applied sciences

MDPI

Article

Semantic-Based Representation Binary Clone Detection for Cross-Architectures in the Internet of Things

Zhenhao Luo, Baosheng Wang *, Yong Tang and Wei Xie

College of Computer, National University of Defense Technology, Changsha 410073, China
* Correspondence: bswang@nudt.edu.cn

Received: 8 July 2019; Accepted: 8 August 2019; Published: 10 August 2019

Abstract: Code reuse is widespread in software development as well as internet of things (IoT) devices. However, code reuse introduces many problems, e.g., software plagiarism and known vulnerabilities. Solving these problems requires extensive manual reverse analysis. Fortunately, binary clone detection can help analysts mitigate manual work by matching reusable code and known parts. However, many binary clone detection methods are not robust to various compiler optimization options and different architectures. While some clone detection methods can be applied across different architectures, they rely on manual features based on human prior knowledge to generate feature vectors for assembly functions and fail to consider the internal associations between features from a semantic perspective. To address this problem, we propose and implement a prototype GeneDiff, a semantic-based representation binary clone detection approach for cross-architectures. GeneDiff utilizes a representation model based on natural language processing (NLP) to generate high-dimensional numeric vectors for each function based on the Valgrind intermediate representation (VEX) representation. This is the first work that translates assembly instructions into an intermediate representation and uses a semantic representation model to implement clone detection for cross-architectures. GeneDiff is robust to various compiler optimization options and different architectures. Compared to approaches using symbolic execution, GeneDiff is significantly more efficient and accurate. The area under the curve (AUC) of the receiver operating characteristic (ROC) of GeneDiff reaches 92.35%, which is considerably higher than the approaches that use symbolic execution. Extensive experiments indicate that GeneDiff can detect similarity with high accuracy even when the code has been compiled with different optimization options and targeted to different architectures. We also use real-world IoT firmware across different architectures as targets, therein proving the practicality of GeneDiff in being able to detect known vulnerabilities.

Keywords: binary clone detection; Semantic representation; cross-architectures; IoT devices; real-world vulnerabilities

1. Introduction

Most software development does not start from scratch; instead, to accelerate innovation and reduce development costs, software developers introduce large amounts of reusable code into their software. Synopsys, after auditing more than 1100 commercial software packages, reported that more than 96% of audited software uses open-source code [1]. More than half of these software files include more than one reusable open-source software package [2].

Code reuse has gradually become an uncontrollable issue. Some developers opportunistically plagiarize other software code [3], possibly causing GNU General Public License (GPL) infringements. In addition, code reuse has exacerbated security issues. These issues also exist in the internet

of things (IoT) devices [4]. The reusable code in IoT devices, as in any traditional application, may contain vulnerabilities or programming bugs. Code containing vulnerabilities is widely used, which accelerates the spread of vulnerabilities and aggravates the hazards presented by these vulnerabilities. For example, Heartbleed (common vulnerabilities and exposures (CVE)-2014-0160) is a vulnerability in OpenSSL, and many IoT device source projects reuse the OpenSSL code, thereby introducing the Heartbleed vulnerability [5,6] into their IoT devices. Statistical studies have shown that 26.66 billion IoT devices will be deployed by 2019 [7], causing the issue of reusable code introducing known vulnerabilities into IoT devices to become a grim concern. More seriously, some IoT device developers, for convenience, may directly copy and paste code from Google searches. Such copied code may come from code snippets shared from various GitHub authors' projects and technical blogs, and this copied code is not maintained. This increases security threats to the software development ecosystem.

To address these issues, researchers need to analyze suspicious software code. However, it is difficult to obtain the source code from IoT devices and commercial software. Therefore, solving these problems usually requires substantial reverse-engineering work. However, reversing software is substantially more difficult than programming software. To address this problem, software clone detection methods can significantly reduce the burden of manual analysis during reverse engineering.

There are many clone detection methods for mono-architecture [3,8–12]. However, many software programs, especially IoT firmware applications, are compiled into binaries for different instruction set architectures (ISAs) depending on the intended runtime environment, which makes mono-architecture clone detection ineffective. Some cross-architecture clone detection methods utilize symbolic execution and theorem provers to evaluate the similarity of binaries [13,14]; however, such methods are too expensive to be applied to large codebases. Other clone detection methods use statistical features [4,15,16], such as the number of specific instructions and the number of basic blocks, to detect the similarity between two binaries. These approaches generally fail to consider the relationships between features. For example, *fopen* and *fclose* functions often appear in pairs.

After disassembly, a binary can be represented in some assembly language. Natural language processing (NLP) has been used to help with various language text analysis tasks, such as semantics analysis [17]. Paragraph vector-distributed memory approach (PV-DM) neural network [17] can learn vector representations for each word and each paragraph, and map words a similar meaning to a similar position in the vector space. Inspired by the PV-DM model in NLP, we found that assembly code analysis and NLP actually share numerous commonalities, including semantic extraction, classification, and similarity comparisons, which are common to both code and articles. The amount of data in NLP is usually very large, indicating high scalability. Therefore, we propose to use a semantic representation model inspired by PV-DM neural network to solve binary clone detection problems. Under this approach, instructions correspond to words, basic blocks correspond to sentences, functions correspond to paragraphs, and binary files correspond to articles.

In this paper, we propose GeneDiff, a semantic-based representation binary clone detection approach for cross-architectures. To address the differences caused by different architectures, GeneDiff converts binaries from different ISAs into a Valgrind intermediate representation (VEX) implementation representation to mitigate the differences. GeneDiff utilizes a novel semantic-based representation learning model for binaries without any prior knowledge. Inspired by the PV-DM model in NLP, GeneDiff uses a large number of binary files to train a semantic representation learning model. Then, GeneDiff utilizes the representation model to map assembly functions from different ISAs into high-dimensional numeric vectors. To match similar functions, GeneDiff uses cosine distance to evaluate the similarity between high-dimensional vectors of the functions, which makes the clone detection approach scalable. In particular, GeneDiff's matching efficiency is much higher than approaches that use symbolic execution (see Section 5.3).

The contributions of this paper are as follows:

- We propose and implement GeneDiff, a semantic-based representation binary clone detection approach for cross-architectures. This is the first work that converts the assembly instructions into a VEX intermediate representation and uses a semantic representation model to implement clone detection for cross-architectures.
- We use real-world IoT firmware across different architectures as targets, therein proving the practicality of GeneDiff in being able to detect known vulnerabilities for IoT devices. In addition, GeneDiff can reduce the burden of manual analysis during reverse engineering, which can be used in many applications, such as detecting code plagiarism or malware.
- The results of our experiments show that it is feasible to analyze the intermediate representation to achieve cross-architecture clone detection using NLP techniques. In addition, many prior systems that use assembly code as a feature for binary clone detection [14–16,18], could benefit from GeneDiff, which uses the intermediate representation to mitigate the differences between different architectures.

The remainder of this paper is organized as follows. In Section 2, we review the related literature. In Section 3, we formally define the problem of this study. In Section 4, we thoroughly describe the design of GeneDiff. Section 5 presents our experiments. Section 6 reports the results of detecting known vulnerabilities using real-world IoT firmware. In Section 7, we discuss why GeneDiff performs well and its limitations. Finally, Section 8 concludes the paper.

2. Related Work

Clone detection, or plagiarism detection, in software has long been a focus of software engineering and security research. Researchers have proposed a variety of solutions; these can generally be divided into mono-architecture detection and cross-architecture detection.

Mono-architecture Detection. Mono-architecture detection methods can be subdivided according to features. CCFinder [19] and CP-Miner [20] find equal suffix chains based on code tokens, which indicate potential code plagiarism. Jiang et al. [21] proposed Deckard to convert abstract syntax trees into numerical vectors for clone detection. However, these solutions require source code support and cannot be applied to closed-source software binaries.

For binaries, BinHunt [9] and its successor iBinHunt [10] relied on symbolic execution and a theorem prover to check the semantic equivalence between basic blocks. CoP [3] also used a theorem prover to detect the parts of cloned code. However, these methods require expensive computation and are not practical for large binary projects.

BinDiff [12] is a commonly used commercial clone detection application supported by IDA Pro that works on multiple architectures. BinDiff can achieve a many-to-many comparison of functions in binaries. However, BinDiff requires manually selected features, and it loses the instruction semantics and dependency information. Tracelet [8] decomposes assembly functions into continuous traces and uses the editing distance between two traces to measure their similarity. Other approaches use birthmarks to facilitate detection. Myles et al. [22] proposed an approach based on opcode-level k-grams as birthmarks. Jhi et al. [11] proposed a method that used value-based program characterization to address software plagiarism. Wang et al. [23] introduced system call-based birthmarks, which can detect similarities in programs that invoke sufficient numbers of system calls.

Cross-architecture detection. As the popularity of IoT device has increased, researchers have proposed clone detection methods applicable to cross-architecture binaries. BinGo [24] used randomly sampled values to compare I/O values. Pewny et al. [13] proposed a method for detecting similarity at the function level across different architectures. Their method translates binary code into the VEX [25] and uses fuzzing basic blocks to match similar parts; however, their method is too expensive to be scalable to large codebases. Although Esh [14] utilizes statistical pre-filtering to boost the theorem prover and has a significant effect, the method is still too computationally expensive.

DiscovRE [4] utilizes pre-filtering to boost the control flow graph (CFG) matching of binary functions. Genius [15] is an advanced bug search tool that uses machine learning to convert the CFGs

of functions into vectors for similarity comparison. Gemini [16] also uses deep learning to convert CFGs and other descriptive statistical features into numeric vectors for comparison.

However, the above methods based on symbolic execution are too time-consuming to be useful on large codebases, while the state-of-the-art static approaches fail to consider the relationships between features. They measure similarity based on statistical features such as the number of specific instructions, the number of basic blocks, CFGs and other features based on prior human knowledge. These approaches assume that each feature is independent and may not consider the associations between features. For example, the memcpy libc function is similar to the strcpy function. Asm2Vec [26] does not require prior knowledge; it uses an NLP model to produce numeric vectors for assembly functions in mono-architecture scenarios. Zuo et al. [18] learned using neural machine translation and proposed a solution based on a siamese network for cross-architecture binaries. Compared with Asm2Vec [26] and reference [18], GeneDiff converts assembly functions of different ISAs into VEX functions and utilizes a semantic representation model to generate representation vectors for all the functions. Another difference is that GeneDiff treats a combination of VEX instructions as a word. The VEX intermediate representation converts an individual assembly instruction into multiple VEX instructions, and generating a semantic representation for each VEX instruction is similar to interpreting the semantics of each letter in "apple", so GeneDiff regards the combinations of VEX instructions as words in NLP.

3. Problem Definition

We focus on binary-oriented clones at the function level. Two assembly functions are similar, if they share similar functional logic in their source code, even though they may be slightly different in terms of syntax.

In order to apply to IoT devices from different architectures, it requires detection approaches to detect similar parts across different architectures. Our research focuses on high-power IoT devices (including their software), which are compiled for 32-bit architectures (i.e., X86, ARM, and MIPS) and 64-bit architectures (i.e., AMD64, AArch64, and MIPS64). These IoT devices usually play essential roles in the IoT, such as routers in industrial IoT and wireless routers in Smart Home. They can provide data communication and management control for other wireless sensors and actuators in the IoT. It is worth noting that low-power IoT devices are not included in our research.

Given an assembly function, our goal is to search for its semantic similar functions from the function representation repository. We formally define the detection problem as follows:

Definition 1. *Given a target binary from one of the architectures in {X86, ARM, MIPS, AMD64, AArch64, MIPS64}, the problem is to match the top-k semantic similar functions from the repository for each assembly function of the binary.*

4. The Design of GeneDiff

4.1. Overview

GeneDiff is a novel binary-oriented clone detection approach based on semantic similarity for cross-architectures. Figure 1 provides a flowchart. GeneDiff includes two operating modes: a training mode and a query mode. In Figure 1, the grey line indicates the flow of the training mode, which is utilized to generate the function representation repository, while the orange line indicates the flow of the query mode, which is used to detect the target binaries. GeneDiff's architecture is shown in Figure 2. GeneDiff has three components: a preprocessor, a semantic representation model and a detection model.

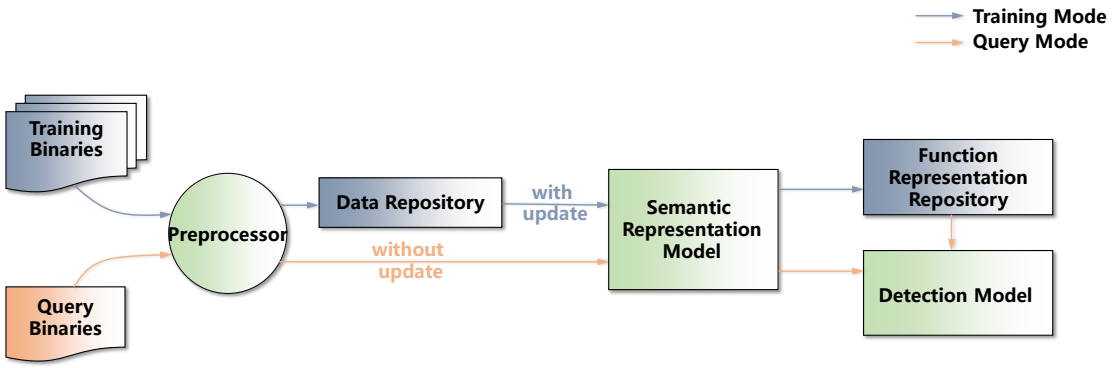


Figure 1. The flowchart of GeneDiff.

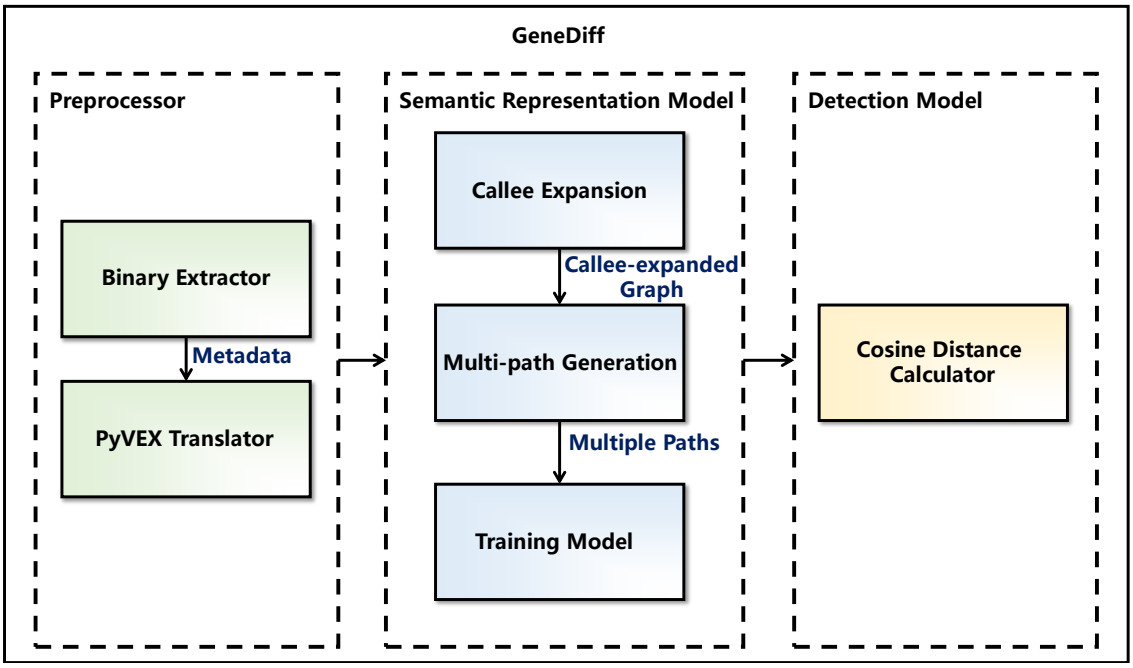


Figure 2. Architecture of GeneDiff. GeneDiff is composed of a preprocessor, a semantic representation model and a detection model.

In GeneDiff's preprocessor, GeneDiff uses a binary extractor to extract data from binaries, including architecture types, compilers, call graphs (callers and callees), function information, basic blocks and assembly code. Then, the assembly code is converted into a VEX intermediate representation by a PyVEX translator at the function level. To address the out-of-vocabulary (OOV) issue, GeneDiff also abstracts some low-frequency (possibly single-occurrence) words in the preprocessor.

In GeneDiff, the semantic representation model is used to build a representation learning model for intermediate representation functions. Before training, GeneDiff used a callee expansion to expand the CFG of functions, and utilized a multi-path generation to generate multiple paths as training sequences. In the training mode, the semantic representation model utilizes the provided training data to update the values of the word matrix without prior knowledge. For each function, the model generates a semantic-based high-dimensional vector representation that is stored in the function representation repository. In the query mode, the model does not update the word matrix; instead, it uses the word matrix constructed by the training model to generate a high-dimensional representation of each function in the target binaries.

After generating vector representations of the target functions, the detection model compares them with the vectors in the function representation repository by using cosine similarity and outputs the top-k results.

4.2. Intermediate Representation

Intermediate representation (IR) is a type of abstract code designed to provide an unambiguous representation of the binary. IR can convert binaries from different ISAs into the same abstract representation, which makes it easier to analyze cross-architecture binary code. Our work uses PyVEX [27] as an intermediate representation.

PyVEX exposes the VEX module from Valgrind [25], which provides a series of python application programming interface (API) interfaces. It is a side-effect-free representation of different ISAs (such as MIPS, ARM, and AMD). To solve the problem of the differences between different architectures, VEX possesses the following features:

Consistent registers. Although the quantity and names of registers vary according to the architecture, each type of architecture contains several general-purpose registers. VEX provides a consistent abstraction interface with memory integer offset for the registers of different architectures.

Memory operation. Different architectures also access memory differently. Some architectures support memory segmentation using special segment registers. VEX abstracts these differences as well.

Side-effects representation. Many instructions have side effects. For example, on the AMD64 architecture, an add instruction may update the condition flags (OF or CF). For instruction side effects, VEX updates only certain operations, such as the stack pointer. Other side effects, such as condition flags are not tracked.

VEX possesses four main abstract object classes: (i) *expressions* represent a calculated or constant value, (ii) *operations* describe a modification of VEX expressions, (iii) statements describe model changes in the state of the target architectures, and (iv) *temporary variables* are used to store the temporary results of *expressions*. Figure 3 shows a snippet that was compiled into AArch64 assembly and AMD64 assembly. The different ISAs and different registers show differences between the two snippets of assembly code. These differences increase the difficulty of binary similarity comparisons. After being converted into VEX code, the snippets have the same format and similar content. In Figure 3, the similar parts are highlighted by same-colour boxes. This converts the problem of binary similarity comparison on different architectures into a similar comparison problem of the same IR, which effectively reduces the analysis difficulty.

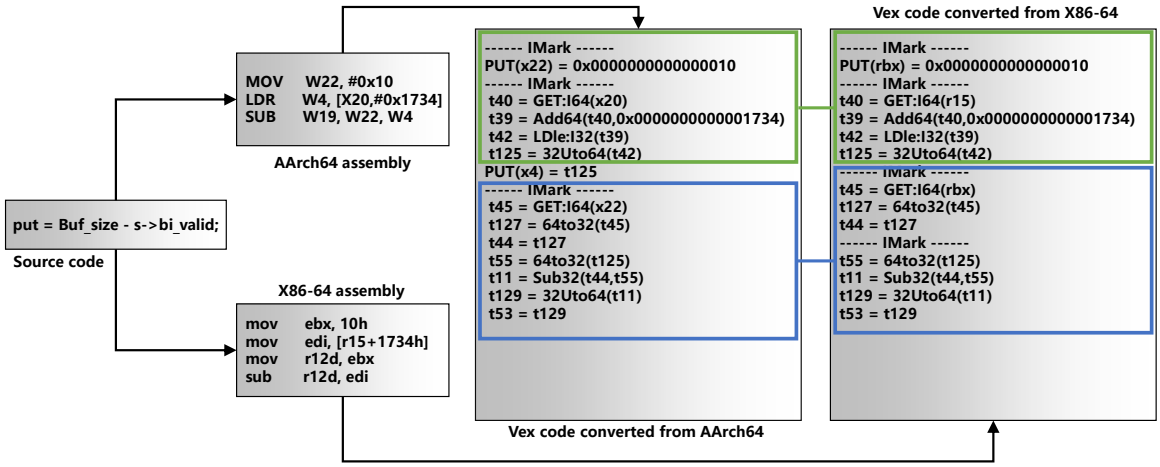


Figure 3. C source of a snippet from deflate.c in zlib and the assembly code for two architectures. Except for the registers, the others are almost identical.

4.3. Word and Paragraph Embedding

Word and paragraph embedding algorithms have been used to help with various text analysis tasks such as semantics analysis [17] and document classification [28].

The PV-DM model [17] is an extension of the original word2vec model and is designed for paragraph vectors and word vectors. A trained PV-DM model can generate paragraph semantic vectors for each paragraph according to the words in the paragraph. These paragraph semantic vectors can be used for semantic analysis and similarity analysis at the paragraph level. In binary semantic analysis inspired by NLP, instructions correspond to words, basic blocks correspond to sentences, functions correspond to paragraphs. GeneDiff is a clone detection approach at the function level, which means that GeneDiff has similar application scenarios with PV-DM model. Besides, PV-DM model is a popular NLP technique for generating paragraph vectors, so we learn from PV-DM model to build a representation learning model for intermediate representation functions.

For example, given a text paragraph that contains many sentences, reference [17] utilized PV-DM to predict a center word from the context words and a paragraph id. Figure 4 details the method. As shown in Figure 4, given a paragraph p from a document $\mathcal{D}$ and given a sentence s, with the words "A cat slept on the sofa", the model utilizes a sliding window to address the words in the sentence. The sliding window, with a size of k, starts from the beginning of the sentence and slides forward the end. The sliding window in Figure 4 contains "a", "cat", "slept", "on" and "the". The model would learn to predict the target word "slept", which is in the middle of the sliding window.

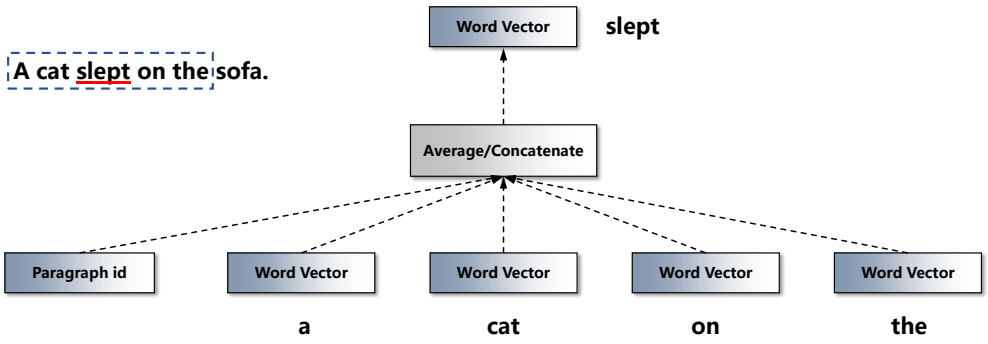


Figure 4. The model of the word and paragraph embedding.

First, the current paragraph is mapped into a high-dimensional vector based on the index of the paragraph id, and the context words are mapped to vectors with the same dimensions. Second, the model averages the above vectors to calculate the vector of the target word. Third, the model utilizes the backpropagation algorithm to update the above vectors. Its purpose is to maximize the following expression:

$$\sum_{p}^{\mathcal{D}} \sum_{s}^{p} \sum_{t=k}^{|s|-k} log(\mathbf{P}(w_t|p, w_{t-k}, \ldots, w_{t+k})), \tag{1}$$

where $|s|$ is the length of sentence s. Finally, the model obtains the appropriate vector representations of the words. However, because assembly code and intermediate representations have more complex associations than plain text, such as control flow and richer syntax, the NLP models cannot be applied directly to binary analysis. One aspect of our work is to propose a representation model that can be applied to the intermediate representation.

4.4. Preprocessor

To find the similarity between different binaries of different architectures and various IoT devices, GeneDiff utilizes the preprocessor to extract data from the binaries. First, the preprocessor utilizes

a Python script via IDA Pro [29] as a binary extractor to extract metadata about the binaries such as their architecture type, imagebase, import tables, hash value, endians and compilers. This metadata effectively helps the preprocessor to select a reasonable solution to further analyze the binaries. For example, architecture type, imagebase and endians can be used as parameters to generate intermediate representations. After confirming the metadata about the binaries, the preprocessor extracts the name, callers, callees, basic blocks, bytecode and assembly code for each function in the binaries. Table 1 shows the details of the extracted information.

Table 1. List of extracted information.

Level	Class	Type
Binary level	Architecture Types	MIPS/ARM/AArch64/X86/X86_64
	Imagebase	Integer
	Import Table	List
	Hash value	Hex
	Endian	Little-Endian/Big-Endian
	Compiler	GNU C++/Visual C++
Function level	Function name	String
	Caller	List
	Callee	List
	Basic blocks	Graph
	Bytecode	Bytes
	Assembly	List

After extracting the metadata required to analyze the binaries, the preprocessor converts the bytecode into VEX via a PyVEX translator. A large number of low-frequency words are imported during this phase. For example, large numbers of numeric constants (e.g., a number in the range $[-2^{32}, 2^{32}]$) and strings would make the word corpus too large, while these words may appear less frequently; in addition, it is difficult to accurately obtain their semantic word vectors. When a word that never appeared during training is present in the target binaries, it is called an OOV word. An attempt to generate a vector for such words will fail. Considering the OOV issue, our Python script abstracts some of the low-frequency words into tags during the conversion. The details of these abstraction rules are listed in Table 2.

Table 2. List of abstract rules.

Tag Name	Representation
<Num_Tag>	Number Constants
<Func_Tag>	Function Names
<Str_Tag>	Strings
<Other_Tag>	Others

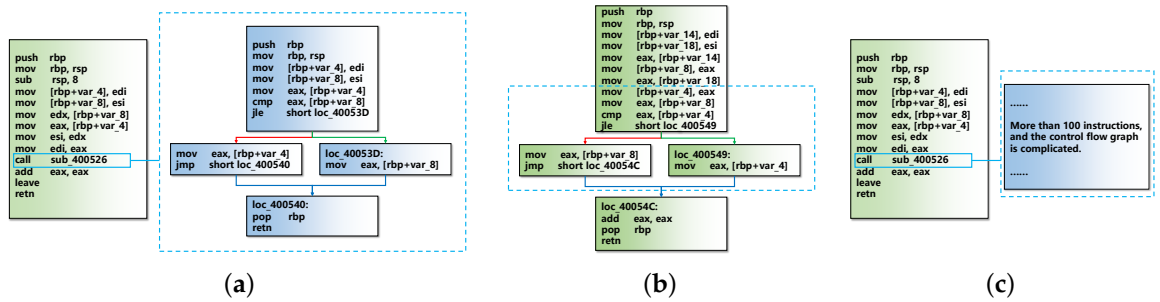
Note that the abstraction rules are applied in both the training and query modes. This approach significantly reduces the size of the word corpus and makes the trained model more practical for further research. Some architectures and some IoT devices have their unique instructions, which may impact the clone detection accuracy. For the unique instructions of different architectures and various IoT devices, GeneDiff removes their unimportant instructions and unifies the necessary instructions (e.g., registers), which can reduce the number of differences and keep the high clone detection accuracy.

4.5. Semantic Representation Model

Inspired by the PV-DM model, we proposed a novel semantic-based representation learning model for binaries. In this semantic representation model, every VEX instruction is treated as a letter, and combinations of VEX instructions are regarded as words in the NLP models, basic blocks are regarded as sentences, and functions are regarded as paragraphs. Therefore, the model for the semantic representation vector of functions can learn from the model of paragraph vectors, although they have certain differences. Compared with the extraction of paragraph vectors, the extraction of the function vector differs primarily in the following ways: (i) the calling relationships between functions are complex, (ii) the execution flows of the functions have multiple paths, and (iii) each expression consists of multiple parts. To address these differences, GeneDiff utilizes the following scheme.

Callee expansion. In software code, a function often calls another function (the *callee*) to achieve some goals. When compiling source code into binary files, the compiler may apply compiler optimization techniques to these functions. Function inlining is one such compiler optimization technique. This technique can extend callees in callers to eliminate the call overhead and improve the performance of the binaries. In binary similarity comparisons, however, such techniques may produce obstacles because they substantially modify the control flow graph of the caller function.

Figure 5 shows three examples of calling a function. Figure 5a,b perform the same function using a normal call and function inlining, respectively. When using function inlining, the CFG of the function of Figure 5b is significantly different from the CFG of the original function in Figure 5a. To perform a similarity comparison in this case, if callee expansion of the original function is not performed, the two functions may be considered as different. If we use callee expansion for all callees without distinction, then the situation in Figure 5c will be questionable. In Figure 5c, the length of the callee is much greater than the length of the original function. Thus, callee expansion of the callee (*sub*_400526) in the original function may cause the expanded original function to be similar to the callee.



(**a**) (**b**) (**c**)

Figure 5. Examples of calling a function. (**a**) An example of calling a function normally; (**b**) an example of calling a function using inlining; (**c**) an example of calling a function that possesses more than 100 instructions and whose control flow graph is complicated.

Considering the above cases, the callee expansion approach used by GeneDiff is shown in Algorithm 1. We use f to represent the original function, f_c to represent a callee function, and $\|f\|$ to indicate the instruction length of f. GeneDiff does not expand callees when they satisfy the following two conditions: (i) callees are not in the list of the data repository (Algorithm 1, Line 3), (ii) the ratio of $\|f_c\|$ to $\|f\|$ is greater than a *threshold*, and $\|f_c\|$ is greater than $numIns$. In this study, $numIns$ and *threshold* are set to 10 and 0.5 respectively.

Multi-path generation. In assembly functions, instructions are usually not executed in order, and there are usually multiple execution paths. Given a function f, we obtain a CFG G_f from the data repository of GeneDiff. Then, we randomly pick edges from G_f and expand them into sequences seq until all the basic blocks in G_f are fully covered. The pseudocode of this multi-path generation process is given in Algorithm 2.

Algorithm 1: Callee expansion

Input: A given function f.
1 $calleeList \leftarrow$ getCallees(f);
2 **for each** f_c ***in*** $calleeList$ **do**
3 **if** f_c **not in** *DataRepository.allFunctions()* **then**
4 **continue**;
5 **if** $\frac{\|f_c\|}{\|f\|} > threshold$ && $\|f_c\| > numIns$ **then**
6 **continue**;
7 Expand(f, f_c)
8 return *Paths*;

Algorithm 2: Generating multiple paths

Input: A given function f.
Output: A collection of multiple paths *Paths*.
1 $G_f \leftarrow$ DataRepository.getCFG(f);
2 $Paths \leftarrow \varnothing$;
3 $blks \leftarrow \varnothing$;
4 **for each** edg ***in*** *shuffle*(G_f) **do**
5 **if** edg **in** $blks$ **then**
6 **continue**;
7 $seq \leftarrow \varnothing$;
8 $node \leftarrow edg$;
9 **while** *prev*($node$) **do**
10 **if** *prev*($node$) $\setminus$ $blks$ **then**
11 $node \leftarrow$ rand(prev($node$) $\setminus$ $blks$);
12 $seq \cup node$;
13 **continue**;
14 $node \leftarrow$ rand(prev($node$));
15 $seq \cup node$;
16 $node \leftarrow edg$;
17 **while** *next*($node$) **do**
18 **if** *next*($node$) $\setminus$ $blks$ **then**
19 $node \leftarrow$ rand(next($node$) $\setminus$ $blks$);
20 $seq \cup node$;
21 **else**
22 $node \leftarrow$ rand(next($node$));
23 $seq \cup node$;
24 $Paths \leftarrow Paths \cup \{seq\}$;
25 $blks \leftarrow blks \cup seq$;
26 return *Paths*;

This method ensures that all the blocks of f are fully covered. Lines 7–23 of Algorithm 2 are used to expand an edg into a sequence, where prev($node$) $\setminus blks$ means $\{x | x \in$ prev($node$) $and\ x \notin blks\}$. To avoid too many blocks being repeated in *Paths*, unselected blocks are selected first when expanding edg (Algorithm 2, lines 10–13 and lines 18–20). Finally, we add all the expanded sequences to *Paths* and return it. In *Paths*, every sequence represents a potential execution trace.

Training model. After generating the multiple paths of functions, we take each path as an input, train a representation learning model based on semantics, and map instructions and functions into high-dimensional vectors. Figure 6 shows the structure of the model.

For functions in the data repository, we map each function f_c into a numeric vector $\overrightarrow{\delta_{f_c}} \in \mathbb{R}^d$. Each dimension of $\overrightarrow{\delta_{f_c}}$ is initialized to a small random value near zero. Similarly, we map each token of the data repository into a vector $\overrightarrow{v} \in \mathbb{R}^d$. Unlike references [18,26], each token in GeneDiff consists of several VEX instructions from the same assembly instruction. For example, t_1 in Figure 6 consists of three VEX instructions (after deletion). These VEX instructions come from the same IMark(0x4, 3, 0),

where an IMark represents an assembly instruction. The initialization method of $\overrightarrow{v}$ is the same as that of $\overrightarrow{\delta_{f_c}}$.

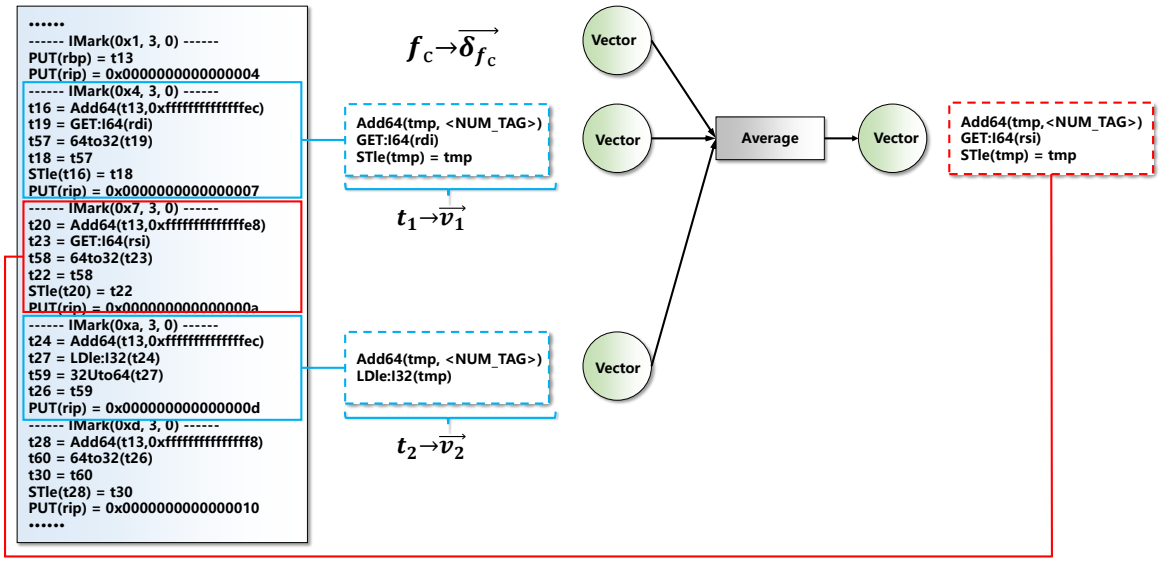


Figure 6. The structure of the proposed model.

Given a function f_c from the data repository, where $paths_{f_c}$ represents the multiple paths of f_c, for each path p_i in $paths_{f_c}$, the model walks through the basic blocks from their beginning. When traversing VEX instructions in a basic block, we treat an IMark as a token. When we calculate $\overrightarrow{v}$ of the current token t_i, the first k tokens and the last k tokens are treated as the context $\mathbb{C}$. Simultaneously, we look up the vector representation $\overrightarrow{\delta_{f_c}}$ in the previously built dictionary. We use $\phi(t_i, f_c)$ to represent the average of the neighbour tokens and $\overrightarrow{\delta_{f_c}}$. This can be formulated as follows:

$$\phi(t_i, f_c) = \frac{1}{|\mathbb{C}_{t_i}| + 1}(\overrightarrow{\delta_{f_c}} + \sum_{\overrightarrow{v}}^{\mathbb{C}_{t_i}} \overrightarrow{v}). \tag{2}$$

The proposed model attempts to maximize the following log probability:

$$\sum_{f_c}^{\mathbb{D}} \sum_{p_i}^{paths_{f_c}} \sum_{t_j}^{p_i} log(\mathbf{P}(t_j | f_c, t_{j-k}, \ldots, t_{j+k})). \tag{3}$$

The prediction task is typically performed via a softmax algorithm, which is a multi-class classifier:

$$\mathbf{P}(t_j | f_c, t_{j-k}, \ldots, t_{j+k}) = \frac{e^{y_{t_j}}}{\sum_d^{\mathbb{D}} e^{y_{t_d}}}, \tag{4}$$

where D denotes all the tokens constructed in the data repository. Each y_{t_d} is an un-normalized log probability for each output token t_d, computed as follows:

$$y = b + Uh(t_{j-k}, \ldots, t_{j+k}; \overrightarrow{v}) \tag{5}$$

where U and b are the softmax parameters. According to Equations (2), Equations (3) and (5) can be rewritten as

$$y_{\overrightarrow{v_j}} = b + Uh(\phi(t_j, f_c)^T \times \overrightarrow{v_j}) \tag{6}$$

$$\begin{aligned} \text{Equation}(3) &= \sum_{f_c}^{\mathbb{D}} \sum_{p_i}^{paths_{f_c}} \sum_{t_j}^{p_i} log(\mathbf{P}(t_j|\phi(t_i, f_c))) \\ &= \sum_{f_c}^{\mathbb{D}} \sum_{p_i}^{paths_{f_c}} \sum_{t_j}^{p_i} log(\frac{e^{y_{t_j}}}{\sum_d^{\mathbb{D}} e^{y_{t_d}}}). \end{aligned} \tag{7}$$

However, it would be computationally extremely expensive to traverse $\mathbb{D}$ for each calculation because the $|\mathbb{D}|$ is too large for the softmax classification. To minimize the computational cost, we use a k negative sampling model [30] to approximate $log(\mathbf{P}(t_j|\phi(t_i, f_c)))$.

Then, we calculate the gradients by taking the derivatives of $\overrightarrow{\delta_{f_c}}$ and $\overrightarrow{v_{t_j}}$ in Equation (7). In training mode, we use back-propagation to update $\overrightarrow{\delta_{f_c}}$ and all the involved $\overrightarrow{v}$ values based on the gradients. In query mode, we update only the $\overrightarrow{\delta_{f_{target}}}$ of the target function.

4.6. Detection Model

In query mode, after generating the vector $\overrightarrow{\delta_{f_{target}}}$ for a target function, we will detect the similarity between it and other functions in the function representation repository. Under continuous development, the function representation repository may eventually possess millions of functions. Considering search scalability, the detection model uses a cosine distance calculator to calculate the similarity. The objects of the search are function vectors with fixed dimensions; thus, we utilize pair-wise similarity to search the top-k vectors closest to $\overrightarrow{\delta_{f_{target}}}$.

5. Experiments

In this section, experimental results and analyses are presented to demonstrate the accuracy, efficiency, and sensitivity of GeneDiff. In our experiments, we utilized the GCC compiler (version 5.4.0) to generate binaries that target different ISAs for the benchmarks. As training and test data, the benchmarks contained some open-source software packages commonly used by IoT devices. All the experiments were performed on a computer running an Ubuntu 16.04 operating system and equipped with a 64-bit Intel(R) Xeon(R) E5-2650 2.00 GHz CPU, 64 GB of RAM, and no GPUs. The experimental results answered the following questions:

- Q. 1 Can the similarity of the instruction vectors generated by GeneDiff indicate the semantic similarity of assembly instructions?
- Q. 2 What is the true positive rate and area under the curve (AUC) of the receiver operating characteristic (ROC) of GeneDiff in detecting the similarity of functions from different architectures?
- Q. 3 How efficient is GeneDiff at detecting function similarity?
- Q. 4 How do the parameters impact GeneDiff's performance?

Section 5.1 answers Q. 1; it indicates that the similarity of the instruction vectors generated by GeneDiff can represent the relevance of two instructions from different ISAs. Section 5.2 answers Q. 2, therein proving that GeneDiff can detect similarity with high AUC-ROC with different compiler optimization options and targeted different architectures, being higher than the approach using symbolic execution. Section 5.3 answers Q. 3. This section verifies that GeneDiff is more efficient than the approach using symbolic execution. Section 5.4 answers Q. 4, revealing the impacts of different parameters on GeneDiff and confirming the reasonable parameter values for GeneDiff.

5.1. Analysis of Instruction Vectors

In this subsection, we show the relevance of instructions from different ISAs, AMD64 and AArch64 using GeneDiff. First, we use the training dataset to train the representation model; the dimension

of the instruction vector d is 100, and the number of iterations (*iteration*) is 50. Then, we extract the numeric vectors of the combinations of VEX instructions from the trained model and establish a map M: assembly instructions $\rightarrow$ combinations of VEX instructions $\rightarrow$ vectors. After obtaining the numeric vectors for each assembly instruction via the map M, we utilize T-SNE [31] for visualization. T-SNE can map high-dimensional vector spaces into two dimensions via nearest neighbour approximation, where a smaller geometric distance indicates a higher lexical semantic similarity. Figure 7 shows the relationship between assembly instruction vectors mapped onto a two-dimensional space. From Zoom 1 in Figure 7, the numeric vectors of the *mov* family instructions from different ISAs are mapped together in two-dimensional space. In zoom 2, the numeric vectors of the *add* family instructions from different ISAs are also grouped together in two-dimensional space.

To better illustrate the similarity in the semantic vectors of instructions of the same family from different ISAs, we obtain the cosine similarity of the instructions in Zoom 2 from Figure 7. Table 3 shows the results of calculating the cosine similarity of some *add* family instructions. The cosine similarity is approximately 1, which indicates that the two instructions are highly similar. It can be found that two instructions processed in the same bit registers have higher similarity; ADDS X0, X21, #1 and ADD EAX,1 have lower similarity because they process registers of different bits, and the ADDS instruction updates the condition flag.

Table 3. The similarity of the *add* family instructions.

Similarity	ADD RAX, 1	ADD EAX, 1	ADD RBX, [RAX+0x10]	ADD RBX, [RSP+0x158]
ADD X20, X20, #0x18	0.9992	0.9631	0.9925	0.9925
ADD X1, X1, #1	0.9992	0.9631	0.9925	0.9925
ADD X0, X0, #0x21	0.9992	0.9631	0.9925	0.9925
ADDS X0, X21, #1	0.9540	0.9271	0.9528	0.9528
ADD W19, W19, #1	0.9631	0.9992	0.9416	0.9416
ADD X2, X29, #0x60	0.9992	0.9631	0.9925	0.9925

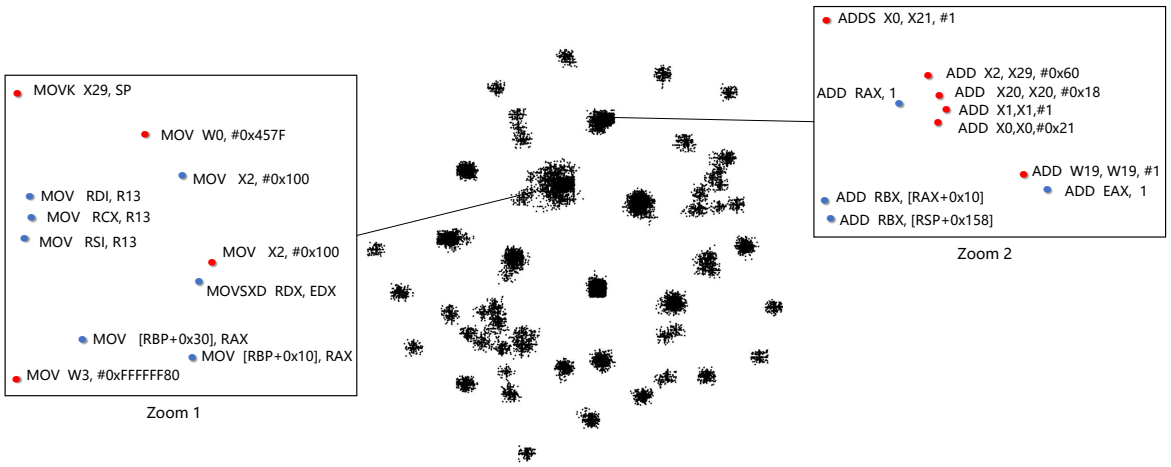


Figure 7. A visualization of assembly instructions from different instruction set architectures.

5.2. Binary Clone Detection

In this experiment, we evaluate whether GeneDiff can detect the similarity of assembly functions. First, we compare GeneDiff with some popular mono-architecture detection methods (i.e., BinDiff [12], Graphlet [32], n-gram and LSH-S [33]) on binaries targeted to the same ISA but with different compiler optimization options. Then, we evaluate GeneDiff in cross-ISAs with different compiler optimization

options (O0-O3) and compare it with the symbolic execution approach, which is widely used by other binary clone detection approaches [9,10,13].

In our experiments, the dataset included binaries that were compiled from OpenSSL-1.0.1f, curl-7.52.1, zlib-1.2.11, libpng-1.6.33, binutils-2.32 and coreutils-8.30, all of which are commonly used by high-power IoT devices. We divided the dataset into two parts: 80% of the binaries were used for training, and the remaining 20% were used for testing. In training mode, we used the binaries in the database to train the representation model to generate numerical vectors for every instruction. In testing, GeneDiff used these instruction vectors to generate semantic vectors for assembly functions and detects the similarity. The control group, which utilized the symbolic execution approach, converts test functions into VEX and uses randomly sampled values to compare I/O values for the functions, where the sampling bound is 2000.

Table 4 shows the experimental results from the same ISA. The results indicate that the average AUC-ROC of GeneDiff is higher than those of the mono-architecture detection methods.

In addition, GeneDiff can also detect binary clones in cross-ISAs. Figure 8 shows the experimental results, where the red line is GeneDiff. Figure 8a shows the ROC curve of the function similarity detection in the same ISA with different compiler optimization options. The two ROC curves are close to the top-left border, which means that GeneDiff and BinDiff achieve good accuracy when comparing binaries targeted to the same ISA. The ROC curve of GeneDiff is closer to the top-left border. Figure 8b–e shows the ROC curves of similarity detection for functions in different ISAs with the same compiler optimization options (O0–O3). Note that the ROC curve of GeneDiff in Figure 8b is very similar to the ROC curve of Figure 8a, but the curve of BinDiff is far from the top-left border; it indicates that the differences between the different ISAs have little effect on GeneDiff's similarity detection performance, but that of BinDiff is seriously affected. Figure 8f shows the ROC curves for detecting the similarities of functions from different ISAs and with different compiler optimization options: the red line is GeneDiff, the green line is BinDiff, and the orange line is the control group. The ROC curve of GeneDiff is significantly closer to the top-left border than the control group, and the AUC (=92.35%) of GeneDiff is greater than that of the control group's AUC (=74.47%) and BinDiff's AUC (=68.71%), which means that GeneDiff is better than the control group and BinDiff at detecting function clones across ISAs and across compiler optimization options. Due to the different compiler optimization options and different ISAs, which change some semantic information of the functions, the AUC of GeneDiff in Figure 8f is slightly lower than the other AUCs.

Table 4. The area under the curve (AUC) of the ROCs of different clone detection approaches for different compiler optimization options in the same ISA.

Baselines	BinDiff [12]	Graphlet [32]	n-gram	LSH-S [33]	GeneDiff
libpng	0.967	0.718	0.845	0.897	0.961
OpenSSL	0.926	0.683	0.819	0.886	0.957
zlib	0.935	0.687	0.786	0.755	0.971
curl	0.924	0.692	0.873	0.904	0.982
Avg.	0.938	0.695	0.831	0.860	0.968

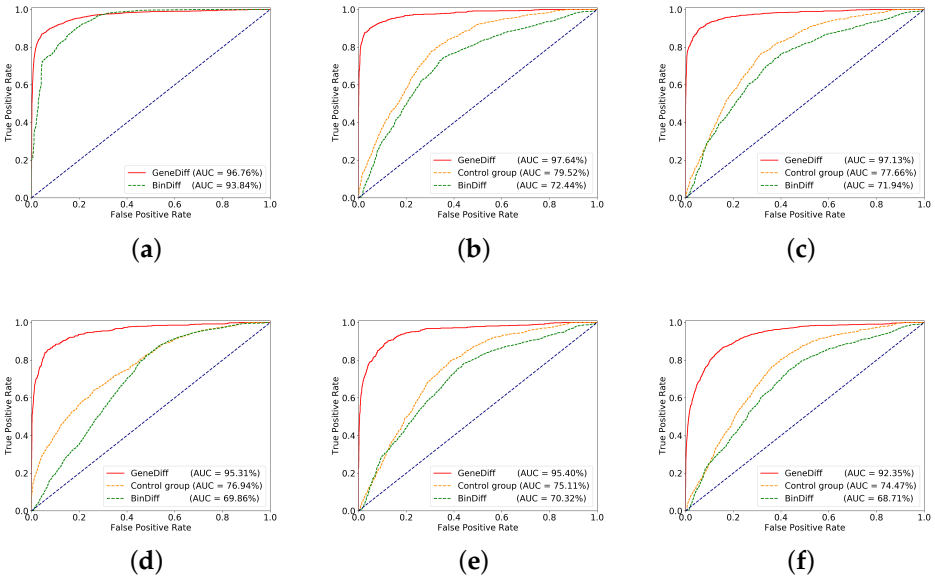


Figure 8. The receiver operating characteristic (ROC) evaluation results based on the test datasets, where the *x*-axis is the true positive rate and the *y*-axis is the false positive rate. (**a**) Same instruction set architecture (ISA), cross opts; (**b**) cross ISAs, O0; (**c**) cross ISAs, O1; (**d**) cross ISAs, O2; (**e**) cross ISAs, O3; (**f**) cross ISAs, cross opts.

5.3. Efficiency Test

An efficient binary clone detection tool can help analysts accelerate the process of binary similarity analysis. In this experiment, we compared GeneDiff with the control group (see Section 5.2) in terms of detection efficiency.

We compiled *binutils-2.32* and *OpenSSL-1.0.1f* into binaries of different ISAs (i.e., AMD64, AArch64, and MIPS64). Then, these binaries were divided into several file pairs according to their file names. We used GeneDiff and the control group to detect file pairs and logged the time taken to detect each function pair. We found that GeneDiff generally completed the task in much less time than did the control group. Figure 9 shows the experimental results of two approaches for detecting the sysinfo file in binutils-2.32 and the libssl file in OpenSSL-1.0.1f; the red line denotes GeneDiff. There are a total of 137 function pairs in the sysinfo file. GeneDiff successfully found 129 of them in 33.02 s, whereas the control group found 93 pairs in 177.69 s. Since the two approaches preprocess the target binaries via IDA Pro [29], the function pairs are not found in either of the approaches in Figure 9 during the initial time period. However, GeneDiff's preprocessing time was shorter than that of the control group, and its detection efficiency was much higher than that of the control group. This is because GeneDiff only converts bytecode into VEX instructions and trims out redundant data during preprocessing, which makes the implementation faster. GeneDiff leverages the lexical semantics based on the intermediate representation rather than on symbolic execution: the former is more scalable and less vulnerable to adverse effects from the differences in architectures and compiler optimization options.

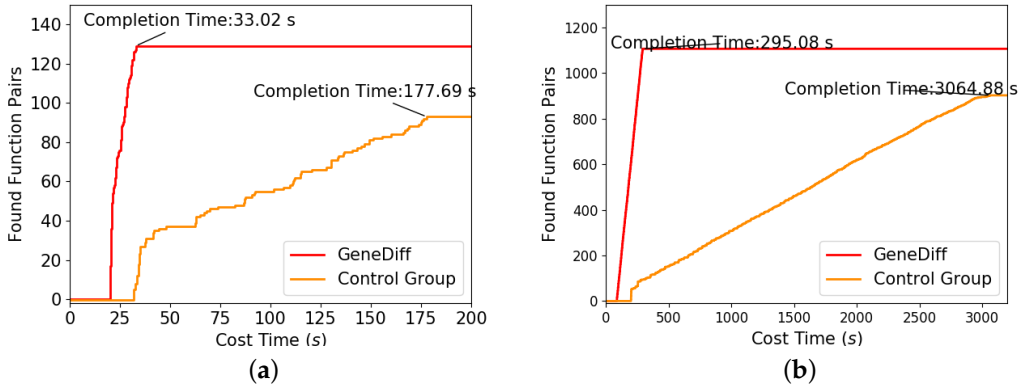


Figure 9. Two examples of detection efficiency: sysinfo in binutils-2.32, libssl in OpenSSL-1.0.1f. (**a**) sysinfo; (**b**) libssl.

5.4. Sensitivity Test

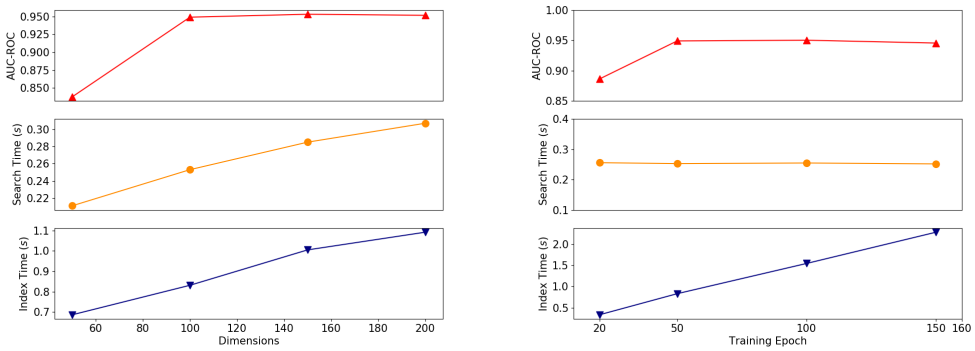
In this experiment, we measured the sensitivity of GeneDiff to the parameters. In particular, we considered the number of training epochs and the dimensions of the vectors. We also measured the impact of the abstraction rules on clone detection. By default, GeneDiff utilizes the abstraction rules, its representation vector dimension is 100, and the number of training epochs is 50.

To better illustrate the impacts of each parameter on clone detection, we used the control variable method in the experiment. Figure 10a shows the impacts of the dimensions of the representation vectors on the AUC-ROC, the average training time per function, and the average search time per function. We note that the AUC-ROC of GeneDiff seriously degrades when the representation vector dimension is low. The distinctions between instructions are not sufficiently obvious when the vector dimension is too low. When the vector dimension is increased, the AUC-ROC increase slightly. However, the average training time per function and the average search time per function increase sharply. After comprehensive testing, we believe that a reasonable dimension for the representation vectors is 100.

Figure 10b shows the impacts of the training epoch on the AUC-ROC, the average training time per function, and the average search time per function. As the number of training epochs increases, the average training time per function increases linearly, while the average search time per function shows almost no fluctuations. When the number of training epochs is less than 50, the AUC-ROC increases rapidly as the number of training epochs increase; when the training epoch number exceeds 50, the AUC-ROC tends to remain stable.

We also evaluated the impacts of the abstraction rules on clone detection. The abstraction rules can mitigate the binary differences of different ISAs, which makes GeneDiff more resilient to cross-architecture differences. As shown in Table 5, using the abstraction rules significantly increases the AUC-ROC and reduces both the training and search time. Using the abstraction rules also reduces the size of the vocabulary and mitigates OOV issues in query mode. Due to a lack of sufficient context, the semantics of words that occur at very low frequencies are difficult to represent accurately, which may increase the convergence time.

Through the sensitivity test, we can confirm the reasonable parameters of GeneDiff, which are a representation vector dimension of 100 and 50 training epochs.



(**a**) (**b**)

Figure 10. The impacts of dimensions and number of epochs on detection performance. From top to bottom, the Y-axes are the area under the curve (AUC) of the ROC score, search time and indexing time. (**a**) Dimensions; (**b**) Epoches.

Table 5. The impacts of the abstraction rules on the clone detection performance.

Abstract Rules	AUC-ROC	Search Time	Index Time
Has Abstract Rules	0.9491	0.253	0.831
No Abstract Rules	0.6575	0.576	1.944

6. Real-World Detection

The above experiments evaluated GeneDiff's overall performance on clone detection for different ISAs. In this section, we apply GeneDiff in real-world situations to evaluate its detection performance for known IoT vulnerabilities. This evaluation searched for the Heartbleed (CVE-2014-0160) vulnerability in OpenSSL binaries from the following sources:

(1) Self-compiled OpenSSL 1.0.1f binaries for different ISAs (i.e., AMD64, AArch64 and MIPS64 (OpenSSL 1.0.1f. Available at https://github.com/openssl/openssl/releases/tag/OpenSSL_1_0_1f));

(2) The OpenWrt r39853 firmware compiled for MIPS, ARM and AArch64 (OpenWrt. Available at https://github.com/openwrt/openwrt/commit/c2bbaf439c);

(3) The DD-WRT r21888 firmware compiled for MIPS, ARM and AArch64 (DD-WRT. Available at https://svn.dd-wrt.com/changeset/21888);

OpenWrt and DD-WRT are two open source projects for routers, and some industrial IoT routers and smart home routers are modified from them. We used the trained representation model to generate representation vectors for functions in binaries. Then, we treat the vulnerability functions (dtls1_process_heartbeat and tls1_process_heartbeat) as the target functions and utilize the query mode of GeneDiff to detect vulnerabilities in binaries from different ISAs. Because the two vulnerability functions are actually identical, when either of them is matched indicates that the vulnerability exists in the binaries. Table 6 shows the results of the Heartbleed vulnerability detection of IoT firmware by GeneDiff on the real-world dataset. These results indicate that using GeneDiff to detect known vulnerabilities in IoT devices is feasible.

Table 6. Detecting the Heartbleed vulnerabilities in real-world internet of things (IoT) firmware.

Binary Name		Target Functions		
		ARM	**AArch64**	**MIPS**
OpenSSL 1.0.1f	AArch64	✓	✓	✓
	AMD64	✓	✓	✓
	MIPS64	✓	✓	✓
OpenWrt	MIPS	✓	✓	✓
	ARM	✓	✓	✓
	AArch64	✓	✓	✓
DD-WRT	MIPS	✓	✓	✓
	ARM	✓	✓	✓
	AArch64	✓	✓	✓

7. Discussion

The results of the extensive experiments show GeneDiff can detect similarity with high accuracy across different compiler optimization options and different targeted architectures. In this section, we discuss why GeneDiff performs well, as well as its limitations.

Unlike existing clone detection methods for cross-ISAs [4,13–16,24], GeneDiff converts assembly functions from different architectures into the same intermediate representation and generates semantic vectors for the functions via a representation model inspired by NLP. To address the unique instructions of different architectures, GeneDiff drops their unimportant instructions and unifies the necessary instructions (e.g., registers), which reduces the number of differences caused by different architectures and improves its clone detection accuracy.

In addition, GeneDiff treats each VEX instruction as a letter instead of a word. An assembly instruction is converted into multiple VEX instructions. Generating a semantic representation for each VEX instruction is similar to interpreting the semantics of each letter in the word "apple". Therefore, in GeneDiff, each VEX instruction is treated as a letter, and the combinations of VEX instructions are regarded as words. This reduces the number of tokens and increases the diversity of the tokens, which enriches the semantics of the words.

GeneDiff also suffers from certain limitations. GeneDiff uses VEX as an intermediate representation. However, VEX is not explicit. For example, VEX uses variables (cc_op, cc_dep1, cc_dep2, and cc_ndep) to store abstract information about the machine status instead of storing the status flags directly, which may introduce some deviations. Besides, GeneDiff is designed for 32-bit architectures (i.e., X86, ARM, and MIPS) and 64-bit architectures (i.e., AMD64, Arch64, and MIPS64). At this stage, it is not directly applicable for semantic clones for eight-bit or 16-bit IoT devices, even though its clone search engine is architecture-agnostic. In the future, we will optimize an intermediate representation to share tokens with low-power IoT devices.

8. Conclusions

Reverse engineering the software of IoT devices targeted to different architectures involves intensive manual analysis. Fortunately, binary clone detection approaches can reduce the burden of manual analysis during reverse engineering. In this paper, we propose and implement a prototype GeneDiff, a semantic-based representation binary clone detection approach for cross-architectures. GeneDiff utilizes a semantic representation model to generate high-dimensional numeric vectors for each assembly function based on an intermediate representation, and uses cosine distance to evaluate function similarity. Compared to the approaches that use symbolic execution, GeneDiff is significantly more efficient and accurate. The extensive experiments indicate that GeneDiff can detect similarity

with high accuracy even when the code has been compiled with different optimization options and targeted to different architectures. We also use real-world IoT firmware across architectures as targets, thereby demonstrating GeneDiff's practicality for detecting known vulnerabilities.

Author Contributions: Z.L. contributed to the design of the study, conducted the experiments, analyzed the results, and wrote the manuscript. Y.T. analyzed the results and proofread the paper. B.W. and W.X. proofread the paper.

Funding: This research was supported in part by National Natural Science Foundation of China under Grant 61472437.

Acknowledgments: This study was financially supported by National Natural Science Foundation of China under Grant 61472437.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application programming interface
AUC	Area under the curve of ROC
CFG	Control flow graph
CVE	Common vulnerabilities and exposures
IoT	Internet of things
IR	Intermediate representation
ISA	Instruction set architecture
NLP	Natural language processing
OOV	Out of vocabulary
PV-DM	Paragraph vector-distributed memory approach
ROC	Receiver operating characteristic curve

References

1. Synopsys. Highlights from the 2018 Open Source Security and Risk Analysis Report. Available online: https://www.synopsys.com/blogs/software-security/infographic-2018-open-source-security-and-risk-analysis-report/ (accessed on 7 March 2019).
2. Mockus, A. Large-scale code reuse in open source software. In Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007), Minneapolis, MN, USA, 20–26 May 2007; p. 7.
3. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1157–1177. [CrossRef]
4. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the 23rd Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 21–24 February 2016.
5. OpenWrt r39853. Available online: https://github.com/openssl/openssl/releases/tag/OpenSSL_1_0_1f (accessed on 20 January 2019).
6. DD-WRT r21888. Available online: https://svn.dd-wrt.com/changeset/21888 (accessed on 20 January 2019).
7. IoT Statistics. Available online: https://safeatlast.co/blog/iot-statistics/ (accessed on 20 March 2019).
8. David, Y.; Yahav, E. Tracelet-based code search in executables. *ACM Sigplan Not.* **2014**, *49*, 349–360. [CrossRef]
9. Gao, D.; Reiter, M.K.; Song, D. K-gram based software birthmarks. In Proceedings of the International Conference on Information and Communications Security, Birmingham, UK, 20–22 October 2008; pp. 238–255.
10. Ming, J.; Pan, M.; Gao, D. iBinHunt: Binary hunting with inter-procedural control flow. In Proceedings of the International Conference on Information Security and Cryptology, Seoul, Korea, 28–30 November 2012; pp. 92–109.

11. Jhi, Y.C.; Wang, X.; Jia, X.; Zhu, S.; Liu, P.; Wu, D. Value-based program characterization and its application to software plagiarism detection. In Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA, 21–28 May 2011; pp. 756–765.
12. Zynamics BinDiff. Available online: http://www.zynamics.com/bindiff.html (accessed on 28 January 2019).
13. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Cross-architecture bug search in binary executables. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–20 May 2015; pp. 709–724.
14. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T. Statistical similarity of binaries. In Proceedings of the ACM Conference on Programming Language Design and Implementation, Santa Barbara, CA, USA, 13–17 June 2016; pp. 266–280.
15. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
16. 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 the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, Texas, USA, 30 October–3 November 2017; pp. 363–376.
17. Le, Q.; Mikolov, T. Distributed representations of sentences and documents. In Proceedings of the International Conference on Machine Learning, Beijing, China, 21–26 June 2014; pp. 1188–1196.
18. Zuo, F.; Li, X.; Zhang, Z.; Young, P.; Luo, L.; Zeng, Q. Neural machine translation inspired binary code similarity comparison beyond function pairs. In Proceedings of the 26rd Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 24–27 February 2019.
19. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
20. Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. *OSdi* **2004**, *4*, 289–302.
21. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, 20–26 May 2007; pp. 96–105.
22. Myles, G.; Collberg, C. K-gram based software birthmarks. In Proceedings of the 2005 ACM Symposium on Applied Computing, Santa Fe, NM, USA, 13–17 March 2005; pp. 314–318.
23. Wang, X.; Jhi, Y.C.; Zhu, S.; Liu, P. Behavior based software theft detection. In Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 9–13 November 2009; pp. 280–290.
24. Chandramohan, M.; Xue, Y.; Xu, Z.; Liu, Y.; Cho, C.Y.; Tan, H.B.K. Bingo: Cross-architecture cross-os binary search. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016; pp. 678–689.
25. Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Proceedings of the ACM Conference on Programming Language Design and Implementation, San Diego, CA, USA, 10–13 June 2007; Volume 42, pp. 89–100.
26. 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, San Francisco, CA, USA, 20–22 May 2019; pp. 38–55.
27. PyVEX. Available online: https://github.com/angr/pyvex (accessed on 28 January 2019).
28. Cambria, E.; White, B. Jumping NLP curves: A review of natural language processing research. *IEEE Comput. Intell. Mag.* **2014**, *9*, 48–57. [CrossRef]
29. IDA Pro. Available online: http://www.hex-rays.com/ (accessed on 26 January 2019).
30. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed representations of words and phrases and their compositionality. In Proceedings of the Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 5–10 December 2013.
31. Maaten, L.V.D.; Hinton, G. Visualizing data using t-SNE. *J. Mach. Learn. Res.* **2008**, *9*, 2579–2605.

32. Khoo, W.M.; Mycroft, A.; Anderson, R. Rendezvous: A search engine for binary code. In Proceedings of the 10th Working Conference on Mining Software Repositories, San Francisco, CA, USA, 18–19 May 2013; pp. 329–338.
33. Sojer, M.; Henkel, J. Code reuse in open source software development: Quantitative evidence, drivers, and impediments. *J. Assoc. Inf. Syst.* **2010**, *11*, 868–901. [CrossRef]