific compilation provenance with a different number of instruction and function features. Interestingly, with only 16 features, VESTIGE can achieve 98% accuracy for optimization level, up to 31% higher than Origin. This clearly shows the impact of binary-level features for provenance identification. AFCG is able to identify optimization level differences since many optimizations work on the binary level. Origin crashes when adding more features beyond 1,024 due to the large intermediate data size crashes the used tool, CRFsuite [31].

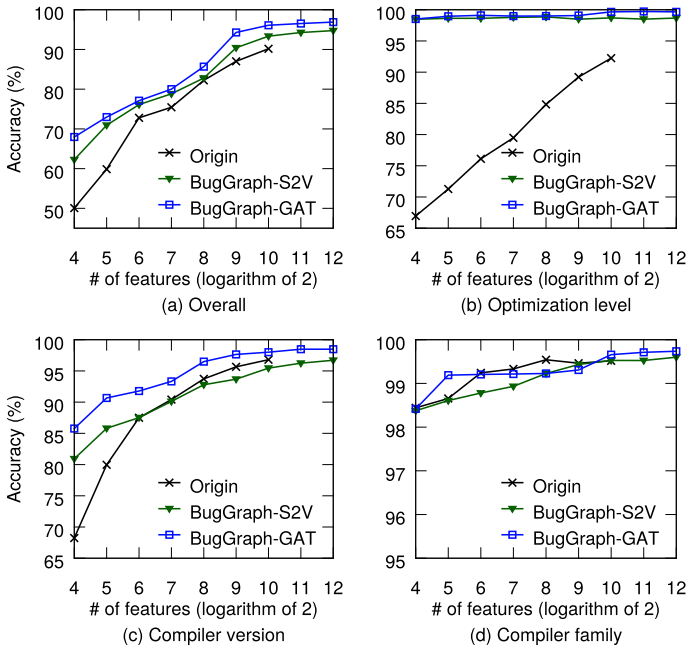


Fig. 8. Accuracy of binary provenance identification with different number of features for (a) overall, (b) optimization level, (c) compiler version, and (d) compiler family.

4.3 Sensitivity Study

This section conducts sensitivity study for the feature count in AFCG and the hyper-parameters in GAT (**RQ3**). We use the same dataset from the previous experiment and also perform 10-fold cross validation.

1) ***Feature count in AFCG*** denotes the total number of instruction and function level features. This is the major parameter for constructing AFCG, as only a small number (thousand scale) of the total features (million scale) will be selected. We perform an extensive study on this parameter by selecting different number of features. Particularly, we test the provenance accuracy of overall, optimization level, compiler version, and compiler family for the number of features from 16 to 4,096 with the power of two. The results are shown in Fig. 8.

We can get two consistent observations in this experiment. First, VESTIGE and the two compared works mostly converge with around 1,024 features. For the overall accuracy, VESTIGE-GAT achieves 96.1% with 1,024 features. With 4,096 features, the accuracy only improves a little bit 0.8%. Similar observation can be concluded on the accuracy of VESTIGE-S2V. To this end, we can conclude that 1,024 instruction and function level features are sufficient for provenance identification.

Second, the binary level feature can help to effectively identify the changes from different provenance, especially the optimization level, which has been studied in the previous experiment. For the compiler version, VESTIGE is able to get 86% with 16 initial features, comparing with VESTIGE-S2V's 81% and Origin's 68%. Interestingly, starting from 128 features, Origin performs better than VESTIGE-S2V-based method. We believe this is because the S2V model is not able to emphasize the nodes relating to the correct provenance since it equally weighs the neighbor nodes. For compiler family, it is relatively easy to predict as all the methods are able to achieve high accuracy, i.e., over 98% from 16 features.

2) ***Hyper-parameters in GAT*** are studied in this experiment. Figure 9 presents the accuracy of VESTIGE with different GAT hyper-parameters, including number of epochs, embedding size, iteration count, and head count. For each parameter test, we keep the others as default (discussed in Sect. 3.4).

Figure 9(a) presents the overall and specific provenance accuracy against the number of epochs. We run the test dataset every 10 training epochs. With only 20 epochs, VESTIGE already reaches a stable state, where the overall accuracy is around 95%, both optimization level and compiler family are close to 100% accuracy, and the compiler version is above 95%. In this experiment, we are training with a large number of binaries, i.e., 5,854, each epoch takes about 10.3 min. That means, one is able to train a usable VESTIGE model within 206 min.

Figure 9(b) presents the accuracy against different embedding size. We test five embedding sizes, i.e., 32, 64, 128, 256, and 512. One can see that the optimization level and compiler family achieve high accuracy regardless of the embedding size. However, the overall accuracy and compiler version increase to a stable state from embedding size 128. As larger embedding sizes take longer time to

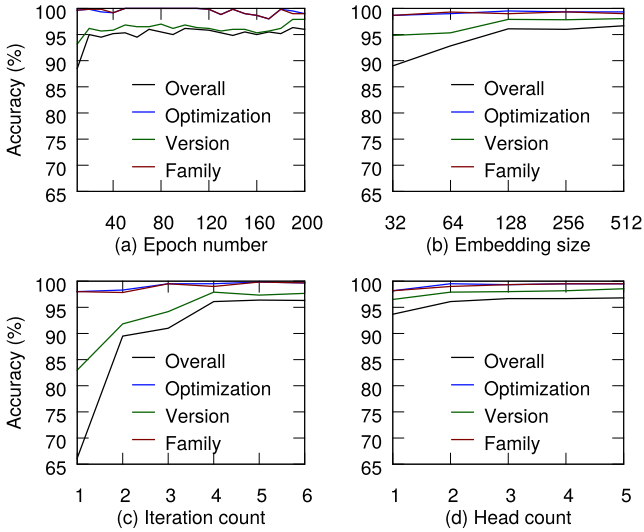


Fig. 9. Sensitivity study of hyper-parameters in GAT, including (a) epoch number, (b) embedding size, (c) iteration count, and (d) head count.

train, e.g., embedding size 512 costs 26% more time than size 128 per epoch, we select 128 as the embedding size.

Figure 9(c) shows the accuracy against different number of iterations. We test six iteration counts from 1 to 6. The optimization level and compiler family achieve high accuracy from 3 iterations, while the overall and compiler version achieve high accuracy from 4 iterations. Although the iteration count does not significantly affect the runtime, training with 6 iterations still costs 3% more time than 4 iterations per epoch. Therefore, we set the iteration count to be 4.

Figure 9(d) shows the accuracy against different number of heads from 1 to 5. One can observe that starting from 2 heads, the model achieves high accuracy for the overall provenance as well as each specific provenance. For the runtime, training with 5 heads would incur 11% more time per epoch. Therefore, we set the head count to 2.

4.4 Case Study: Code Similarity Detection

This section applies VESTIGE to binary code similarity detection, and later vulnerability detection (**RQ4**). Particularly, we apply three recent code similarity detection methods, i.e., Gemini [40], Genius [14], and BGM [14]. They convert each binary function as an attributed control flow graph (ACFG), which is presented in §2. We illustrate their details in the following.

– *Bipartite graph matching (BGM)* is a baseline method to evaluate the pairwise graph-based matching approaches [14,40]. BGM regards the similarity score

of two binary functions as the graph edit distance-based similarity between the two ACFGs. We get its source code from the authors of Genius [14].
- *Genius* is the first work of using graph embedding for binary code similarity detection [14]. Since each function is represented as an ACFG, Genius uses graph edit distance to compute the similarity of two functions. Later, it applies the *bag-of-words* method to create a high-level embedding for each function. During online searching, it uses semantic hashing on the embeddings to quickly get the similar code. We get part of the source code from the authors and reimplement the rest.
- *Gemini* presents the first work of using a graph neural network to generate embeddings for binary code similarity detection [40]. It uses the Siamese network to supervise the embedding generation. The Siamese network takes two embeddings as input with the label as either $+1$ for similar and -1 for different, computes the loss value, and back propagates it to embedding generation. We get its source code from the authors.

1) Case #1: Code Similarity Detection.

The code similarity dataset (Dataset II) is used for this experiment. BGM does not need training, but it needs tuning the cost weight for the eight attributes in ACFG. We use the default values from [40] as they get them through large scale testing. Both Genius and Gemini need the training to be able to identify similar code. Realizing the binaries from the same software may share similar code, we split the dataset into training and testing from the software level. That is, we use the 600 binaries from software SNNS and PostgreSQL as training dataset, and the 5,568 binaries from Binutils-{2.25, 2.30} and Coreutils-{8.21, 8.29} as testing dataset. VESTIGE uses the same split for the training and testing dataset. Such dataset splitting would show the generability of both VESTIGE and code similarity detection methods.

Table 2. Top-1 and top-5 hit rate for the original methods and new solutions with VESTIGE on binary code similarity detection.

	Top-1		Top-5	
	Original	+ VESTIGE	Original	+ VESTIGE
BGM	29%	56%	45%	89%
Genius	51%	64%	69%	91%
Gemini	66%	87%	77%	94%

During testing, we randomly select 1,000 different query functions from the testing dataset. For each query function, we will search the targeted binaries, which are known to have matches to the query function. In the end, we compute the average hit rate of the 1,000 queries under different top-k values.

To integrate with the code similarity methods, during the online phase, we first use VESTIGE to figure out the compilation provenance of the query binary,

and later apply the code similarity methods to compute function similarities with the predicted provenance. In this case, VESTIGE gets 82% accuracy for the whole compilation provenance, and 100%, 96% and 84% accuracy for compiler family, compiler version, and optimization level, respectively. Although the performance drops compared with cross validation result from Sect. 4.2, it is reasonable as the training and testing are on a completely different dataset and the size of the testing dataset is much larger than the training. In fact, this shows the generability of our method in a practical scenario.

Table 2 shows the hit rate of top-1 and top-5 for the three code similarity detection methods without and with our provenance identifier. We can observe that provenance identification is able to significantly improve the performance of all the works. Particularly, for the top-1 hit rate, the original code similarity detection methods, BGM, Genius, and Gemini get 29%, 51%, and 66%, respectively. VESTIGE is able to improve the hit rate by 27%, 13%, and 22%, for BGM, Genius, and Gemini, respectively. Top-1 hit rate is most important because the security analysts would start the manual investigation from the first one. Further, for the top-5 hit rate, the provenance identifier is able to improve BGM, Genius, and Gemini by 44%, 22%, and 17%, respectively. Interestingly, the simple baseline method, BGM, with VESTIGE is able to reach a rather high 89% hit rate. Note that, the hit rates of prior works align with their original reports since we only pick up the most strict and meaningful top-1 and top-5 hit rates.

2) Case #2: Vulnerability Detection.

In this study, we extend the evaluation of general code similarity detection to the specific case of vulnerability detection. Particularly, we reuse the trained models of the three code similarity works and VESTIGE from the previous experiment.

We use the vulnerability dataset (Dataset III), and take the ACFGs from OpenSSL-1.0.1f as the vulnerability database. For each binary, we will query against the binaries with 24 different compilation provenances from OpenSSL-1.0.1f. We report the average results of all the binaries with different compilation provenances for that OpenSSL version. Our provenance identifier is able to get 71% accuracy for the overall provenance, and 94%, 78%, 75% for compiler family, compiler version, and optimization level, respectively.

Table 3 shows the top-1 hit rate of the three works without and with VESTIGE for the 20 CVEs. Interestingly, the provenance identification of VESTIGE is able to significantly improve the performance of the original works on code similarity detection. One can see that, the original code similarity works get 50%, 39%, 33% top-1 hit rate for Gemini, Genius, and BGM, respectively. With VESTIGE, the top-1 hit rate improves to 76%, 58%, and 49% for Gemini, Genius, and BGM, respectively.

To understand the false positives of vulnerability detection, we take a deep look at the specific cases. A false positive is likely to happen if a queried normal function shares similar ACFG with the vulnerable function. Although it is uncommon for two completely different functions to have similar ACFGs, we do observe some occurrences, e.g., CVE-2016-0705 which is a double free vul-

Table 3. The average hit rate (%) of top-1 candidates on the 20 CVEs. +P represents adding our provenance identifier VESTIGE to their methods.

CVE	Query	BGM : +P	Genius : +P	Gemini : +P
2015-0209	1.0.1f	46 : 67	50 : 71	50 : 88
2014-0195	1.0.1f	33 : 42	42 : 58	54 : 92
2016-2106	1.0.1f	58 : 63	58 : 63	63 : 83
2012-0027	1.0.1f	42 : 58	58 : 67	63 : 88
2014-3513	1.0.1f	46 : 71	50 : 83	67 : 92
2015-1791	1.0.1f	50 : 67	50 : 83	71 : 96
2015-3196	1.0.1f	42 : 67	38 : 75	58 : 92
2014-3567	1.0.1f	22 : 56	33 : 67	50 : 79
2016-0797	1.0.1n	21 : 41	25 : 41	38 : 83
2016-2180	1.0.1n	25 : 42	29 : 83	46 : 95
2016-2105	0.9.7f	58 : 67	47 : 58	58 : 83
2016-2176	1.0.1n	38 : 42	38 : 46	50 : 67
2016-2109	0.9.7f	10 : 29	30 : 38	40 : 54
2015-3195	0.9.7f	25 : 42	50 : 63	58 : 83
2016-2182	0.9.7f	25 : 42	33 : 50	46 : 58
2016-2178	0.9.7f	13 : 25	19 : 42	25 : 63
2015-0292	0.9.7f	37 : 42	46 : 54	50 : 67
2016-2105	0.9.7f	58 : 63	58 : 63	67 : 71
2016-2842	1.0.1n	5 : 25	10 : 33	19 : 50
2016-0705	1.0.1n	13 : 21	17 : 21	19 : 42
Average	–	33 : 49	39 : 58	50 : 76

nerability in function dsa_priv_decode. In this case, another function, named d2i_ECPrivateKey, is ranked as top-1 for some queries, which encodes and decodes functions for saving and reading a key data structure. We have identified two factors that explain this false positive. First, although the source codes of the two functions are different, they share some similarities. Both of them are related to private key decode, which results in them both invoking a number of similar private key related functions. And the two functions have similar coding characteristics, i.e., both have many conditional branches (including if and goto), and neither has loop operations. Thus, their structures and code features are similar. Second, from the binary level, these two functions share quite similar graph structures. For example, for OpenSSL version 1.0.1f compiled with Clang-3.3-O0, the CFG of dsa_priv_decode has 44 nodes and 61 edges, while d2i_ECPrivateKey has 40 nodes and 58 edges. In short, in this case, the best traditional solution, i.e., Gemini, shows the top-1 hit rate of 50% across 24 varieties. However, with VESTIGE, the hit rate improves to 76%.

5 Related Work

The first work on binary code compilation provenance identification is done by Rosenblum *et al.* [36]. They extract the instruction level features, i.e., idioms, and train a provenance identification model with the linear conditional random fields (CRF) method. Further, they design Origin by adding another type of feature, i.e., the graphlet features from function level [35]. They still use the same linear CRF model. Although there are two more proposed works, Origin still achieves the best performance [28]. BinComp builds a three-layer provenance identification model [34]. The first layer learns the code transformation rules with supervised learning. The second layer extracts the statistical features and labels compiler-related functions. The third layer identifies the compiler version and optimization level from the semantic features. BinComp relies on compiler helper information, which is affected by a complete strip. Massarelli *et al.* design a graph embedding neural network for provenance identification [28]. They build an attributed control flow graph by representing each basic block as an embedding with natural language processing (NLP) models. Using NLP model is promising, but they also miss the important binary level features. o-glassesX [32] identifies the compilation provenance from a short code fragment using a deep learning model with attention mechanism and convolutional neural network. However, the compiler version and optimization level are not well differentiated. For example, the optimization level is only classified as either low (O0) or high (O3). We only compared with Origin and a variant of VESTIGE for two reasons. First, we were not able to find the source code of some other works, e.g., BinComp [34]. Second, even if we got the source code, e.g., o-glassesX [32] (we really appreciate their efforts of releasing them), we were not able to run them on our dataset due to failures on configuring the required disassembly tools.

6 Discussion and Conclusion

The interpretation of machine learning methods, especially neural networks, is an open challenge. VESTIGE uses one graph-based neural network, i.e., graph attention network. We have tried to interpret VESTIGE from the perspective of extracting useful features towards code provenance identification, i.e., the three-level features investigated in Sect. 3. Recently, we see several interesting methods for graph neural network explanation [42]. In the future, we would try to explain VESTIGE with such methods. Though VESTIGE uses a graph neural network, it takes reasonable time for training and inference. We also see some interesting works on accelerating the computation of graph algorithms [18–20], which we would like to leverage in the future to further improve the runtime performance.

In this work, we designed VESTIGE, a binary code provenance identification framework with a graph neural network. VESTIGE designs a new representation of binary code, i.e., attributed function call graph (AFCG) and applies an attention-based graph neural network, graph attention network. We have tested VESTIGE on several publicly available datasets with more than six thousand binaries. VESTIGE outperforms state-of-the-art by 6% for overall provenance.

Acknowledgment. The authors would like to thank the anonymous reviewers from ACNS'21 for their help in improving this paper. We would also like to express our grateful thanks to the authors of Genius, Gemini, and Origin (including Xiaozhu Meng) for sharing the source code and dataset with us. Lei Cui participated in this work while working as a postdoctoral researcher at the George Washington University from June 2017 to July 2018. 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. 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, or the U.S. Government.

References

1. Cisco confirms 5 serious security threats to 'tens of millions' of network devices, February 2020. https://www.forbes.com/sites/daveywinder/2020/02/05/cisco-confirms-5-serious-security-threats-to-tens-of-millions-of-network-devices
2. Download LLVM releases, December 2019. https://releases.llvm.org/
3. GCC releases - GNU project, March 2020. https://gcc.gnu.org/releases.html
4. Ida pro - interactive disassembler. https://www.hex-rays.com/products/ida/
5. Researchers uncover 125 vulnerabilities across 13 routers and NAS devices (2019). https://www.helpnetsecurity.com/2019/09/17/vulnerabilities-iot-devices/
6. Using the GNU compiler collection (GCC): Optimize options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
7. Batchelor, J., Andersen, H.R.: Bridging the product configuration gap between PLM and ERP–an automotive case study. In: 19th International Product Development Management Conference (2012)
8. Bowman, B., Laprade, C., Ji, Y., Huang, H.H.: Detecting lateral movement in enterprise computer networks with unsupervised graph AI. In: Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2020)
9. Dabrowski, A., Echizen, I., Weippl, E.R.: Error-correcting codes as source for decoding ambiguity. In: 2015 IEEE Security and Privacy Workshops (2015)
10. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data. In: International Conference on Machine Learning (2016)
11. Ding, S.H., Fung, B.C., Charland, P.: Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: Proceedings of the IEEE Symposium on Security and Privacy (2019)
12. Egele, M., Woo, M., Chapman, P., Brumley, D.: Blanket execution: dynamic similarity testing for program binaries and components. In: USENIX Security (2014)
13. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovRE: efficient cross-architecture identification of bugs in binary code. In: Proceedings of NDSS (2016)
14. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of ACM CCS (2016)
15. Grochow, J.A., Kellis, M.: Network motif discovery using subgraph enumeration and symmetry-breaking. In: Speed, T., Huang, H. (eds.) RECOMB 2007. LNCS, vol. 4453, pp. 92–106. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71681-5_7

16. Ji, Y., Bowman, B., Huang, H.H.: Securing malware cognitive systems against adversarial attacks. In: International Conference on Cognitive Computing (ICCC). IEEE (2019)
17. Ji, Y., Cui, L., Huang, H.H.: BugGraph: differentiating source-binary code similarity with graph triplet-loss network. In: 16th ACM ASIA Conference on Computer and Communications Security (ASIACCS) (2021)
18. Ji, Y., Huang, H.H.: Aquila: adaptive parallel computation of graph connectivity queries. In: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC) (2020)
19. Ji, Y., Liu, H., Huang, H.H.: iSpan: parallel identification of strongly connected components with spanning trees. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 731–742. IEEE (2018)
20. Ji, Y., Liu, H., Huang, H.H.: SWARMGRAPH: analyzing large-scale in-memory graphs on GPUs. In: International Conference on High Performance Computing and Communications (HPCC). IEEE (2020)
21. Kharaz, A., Arshad, S., Mulliner, C., Robertson, W., Kirda, E.: UNVEIL: a large-scale, automated approach to detecting ransomware. In: USENIX Security (2016)
22. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
23. Kotzias, P., Bilge, L., Vervier, P.A., Caballero, J.: Mind your own business: a longitudinal study of threats and vulnerabilities in enterprises. In: NDSS (2019)
24. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006). https://doi.org/10.1007/11663812_11
25. Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W.: α Diff: cross-version binary code similarity detection with DNN. In: Proceedings of ASE (2018)
26. Liu, H., Motoda, H.: Feature selection for knowledge discovery and data mining (2012)
27. Marcantoni, F., Diamantaris, M., Ioannidis, S., Polakis, J.: A large-scale study on the risks of the HTML5 WebAPI for mobile sensor-based attacks. In: WWW (2019)
28. Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R.: Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In: Proceedings of the 2nd Workshop on Binary Analysis Research (2019)
29. Meng, X., Miller, B.P.: Binary code multi-author identification in multi-toolchain scenarios (2018)
30. Meng, X., Miller, B.P., Jun, K.-S.: Identifying multiple authors in a binary program. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) ESORICS 2017. LNCS, vol. 10493, pp. 286–304. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_16
31. Okazaki, N.: CRFsuite: a fast implementation of conditional random fields (CRFs) (2007). http://www.chokkan.org/software/crfsuite/
32. Otsubo, Y., Otsuka, A., Mimura, M., Sakaki, T., Ukegawa, H.: o-glassesX: compiler provenance recovery with attention mechanism from a short code fragment. In: Proceedings of the 3nd Workshop on Binary Analysis Research (2020)
33. Possemato, A., Lanzi, A., Chung, S.P.H., Lee, W., Fratantonio, Y.: ClickShield: are you hiding something? Towards eradicating clickjacking on android. In: Proceedings of ACM CCS (2018)

34. Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., Debbabi, M.: Bincomp: a stratified approach to compiler provenance attribution (2015)
35. Rosenblum, N., Miller, B.P., Zhu, X.: Recovering the toolchain provenance of binary code. In: Proceedings of ISSTA (2011)
36. Rosenblum, N.E., Miller, B.P., Zhu, X.: Extracting compiler provenance from program binaries. In: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (2010)
37. Rosenblum, N.E., Zhu, X., Miller, B.P., Hunt, K.: Learning to analyze binary computer code. In: AAAI, pp. 798–804 (2008)
38. Open Source: Dyninst: an application program interface (API) for runtime code generation (2016). http://www.dyninst.org
39. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)
40. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of ACM CCS (2017)
41. Xu, Z., Zhang, J., Gu, G., Lin, Z.: GoldenEye: efficiently and effectively unveiling malware's targeted environment. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 22–45. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11379-1_2
42. Ying, Z., Bourgeois, D., You, J., Zitnik, M., Leskovec, J.: GNNExplainer: generating explanations for graph neural networks. In: Proceedings of NeurIPS (2019)
43. Zuo, F., Li, X., Zhang, Z., Young, P., Luo, L., Zeng, Q.: Neural machine translation inspired binary code similarity comparison beyond function pairs. In: NDSS (2019)