αDiff: Cross-Version Binary Code Similarity Detection with DNN

Bingchang Liu*
liubingchang@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences

Wei Huo*†
huowei@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences

Chao Zhang†
chaoz@tsinghua.edu.cn
Institute for Network Science and Cyberspace, Tsinghua University

Wenchao Li*
Institute of Information Engineering, Chinese Academy of Sciences

Feng Li*
Institute of Information Engineering, Chinese Academy of Sciences

Aihua Piao
Institute of Information Engineering, Chinese Academy of Sciences

Wei Zou*
zouwei@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences

ABSTRACT

Binary code similarity detection (BCSD) has many applications, including patch analysis, plagiarism detection, malware detection, and vulnerability search etc. Existing solutions usually perform comparisons over specific syntactic features extracted from binary code, based on expert knowledge. They have either high performance overheads or low detection accuracy. Moreover, few solutions are suitable for detecting similarities between *cross-version* binaries, which may not only diverge in syntactic structures but also diverge slightly in semantics.

In this paper, we propose a solution αDiff, employing three semantic features, to address the cross-version BCSD challenge. It first extracts the *intra-function feature* of each binary function using a deep neural network (DNN). The DNN works directly on raw bytes of each function, rather than features (e.g., syntactic structures) provided by experts. αDiff further analyzes the function call graph of each binary, which are relatively stable in cross-version binaries, and extracts the inter-function and inter-module features. Then, a distance is computed based on these three features and used for BCSD. We have implemented a prototype of αDiff, and evaluated it on a dataset with about 2.5 million samples. The result shows that αDiff outperforms state-of-the-art static solutions by over 10 percentages on average in different BCSD settings.

CCS CONCEPTS

• **Security and privacy** → *Software reverse engineering*; • **Computing methodologies** → *Machine learning*;

*Also with: School of Cyber Security, University of Chinese Academy of Sciences.
†corresponding author

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238199

KEYWORDS

Code Similarity Detection, DNN

ACM Reference Format:
Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-Version Binary Code Similarity Detection with DNN . In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238199

1 INTRODUCTION

Given two binary functions, the problem of evaluating whether they are similar is called binary code similarity detection (BCSD). It plays an important role in many applications, including code plagiarism detection [32, 33, 43] and malware family and lineage analysis [2, 26, 28]. It could also be used to analyze 1-day (i.e., patched) vulnerabilities [5], or summarize vulnerability patterns [53], when applying BCSD on pre-patch and post-patch binaries. Moreover, it could even be used in cross-architecture bug searching [16, 17, 52], when applying BCSD on a known bug and target applications.

However, BCSD faces several challenges. First, different compiler optimizations yield *cross-optimization* binaries. Second, compilers with different algorithms (e.g., register allocation) generate *cross-compiler* binaries. Third, the source code compiled on different platforms (e.g., with different instruction sets) yields *cross-architecture* binaries. These binaries are semantic-equivalent, but have different syntactic structures. On the other hand, the source code itself may evolve over time (e.g., being patched), yielding *cross-version* binaries. These binaries by nature are similar, because they have a same root. But they have different syntactic structures and slightly different semantics. Existing solutions could address these BCSD challenges to some extent, but perform poorly in *cross-version* binaries.

State-of-the-art BCSD solutions heavily rely on a specific syntactic feature of binary code, i.e., control flow graphs (CFGs) of functions. The most widely used tool BinDiff [55] utilizes graph-isomorphism (GI) theory [14, 18] to compare functions' CFGs. However, GI algorithms are time consuming and lack polynomial time

solutions. Moreover, GI is vulnerable to (even minor) CFG changes, and thus has a low accuracy. BinHunt [19] and iBinHunt [34] extend GI with symbolic execution and taint analysis to address these challenges, but still have low accuracy and high overheads.

BinGo [6], Esh [12] and CABS [38] provide more resilience to CFG changes, by computing the similarities of CFG fragments and composing the overall CFG similarity. DiscovRE [16] provides better performance, by employing a filter on CFGs to reduce the number of GI comparisons. It extracts some numeric features from CFGs, e.g., counts of instructions or basic blocks (BBs), and use the kNN algorithm to pre-filter similar CFGs. Genius [17] extracts similar numeric attributes from BBs, and use them to augment CFG nodes and get Attributed CFGs (ACFGs), to support cross-architecture BCSD. Gemini [52] uses an end-to-end neural network to embed ACFGs, providing better performance and accuracy.

These solutions all rely on the syntactic feature, i.e., CFGs. These features are derived from expert knowledge, which could introduce bias sometimes. For example, CFGs could change dramatically even if there is none or minor code changes, and cause noticeable deviations in the BCSD results. The first research question addressed in this paper is: *RQ1: How to extract features from binary code with as little human bias as possible?*

Few solutions consider the semantics of the binary code, except BinGo [6] and Esh [12]. These two use theorem proving to check semantic equivalence of CFG fragments, and thus are computationally expensive. On the other hand, the semantics of cross-version binaries may change slightly, e.g., due to patching. So, strict semantic equivalence comparisons are not suitable neither. The second research question addressed in this paper is: *RQ2: How to efficiently utilize semantic features to improve the accuracy of BCSD?*

Cross-version BCSD is demanded for two decades, e.g., in patch analysis [5] and knowledge transfer [50]. It is also one of the most attractive functionalities provided by the popular tool BinDiff [55]. However, this problem is far from being solved. For example, the average accuracy of BinDiff is less than 0.5 when comparing coreutils 5.0 with coreutils 8.29 that consist of hundreds of binaries. But researchers have paid few attentions to this specific topic. The third research question addressed in this paper is: *RQ3: How to build a solution fit for cross-version BCSD?*

In this paper, we propose a solution αDiff to address the aforementioned questions. In short, it extracts proper semantic features from binaries, and uses them to compute similarity scores to perform BCSD. To fit for cross-version BCSD, each binary function is characterized as three semantic features, i.e., the function code's (i.e., intra-function) features, function invocation (i.e., inter-function) features, and module interactions (i.e., inter-module) features.

First, to characterize a function's intra-function feature, we do not use its CFG or other attributes derived from expert knowledge. We notice that, the raw bytes contain all semantic information of the function, and neural networks could automatically retrieve unbiased features from them. Thus, we propose a neural network to extract features from the function's raw bytes, inspired by previous works [45]. More specifically, we represent the raw bytes as a matrix, and use convolutional neural network (CNN) to convert it into an embedding (i.e. a vector). Further, in order to ensure similar functions' embeddings are close to each other, we embed this CNN into a Siamese network, i.e., a popular solution used in fine-grained visual similarity recognition [3, 4, 44, 47].

Second, we notice that similar functions have similar call graphs but not the opposite. So we analyze each function's call graph to extract its inter-function feature. Ideally, the whole call graph should be considered. But in our solution, we extract only the in-degree and out-degree of a function node in the call graph as the function's feature, for performance reason.

Third, we also notice that similar functions have similar imported functions (even in different architectures), but not the opposite. So we analyze each function's imported function set and use it as inter-module feature. A specific algorithm (Section 3.4) is proposed to embed this set into a vector, to support distance computation.

So, given any two binary functions, we could extract their intra-function, inter-function and inter-module features. Then, we could compute their distances in terms of each feature respectively. Finally, we will merge these three distances to measure the overall similarity of these two functions.

We have implemented a prototype of our solution αDiff, and evaluated it on a custom dataset consisting of about 2.5 millions pairs of cross-version functions, which are collected from public repositories. The results showed that, αDiff outperforms BinDiff by 11 percentage on average, up to 52% for some binary pairs. With only the intra-function feature, αDiff outperforms BinDiff by 6 percentage on average, up to 43% for some binary pairs.

More importantly, although our training data is composed of cross-version binaries, our model is also good for cross-compiler and cross-architecture binary code similarity detection, as well as in a specific application, i.e., vulnerability search. The results showed that αDiff in general outperforms state-of-the-art solutions.

Overall, we made the following contributions:

(1) We proposed a neural network solution to extract intra-function semantic features from raw bytes of binary functions, without interference of expert knowledge. Together with two other proposed semantic features, i.e., inter-function and inter-module features, we built an end-to-end system αDiff able to perform cross-version BCSD.
(2) We built a labelled dataset for deep learning, which contains 66,823 pairs of binaries and about 2.5 million pairs of functions. Researchers can freely use this dataset[1], to design other neural network models and solve other problems.
(3) We developed a prototype αDiff and evaluated it on this dataset. The results showed that it outperforms state-of-the-art solutions, in all of cross-compiler, cross-architecture and cross-version BCSD settings.

2 PROBLEM DEFINITION

In this section, we will introduce the definition of the cross-version BCSD (binary code similarity detection) problem.

2.1 Notation and Assumption

We assume all binaries are compiled from source code written in high-level languages, not assembled from hand-written assembly or generated by packers, which aim at obfuscating the binaries.

[1] https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix

To be practical, we also assume the debug symbols in binaries are stripped, which makes binary analysis more challenging.

A binary B_i consists of a set of functions $f_{i1}, f_{i2}, ..., f_{in}$. Binary function identification, which is out of the scope of this paper, could be handled well by existing binary disassembly solutions. We hereby assume each binary's functions could be identified correctly, i.e., all bytes of each function f_{ij} in a binary B_i can be determined.

A core task of BCSD is to find each function's matching counterpart. Two binary functions are considered as matching, if they are compiled from functions with the same name (including namespace and class etc.) and used in similar contexts. It is worth noting that, identical functions (i.e., with same raw bytes) are matching, but matching functions could be non-identical.

2.2 Cross-version BCSD Problem

The *cross-version BCSD problem* focuses on analyzing two binaries B_1 and B_2 compiled from a same source code project, which could evolve over time. It is related to the following tasks:

(1) *function matching:* for each function f_{1i} in a binary B_1, if exist, find its match f_{2j} in the other binary B_2.
(2) *similarity score*: for each pair of functions f_{1i} and f_{2j}, compute a semantic similarity score ranging from 0 to 1 between them, indicating how likely they are similar to each other.
(3) *difference identification*: for each matching pair of functions f_{1i} and f_{2j}, identify the exact differences in their code bytes, if their similarity score is less than 1 (i.e., non-identical).

In this paper, we only focus on the first 2 tasks.

2.3 Variant BCSD Problems

In this paper, we aim to solve the challenges in the cross-version BCSD problem, which is more challenging than other BCSD problems. As the evaluation results showed, our solution could be used directly for the following variant settings and received good results.

(1) *Cross-optimization BCSD*: It aims at analyzing two binaries compiled from a same copy of code, using a same compiler but with different compilation optimizations.
(2) *Cross-compiler BCSD*: It aims at analyzing two binaries compiled from a same copy of code, using different compilers (e.g., different vendors.).
(3) *Cross-architecture BCSD*: It aims at analyzing two binaries compiled from a same copy of code, targeting different architectures (e.g., with different instruction sets).

2.4 Evaluation Metric

The goal of BCSD solutions is identifying matching functions accurately. We thus evaluate whether the matching function is in the top K matching candidates reported by a given BCSD, namely **Recall@K**, similar to related works [29, 47],

Given two binaries $B_1 = f_{11}, f_{12}, ..., f_{1n}$ and $B_2 = f_{21}, f_{22}, ..., f_{2m}$, for simplicity, we assume they have T pairs of matching functions, i.e., (f_{11}, f_{21}), (f_{12}, f_{22}), ..., and (f_{1T}, f_{2T}) respectively. The rest of functions do not match.

For any function f_{1i} in B_1, the BCSD solution could sort functions in the other binary B_2, based on their similarities with f_{1i}. We denote the top K similar functions as $topK(f_{1i})$, and denote $hit@K(f_{1i})$ as whether f_{1i}'s matching function exists in $topK(f_{1i})$.

$$hit@K(f_{1i}) = \begin{cases} 1, & f_{2i} \in topK(f_{1i}) \text{ and } i \leq T \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The evaluation metric of BCSD is thus defined as follows.

$$Recall@K(B1, B2) = \frac{\sum_{i=1}^{T} hit@K(f_{1i})}{T} \quad (2)$$

3 APPROACH

In this section, we present the key idea of our solution to the problem of *cross-version* binary code similarity detection.

3.1 Overview

Traditional solutions based on syntactic attributes are inadequate for cross-version BCSD. The similarity of two cross-version binary functions should be estimated by their semantics, i.e., their raw bytes, their relationships with other functions defined in the same binaries, and their relationships to imported functions defined in external modules. For simplicity, we name these features as intra-function, inter-function and inter-module features respectively.

As shown in Figure 1, our solution αDiff first extracts these features from two binary functions, then calculates the distances between each pair of features, and finally evaluates an overall similarity score based on these three distances. The final score indicates the similarity between the two functions.

Unlike traditional solutions which use CFG and other syntactic attributes as features, we apply a deep neural network to directly extract intra-function features from each function's raw bytes. An embedding is generated by this neural network, to represent the binary function's semantic feature.

Furthermore, we use the call graph (CG) to characterize the inter-function semantic feature, and use the imported function invocation relationship to characterize the inter-module semantic feature. These two features could be extracted from binaries with traditional lightweight program analysis.

3.2 Intra-function Semantic Feature

Inspired by previous binary analysis solutions [45], we also utilize a neural network to extract intra-function semantic features from the raw bytes of binary functions. After many trials (e.g., Conv1D, LSTM and convolutional LSTM), we find out the convolutional neural network (CNN) is the best fit.

In our solution, the CNN takes the raw bytes of a function I_q as input, and maps it to an embedding $f(I_q)$, i.e., a vector in a d-dimensional Euclidean space $\mathbb{R}^d$. Then we could calculate the distance between any two functions using their embeddings.

In order to detect similarity, we have to train the model to satisfy the following requirement. *RQ: The distance between (embeddings of) two similar functions should be small, while the distance between (embeddings of) two dissimilar functions should be large.*

Inspired by deep metric learning [3, 44, 47], we also embed two identical CNNs into a Siamese architecture [4], to comply with the requirement RQ and train the CNN's parameters. Unlike the recent work Gemini [52], which generates embeddings with DNN

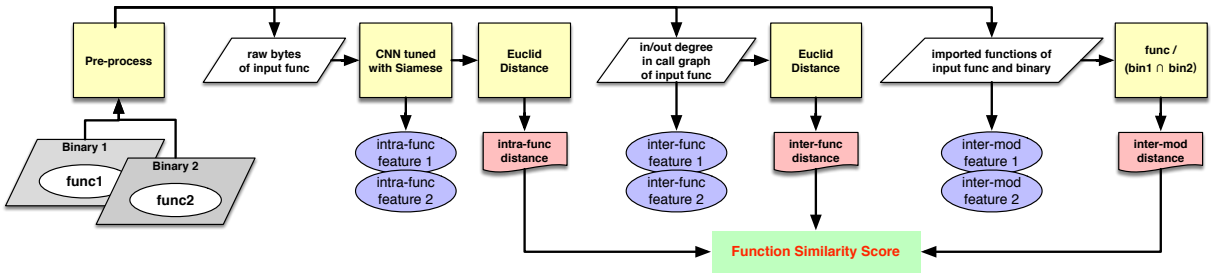


Figure 1: Overview of the cross-version binary code similarity detection solution αDiff.

based on some engineered syntactic features, our solution does not require expert knowledge and is suitable for cross-version BCSD.

3.2.1 Embedding Functions with CNN. The convolutional neural network (CNN) is a specific kind of neural network for processing data that has a known, grid-like topology [20, 31]. It has achieved great success in many applications, e.g., AlexNet [30].

However, classical CNNs are specifically designed for image classification, requiring inputs similar to RGB images, which have at least 3 channels. This is not suitable for our problem scope and our input format. After many trials, we design a new CNN as follows.

Network structure. The CNN that we propose consists of 8 convolutional layers, 8 batch normalization layers, 4 max pooling layers and 2 full-connected layers. The whole model uses rectified linear units, i.e. ReLU [10], as the non-linear activation function. In total, there are more than 1.6 million parameters in this network [2].

Network I/O. This CNN takes a 100×100×1 tensor $\mathbb{T}$ as input, and outputs a 64-dimensional vector (i.e., embedding). We fill the raw bytes of a function into $\mathbb{T}$ byte by byte. If the function has less than 10,000 bytes, we will fill the tensor $\mathbb{T}$ with zero byte paddings. Otherwise, we will discard the redundant bytes of this function.

It is worth noting that, few functions (e.g., less than 0.01% in our dataset) have more than 10,000 bytes. Moreover, if two functions' first 10,000 bytes are similar, they are likely similar too. So, it is reasonable to simply discard redundant bytes of the function.

Data augmentation. In image classification applications, data augmentation is a popular measure to improve datasets for CNN training. Unlike images pixels that are tolerant to minor modifications, function bytes are vulnerable to changes, since they will alter the function semantics. So during the training of our model, we do not apply any data augmentation measures.

Overfitting issue. We also investigated the measures used in AlexNet etc., i.e., layer stacking style and solutions, to avoid model overfitting. In particular, we adopt *batch normalization* [27] to address the overfitting issue.

3.2.2 Learning Parameters Using Siamese Network. In order to train the parameters of this CNN embedding network, we use the Siamese architecture [4]. As shown in Figure 2, the Siamese architecture uses two identical CNN embedding networks. Each CNN takes one function as input, namely I_q and I_t, and outputs the corresponding

[2]Details could be found at: https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix

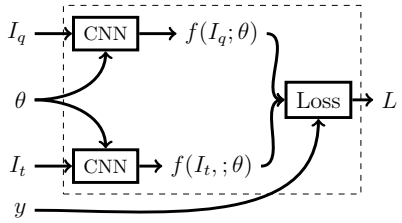


Figure 2: Siamese network illustration.

embeddings, namely $f(I_q;\theta)$ and $f(I_t;\theta)$ respectively, where f represents the network structure and θ represents the network parameters.

In addition to the input pair (I_q, I_t), the Siamese architecture also accepts an indicator input y. This input y indicates whether the two functions I_q and I_t are similar or not. If they are similar, $y = 1$, otherwise $y = 0$.

The goal of the training is to find the best parameter θ, to satisfy the aforementioned requirement RQ, i.e., the distance between functions I_q and I_t is small if they are similar otherwise large. Formally, the distance of two functions' intra-function features is defined as follows.

$$D1(I_q, I_t) = \left\| f(I_q;\theta) - f(I_t;\theta) \right\| \tag{3}$$

To achieve this goal, we evaluate a *contrastive loss function* [22] of this Siamese network as follows.

$$L(\theta) = \underset{(I_q,I_t)}{Average}\{y \cdot D1(I_q, I_t) + (1-y) \cdot \max(0, m - D1(I_q, I_t))\} \tag{4}$$

where m is a pre-defined hyper-parameter, i.e., the minimal margin distance that dissimilar functions are expected to have.

We can infer that, if this loss function gets a minimal value, $D1(I_q, I_t)$ is close to 0 when $y = 1$, and $\max(0, m - D1(I_q, I_t))$ is close to 0 when $y = 0$. In other words, each function will be close to similar ones and far from dissimilar ones, in the embedding space. So, the aforementioned requirement RQ is satisfied.

The objective of the training thus becomes to find the parameter θ to minimize the Siamese network's loss function, i.e.

$$arg\min_{\theta} L(\theta) \tag{5}$$

This objective function can be solved using Stochastic Gradient Descent (SGD) with standard back propagation algorithms [31, 41].

3.2.3 Negative Training Samples. It is crucial to build a set of positive samples (i.e., pairs of similar functions) and negative samples (i.e., pairs of dissimilar functions), in order to get desirable convergence in the aforementioned CNN and Siamese network.

We have collected about 2.5 million positive samples from public repositories. We thus need a way to either collect or generate sufficient negative samples for training. Similar to [37, 44, 47], we also generate negative samples in each mini-batch during training, based on the positive samples.

More specifically, for each positive sample (I_q, I_p) in a mini-batch, we will generate two *semi-hard* [44] negative samples, namely (I_q, I_{n1}) and (I_p, I_{n2}). Take the function I_q as an example, we will look for function I_n that satisfies the following equation[3].

$$0 < D1(I_q, I_n) < m \tag{6}$$

We randomly select one function I_{n1} that satisfies this constraint as the negative function. But we will skip the hardest negative function (i.e. $arg\min D1(I_q, I_n)$), because such samples can easily lead the model to bad local minima during training.

In order to get sufficient different negative samples, we will shuffle the mini-batch of positive samples in each epoch during training. More specifically, in each epoch, we first randomly sort the binary file pairs, then randomly sort the positive function pairs between each binary file pair. The randomly sorted positive samples (function pairs) will then be divided into mini-batches, and new negative samples could be generated from these new mini-batches.

3.3 Inter-function Semantic Feature

Functions do not work solely, i.e., they will call other functions or be called by others. The interactive relationship with other functions in the same binary (including themselves) is an important semantic feature, i.e. inter-function feature. This feature can be represented as the function call graph. We notice that similar functions have similar call graphs.

Ideally, the whole call graph should be considered. For example, SMIT [26] uses the call graph matching to detect similarity between malware samples. Although they propose an efficient Graph Edit Distance algorithm, the computation cost is still too high to deploy.

In our solution, we extract only the in-degree and out-degree of a node (i.e., function) in the call graph as its inter-function feature. More specifically, for each function I_q, we embed its inter-function feature as a 2-dimensional vector as follows.

$$g(I_q) = (in(I_q), out(I_q)) \tag{7}$$

where $in(I_q)$ and $out(I_q)$ are the in-degree and out-degree of the function I_q in the call graph respectively. Formally, the (Euclidean) distance of two functions' inter-function features is defined as:

$$D2(I_q, I_t) = \left\| g(I_q) - g(I_t) \right\| \tag{8}$$

3.4 Inter-module Semantic Feature

A function I_q also invokes a set of imported functions, denoted as $imp(I_q)$, which are defined in external modules (libraries). We notice that similar functions invoke similar imported functions but not the opposite. Moreover, the set $imp(I_q)$ is relatively stable even if I_q changes across versions, due to the modular development process. As a result, the imported function set is also an important semantic feature, i.e. inter-module feature.

For consistency, we also convert the inter-module feature, i.e., the imported function set, into a vector for distance computation. Therefore, we use the following element-testing formula to embed a set into the superset's space.

$$h(set, superset) =< x_1, x_2, ..., x_N > \tag{9}$$

where N is the size of the superset, and $x_i = 1$ if the i-th element of $superset$ is in set; otherwise 0.

For two functions I_q and I_t, assuming their binaries are B_q and B_t, we will get their imported function set $imp(B_q)$ and $imp(B_t)$ too. Then we take $imp(B_q) \cap imp(B_t)$ as the superset, and use the aforementioned formula to encode each inter-module feature, then compute their distance as follows.

$$\begin{aligned} D3(I_q, I_t) = \| & h(imp(I_q), imp(B_q) \cap imp(B_t)) - \\ & h(imp(I_t), imp(B_q) \cap imp(B_t)) \| \end{aligned} \tag{10}$$

It is worth noting that, $imp(B_q)$ is a superset of $imp(I_q)$, and $imp(B_t)$ is a superset of $imp(I_t)$. Moreover, although symbols (e.g., function names) may be stripped from binaries, the names of imported functions will always be kept so that the linker could link modules together. As a result, it is easy to extract the set of imported functions for a binary or a function.

3.5 Overall Similarity Computation

Given any two functions I_q and I_t, we could thus compute their intra-function distance ($D1$), inter-function distance ($D2$) and inter-module distance ($D3$), following Equation 3, Equation 8 and Equation 10 respectively.

As aforementioned, the imported function set of a function is usually stable, so the inter-module distance $D3$ between similar functions in general is small. Moreover, the intra-function distance $D1$ between similar functions is also small, usually smaller than the minimal margin of dissimilar functions, i.e., the parameter m in Equation 4. But, similar functions in cross-version binaries could have different call graphs, especially different in-degree and out-degree, resulting a relatively large inter-function distance $D2$.

So, we eventually compute an overall distance to represent the overall similarity of these two functions as follows.

$$D(I_q, I_t) = D1(I_q, I_t) + (1 - \xi^{D2(I_q, I_t)}) + D3(I_q, I_t) \tag{11}$$

where ξ is a pre-defined hyper-parameter in the range $(0, 1)$, used for suppressing the effect of $D2$.

For any function I_q in question, we will compute the overall distance D with each target function, and then sort all target functions by the distance. The closest target functions (i.e., $topK$ defined in Section 2.4) are more likely similar to I_q.

4 EVALUATION

4.1 Implementation

We have implemented a prototype of αDiff. It consists of three major components: *preprocessor, feature generator*, and *neural network model*. The preprocessor is implemented as a plug-in of the binary analysis tool IDA Pro 6.8 [24]. From each function in a binary, three types of information are extracted, i.e. its raw bytes, its

[3] This is different from FaceNet [44] and VGGFace [37].

Table 1: Sources of the dataset

data-src	projects/ packages	versioned proj/pkg	cross-version binary pairs	cross-version function pairs
GitHub repo	31	9,419	8,510	166,541
Debian repo	895	1,842	58,313	2,323,252
TOTAL	926	11,261	66,823	2,489,793
for training	-	-	44,526	1,665,025
for validation	-	-	11,150	417,158
for testing	-	-	11,147	407,610

in/out-degree in the call graph and its imported function set. These raw information are then encoded into embeddings, as discussed in Section 3. Specifically, the raw bytes are converted into embeddings, using a special neural network. This network model is implemented in TensorFlow-1.3 [1] and Keras-2.0 [8].

4.2 Evaluation Setup

Our experiments are conducted on a server equipped with two Intel Xeon E5-2650v4 CPUs (24 cores in total) running at 2.20 GHz, 128 GB memory, 12TB hard drives, and 4 NVIDIA Tesla P100 PCIE-16G GPU cards. During both training and evaluation, only 1 GPU card was used.

4.2.1 Dataset. A dataset is needed to train neural network model and evaluate its effectiveness. We collected a set of 2,489,793 positive samples (i.e., pairs of matching functions) from 66,823 pairs of *cross-version* binaries in x86 Linux platform. As shown in Table 1, the dataset has two sources.

The first source is the GitHub repository, where we collected source code from 31 projects with 9,419 releases. Each release is then compiled with the compiler `GCC-5.4` with the default optimization options. We placed each project's two successive releases of binaries into one pair, and got 8,510 pairs in total.

The second source is Debian package repository, where we directly collected binaries from `.deb` packages. We have collected 895 packages with 1,842 versions, from the Ubuntu 12.04, 14.04 and 16.04 platform. Each package may contain more than one binaries. We grouped each version of binary with its closest version as a pair, and got 58,313 pairs in total.

For each pair of cross-version binaries, we then retrieved pairs of matching functions, which have a same name but are not identical. To increase the diversity, we also extracted some pairs of functions that are identical in cross-version binaries. Finally, in total we have 2,489,793 pairs of cross-version matching functions, from 66,823 pairs of cross-version binaries. Among them, about 1.52 percents of pairs of cross-version functions are identical. It is worth noting that, BinDiff reports that 29.4 percent of pairs are identical, due to the inaccuracy introduced in its GI-based algorithm.

Ground Truth. As aforementioned, to get the ground truth of matching functions, we utilized function names and thus relied on debug symbols optionally shipped with binaries. For binaries that are compiled from GitHub code, the compilation option `-g` is added when building. For binaries from Debina package repository, we only collected packages with symbolic files (e.g., .ddeb packages). After collecting the ground truth, we stripped all debug symbols from binaries, and evaluate our tool αDiff and other tools on the stripped binaries only.

4.2.2 Dataset Split. Similar to other works, we also split the dataset into three disjoint subsets for training, validation and testing, in order to evaluate the generalization capability of the trained model on unseen binaries. Roughly, we set the number of positive samples (i.e., pairs of matching functions) in these three subsets proportion to 4:1:1. Moreover, we ensure that matching pairs from one pair of binaries will be placed in one subset. Table 1 shows the size of each subset at the bottom.

4.2.3 Neural Network Training. This dataset is used to train the neural network model for intra-function feature extraction. In the CNN model, we use the RMSProp optimizer [25], set the learning rate to 0.001, and set the forgetting factor to 0.9. In the Siamese network (Eq.4), we set the margin m, i.e., the minimal distance between dissimilar functions, to 1.0. Furthermore, we set the ξ in the overall similarity score formula (e.g., Equation 11) to 0.75. For each mini-batch, 100 positive samples are selected and 200 semi-hard negative samples are generated online. The Siamese network is trained for 200 epochs (3.075 h/epoch), to tune the parameters in the CNN embedding network.

4.3 Hyper-parameters in the Siamese Network

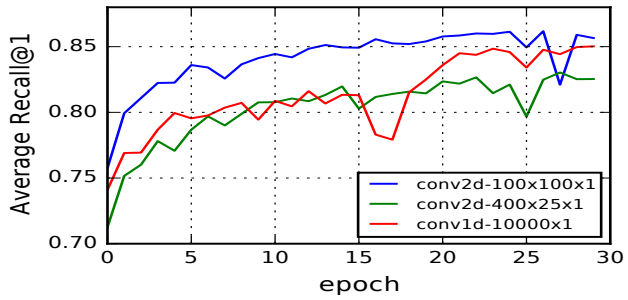
In addition, the DNN involves several other hyper-parameters and design decisions, e.g., the shape of input tensor, the embedding size, the negative sample mining method and the network architecture etc. The choices of these parameters and design decisions could also affect the effectiveness of the model.

We have conducted a set of experiments to select proper parameters and design decisions. Due to the time and resource limitation, we train each model setting with 25% samples of the training set for 30 epochs. We evaluate each model's performance on a subset of the testing set, in which the number of positive samples of each binary pair is no less than 100.

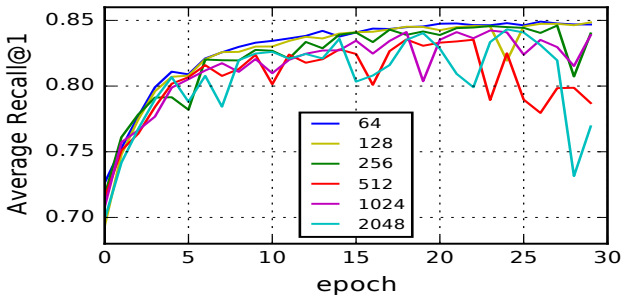
4.3.1 Input Shape and Convolutional Layer Type. We have evaluated the performance of the network in different input shape and convolutional layer, as shown in Figure 3a. We can see that, the model's performance is affected by the shape of input tensor. Besides, we also evaluate the performance of 1D-CNN and find it doesn't perform better than 2D-CNN with 100x100x1 input tensor. Section 5 will discuss more about it.

4.3.2 Embedding Size. We have evaluated the performance of the network in different embedding size, i.e., the dimension of the CNN's output vector, as shown in Figure 3b. It shows that if the embedding size is set to 64, the model in general performs best and get the highest average Recall@1 accuracy. Thus, in our model, we set the embedding size to 64.

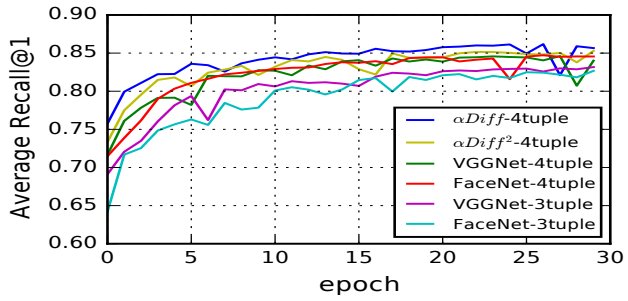
4.3.3 Hard Negative Sample Mining Method. We evaluate the performance of our hard negative samples mining method and compare it with another two typical mining methods, i.e., FaceNet [44] and VGGNet [37]. Further, we evaluate the performance of network in different count of hard negative samples corresponding to each positive sample. In Figure 3c, *FaceNet-3tuple* means one semi-hard negative sample is mined by FaceNet method [44] for each positive



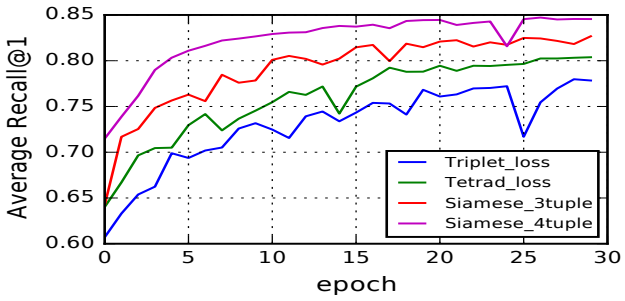
(a) Average Recall@1 in different input shape and convolutional layer.



(b) Average Recall@1 accuracy in different embedding sizes.



(c) Average Recall@1 in different negative sample mining methods.



(d) Average Recall@1 in different network architecture.

Figure 3: Evaluation of hyper-parameters and design decisions.

Table 2: The recall accuracy of αDiff-1f and αDiff-3f on the testing set of 9,308 pairs of binaries.

	Whole set		Big subset	
	αDiff-1f	αDiff-3f	αDiff-1f	αDiff-3f
Avg. Recall@1	0.953	0.955	0.885	0.900
Avg. Recall@5	0.996	0.997	0.968	0.974
Avg. MRR	0.973	0.975	0.922	0.933

sample, while *FaceNet-4tuple* means two semi-hard negative samples, corresponding to the left and the right of each positive sample (i.e. a 2-tuple).

Besides, we have also tried a more gentle mining criterion[4] as described by the line $\alpha Diff^2$-4tuple in Figure 3c. However, it doesn't performs better than ours. We can see that, our method performs better than another two methods. And 4-tuple (tetrad) mining method performs better than 3-tuple (triplet) mining.

4.3.4 Network Architecture. We have also evaluated the performance of different network architectures, as shown in Figure 3d.

We evaluated the Triplet architecure of FaceNet [44] and Tetrad architecture of [47]. We also evaluated Siamese architecture with triplet mining and tetrad mining method. We can see that, Siamese architecture with tetrad mining method performs best.

[4] $D1(I_q, I_p) < D1(I_q, I_{n1}) < m$

4.4 Accuracy in Cross-version BCSD

In this section, we evaluated the accuracy of αDiff, with only the intra-function feature enabled (denoted as αDiff-1f) and with all three features enabled (denoted as αDiff-3f or αDiff), using the metric **Recall@K**. The task is essentially a ranking task and every query has only one correct answer (matched function). So we thus also evaluated the MRR (Mean Reciprocal Rank) [39].

4.4.1 Evaluation on Testing Set. We first evaluated αDiff-1f and αDiff-3f on the testing dataset consisting of 9,308 pairs of cross-version binaries, and calculated the metrics of Recall@1 and Recall@5 for each pair of binaries. In order to evaluate αDiff's performance on big binary pair (more function pairs), we split the testing set into big subset and small subset. The big subset is consisted of 647 binary pairs and each binary pair contains more than 300 function pairs. Table 2 shows the average recall and MRR results.

4.4.2 Evaluation on coreutils. We further evaluated the accuracy of αDiff on unseen binaries, e.g., `coreutils` that is commonly used target in other BCSD solutions [6, 15, 49]. We collected 7 versions of `coreutils`, including the latest version (i.e., v8.29 at the time of writing), and got 604 pairs of cross-version binaries.

Table 4 shows the results of accuracy, when matching the old version of `coreutils` to its latest version, compared with state-of-the-art cross-version BCSD tool BinDiff.

First, BinDiff becomes less accurate when the version gap gets larger. For example, the accuracy of matching v5.0 to v8.29 is only 0.486, less than half of the accuracy of matching v8.28 to v8.29. αDiff

Table 3: Accuracy of αDiff and BinDiff in cross-compiler-vendor & cross-version BCSD.

vulnerability	alias	binaries		BinDiff		αDiff		
		pre	post	vul-func found?	Recall@1	vul-func found in top-1?	Recall@1	MRR
CVE-2014-0160	Heartbleed	openssl-1.0.1f	openssl-1.0.1g	×	0.371	√	0.609	0.695
CVE-2014-6271	Shellshock	bash-4.3	bash-4.3.30	×	0.485	√	0.559	0.577
CVE-2014-4877		wget-1.15	wget-1.16	×	0.555	√	0.691	0.664
CVE-2014-7169	Shellshock2	bash-4.3	bash-4.3.30	×	0.485	×	0.559	0.577
CVE-2014-9295	Clobberin Time	ntpd-4.27p10	ntpd-4.28	×	0.434	√	0.422	0.364
CVE-2015-3456	Venom	qemu-2.30	qemu-2.40	×	0.276	×	0.301	0.261

Table 4: Comparison between αDiff and BinDiff on unseen binaries from `coreutils`. Each version of binary is evaluated against its latest version, i.e., v8.29 released on 2017-12-31. The average accuracy is shown here. @1 means Recall@1 and @5 means Recall@5.

Ver#	Date	BinDiff	αDiff-1f			αDiff-3f		
			@1	@5	MRR	@1	@5	MRR
5.0	2003-04-02	0.486	0.649	0.756	0.708	0.738	0.821	0.782
6.3	2006-09-30	0.606	0.677	0.844	0.756	0.778	0.892	0.836
7.1	2009-02-21	0.618	0.743	0.870	0.809	0.804	0.896	0.853
8.10	2011-02-04	0.776	0.827	0.906	0.868	0.864	0.926	0.896
8.26	2016-11-30	0.992	0.958	0.987	0.972	0.977	0.999	0.987
8.28	2017-09-01	0.999	0.995	0.999	0.997	0.996	0.999	0.997

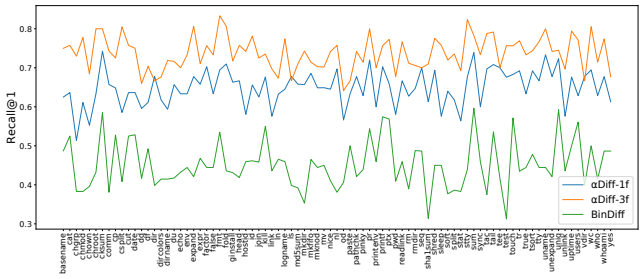


Figure 4: Comparison of the exact matching accuracy (i.e., Recall@1) between αDiff and Bindiff, when comparing each binary in `coreutils` of version v5.0 to version v8.29.

has a much better performance in detect similarities between versions spanning a long time period. For example, as shown in Figure 4, αDiff outperforms BinDiff by more than 52% when comparing some binaries from v5.0 to v8.29.

Second, αDiff-1f is also better than BinDiff when the version gap is large. For example, it outperforms BinDiff by over 16% on average, as shown in Figure 4. It shows that the sole intra-function feature, which is identified by the Siamese network, is a very strong feature for cross-version BCSD.

Third, BinDiff performs slightly better than αDiff when the version gap is small. For example, the Recall@1 of αDiff is 0.996, smaller than the accuracy of BinDiff (i.e., 0.999), when comparing binaries of version v8.28 to v8.29. However, the Recall@5 of αDiff is better than BinDiff, even if the version gap is small. It shows that αDiff could get better match in the top 5 candidates than BinDiff.

4.5 Performance in Cross-compiler BCSD

Cross-compiler BCSD has three sub-types: cross-compiler-vendor, cross-compiler-version and cross-optimization-level. Several approaches have been proposed to solve one or two of them, however, neither one can solve all of them well. Our solution αDiff employs semantic features to solve BCSD, and brings a chance to solve cross-compiler BCSD too.

4.5.1 Cross-compiler-vendor & Cross-compiler-version. Esh [12] is one of the representative solutions for this problem. However, it is too slow, taking about 3 minutes to compare a pair of functions. But we have a bunch of pairs to analyze. So here we only compared αDiff with the state-of-the-art industrial tool BinDiff, not Esh.

As shown in Table 3, we collected six projects with known vulnerabilities, which are also evaluated in [12]. To construct cross-compiler-vendor and cross-compiler-version binary pairs, we first selected both vulnerable version and patched version for each project. Then we compiled the pre-patch version with gcc-4.6.3, and compiled the post-patch version with clang-3.8. To evaluate the accuracy of similarity detection, we then designed two experiments.

We first queried the vulnerable function(s) in the patched binary. If the tool returns the matching vulnerable function or puts it in the top-1 candidate, we mark a √ otherwise × in the table. The results showed that, αDiff succeeded in four out of six cases, whereas Bindiff failed in all cases.

Then we computed the overall accuracy of function matching, i.e., Recall@1, between these two binaries. In all cases except CVE-2014-9295, αDiff outperformed BinDiff by 10 percentage on average.

4.5.2 Cross-compiler-vendor & Cross-optimization-level. BinGo [6] is specialized on cross-compiler and cross-architecture BCSD problems. Here, we compared αDiff with BinGo and BinDiff, using same experiment configurations as BinGo.

More specifically, we compiled coreutils for x86 32-bit and x86 64-bit architectures, using gcc (v4.8.2) and clang (v3.0) with various optimization levels (`O0` to `O3`).

To make a head-to-head comparison with BinGo, we also use the same metric as BinGo. More specifically, it evaluates the percentage of functions in the first binary, whose matching function in the second binary is ranked one (i.e., best match) by the tool. We can infer that, this percentage is similar to Recall@1, except with a different denominator in Equation 2.

We evaluated six settings and listed the results in Table 5. In the x86 architecture, we can see that αDiff outperforms BinGo in all cases by 20% on average, and outperforms BinDiff in all cases except the two `clang-O2 vs. clang-O3` settings. In the x64 architecture,

Table 5: Comparing percentage of matching functions ranked #1 by different tools, i.e., αDiff, BinGo and BinDiff, in cross-compiler-vendor & cross-optimization-level settings. C is short for clang and G is short for gcc.

	x86_64			x86		
	αDiff	BinGo	BinDiff	αDiff	BinGo	BinDiff
C-O0 vs. G-O3	0.403	0.265	0.229	0.462	0.332	0.271
C-O0 vs. C-O3	0.461	0.305	0.285	0.492	0.372	0.315
C-O2 vs. C-O3	0.960	0.561	0.996	0.969	0.576	0.994
G-O0 vs. C-O3	0.446	0.307	0.199	0.484	0.333	0.258
G-O0 vs. G-O3	0.428	0.257	0.192	0.441	0.302	0.255
G-O2 vs. G-O3	0.577	0.470	0.780	0.765	0.480	0.757

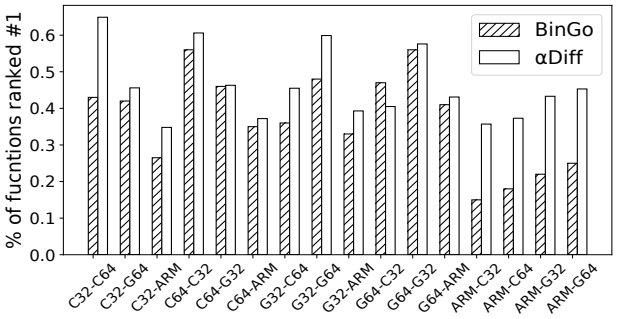


Figure 5: Comparing αDiff with BinGo in cross-compiler-vendor and cross-architecture settings. C32 is short for clang x86-32bit and G64 for gcc x86-64bit.

we could also draw similar results. In general, αDiff outperforms BinGo and BinDiff in this setting of BCSD problems.

4.6 Performance in Cross-architecture BCSD

Since BinGo is also specialized on cross-architecture BCSD, we still made a comparison with it here, using a same experiment configuration. More specifically, we matched all functions in CORE-UTILS binaries compiled for one architecture (i.e., x86 32-bit, x86 64-bit and ARM) to namesakes in binaries compiled for another architecture. We also used the percentage of matching functions ranked one as metric.

The evaluation result is shown in Figure 5. For BinGo, we took its best result (i.e., the one with selective inlining). The data in the plot can be read in the same way as BinGo. For example, the bar $C32 - G64$ means that, when querying functions compiled using clang for x86 architecture, around 42% of their matching functions compiled using gcc for x64 architecture are ranked 1 by the tool BinGo, while 46% are ranked 1 by αDiff. Overall, αDiff outperforms BinGo in all settings except $G64 - C32$.

4.7 Application in Vulnerability Search

Many BCSD solutions are proposed to solve vulnerability search problem. Here we also evaluated the performance of αDiff in vulnerability search, in the cross-architecture setting, as done by DiscovRE [16], Multi-k-MH [38] and Genius [17].

We first compiled the vulnerable OpenSSL library, which have two virtually identical vulnerabilities TLS and DTLS, on platforms ARM, MIPS and x86. Then we queried one vulnerable function in one binary, and examined the ranks of the two matching functions in another binary, reported by each tool. The results are listed in Table 6.

It shows that, when searching one function (e.g., TLS), αDiff could always rank the two matching functions at place 1 and 2. In general, it outperforms other tools. For example, when querying the x86 TLS function in the MIPS binary, DiscovRE ranks the DTLS function at place 4, while αDiff ranks it at place 2.

It shows that, the semantic feature automatically extracted by neural network (used in αDiff) is effective, even better than the CFG and other attributes (used in DiscovRE) provided by human experts.

It is worth noting that, Gemini [52] also uses a network to search bugs in cross-architecture. However, it uses features (e.g., CFG and node attributes) provided by human experts to pre-process input binaries, which we believe would introduce bias. But we do not have the benchmark they used to evaluate the effectiveness. So we omit the comparison between αDiff and Gemini.

5 DISCUSSION

In our work, we use CNN to encode the raw bytes of a function into an embedding. More specifically, we take a function as a 2D grid of bytes through 2-dimensional convolutional network (2D-CNN). 2D-CNN is usually used for image processing, because an image presents strong spatial locality on its the two dimensions (i.e. width and height). A function is different from this and more similar to text, meaning 1D-CNN seems to be more appropriate. In our evaluations, 1D-CNN performs not bad, however, 2D-CONV with the specific configuration performs better.

We can't explain the reason behind this and plan to explore it based on the advances on neural network visualization-related researches [35, 36] in the future work. Although we don't think we have found the best configuration for both 2D-CNN and 1D-CNN, we show the feasibility of extracting similarity features from raw bytes with 2D-CNN. Other researchers can also continue to find better models and configurations with our dataset.

Although αDiff outperforms BinGo [6] in cross-architecture evaluations, in fact, the inter-function feature and inter-module feature play important roles in our evaluations. In the future, we plan to transfer our approach to the cross-architecture settings, i.e. training a model with cross-architecture dataset.

6 RELATED WORK

In this section, we briefly survey closely related work.

6.1 Binary Code Similarity Analysis

Staitic Analysis. BinDiff [55], DiscovRE [16] and Genius [17] are based on CFG/CG graph-isomorphism (GI) theory [14, 18]. DiscovRE [16] identifies a set of lightweight numeric features and builds a pre-filter based on the features to quickly identify a small set of candidate functions. Genius [17] encodes the CFGs into high-level numeric vectors to achieve realtime vulnerability search in a large set of firmware images. These approaches depend on graph

Table 6: Accuracy of searching the two vulnerable functions of Heartbleed, namely 'tls1_process_heartbeat' (denoted as TLS) and 'dtls1_process_heartbeat' (denoted as DTLS), in OpenSSL binaries compiled for ARM, MIPS and x86. When searching one function, both two should appear in the top candidates, since they are virtually identical. When searching TLS (or DTLS), the value in the cell means the ranks of the matching TLS;DTLS functions (or DTLS;TLS) reported by the tool.

	Multi-MH [38]		Multi-k-MH [38]		DiscovRE [16]		Genius [17]		Centroid [7]		αDiff	
From → to	TLS	DTLS	TLS	DTLS	TLS	DTLS	TLS	DTLS	TLS	DTLS	TLS	DTLS
ARM → x86	1;2	1;2	1;2	1;2	1;2	1;2	*	*	*	*	1;2	1;2
ARM → MIPS	1;2	1;2	1;2	1;2	1;2	1;2	*	*	*	*	1;2	1;2
ARM → ReadyNAS [40]	1;2	1;2	1;2	1;2	1;2	1;2	*	*	*	*	1;2	2;1
ARM → DD-WRT [13]	1;2	1;2	1;2	1;2	1;2	1;2	*	*	*	*	1;2	1;2
MIPS → ARM	2;3	3;4	1;2	1;2	1;2	1;2	*	*	*	*	2;1	2;1
MIPS → x86	1;4	1;3	1;2	1;3	1;2	1;2	*	*	*	*	2;1	2;1
MIPS → ReadyNAS	2;4	6;16	1;2	1;4	1;2	1;2	1;2	1;2	88;190	678;988	2;1	1;2
MIPS → DD-WRT	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2	46;100	87;99	1;2	2;1
x86 → ARM	1;2	1;2	1;2	1;2	1;2	1;2	*	*	*	*	2;1	2;1
x86 → MIPS	1;7	11;21	1;2	1;6	1;4	1;3	*	*	*	*	1;2	1;2
x86 → ReadyNAS	1;2	1;2	1;2	1;2	1;2	1;2	1;2	1;2	145;238	333;127	2;1	2;1
x86 → DD-WRT	70;78	1;2	5;33	1;2	1;2	1;2	1;2	1;2	97;255	102;89	1;2	1;2

matching, which has no known polynomial time algorithm, and ignore the semantics of concrete assembly-level instructions.

Inspired by DiscovRE and Genius, Gemini [52] assumes a function can be represented as an ACFG, a CFG with numeric attributes. It converts each ACFG to an embedding through Siamese architecture and Structure2vec [11] network, which is similar with ours. However, Gemini relies on hand-tuned features, such as CFG structures and numeric features. αDiff extracts the intra-function feature from the raw bytes of functions, without human interference.

BinHunt [19] and iBinHunt [34] extend GI with symbolic execution and taint analysis to find semantic differences. BinGo [6] captures the complete function semantics by a selective inlining technique and then utilizes length variant partial traces to model binary functions in a program structure agnostic fashion. Esh [12] statistically reasons similarity of functions based on smaller fragments' semantic similarities computed by a program verifier. These approaches are computationally expensive. For example, Esh takes 3 minutes on average to compare a pair of functions.

Dynamic Analysis. Under the assumption that similar code has similar runtime behaviors, BELX [15] executes each function for several calling contexts and collects runtime behaviors of functions under a controlled randomized environment. IMF-SIM [49] introduces in-memory fuzzing to solve the coverage issue of dynamic approaches. These approaches rely on architecture-specific tools to execute or emulate binaries, and are inconvenient to apply.

6.2 Deep Metric Learning

Bromley et al. [4] paves the way on deep metric learning and trained Siamese networks for signature verification. Chopra et al. [9] presents a method for training a similarity metric from data and applied it to face verification. Sean et al. [3] learns fine-grained visual similarity for product design with deep convolutional neural network and siamese network. FaceNet [44] uses a deep convolutional network and triplet embedding [51] to learn unified embedding on faces for face verification and identification. In contrast to contrastive embedding [22] and triplet embedding [51], Song et al. [47] proposes a new deep feature embedding algorithm by taking full advantage of the training batches.

6.3 Convolutional Neural Network

Convolutional networks are a specialized kind of neural network for processing data that has a known and grid-like topology [20, 31]. CNNs typically consist of multiple interleaved layers of convolutions, non-linear activations, local response normalizations, pooling layers and one or more full-connected layers. Since the notable success of AlexNet [30] in ILSVRC2012 [42], there has been an explosion of interest in CNNs and many successful variants, such as VGGNet [46], Inception-v3 [48] and ResNet [23], have been presented. A full review of CNNs is beyond the scope of this paper and more information can be found in [20, 21, 54].

7 CONCLUSION

In this paper, we propose a DNN augmented solution αDiff to solve the cross-version BCSD problem. It employs three semantic features, i.e., intra-function, inter-function and inter-module features, which are exacted from binary code with lightweight solution. We have implemented a prototype of αDiff, and evaluated it on a dataset with about 2.5 million samples. The result shows that αDiff outperforms state-of-the-art static solutions by over 10 percentages on average, in detecting similarities between cross-version, cross-compiler and cross-architecture binaries.

ACKNOWLEDGMENTS

We would thank Xiaoyu He, Jiaqi Peng and Shuai Wang for their help in dataset preparing and paper comments. The work is supported by the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology, as well as National Key R&D Program of China under Grant No.: 2016QY071405, NSFC under Grant No.: 61572481, 61602470, 61772308, 61472209, 61502536, and U1736209. and Young Elite Scientists Sponsorship Program by CAST (Grant No. 2016QNRC001).

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[2] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering.. In *NDSS*, Vol. 9. Citeseer, 8–11.

[3] Sean Bell and Kavita Bala. 2015. Learning visual similarity for product design with convolutional neural networks. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 98.

[4] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a" siamese" time delay neural network. In *Advances in Neural Information Processing Systems*. 737–744.

[5] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 143–157.

[6] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.

[7] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale.. In *USENIX Security Symposium*, Vol. 15.

[8] François Chollet et al. 2015. Keras. Retrieved April 10, 2018 from https://keras.io/

[9] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, Vol. 1. IEEE, 539–546.

[10] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 8609–8613.

[11] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*. 2702–2711.

[12] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.

[13] DDWRT 2013. DD-WRT Firmware Image r21676. Retrieved April 26, 2018 from ftp://ftp.dd-wrt.com/betas/2013/05-27-2013-r21676/senao-eoc5610/linux.bin

[14] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *Sstic* (2005), 1–13.

[15] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX.

[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *NDSS*.

[17] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.

[18] Halvar Flake. 2004. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*. Citeseer, 161–174.

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

[20] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep learning*. Vol. 1. MIT press Cambridge.

[21] Isma Hadji and Richard P Wildes. 2018. What Do We Understand About Convolutional Networks? *arXiv preprint arXiv:1803.08834* (2018).

[22] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, Vol. 2. IEEE, 1735–1742.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[24] Hex-Rays. 2015. IDA Pro Disassembler and Debugger. Retrieved April 10, 2018 from https://www.hex-rays.com/products/ida/index.shtml

[25] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. 2012. Neural Networks for Machine Learning-Lecture 6a-Overview of mini-batch gradient descent.

[26] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 611–620.

[27] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[28] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference.. In *USENIX Security Symposium*. 81–96.

[29] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2011), 117–128.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[31] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[32] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 389–400.

[33] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177.

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

[35] Anh Nguyen, Jason Yosinski, Yoshua Bengio, Alexey Dosovitskiy, and Jeff Clune. 2016. Plug & play generative networks: Conditional iterative generation of images in latent space. *arXiv preprint arXiv:1612.00005* (2016).

[36] Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. 2018. The Building Blocks of Interpretability. *Distill* 3, 3 (2018), e10.

[37] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. 2015. Deep face recognition.. In *BMVC*, Vol. 1. 6.

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

[39] Dragomir R Radev, Hong Qi, Harris Wu, and Weiguo Fan. 2002. Evaluating web-based question answering systems. *Ann Arbor* 1001 (2002), 48109.

[40] ReadyNAS 2014. ReadyNAS Firmware Image v6.1.6. Retrieved April 26, 2018 from http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip

[41] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[43] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 117–128.

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

[45] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks.. In *USENIX Security Symposium*. 611–626.

[46] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[47] Hyun Oh Song, Yu Xiang, Stefanie Jegelka, and Silvio Savarese. 2016. Deep metric learning via lifted structured feature embedding. In *Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on*. IEEE, 4004–4012.

[48] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2818–2826.

[49] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 319–330.

[50] Zheng Wang, Ken Pierce, and Scott McFarling. 2000. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism* 2 (2000), 1–20.

[51] Kilian Q Weinberger, John Blitzer, and Lawrence K Saul. 2006. Distance metric learning for large margin nearest neighbor classification. In *Advances in neural information processing systems*. 1473–1480.

[52] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 363–376.

[53] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 462–472.

[54] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[55] zynamics. [n. d.]. BinDiff. Retrieved April 09, 2018 from https://www.zynamics.com/bindiff.html