DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing

Yue Duan*, Xuezixiang Li†, Jinghan Wang†, and Heng Yin†

*Cornell University †UC Riverside

yd375@cornell.edu, {xli287, jwang131}@ucr.edu, heng@cs.ucr.edu

***Abstract*—Binary diffing analysis quantitatively measures the differences between two given binaries and produces fine-grained basic block level matching. It has been widely used to enable different kinds of critical security analysis. However, all existing program analysis and machine learning based techniques suffer from low accuracy, poor scalability, coarse granularity, or require extensive labeled training data to function. In this paper, we propose an unsupervised program-wide code representation learning technique to solve the problem. We rely on both the code semantic information and the program-wide control flow information to generate basic block embeddings. Furthermore, we propose a k-hop greedy matching algorithm to find the optimal diffing results using the generated block embeddings. We implement a prototype called DEEPBINDIFF and evaluate its effectiveness and efficiency with a large number of binaries. The results show that our tool outperforms the state-of-the-art binary diffing tools by a large margin for both cross-version and cross-optimization-level diffing. A case study for OpenSSL using real-world vulnerabilities further demonstrates the usefulness of our system.**

I. INTRODUCTION

Binary Code Differential Analysis, a.k.a, binary diffing, is a fundamental analysis capability, which aims to quantitatively measure the similarity between two given binaries and produce the fine-grained basic block level matching. Given two input binaries, it precisely characterizes the program-wide differences by generating the optimal matching among basic blocks with quantitative similarity scores. It not only presents precise, fine-grained and quantitative results about the differences at a whole binary scale but also explicitly reveals how code evolves across different versions or optimization levels. Because of the precision and fine-granularity, it has enabled many critical security usages in various scenarios when program-wide analysis is required, such as changed parts locating [1], malware analysis [28], [45], security patch analysis [55], [38], binary wide plagiarism detection [40] and patch-based exploit generation [11]. As a result, binary diffing has been an active research focus. In general, existing works can be put into two categories.

Traditional Approaches. BinDiff [10], which is the de facto commercial binary diffing tool, performs many-to-many graph isomorphism detection [35] on callgraph and control-flow graph (CFG), and leverages heuristics (e.g., function name, graph edge MD index) to match functions and basic blocks. Other static analysis based techniques perform matching on the generated control and data flow graphs [25], [30], [49], [27] or decompose the graphs into fragments [20], [17], [18], [40] for similarity detection. Most of these approaches consider only the syntax of instructions rather than the semantics, which can be critical during analysis, especially when dealing with different compiler optimization techniques. Moreover, graph matching algorithms such as Hungarian algorithm [35] are expensive and cannot guarantee optimal matching.

Another line of research utilizes dynamic analysis. These techniques carry out the analysis by directly executing the given code [26], [53], performing dynamic slicing [44] or tainting [43] on the given binaries, and checking the semantic level equivalence based on the information collected during the execution. In general, these techniques excel at extracting semantics of the code and have good resilience against compiler optimizations and code obfuscation but usually suffer from poor scalability and incomplete code coverage, due to the nature of the dynamic analysis.

Learning-based Approaches. Recent works have leveraged the advance of machine learning to tackle the binary diffing problem. Various techniques [29], [54], [58], [23] have been proposed to leverage graph representation learning techniques [16], [42], [37] and incorporate code information into embeddings (i.e., high dimensional numerical vectors). Then they use these embeddings for similarity detection. InnerEye [58] and Asm2Vec [23] further rely on NLP techniques to automatically extract semantic information and generate embeddings for diffing. These approaches embrace two major advantages over the traditional static and dynamic approaches: 1) higher accuracy as they incorporate unique features of the code into the analysis by using either manual engineered features [29], [54] or deep learning based automatic methods [23], [58]; 2) better scalability since they avoid heavy graph matching algorithm or dynamic execution. What's more, the learning process can be significantly accelerated by GPUs.

*This work was conducted while Yue Duan was a PhD student at UC Riverside, advised by Prof. Heng Yin.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24311
www.ndss-symposium.org

Limitations. Despite the advantages, we identify three major limitations of the existing learning-based approaches.

First, no existing learning-based technique can perform efficient program-wide binary diffing at a fine-grained basic block level. Most of the current techniques conduct diffing at a granularity of functions [29], [54], [36], [23], [39]. InnerEye [58] is the only learning-based technique that achieves basic block level granularity. Nevertheless, it is not scalable enough for program-wide binary diffing due to its design. Each calculation of basic block similarity has to go through a complex neural network (with a total of 3,700,440 parameters in the current implementation), which significantly affects the performance. To evaluate the scalability, we use pre-trained models from the authors [7] and measure the performance. The results show that it takes an average of 0.6ms for InnerEye to process one pair of blocks. Note that a binary diffing could easily involve millions of basic block distance calculations. Therefore, it takes hours for InnerEye to finish diffing between two small binaries unless a powerful GPU is used. As aforementioned, fine-grained binary diffing is an essential analysis, upon which many critical security analyses can be built. Hence, a fine-grained and efficient diffing tool is strongly desired.

Second, none of the learning-based techniques considers both program-wide dependency information and basic block semantic information during analysis. The program-wide dependency information, which can be extracted from the interprocedural control flow graph (ICFG), provides the contextual information of a basic block. It is particularly useful in binary diffing as one binary could contain multiple very similar functions, and the program-wide contextual information can be vital to differentiate these functions as well as the blocks within them. Basic block semantic information, on the other hand, characterizes the uniqueness of each basic block. InnerEye [58] extracts basic block semantic information with NLP techniques [42] but only considers local control dependency information within a small code component by adopting the Longest Common Subsequence. Asm2Vec [23] generates random walks only within functions to learn token and function embeddings.

Third, most of the existing learning-based techniques [55], [36], [58], [39] are built on top of supervised learning. Thus, the performance is heavily dependent on the quality of training data. To begin with, we argue that a large, representative and balanced training dataset can be very hard to collect, because of the extreme diversity of the binary programs. Also, supervised learning could suffer from the overfitting problem [22]. Moreover, state-of-the-art supervised learning technique InnerEye [58] considers a whole instruction (opcode + operands) as a word, therefore, may lead to serious out-of-vocabulary (OOV) problem. To show this, we follow the same preprocessing step described in the paper and evaluate the pre-trained model using the same dataset CoreUtils v8.29. The results show that the pre-trained model could only achieve an average of 78.37% instruction coverage for the binaries in CoreUtils v8.29. In other words, 21.63% of the instruction cannot be modeled, as compared to only 3.7% reported in the paper. This is due to the fact that we use GCC compiler while clang was used in the InnerEye paper. Nonetheless, it shows that the InnerEye model could easily become much less useful when facing even a small change in the test setting.

Our Approach. To this end, we propose an unsupervised deep neural network based program-wide code representation learning technique for binary diffing. In particular, our technique first learns basic block embeddings via unsupervised deep learning. Each learned embedding represents a specific basic block by carrying both the semantic information of the basic block and the contextual information from the ICFG. These embeddings are then used to efficiently and accurately calculate the similarities among basic blocks.

To achieve this goal, we modify state-of-the-art NLP technique Word2Vec [42] to extract semantic information for tokens (opcode and operands), and further assemble basic block level feature vectors. Hence, these feature vectors contain the semantic information for blocks. Modeling opcode and operand separately also eliminates the OOV problem. Then, we model the basic block embedding generation as a network representation learning problem and feed the feature vectors into Text-associated DeepWalk algorithm (TADW) [56] to generate basic block embeddings that contain program-wide control flow contextual information. Consequently, these basic block embeddings contain both the program-wide contextual information and the semantics from the basic blocks. Finally, we present a k-hop greedy matching algorithm to match basic blocks to cope with compiler optimizations including function inlining and basic block reordering.

We implement a prototype DEEPBINDIFF, and conduct an extensive evaluation with representative datasets containing 113 C binaries and 10 C++ binaries. The evaluation shows that our tool soundly outperforms state-of-the-art techniques BinDiff and Asm2Vec, for both cross-version and cross-optimization-level diffing. Furthermore, we conduct a case study using real-world vulnerabilities in OpenSSL [9] and show that our tool has unique advantages when analyzing vulnerabilities.

Contributions. The contributions of this paper are as follows:

- We propose a novel unsupervised program-wide code representation learning technique for binary diffing. Our technique relies on both the code semantic information and the program-wide control-flow graph contextual information to generate high quality basic block embeddings. Then we propose a k-hop greedy matching algorithm to obtain the optimal results.
- We implement a prototype DEEPBINDIFF. It first extracts semantic information by leveraging NLP techniques. Then, it performs TADW algorithm to generate basic block embeddings that contain both the semantic and the program-wide dependency information.
- An extensive evaluation shows that DEEPBINDIFF could outperform state-of-the-art binary diffing tools for both cross-version and cross-optimization-level diffing. A case study further demonstrates the usefulness of DEEPBINDIFF with real-world vulnerabilities.

To facilitate further research, we have made the source code and dataset publicly available.[1]

[1] https://github.com/deepbindiff/DeepBinDiff

II. Problem Statement

A. Problem Definition

Given two binary programs, binary diffing precisely measures the similarity and characterizes the differences between the two binaries at a fine-grained basic block level. We formally define ***binary diffing problem*** as follows:

Definition 1. Given two binary programs $p_1 = (B_1, E_1)$ and $p_2 = (B_2, E_2)$, binary diffing aims to find the optimal basic block matching that maximizes the similarity between p_1 and p_2:

$$SIM(p1, p2) = \max_{m_1, m_2, ..., m_k \in M(p_1, p_2)} \sum_{i=1}^{k} sim(m_i), \text{ where:}$$

- $B_1 = \{b_1,\ b_2,\ ...,\ b_n\}$ and $B_2 = \{b_1^{'}, b_2^{'}, ..., b_m^{'}\}$ are two sets containing all the basic blocks in p_1 and p_2;
- Each element e in $E \subseteq B \times B$ corresponds to *control flow dependency* between two basic blocks;
- Each element m_i in $M(p_1, p_2)$ represents a matching pair between b_i and $b_j^{'}$;
- $sim(m_i)$ defines the quantitative similarity score between two matching basic blocks.

Therefore, the problem can be transformed into two subtasks: 1) discover $sim(m_i)$ that quantitatively measures the similarity between two basic blocks; 2) find the optimal matching between two sets of basic blocks $M(p1, p2)$.

B. Assumptions

We list the following assumptions on the given inputs:

- Only stripped binaries, no source or symbol information is given. COTS binaries are often stripped and malicious binaries do not carry symbols for obvious reasons.
- Binaries are not packed, but can be transformed with different compiler optimization techniques, which can lead to distinctive binaries even with the same source code input. For packed malware binaries, we assume they are first unpacked before they are presented to our tool.
- Two input binaries are for the same architecture. So far DeepBinDiff supports x86 binaries since they are the most prevalent in real world. DeepBinDiff could be extended to handle cross-architecture diffing via analysis on an Intermediate Representation (IR) level. We leave it as future work.

III. Approach Overview

Figure 1 delineates the system architecture of DeepBinDiff. Red squares represent generated intermediate data during analysis. As shown, the system takes as input two binaries and outputs the basic block level diffing results. The system solves the two tasks mentioned in Section II-A by using two major techniques. First, to calculate $sim(m_i)$ that quantitatively measures basic block similarity, DeepBinDiff embraces an unsupervised learning approach to generate embeddings and utilizes them to efficiently calculate the similarity scores between basic blocks. Second, our system uses a k-hop greedy matching algorithm to generate the matching $M(p_1, p_2)$.

The whole system consists of three major components: 1) pre-processing; 2) embedding generation and 3) code diffing. Pre-processing, which can be further divided into two subcomponents: CFG generation and feature vector generation, is responsible for generating two pieces of information: inter-procedural control-flow graphs (ICFGs) and feature vectors for basic blocks. Once generated, the two results are sent to embedding generation component that utilizes TADW technique [48] to learn the graph embeddings for each basic block. DeepBinDiff then makes use of the generated basic block embeddings and performs a k-hop greedy matching algorithm for code diffing at basic block level.

IV. Pre-processing

Pre-processing analyzes binaries and produces inputs for embedding generation. More specifically, it produces inter-procedural CFGs for binaries and applies a token embedding generation model to generate embeddings for each token (opcode and operands). These generated token embeddings are further transformed into basic block level feature vectors.

A. CFG Generation

By combining the call graph with the control-flow graphs of each function, DeepBinDiff leverages IDA pro [5] to extract basic block information, and generates an inter-procedural CFG (ICFG) that provides program-wide contextual information. This information is particularly useful when differentiating semantically similar basic blocks in dissimilar contexts.

B. Feature Vector Generation

Besides the control dependency information carried by ICFGs, DeepBinDiff also takes into account the semantic information by generating feature vector for each basic block. The whole process consists of two subtasks: token embedding generation and feature vector generation. More specifically, we first train a token embedding model derived from Word2Vec algorithm [42], and then use this model to generate token (opcode or operand) embeddings. And eventually we generate feature vectors for basic blocks from token embeddings. Figure 2 shows the four major steps for the feature vector generation process.

Random Walks. When distilling semantics of each token, we would like to make use of the instructions around it as its context. Therefore, we need to serialize ICFGs to extract control flow dependency information. As depicted in Step 1 in Figure 2, we generate random walks in ICFGs so that each walk contains one possible execution path of the binary. To ensure the completeness of basic block coverage, we configure the walking engine so that every basic block is guaranteed to be contained by at least 2 random walks. Further, each random walk is set to have a length of 5 basic blocks to carry enough control flow information. Then, we put random walks together to generate a complete instruction sequence for training.

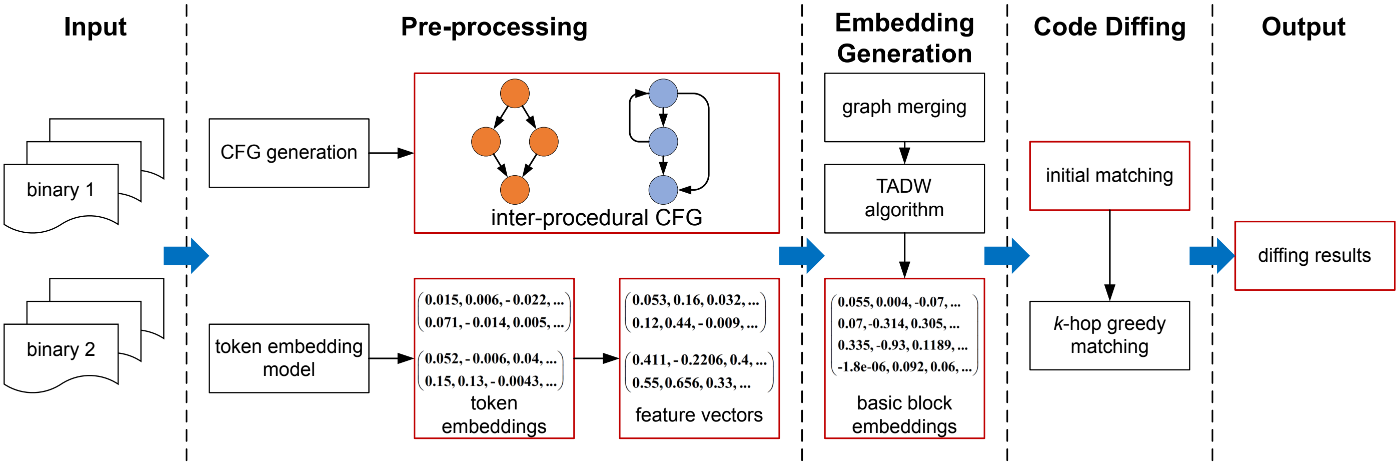


Fig. 1: Overview of DeepBinDiff.

Normalization. Before sending the instruction sequence to train our Word2Vec model, the serialized codes may still contain some differences due to various compilation choices. To refine the code, DeepBinDiff adopts a normalization process shown as Step 2 in Figure 2. Our system conducts the normalization using the following rules : 1) all numeric constant values are replaced with string 'im'; 2) all general registers are renamed according to their lengths; 3) pointers are replaced with string 'ptr'. Notice that we do not follow InnerEye [58] where all the string literals are replaced with <STR>, because the string literals can be useful to distinguish different basic blocks.

Model Training. DeepBinDiff considers the generated random walks as sentence for our modified version of Word2Vec algorithm [42] and learns the token embeddings by training a token embedding model to the normalized random walks. Note that model training is only a one-time effort.

A word embedding is simply a vector, which is learned from the given articles to capture the contextual semantic meaning of the word. There exist multiple methods to generate vector representations of words including the most popular Continuous Bag-of-Words model (CBOW) and Skip-Gram model proposed by Mikolov et al. [42]. Here we utilize the CBOW model which predicates target from its context.

Given a sequence of training words $w_1, w_2, ..., w_t$, the objective of the model is to maximize the average log probability $J(w)$ as shown in Equation 1

$$J(w) = \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c} \log p(w_{t+j} | w_t) \quad (1)$$

where c is the sliding window for context and $p(w_{t+j}|w_t)$ is the softmax function defined as Equation 2.

$$p(w_k \in C_t | w_t) = \frac{\exp(v_{w_t}^T v_{w_k})}{\sum_{w_i \in C_t} \exp(v_{w_t}^T v_{w_i})} \quad (2)$$

where v_{w_t}, v_{w_k} and v_{w_i} are the vector representations of w_t, w_k and w_i. To further improve the efficiency of the computation, Word2Vec adopts the hierarchical softmax as a computationally efficient approximation [42].

To train the token embedding generation model, we modify the Word2Vec CBOW model which uses words around a target word as context. In our case, we consider each token (opcode or operand) as word, normalized random walks on top of ICFGs to be sentences and instructions around each token as its context. For example, step 3 in Figure 2 shows that the current token is *cmp* (shown in red), so we use one instruction before and another instruction after (shown in green) in the random walk as the context. If the target instruction is at the block boundary (e.g., first instruction in the block), then only one adjacent instruction will be considered as its context.

Feature Vector Generation. Feature vectors for basic blocks are then generated based on the token embeddings. Since each basic block could contain multiple instructions and each instruction in turn involves one opcode and potentially multiple operands, we calculate the average of the operand embeddings, concatenate with the opcode embedding to generate instruction embedding, and further sum up the instructions within the block to formulate the block feature vector.

Additionally, because of different compilers and optimizations, instructions may not be of equal importance in terms of diffing. For instance, GCC v5.4 compiler uses 3 *mov* instructions to set up a *printf* call under O0 optimization but uses only 1 *mov* instruction with O3 optimization. In this case, *mov* instruction is less important than *call* instruction during matching. To tackle this problem, DeepBinDiff adopts a weighting strategy to adjust the weights of opcodes based on the opcodes importance with TF-IDF model [50]. The calculated weight indicates how important one instruction is to the block that contains it in all the blocks within two input binaries.

Particularly, for an instruction in_i containing an opcode p_i and a set of k (could be zero) operands Set_{t_i}, we model the instruction embedding as the concatenation of two terms: 1) opcode embedding $embed_{p_i}$ multiplies by its TF-IDF weight $weight_{p_i}$; 2) the average of operand embeddings $embed_{t_{i_n}}$. Therefore, for a block $b = \{in_1, in_2, .., in_j\}$ containing j instructions, its feature vector FV_b is the sum of its instruction embeddings, as depicted in Equation 3.

$$FV_b = \sum_{i=1}^{j} (embed_{p_i} * weight_{p_i} || \frac{1}{|Set_{t_i}|} * \sum_{n=1}^{k} embed_{t_{in}}) \quad (3)$$

Our token embedding generation model shares some similarity with Asm2Vec [23], which also uses instructions around a target token as context. Nonetheless, our model has a fundamentally different design goal. DEEPBINDIFF learns token embeddings via program-wide random walks while Asm2Vec is trying to learn function and token embeddings at the same time and only within the function. Therefore, we choose to modify Word2Vec CBOW model while Asm2Vec leverages the PV-DM model.

V. EMBEDDING GENERATION

Based on the ICFGs and feature vectors generated in the prior steps, basic block embeddings are generated so that similar basic blocks can be associated with similar embeddings. To do so, DEEPBINDIFF first merges the two ICFGs into one graph and then models the problem as a network representation learning problem to generate basic block embeddings using Text-associated DeepWalk algorithm (TADW) [56].

Since the most important building basic block for this component is the TADW algorithm, we first describe the algorithm in detail and present how basic block embeddings are generated. Then, we justify the need of graph merging and report how DEEPBINDIFF accomplishes it.

A. TADW algorithm

Text-associated DeepWalk [56] is an unsupervised graph embedding learning technique. As the name suggests, it is an improvement over the DeepWalk algorithm [48].

DeepWalk algorithm is an online graph embedding learning algorithm that considers a set of short truncated random walks as a corpus in language modeling problem, and the graph vertices as its own vocabulary. The embeddings are then learned using random walks on the vertices in the graph. Accordingly, vertices that share similar neighbours will have similar embeddings. It excels at learning the contextual information from a graph. Nevertheless, it does not consider the node features during analysis.

As a result, Yang at el. [56] propose an improved algorithm called Text-associated DeepWalk (TADW), which is able to incorporate features of vertices into the network representation learning process. They prove that DeepWalk is equivalent to factorizing a matrix $M \in R^{|v| \times |v|}$ where each entry M_{ij} is logarithm of the average probability that vertex v_i randomly walks to vertex v_j in fixed steps. This discovery further leads to TADW algorithm depicted in Figure 3. It shows that it is possible to factorize the matrix M into the product of three matrices: $W \in R^{k \times |v|}$, $H \in R^{k \times f}$ and a text feature $T \in R^{f \times |v|}$. Then, W is concatenated with HT to produce $2k$-dimensional representations of vertices (embeddings).

B. Graph Merging

Since we have two ICFGs (one for each binary), the most intuitive way is to run TADW twice for the two graphs. However, this method has two drawbacks. First, it is less efficient to perform matrix factorization twice. Second, generating embeddings separately can miss some important indicators for similarity detection.

For example, Figure 4 shows two ICFGs and each has a basic block that calls *fread* and another basic block that has a reference to string *'hello'*. Ideally, these two pairs of nodes ('a' and '1', 'd' and '3') are likely to match. However, in practice, the feature vectors of these basic blocks may not look very similar as one basic block could contain multiple instructions while the call or the string reference is just one of them. Besides, the two pairs also have different contextual information (node 'a' has no incoming edge but '1' does). As a result, TADW may not generate similar embeddings for the two pairs of nodes.

We propose graph merging to alleviate this problem. That is, the two ICFGs are merged and TADW runs only once on the merged graph. Particularly, DEEPBINDIFF extracts the string references and detects external library calls and system calls. Then, it creates virtual nodes for strings and library functions, and draws edges from the callsites to these virtual nodes. Hence, two graphs are merged into one on terminal virtual nodes. By doing so, node 'a' and '1' have at least one common neighbor, which boosts the similarity between them. Further, neighbors of node 'a' and '1' also have higher similarity since they share similar neighbors. Moreover, since we only merge the graphs on terminal nodes, the original graph structures stay unchanged.

C. Basic Block Embeddings

With the merged graph, DEEPBINDIFF leverages TADW algorithm to generate basic block embeddings. More specifically, DEEPBINDIFF feeds the merged graph and the basic block feature vectors into TADW for multiple iterations of optimization. The algorithm factorizes the matrix M into three matrices by minimizing the loss function depicted in Equation 4 using Alternating Least Squares (ALS) algorithm [34]. It stops when the loss converges or after a fixed n iterations.

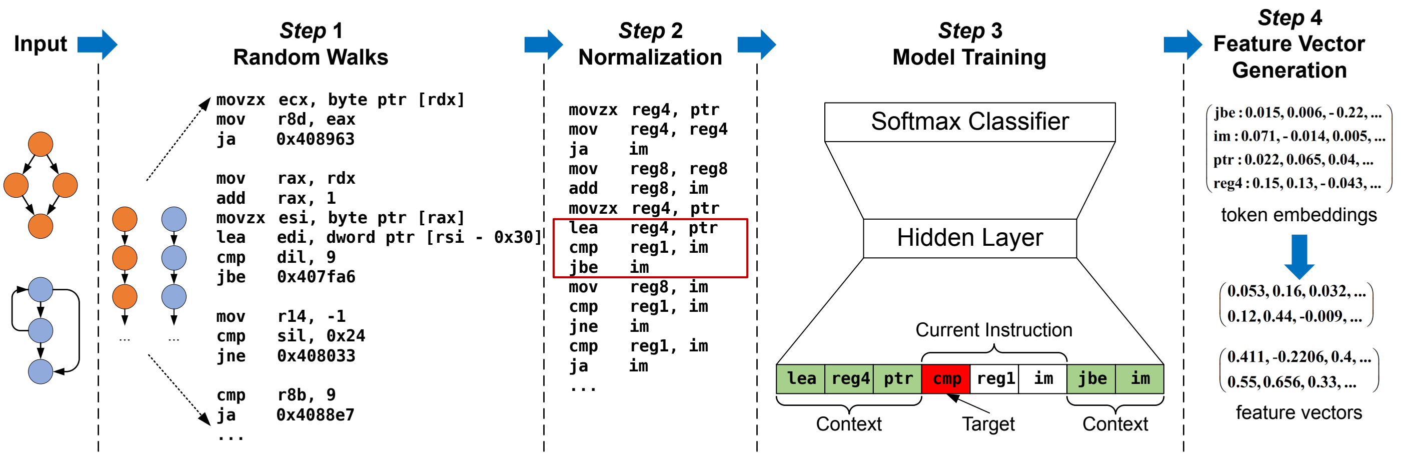
$$\min_{W,H} ||M - W^T HT||_F^2 + \frac{\lambda^2}{2}(||W||_F^2 + ||H||_F^2) \quad (4)$$

On that account, each generated basic block embedding contains not only the semantic information about the basic block itself, but also the information from the ICFG structure.

VI. CODE DIFFING

DEEPBINDIFF then performs code diffing. The goal is to find a basic block level matching solution that maximizes the similarity for the two input binaries. One spontaneous choice is to perform linear assignment based on basic block embeddings to produce an optimal matching. This method

[2] λ is a harmonic factor to balance two components



```
movzx ecx, byte ptr [rdx]
mov   r8d, eax
ja    0x408963

mov   rax, rdx
add   rax, 1
movzx esi, byte ptr [rax]
lea   edi, dword ptr [rsi - 0x30]
cmp   dil, 9
jbe   0x407fa6

mov   r14, -1
cmp   sil, 0x24
jne   0x408033

cmp   r8b, 9
ja    0x4088e7
...
```

```
movzx reg4, ptr
mov   reg4, reg4
ja    im
mov   reg8, reg8
add   reg8, im
movzx reg4, ptr
lea   reg4, ptr
cmp   reg1, im
jbe   im
mov   reg8, im
cmp   reg1, im
jne   im
cmp   reg1, im
ja    im
...
```

Fig. 2: Basic Block Feature Vector Generation.

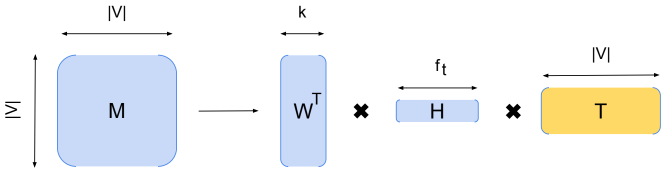


Fig. 3: TADW

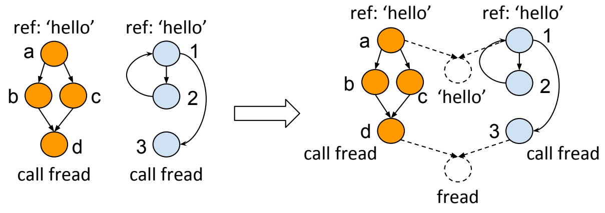


Fig. 4: Graph Merging

suffers from two major limitations. First, linear assignment can be inefficient as binaries could contain enormous amount of blocks. Second, although embeddings include some contextual information, linear assignment itself does not consider any graph information. Thus, it is still likely to make mistakes when matching very similar basic blocks. A possible improvement is to conduct linear assignment at two levels. Rather than matching basic blocks directly, we could match functions first by generating function level embeddings. Then, basic blocks within the matched functions can be further matched using basic block embeddings. This approach, however, can be severely thwarted by compiler optimizations that alter the function boundary such as function inlining.

A. *k-Hop Greedy Matching*

To address this problem, we introduce a k-hop greedy matching algorithm. The high-level idea is to benefit from the ICFG contextual information and find matching basic blocks based on the similarity calculated from basic block embeddings within the k-hop neighbors of already matched ones.

As presented in Algorithm 1, the initial matching set $Set_{initial}$ are computed by using the virtual nodes during the graph merging. Starting from the virtual nodes, the algorithm calls $ComputeInitialSet()$ in Ln.2 to extract direct neighbors of the virtual nodes and finds best matching pairs among the neighbors based on embeddings. For example, node 'a' and '1' in Figure 4 will become one of the pairs in initial set.

Starting from there, the algorithm loops and explores the neighbors of the already matched pairs in $GetKHopNeighbors()$ in Ln.7-8 by exploring the merged ICFG. It then sorts the similarities between neighbor basic blocks and picks the pair that bears highest similarity with a predefined threshold t of 0.6 by calling $FindMaxUnmatched()$ in Ln.9. This process is repeated until all k-hop neighbors of matched pairs are explored and matched. Note that after the loop, there may still exist unmatched basic blocks due to unreachable code (dead code) or low similarity within k-hop neighbors. Our method then performs linear assignment using Hungarian algorithm [35] and finds the optimal matching among them in Ln.16. Please note that we only use the Hungarian algorithm occasionally for small numbers of unmatched basic blocks, hence, its impact on accuracy and efficiency is very minimal. Finally, it returns $Set_{matched}$ as the matching result, Set_i as insertion basic blocks and Set_d as deletion basic blocks. [3] We set k to 4. More details about parameter selection is presented in Section VII.

VII. Evaluation

In this section, we evaluate DeepBinDiff with respect to its effectiveness and efficiency for two different diffing scenarios: cross-version and cross-optimization-level. To our best knowledge, this is the first research work that comprehensively examines the effectiveness of program-wide binary diffing tools under the cross-version setting. Furthermore, we conduct a case study to demonstrate the usefulness of DeepBinDiff in real-world vulnerability analysis.

A. *Experimental Setup, Datasets & Baseline Techniques*

Our experiments are performed on a moderate desktop computer running Ubuntu 18.04LTS operating system with

[3] Insertions and deletions can happen when diffing between two different versions of the binary.

Algorithm 1 k-Hop Greedy Matching Algorithm

```
$Set_{virtualnodes} \leftarrow$ {virtual nodes from merged graphs}
$Set_{initial} \leftarrow$ ComputeInitialSet($Set_{virtualnodes}$)
$Set_{matched} \leftarrow Set_{initial}$; $Set_{currPairs} \leftarrow Set_{initial}$

while $Set_{currPairs}$ != empty do
    $(node_1, node_2) \leftarrow Set_{currPairs}.pop()$
    $nb_{node_1} \leftarrow$ GetKHopNeighbors($node_1$)
    $nb_{node_2} \leftarrow$ GetKHopNeighbors($node_2$)
    $newPair \leftarrow$ FindMaxUnmatched($nb_{node_1}$, $nb_{node_2}$)
    if $newPair$ != null then
        $Set_{matched} \leftarrow Set_{matched} \cup newPair$
        $Set_{currPairs} \leftarrow Set_{currPairs} \cup newPair$
    end if
end while
$Set_{unreached} \leftarrow$ {basic blocks that are not yet matched}
$\{Set_m, Set_i, Set_d\} \leftarrow$ LinearAssign($Set_{unreached}$)
$Set_{matched} \leftarrow Set_{matched} \cup Set_m$
output $Set_{matched}, Set_i, Set_d$ as the diffing result
```

Intel Core i7 CPU, 16GB memory and no GPU. The feature vector generation and basic block embedding generation components in DeepBinDiff are expected to be significantly faster if GPUs are utilized since they are built upon deep learning models.

Datasets. To thoroughly evaluate the effectiveness of DeepBinDiff, we utilize three popular binary sets - Coreutils [2], Diffutils [3] and Findutils [4] with a total of 113 binaries. Multiple different versions of the binaries (5 versions for Coreutils, 4 versions for Diffutils and 3 versions of Findutils) are collected with wide time spans between the oldest and newest versions (13, 15, and 7 years respectively). This setting ensures that each version has enough distinctions so that binary diffing results among them are meaningful and representative.

We then compile them using GCC v5.4 with 4 different compiler optimization levels (O0, O1, O2 and O3) in order to produce binaries equipped with different optimization techniques. This dataset is to show the effectiveness of DeepBinDiff in terms of cross-optimization-level diffing. We randomly select half of the binaries in our dataset for token embedding model training.

To demonstrate the effectiveness with C++ programs, we also collect 2 popular open-source C++ projects LSHBOX [8] and indicators [6], which contain plenty of virtual functions, from GitHub. The two projects include 4 and 6 binaries respectively. In LSHBOX, the 4 binaries are psdlsh, rbslsh, rhplsh and thlsh. And in indicators, there exist 6 binaries - blockprogressbar, multithreadedbar, progressbarsetprogress, progressbartick, progressspinner and timemeter. For each project, we select 3 major versions and compile them with the default optimization levels for testing.

Finally, we leverage two different real-world vulnerabilities in a popular crypto library OpenSSL [9] for a case study to demonstrate the usefulness of DeepBinDiff in practice.

Baseline Techniques. With the aforementioned datasets, we compare DeepBinDiff with two state-of-the-art baseline techniques (Asm2Vec [23] and BinDiff [10]). Note that Asm2Vec is designed only for function level similarity detection. We leverage its algorithm to generate embeddings, and use the same k-hop greedy matching algorithm to perform diffing. Therefore, we denote it as Asm2Vec+k-Hop. Also, to demonstrate the usefulness of the contextual information, we modify DeepBinDiff to exclude contextual information and only include semantics information for embedding generation, shown as DeepBinDiff-Ctx.

As mentioned in Section I, another state-of-the-art technique InnerEye [58] has scalability issue for binary diffing. Hence, we only compare it with DeepBinDiff using a set of small binaries in Coreutils. Note that we also apply the same k-hop greedy matching algorithm in InnerEye, and denote it as InnerEye+k-Hop.

B. Ground Truth Collection

For the purpose of evaluation, we rely on source code level matching and debug symbol information to conservatively collect ground truth that indicates how basic blocks from two binaries should match.

Particularly, for two input binaries, we first extract source file names from the binaries and use Myers algorithm [46] to perform text based matching for the source code in order to get the line number matching. To ensure the soundness of our extracted ground truth, 1) we only collect identical lines of source code as matching but ignore the modified ones; 2) our ground truth collection conservatively removes the code statements that lead to multiple basic blocks. Therefore, although our source code matching is by no means complete, it is guaranteed to be sound. Once we have the line number mapping between the two binaries, we extract debug information to understand the mapping between line numbers and program addresses. Eventually, the ground truth is collected by examining the basic blocks of the two binaries containing program addresses that map to the matched line numbers.

Example. To collect the ground truth for basic block matching between v5.93 and v8.30 of Coreutils binary `chown`, we first extract the names of the source files and perform text-based matching between the corresponding source files. By matching the source files `chown.c` in the two versions, we know Ln.288 in v5.93 should be matched to Ln.273 in v8.30. Together with the debug information extracted, a matching between address `0x401cf8` in v5.93 and address `0x4023fc` in v8.30 can be established. Finally, we generate basic blocks for the two binaries. By checking the basic block addresses, we know basic block 3 in v5.93 should be matched to basic block 13 in v8.30.

C. Effectiveness

With the datasets and ground truth information, we evaluate the effectiveness of DeepBinDiff by performing diffing between binaries across different versions and optimization levels, and comparing the results with the baseline techniques.

Evaluation Metrics. We use precision and recall metrics to measure the effectiveness of the diffing results produced by diffing tools. The matching result M from DeepBinDiff can be presented as a set of basic block matching pairs with a length of x as Equation 5. Similarly, the ground truth information G for the two binaries can be presented as a set of basic block matching pairs with a length of y as Equation 6.

$$M = \{(m_1, m_1^{'}), (m_2, m_2^{'}), ..., (m_x, m_x^{'})\} \tag{5}$$

$$G = \{(g_1, g_1^{'}), (g_2, g_2^{'}), ..., (g_y, g_y^{'})\} \tag{6}$$

We then introduce two subsets, M_c and M_u, which represent correct matching and unknown matching respectively. Correct match $M_c = M \cap G$ is the intersection of our result M and ground truth G. It gives us the correct basic block matching pairs. Unknown matching result M_u represents the basic block matching pairs in which no basic block ever appears in ground truth. Thus, we have no idea whether these matching pairs are correct. This could happen because of the conservativeness of our ground truth collection process. Consequently, $M - M_u - M_c$ portrays the matching pairs in M that are not in M_c nor in M_u, therefore, all pairs in $M - M_u - M_c$ are confirmed to be incorrect matching pairs.

Once M and G are formally presented, the precision metric presented in Equation 7 gives the percentage of correct matching pairs among all the known pairs (correct and incorrect).

$$Precision = \frac{||M \cap G||}{||M \cap G|| + ||M - M_u - M_c||} \tag{7}$$

The recall metric shown in Equation 8 is produced by dividing the size of intersection of M and G with the size of G. This metric shows the percentage of ground truth pairs that are confirmed to be correctly matched.

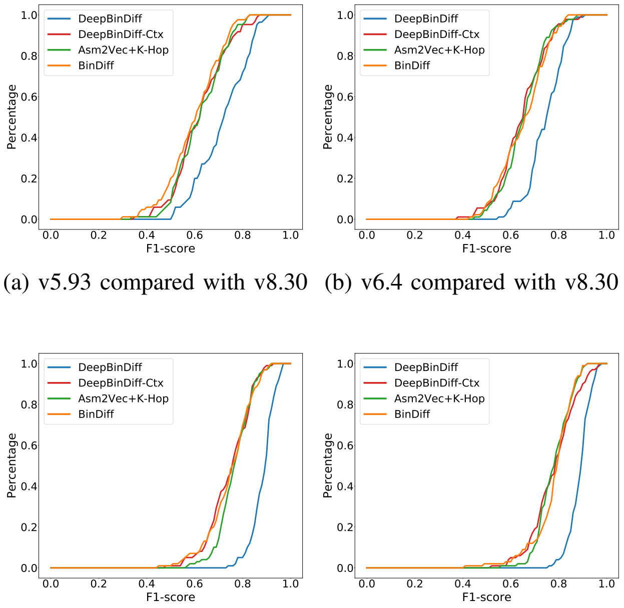
$$Recall = \frac{||M \cap G||}{||G||} \tag{8}$$

Cross-version Diffing. In this experiment, we benchmark the performance of DeepBinDiff, BinDiff, DeepBinDiff-Ctx and Asm2Vec+k-Hop using different versions of binaries (with the default O1 optimization level) in Coreutils, Diffutils and Findutils. We report the average recall and precision results for each tool under different experimental settings in Table I.

As shown, DeepBinDiff outperforms DeepBinDiff-Ctx, Asm2Vec+k-Hop and BinDiff across all versions of the three datasets in terms of recall, especially when the two diffed versions have a large gap. For example, for Coretuils diffing between v5.93 and v8.30, DeepBinDiff improves the recall by 11.4%, 10% and 36.9% over DeepBinDiff-Ctx, Asm2Vec+k-Hop and BinDiff. Also, we can observe that Asm2Vec+k-Hop and DeepBinDiff-Ctx, which carry the semantic information for tokens, has better recall than the defacto commercial tool BinDiff. This result shows that including semantic information during analysis can indeed improve the effectiveness. Moreover, the performance difference between DeepBinDiff-Ctx and DeepBinDiff shows that contextual information can help boost the quality of diffing results by a large margin.

Notice that although DeepBinDiff outperforms BinDiff in terms of precision for most binaries, BinDiff can produce higher precision in certain cases, such as the diffing between v5.93 and v8.30. We investigate the details and see that BinDiff has a very conservative matching strategy. It usually only matches the basic blocks with very high similarity score and leaves the other basic blocks unmatched. Therefore, BinDiff generates much shorter matching list than DeepBinDiff that uses k-hop greedy matching to maximize the matching. Nonetheless, DeepBinDiff can achieve a higher average precision by a large margin.

Figure 5 and 6 further present the Cumulative Distribution Function (CDF) figures of the F1-scores for three diffing techniques on Coreutils binaries in both cross-version binary diffing and cross-optimization-level diffing settings. Again, from the CDF figures we can see that DeepBinDiff-Ctx, Asm2Vec+k-Hop and BinDiff have somewhat similar F1-scores, while DeepBinDiff performs much better. In a nutshell, DeepBinDiff can exceed three baseline techniques by large margins with respect to cross-version binary diffing.



(a) v5.93 compared with v8.30 (b) v6.4 compared with v8.30

(c) v7.6 compared with v8.30 (d) v8.1 compared with v8.30

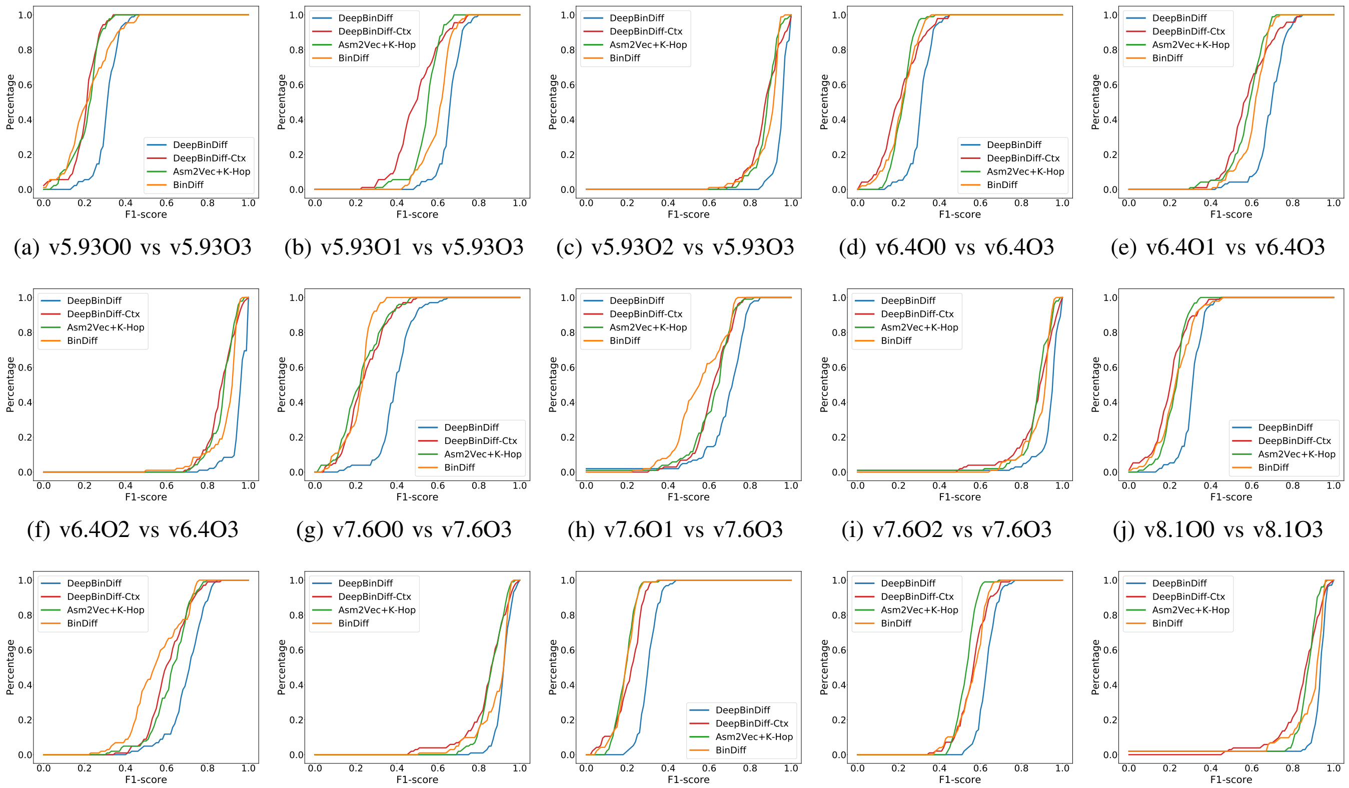
Fig. 5: Cross-version Diffing F1-score CDF

Cross-optimization-level Diffing. We then conduct experiments to measure the effectiveness for cross-optimization-level setting. Particularly, each binary is compiled and diffed three times (O0 vs O3, O1 vs O3, O2 vs O3) and average recall and precision results are reported in Table II.

As shown, DeepBinDiff outperforms DeepBinDiff-Ctx, Asm2Vec+k-Hop and BinDiff for most of the settings with respect to recall as well as precision. There exist only two exception for recall. In Diffutils v3.1 O1 vs O3, DeepBinDiff-Ctx has a recall rate of 0.826, while DeepBinDiff obtains a similar recall of 0.825. And in Diffutils v3.6 O1 vs O3, Asm2Vec+k-Hop has a slightly better recall than DeepBinDiff. This is because there are only 4 binaries in Difftuils and most of them are small. In this special case, program-wide structural information becomes less useful. As a result, DeepBinDiff-Ctx and DeepBinDiff also share

TABLE I: Cross-version Binary Diffing Results

		Recall				**Precision**			
		BinDiff	Asm2Vec+k-Hop	DeepBinDiff-Ctx	DeepBinDiff	BinDiff	Asm2Vec+k-Hop	DeepBinDiff-Ctx	DeepBinDiff
Coreutils	v5.93 - v8.30	0.506	0.635	0.622	**0.693**	**0.775**	0.613	0.611	0.761
	v6.4 - v8.30	0.572	0.654	0.656	**0.748**	0.784	0.643	0.645	**0.805**
	v7.6 - v8.30	0.748	0.771	0.752	**0.867**	0.771	0.746	0.751	**0.904**
	v8.1 - v8.30	0.756	0.785	0.788	**0.872**	0.821	0.765	0.755	**0.903**
	Average	0.646	0.711	0.705	**0.795**	0.788	0.692	0.691	**0.843**
Diffutils	v2.8 - v3.6	0.354	0.741	0.733	**0.778**	0.662	0.742	0.752	**0.783**
	v3.1 - v3.6	0.905	0.933	0.915	**0.961**	0.949	0.931	0.932	**0.949**
	v3.4 - v3.6	0.925	0.955	0.947	**0.972**	**0.964**	0.94	0.935	0.941
	Average	0.728	0.876	0.865	**0.904**	0.858	0.871	0.873	**0.891**
Findutils	v4.2.33 - v4.6.0	0.511	0.688	0.673	**0.725**	0.631	0.704	0.711	**0.748**
	v4.4.1 - v4.6.0	0.736	0.813	0.821	**0.911**	**0.898**	0.877	0.855	0.885
	Average	0.624	0.751	0.747	**0.818**	0.765	0.791	0.783	**0.817**



(a) v5.93O0 vs v5.93O3 (b) v5.93O1 vs v5.93O3 (c) v5.93O2 vs v5.93O3 (d) v6.4O0 vs v6.4O3 (e) v6.4O1 vs v6.4O3

(f) v6.4O2 vs v6.4O3 (g) v7.6O0 vs v7.6O3 (h) v7.6O1 vs v7.6O3 (i) v7.6O2 vs v7.6O3 (j) v8.1O0 vs v8.1O3

(k) v8.1O1 vs v8.1O3 (l) v8.1O2 vs v8.1O3 (m) v8.30O0 vs v8.30O3 (n) v8.30O1 vs v8.30O3 (o) v8.30O2 vs v8.30O3

Fig. 6: Cross-optimization-level Diffing F1-score CDF

similar recall rates for these binaries. Still, DeepBinDiff could outperform the baseline techniques for all other settings, even for Diffutils. For precision, because of the conservativeness of BinDiff, it could achieve slightly better in some cases. However, DeepBinDiff still outperforms BinDiff for most binaries as shown in average results. Also, the evaluation shows that cross-optimization-level binary diffing is more difficult than cross-version diffing since the recall and precision rates are lower, because the compiler optimization techniques could greatly transform the binaries.

Binary Size vs. Accuracy. We further examine the relationship between binary size and accuracy to see how the effectiveness of our tool could be affected by the binary size. Intuitively, a larger binary tends to have more basic blocks, and therefore, resulting in a more complicated ICFG. Although a very complicated ICFG may reduce our chance of matching the correct basic blocks within a fix k hops, our algorithm should

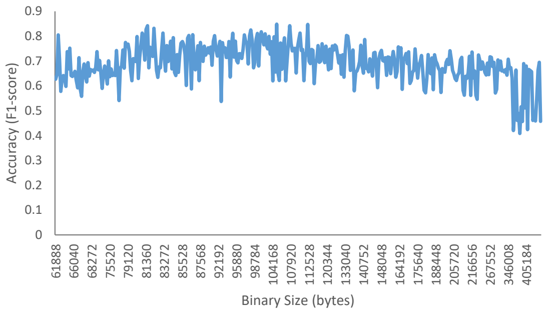


Fig. 7: Binary Size vs. Accuracy

TABLE II: Cross-optimization-level Binary Diffing Results

		Recall				Precision			
		BinDiff	ASM2VEC+k-HOP	DEEPBINDIFF-CTX	DEEPBINDIFF	BinDiff	ASM2VEC+k-HOP	DEEPBINDIFF-CTX	DEEPBINDIFF
Coreutils	v5.93 O0 - O3	0.176	0.155	0.163	**0.311**	0.291	0.211	0.235	**0.315**
	v5.93 O1 - O3	0.571	0.545	0.497	**0.666**	0.638	0.544	0.515	**0.681**
	v5.93 O2 - O3	0.837	0.911	0.912	**0.975**	0.944	0.859	0.813	**0.955**
	v6.4 O0 - O3	0.166	0.201	0.221	**0.391**	0.262	0.235	0.218	**0.301**
	v6.4 O1 - O3	0.576	0.579	0.599	**0.703**	0.646	0.563	0.579	**0.709**
	v6.4 O2 - O3	0.838	0.893	0.871	**0.949**	0.947	0.851	0.852	**0.953**
	v7.6 O0 - O3	0.156	0.225	0.213	**0.391**	0.331	0.291	0.277	**0.40**
	v7.6 O1 - O3	0.484	0.618	0.653	**0.672**	0.674	0.599	0.633	**0.761**
	v7.6 O2 - O3	0.840	0.903	0.911	**0.946**	**0.942**	0.861	0.855	0.941
	v8.1 O0 - O3	0.166	0.169	0.155	**0.305**	0.334	0.291	0.287	**0.351**
	v8.1 O1 - O3	0.480	0.625	0.602	**0.679**	0.677	0.612	0.621	**0.721**
	v8.1 O2 - O3	0.835	0.871	0.881	**0.915**	**0.942**	0.828	0.833	0.912
	v8.30 O0 - O3	0.135	0.144	0.151	**0.292**	0.285	0.275	0.261	**0.315**
	v8.30 O1 - O3	0.508	0.521	0.506	**0.602**	0.620	0.493	0.471	**0.665**
	v8.30 O2 - O3	0.842	0.875	0.856	**0.956**	**0.954**	0.843	0.853	0.908
	Average	0.507	0.549	0.546	**0.65**	0.632	0.557	0.553	**0.659**
Diffutils	v2.8 O0 - O3	0.236	0.321	0.327	**0.421**	0.454	0.392	0.381	**0.575**
	v2.8 O1 - O3	0.467	0.781	0.745	**0.833**	0.713	0.751	0.765	**0.831**
	v2.8 O2 - O3	0.863	0.955	0.961	**0.981**	0.953	0.939	0.927	**0.971**
	v3.1 O0 - O3	0.125	0.211	0.208	**0.379**	0.251	0.217	0.239	**0.314**
	v3.1 O1 - O3	0.633	0.819	**0.826**	0.825	0.655	0.652	0.641	**0.771**
	v3.1 O2 - O3	0.898	0.905	0.901	**0.939**	0.966	0.921	0.936	**0.967**
	v3.4 O0 - O3	0.171	0.199	0.187	**0.355**	0.271	0.233	0.249	**0.336**
	v3.4 O1 - O3	0.577	0.711	0.737	**0.761**	0.708	0.681	0.697	**0.715**
	v3.4 O2 - O3	0.903	0.907	0.901	**0.933**	0.953	0.941	0.956	**0.967**
	v3.6 O0 - O3	0.159	0.228	0.232	**0.373**	0.247	0.215	0.227	**0.292**
	v3.6 O1 - O3	0.735	**0.881**	0.871	0.871	0.815	0.817	0.831	**0.861**
	v3.6 O2 - O3	0.919	0.952	0.957	**0.962**	**0.964**	0.925	0.921	0.949
	Average	0.557	0.656	0.654	**0.719**	0.663	0.64	0.648	**0.712**
Findutils	v4.233 O0 - O3	0.144	0.192	0.217	**0.314**	0.225	0.211	0.207	**0.249**
	v4.233 O1 - O3	0.633	0.687	0.696	**0.791**	0.768	0.621	0.633	**0.787**
	v4.233 O2 - O3	0.933	0.951	0.945	**0.981**	0.968	0.936	0.927	**0.985**
	v4.41 O0 - O3	0.084	0.142	0.137	**0.295**	0.133	0.135	0.145	**0.242**
	v4.41 O1 - O3	0.677	0.711	0.696	**0.83**	0.731	0.692	0.678	**0.885**
	v4.41 O2 - O3	0.839	0.917	0.901	**0.945**	0.964	0.952	0.961	**0.962**
	v4.6 O0 - O3	0.075	0.151	0.139	**0.292**	0.132	0.172	0.185	**0.315**
	v4.6 O1 - O3	0.563	0.645	0.627	**0.761**	0.633	0.727	0.705	**0.806**
	v4.6 O2 - O3	0.958	0.935	0.923	**0.957**	0.932	0.914	0.921	**0.957**
	Average	0.545	0.592	0.587	**0.685**	0.609	0.596	0.598	**0.688**

be able to extract more useful contextual information from the ICFG. To evaluate this, we use all the O1 and O3 binaries in our dataset to perform diffing, and record the results.

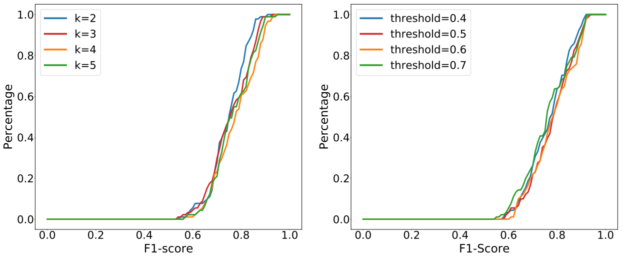
Figure 7 shows the relationship between binary size and accuracy. As shown in the figure, DEEPBINDIFF can perform quite stably regardless of the binary size. For example, DEEPBINDIFF can achieve an average F1-score of 0.7 for binaries with sizes of 60KB, and it can still achieve an average score of 0.65 for binaries that are larger than 400KB.

D. Parameter Selection

Parameter selection, which contains hyperparameters in embedding generation and parameters in k-hop greedy matching, is of great importance to the effectiveness of our system. Hyperparameters such as the number of latent dimensions, the number of walks started per vertex, and the harmonic factor λ have been extensively discussed in DeepWalk [48] and TADW [56]. Therefore, we simply use the default values.

For k-hop greedy matching, we need to tune two important parameters: the number of hops k and the threshold t for filtering matched pairs in $FindMaxUnmatched()$. In particular, we choose different values for these parameters and use DEEPBINDIFF to perform binary diffing for 95 binaries in CoreUtils version 6.4 and version 8.30 to understand the impact.

Number of Hops. To see how the effectiveness of DEEPBINDIFF can be affected by the number of hops during matching, we choose k to be 2, 3, 4 and 5 (with a fixed $threshold$=0.6), and check the matching results respectively. The F1-score CDF is displayed in Figure 8a. Results show that the average F1-score steadily increases when a bigger k is chosen, and becomes the highest when k=4. This means that when we enlarge the search space up to 4 hops for matching basic block pairs, the chance of finding the right match increases. However, if we try to search within very large space (too many hops), DEEPBINDIFF may incorrectly match unrelated but somehow similar basic blocks. Also, more runtime overhead can be introduced.



(a) k selection

(b) threshold selection

Fig. 8: Parameter Selection

Threshold. We then evaluate how $threshold$, which is to guarantee that every pair of matching blocks should be similar enough to certain degree, could affect the matching results. It is distinctly useful when dealing with basic block deletions as it ensures that DeepBinDiff will not match two basic blocks that are dissimilar but share the same context. Figure 8b illustrates the F1-score CDF when choosing different $threshold$. More specifically, we set $threshold$ to be 0.4, 0.5, 0.6 and 0.7 (with a fixed k=4). As we can observe, the average F1-score reaches the highest when $threshold$=0.6, but starts to decline when choosing 0.7. This indicates that there exist some matched basic block pairs that are transformed by compilers and become less similar to each other. DeepBinDiff is able to tolerate these cases by selecting a proper $threshold$.

E. *Efficiency*

We then evaluate the efficiency of DeepBinDiff, which can be split into four: training time, preprocessing time, embedding generation time and matching time.

Training Time. Training is a one-time effort. We train our token embedding generation model with the binaries in our dataset. We stop the training for each binary when loss converges or it hits 10000 steps. In total, It takes about 16 hours to finish the whole training process. Asm2Vec+k-Hop also needs to train its model, while BinDiff does not need any training time. Note that the training process could be significantly accelerated if GPUs are used.

Preprocessing Time. DeepBinDiff takes only an average of 8.2s to finish the graph generation on one binary using IDA pro. Then, it applies the pre-trained model to generate token embeddings and calculates the feature vectors for each basic block. These two steps take less than 100ms for one binary.

Embedding Generation. The most heavy part of DeepBinDiff is the embedding generation, which utilizes TADW to factorize a matrix. On average, it takes 591s to finish one binary diffing. One way to accelerate the process is to use a more efficient algorithm instead of the ALS algorithm used in TADW. For example, CCD++ [57] is demonstrated to be 40 times faster than ALS.

Matching Time. k-hop greedy matching algorithm is efficient in that it limits the search space by searching only within the k-hop neighbors for the two matched basic blocks. On average, it takes about 42s to finish the matching on average.

Binary Size vs. Runtime. The runtime overhead of DeepBinDiff has a high positive correlation to binary size. This is due the to fact that the two most time-consuming steps in DeepBinDiff, namely embedding generation and matching, are all directly bounded by the sizes of the ICFG, which are largely decided by the binary sizes.

Figure 9 delineates how the binary size can affect the runtime overhead in DeepBinDiff. As we can see, the relationship between binary size and runtime overhead is somewhat linear. For example, it takes less than 100s for diffing two binaries with sizes around 60kb. And for binaries with sizes of 400kb, DeepBinDiff needs about 1000s to finish. Therefore, considering the linear relationship, we can draw a conclusion that DeepBinDiff scales reasonably well with respect to the runtime overhead. Note that the embedding generation component, which is responsible for the majority of runtime overhead, can be significantly accelerated if GPUs are used.

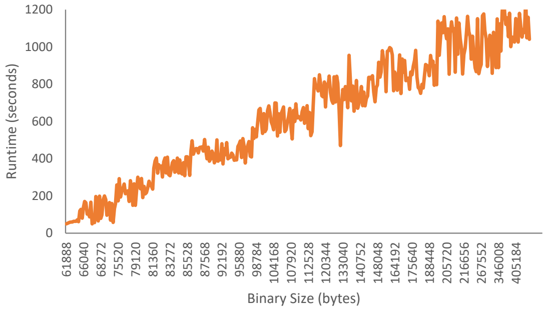


Fig. 9: Binary Size vs. Runtime

F. *Comparison with* InnerEye+k-Hop

InnerEye [58] is a recent research work that also leverages deep learning technique to perform binary code similarity comparison. To evaluate the performance improvement of DeepBinDiff, we compare our system with InnerEye+k-Hop, which is an InnerEye variant that incorporates k-hop greedy matching algorithm for optimal basic block matching. We directly leverage the instruction embedding model and similarity calculation model provided by InnerEye authors [7]. As discussed in Section I, InnerEye has serious scalability issues for whole program binary diffing. Hence, we only select 10 small binaries listed in Table III for testing.

More specifically, we use the tools to perform both cross-version and cross-optimization-level diffing for the selected binaries. The results show that DeepBinDiff can outperform InnerEye+k-Hop in all settings for both recall and precision. For example, our system can achieve an average recall rate of 0.553 for cross-version diffing, while InnerEye+k-Hop can only reach an average recall rate of 0.384. We further compare the two tools from an efficiency perspective. On average, it takes InnerEye+k-Hop 1953 seconds to finish one binary diffing, whereas DeepBinDiff only needs an average time of 62 seconds to finish the diffing.

We carefully investigate the results and observe two major reasons. First, as mentioned in Section I, InnerEye+k-Hop suffers from a serious out-of-vocabulary (OOV) problem. In our experiment, it can only model 85.15% of the instructions correctly, and will simply assign 0 to the embeddings of the unmodeled instructions. Therefore, semantic information of a large number of instructions is missing. Second, the pre-trained model provided by the authors is trained specifically to handle similarity detection between X86 and ARM basic blocks. It is not suitable to perform cross-version and cross-optimization-level binary diffing.

In a nutshell, the scalability issue and OOV problem render InnerEye unsuitable for common binary diffing scenarios. The usage of supervised learning also makes it hard to have balanced and representative dataset and generate a well-trained model for different test settings.

G. *C++ Programs Testing*

As aforementioned, DeepBinDiff relies on IDA Pro for inter-procedural CFG generation. However, C++ programs

TABLE III: Effectiveness Comparison with INNEREYE+k-HOP

	Cross-version (Coreutils v5.93 - v.830)				**Cross-optimization-level (Coreutils v5.93O1 - v5.93O3)**			
	Recall		Precision		Recall		Precision	
	INNEREYE+k-HOP	DEEPBINDIFF	INNEREYE+k-HOP	DEEPBINDIFF	INNEREYE+k-HOP	DEEPBINDIFF	INNEREYE+k-HOP	DEEPBINDIFF
env	0.360	**0.479**	0.375	**0.667**	0.266	**0.626**	0.271	**0.631**
false	0.397	**0.477**	0.403	**0.600**	0.275	**0.605**	0.281	**0.652**
hostid	0.371	**0.533**	0.382	**0.692**	0.282	**0.639**	0.288	**0.701**
printenv	0.405	**0.443**	0.427	**0.604**	0.221	**0.591**	0.224	**0.605**
rmdir	0.398	**0.675**	0.411	**0.753**	0.393	**0.632**	0.396	**0.663**
sync	0.348	**0.499**	0.364	**0.509**	0.334	**0.641**	0.341	**0.641**
true	0.412	**0.829**	0.418	**0.824**	0.256	**0.647**	0.262	**0.643**
tty	0.316	**0.605**	0.323	**0.675**	0.331	**0.563**	0.338	**0.594**
uname	0.471	**0.497**	0.491	**0.587**	0.258	**0.649**	0.269	**0.679**
yes	0.377	**0.616**	0.406	**0.631**	0.252	**0.625**	0.259	**0.632**
Average	0.384	**0.553**	0.401	**0.636**	0.296	**0.620**	0.302	**0.645**

TABLE IV: C++ Programs Testing

		F1-score	
		BinDiff	DEEPBINDIFF
LSHBOX v1.0 vs v3.0	psdlsh	0.556	**0.876**
	rbslsh	0.526	**0.889**
	rhplsh	0.528	**0.895**
	thlsh	0.539	**0.877**
	Average	0.534	**0.884**
LSHBOX v2.0 vs v3.0	psdlsh	0.906	**0.975**
	rbslsh	0.876	**1**
	rhplsh	0.912	**1**
	thlsh	0.873	**1**
	Average	0.892	**0.994**
indicators v1.2 vs v1.4	blockprogressbar	0.943	**0.958**
	multithreadedbar	0.796	**0.843**
	progressbarsetprogress	0.814	**0.873**
	progressbartick	0.659	**0.852**
	progressspinner	0.920	**0.989**
	timemeter	0.637	**0.814**
	Average	0.795	**0.888**
indicators v1.3 vs v1.4	blockprogressbar	0.796	**0.867**
	multithreadedbar	**0.945**	0.929
	progressbarsetprogress	0.814	**0.924**
	progressbartick	0.659	**0.936**
	progressspinner	0.920	**1**
	timemeter	0.637	**0.940**
	Average	0.795	**0.932**

TABLE V: C++ Programs Testing Detailed Numbers

		Total	GT	M_c	M_w
LSHBOX v1.0 vs v3.0	psdlsh	650	89	78	11
	rbslsh	843	108	9	12
	rhplsh	812	107	96	11
	thlsh	968	90	79	11
LSHBOX v2.0 vs v3.0	psdlsh	646	240	234	6
	rbslsh	715	252	252	0
	rhplsh	728	269	269	0
	thlsh	736	249	249	0
indicators v1.2 vs v1.4	blockprogressbar	675	82	79	3
	multithreadedbar	793	83	70	13
	progressbarsetprogress	668	79	69	10
	progressbartick	633	70	62	8
	progressspinner	905	93	92	1
	timemeter	636	70	57	13
indicators v1.3 vs v1.4	blockprogressbar	793	83	72	11
	multithreadedbar	695	85	79	6
	progressbarsetprogress	669	79	73	6
	progressbartick	633	79	68	11
	progressspinner	905	107	107	0
	timemeter	636	70	67	2

may expose additional challenges for generating complete CFGs, and could have negative impact on the performance. To this end, we leverage 2 popular open-source C++ projects LSHBOX [8] and indicators [6] from GitHub, and select 3 major versions from each project to evaluate the effectiveness of DEEPBINDIFF with respect to C++ programs.

We report the experimental results in Table IV. Among all the 10 C++ binaries, we can clearly see that DEEPBINDIFF outperforms BinDiff for 9 of them, and achieves a slightly lower F1-score for only 1 binary (0.929 vs 0.945). When the differences between the testing binaries are bigger, DEEPBINDIFF performs much more stable than BinDiff. We further present the detailed numbers for DEEPBINDIFF during the diffing in Table V. Columns 3-6 represent the total number of basic blocks for the diff'ed binary, the number of basic blocks in our collected ground truth, the number of correctly matched basic blocks, and the number of wrongly matched basic blocks, both by DEEPBINDIFF.

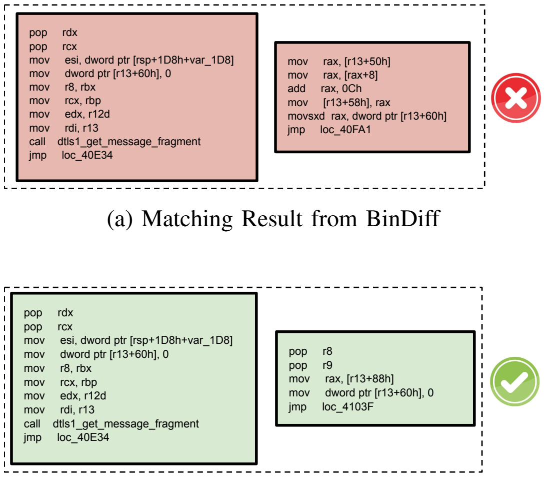
H. Case Study

We further showcase the efficacy of DEEPBINDIFF with real-world vulnerability analysis. Two security vulnerabilities in OpenSSL are used for an in-depth comparison between our tool and the state-of-the-art commercial tool BinDiff.

DTLS Recursion Flaw. The first vulnerability (CVE-2014-0221) is in OpenSSL v1.0.1g and prior versions, and gets fixed in v1.0.1h. It is a Datagram Transport Layer Security (DTLS) recursion flaw vulnerability, which allows attackers to send an invalid DTLS handshake to OpenSSL client to cause recursion and eventually crash. Listing 1 shows the vulnerability along with the patched code. As listed, patching is made to avoid the recursive call by using a *goto* statement (Ln.9-10).

To analyze this vulnerability, we feed a vulnerable version (1.01h) as well as a patched version (1.01g) of OpenSSL into the diffing tools and see if the tools can generate correct matching for all the basic blocks, including the ones that contain the vulnerability and the patch.

The total number of basic blocks is 11734. And our ground truth collection process is able to collect 9591 blocks, achieving a coverage of 82%. Both BinDiff and DEEPBINDIFF can achieve very high F1-scores of 0.947 and 0.962, because of the high similarity between two adjacent versions. More precisely, the two tools can correctly match 8659 and 9172 basic blocks respectively. Despite the high accuracy for both tools, only DEEPBINDIFF is able to match the vulnerability correctly due to the function inlining technique.



```
pop    rdx
pop    rcx
mov    esi, dword ptr [rsp+1D8h+var_1D8]
mov    dword ptr [r13+60h], 0
mov    r8, rbx
mov    rcx, rbp
mov    edx, r12d
mov    rdi, r13
call   dtls1_get_message_fragment
jmp    loc_40E34
```

```
mov     rax, [r13+50h]
mov     rax, [rax+8]
add     rax, 0Ch
mov     [r13+58h], rax
movsxd  rax, dword ptr [r13+60h]
jmp     loc_40FA1
```

(a) Matching Result from BinDiff

```
pop    rdx
pop    rcx
mov    esi, dword ptr [rsp+1D8h+var_1D8]
mov    dword ptr [r13+60h], 0
mov    r8, rbx
mov    rcx, rbp
mov    edx, r12d
mov    rdi, r13
call   dtls1_get_message_fragment
jmp    loc_40E34
```

```
pop    r8
pop    r9
mov    rax, [r13+88h]
mov    dword ptr [r13+60h], 0
jmp    loc_4103F
```

(b) Matching Result from DEEPBINDIFF

Fig. 10: DTLS Recursion Flaw

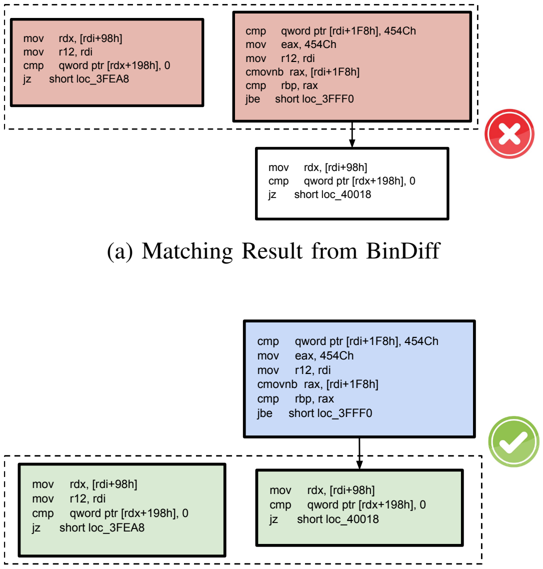
Listing 1: DTLS Recursion Flaw

```
static long dtls1_get_message_fragment() {
    ...
  + redo: ;
    if((frag_len = fragment(s, max, ok)) {
      ....
      if (s->msg_callback) {
          s->msg_callback(0, s->version)
          - return dtls1_get_message_fragment();
          + goto redo;
      }
```

The patched function $dts1\_get\_message\_fragment()$ is inlined into function $dts1\_get\_message()$ in v1.0.1h. As shown in Figure 10a, BinDiff fails to handle this case and incorrectly matches the vulnerable function in v1.0.1h with its caller in v1.0.1g, leaving $dts1\_get\_message()$ in v1.0.1g unmatched. Therefore, it could not match the basic block containing the recursive call to the basic block that has the *goto* statement. Instead, it mistakenly matches the basic block to another similar basic block but with completely different context. Meanwhile, DEEPBINDIFF finds the correct matching shown in Figure 10b by considering both the semantics and the program-wide structural information.

This real-world case study shows that DEEPBINDIFF is very useful even when BinDiff can also achieve a very high accuracy. The unique design of our system makes it more accurate in harder scenarios that include function inline.

Memory Boundary Checking Failure. The second vulnerability (CVE-2016-6308) exists in OpenSSL v1.1.0 and the prior versions, and gets fixed in v1.1.0a. The program fails to check the length before memory allocation, allowing attackers to allocate excessive amount of memory. As shown in Listing 2, the patch inserts a new condition check.



```
mov    rdx, [rdi+98h]
mov    r12, rdi
cmp    qword ptr [rdx+198h], 0
jz     short loc_3FEA8
```

```
cmp     qword ptr [rdi+1F8h], 454Ch
mov     eax, 454Ch
mov     r12, rdi
cmovnb  rax, [rdi+1F8h]
cmp     rbp, rax
jbe     short loc_3FFF0
```

```
mov    rdx, [rdi+98h]
cmp    qword ptr [rdx+198h], 0
jz     short loc_40018
```

(a) Matching Result from BinDiff

```
cmp     qword ptr [rdi+1F8h], 454Ch
mov     eax, 454Ch
mov     r12, rdi
cmovnb  rax, [rdi+1F8h]
cmp     rbp, rax
jbe     short loc_3FFF0
```

```
mov    rdx, [rdi+98h]
mov    r12, rdi
cmp    qword ptr [rdx+198h], 0
jz     short loc_3FEA8
```

```
mov    rdx, [rdi+98h]
cmp    qword ptr [rdx+198h], 0
jz     short loc_40018
```

(b) Matching Result from DEEPBINDIFF

Fig. 11: Memory Boundary Checking Failure

Listing 2: Memory Boundary Checking Failure

```
static int dtls1_preprocess_fragment() {
    size_t frag_off;
    frag_len = msg_hdr->frag_len;
    if ((frag_off + frag_len) > len) ||
        + len > max_handshake_message_len(s)) {
        SSLerr();
        return SSL_AD_ILLEGAL_PARAMETER;
    }
    // memory allocation using len
```

We then feed the vulnerable version (1.1.0) and a patched version (1.1.0a) into the tools and observe the results. There are a total of 10359 basic blocks for the version 1.1.0 binary, and we collect a ground truth of 8622 basic blocks. BinDiff and DEEPBINDIFF can correctly match 7204 and 7815 basic blocks.

While both tools can achieve high accuracy, still, only DEEPBINDIFF is able to correctly match the vulnerability patch. More specifically, we expect diffing tools to identify the patch as a new insertion, while still matching the original basic blocks, for the vulnerability analysis. Depicted in Figure 11a, BinDiff mismatches the vulnerable basic block with the new condition check basic block, rendering the real matching basic block unmatched (shown as white block). For DEEPBINDIFF, it successfully matches the basic blocks and identifies the new condition check as an insertion. This case study shows that DEEPBINDIFF can identify inserted basic blocks accurately, with the help of its design choices and Algorithm 1.

VIII. DISCUSSION

A. Compiler optimizations

Different compiler optimization techniques are one of the major reasons for code transformation. For example, it is

TABLE VI: Compiler Optimizations

Optimizations	DeepBinDiff Design
instruction scheduling	1) choose not to use sequential info as a part of context for each instruction 2) exclude instruction sequence during block feature vector generation
instruction replacement	1) NLP technique is used to distill semantic information
block reordering	1) treat the merged ICFG as an undirected graph
function inlining	1) generate random walks across function boundaries 2) avoid function level matching 3) k-Hop greedy matching is done on top of ICFG rather than CFG
register allocation	1) register name normalization

reported that about 10% functions with GCC O2 optimization have been transformed by function inlining technique [21]. DeepBinDiff is designed to handle common optimization techniques as so to achieve high matching accuracy.

As shown in Table VI, the design of DeepBinDiff takes care of multiple most common compiler optimization techniques [15]. We deliberately choose to exclude some easy-to-break assumptions such as the order of instructions and blocks, CFG directions, function boundaries.

Particularly, when training the block feature vector generation model, instruction sequential information is not part of context information. By considering ICFG as an undirected graph, our system will not be affected by block reordering as long as the two blocks are still k-hop neighbors. For function inlining, DeepBinDiff generates random walks based on ICFG, meaning control dependency information is extracted regardless of function boundary changes. Also, we choose not to perform function level matching as in BinDiff to avoid being affected by function boundary changes. Finally, k-Hop greedy matching is based on ICFG other than CFG, therefore, block matching is not affected.

B. Limitations

Besides all the advantages, DeepBinDiff still has a few limitations. First, in practice, certain blocks are often merged by the compiler to reduce branch mispredictions. Hence, the numbers of blocks for the two binaries can be changed. In this case, DeepBinDiff could mistakenly categorize some blocks as insertions or deletions since our system performs one-to-one block matching. We argue that our tool can still match the merged blocks to their most similar counterparts. Therefore, if a block is semantic-rich (i.e., contains multiple instructions), it will still be matched correctly to the merged block, leaving only the less meaningful block unmatched.

Second, optimizations that drastically change the control flow could thwart the effectiveness of DeepBinDiff. This is due to the fact that our analysis heavily relies on ICFG to extract graph structural information in order to differentiate semantically similar blocks, big change of control flow can significantly affect the results. Consequently, DeepBinDiff is vulnerable to obfuscation techniques that completely alter the CFG. Packing techniques [51], [24] that encrypt the code can also defeat our system. But please note that no existing learning-based techniques can do a better job since they all rely on control flow information.

Third, DeepBinDiff currently has no support for cross-architecture diffing, which has become quite popular, especially in IoT related security research [29], [27], [19]. Potentially, DeepBinDiff could solve this issue by lifting binaries into IR first and then perform diffing in the same way. We leave this as a furture work.

IX. Related Work

A. Code Similarity Detection

Static Approaches. Static approaches usually transform binary code into graphs and then perform the comparison. Bindiff [10], [25] performs many-to-many graph isomorphism detection on callgraph to match functions and leverages CFG matching for basic blocks. Binslayer [12] further augments the graph matching with the Hungarian algorithm to improve the matching results. Pewny et.al. [49] searches bugs by collecting input/output pairs to capture the semantics of a basic block and perform graph matching. To improve runtime performance, discovRE [27] uses lightweight syntax features and applies pre-filtering before matching. However, the pre-filtering may significantly affect the accuracy [29]. To avoid heavy graph matching, Tracelet [20] converts CFGs into a number of paths with fixed-length called *tracelets* and then matches them via rewriting. Esh [17] decomposes the functions into data-flow dependent segments named *strands* and uses statistical reasoning to calculate similarities. GitZ [18] finds strands equality through re-optimization. BinGo [14] performs selective function inlining and extracts partial traces for function similarity detection. These techniques can only decompose within functions to abstain from massive number of fragments. BinHunt [30] uses static symbolic execution and theorem proving to extract semantics. CoP [40] also uses symbolic execution to compute the semantic similarity of blocks and leverages the longest common sub-sequence of linearly independent paths to measure the similarity.

Dynamic Approaches. Blanket Execution [26] executes functions of the two input binaries with the same inputs and compares monitored behaviors for similarity. iBinHunt [43] extends the comparison to inter-procedural CFGs and reduces the number of candidates of basic block matching by monitoring the execution under a common input. BinSim [44], which is specifically proposed to compare binaries with code obfuscation techniques, relies on system calls to perform dynamic slicing and then check the equivalence with symbolic execution. Essentially, dynamic analysis based approaches by nature suffer from poor scalability and incomplete code coverage problem.

Learning based Approaches. Genius [29] forms attributed CFGs and calculates the similarity via graph embeddings generated through comparing with a set of representative graphs named *codebook*. Gemini [54] improves Genius by leveraging neural network to generate function embeddings and trains a Siamese network for similarity detection. αDiff [39] uses a similar Siamese network with CNN to generate function embeddings. This eliminates the need of manually-crafted features. InnerEye [58] utilizes NLP techniques and LSTM-RNN to automatically encode the information of basic blocks. Asm2Vec [23] adopts an unsupervised learning approach by generating token and function embeddings using PV-DM model. However, it only works on function comparison. Also, it does not consider any program-wide CFG structural information during analysis. SAFE [41] leverages a self-attentive neural network to generate function embeddings.

B. Graph Embedding Learning

HOPE [47] preserves higher-order proximity and uses generalized Singular Value Decomposition to improve efficiency. TADW [56] is the first work that considers feature vectors for nodes during matrix factorization. REGAL [32] also performs factorization with node features. However, it only checks the existence of features without considering the numeric values.

DeepWalk [48] is proposed to learn latent representations of nodes in a graph using local information from truncated uniform random walks. node2vec [31] specifically designs a biased random walk procedure that efficiently explores diverse neighborhoods of a node to learn continuous feature representations of nodes.

DNGR [13] proposes a graph representation model based on deep neural networks that captures the graph structure information directly. SDNE [52] designs a semi-supervised model that has multiple layers of non-linear functions to capture both the local and global graph structures. GCN [33] uses a localized first-order approximation of spectral graph convolutions to perform semi-supervised learning on graphs in a scalable way. Structure2Vec [16] is proposed for structured data representation via learning features spaces that embeds latent variable models.

X. Conclusion

In this paper, we propose a novel unsupervised learning based program-wide code representation learning technique to perform binary diffing. To precisely match the blocks within given binaries, we leverage NLP techniques to generate token embeddings which are further used to generate block feature vectors containing semantic information. We then generate inter-procedural CFGs (ICFGs), extract the program-wide structural information from the ICFGs using TADW algorithm and generate basic block level embeddings. Finally, we propose a k-hop greedy matching algorithm to find optimal matching for the blocks. We implement a prototype named DeepBinDiff and evaluate it against 113 binaries from Coreutils, Diffutils and Findutils, 10 C++ binaries, and 2 real-world vulnerabilities in OpenSSL under the scenarios of cross-version and cross-optimization-level diffing. The results show that our system could outperform state-of-the-art techniques by a large margin.

Acknowledgement

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported in part by National Science Foundation under grant No. 1719175, DARPA under grant FA8750-16-C-0044, and Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "diffing-with-kam1n0," https://www.whitehatters.academy/diffing-with-kam1n0/, 2019.

[2] "GNU Coretuils," https://www.gnu.org/software/coreutils/, 2019.

[3] "GNU Difftuils," https://www.gnu.org/software/diffutils/, 2019.

[4] "GNU Findutils," https://www.gnu.org/software/findutils/, 2019.

[5] "IDA Disassembler and debugger," https://www.hex-rays.com/products/ida/, 2019.

[6] "indicators," https://github.com/p-ranav/indicators/, 2019.

[7] "InnerEye," https://nmt4binaries.github.io//, 2019.

[8] "LSHBOX," https://github.com/RSIA-LIESMARS-WHU/LSHBOX/, 2019.

[9] "OpenSSL," https://www.openssl.org/, 2019.

[10] "zynamics BinDiff," https://www.zynamics.com/bindiff.html, 2019.

[11] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," 2011.

[12] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.

[13] S. Cao, W. Lu, and Q. Xu, "Deep neural networks for learning graph representations." in *AAAI*, 2016, pp. 1145–1152.

[14] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 678–689.

[15] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[16] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International Conference on Machine Learning*, 2016, pp. 2702–2711.

[17] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.

[18] ——, "Similarity of binaries through re-optimization," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 79–94.

[19] ——, "Firmup: Precise static detection of common vulnerabilities in firmware," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 392–404.

[20] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014.

[21] F. De Goër, S. Rawat, D. Andriesse, H. Bos, and R. Groz, "Now you see me: Real-time dynamic function call detection," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 618–628.

[22] T. Dietterich, "Overfitting and undercomputing in machine learning," *ACM computing surveys*, vol. 27, no. 3, pp. 326–327, 1995.

[23] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Security and Privacy (SP), 2019 IEEE Symposium on*. IEEE, 2019.

[24] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: A systematic study based on whole-system emulation." in *NDSS*, 2018.

[25] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," in *Proceedings of the Symposium sur la Securite des Technologies de Linformation et des communications*, 2005.

[26] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2014.

[27] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[28] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *Software Security and Reliability (SERE), 2014 Eighth International Conference on*. IEEE, 2014, pp. 78–87.

[29] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 480–491.

[30] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[31] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 855–864.

[32] M. Heimann, H. Shen, T. Safavi, and D. Koutra, "Regal: Representation learning-based graph alignment," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM, 2018, pp. 117–126.

[33] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[34] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.

[35] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics (NRL)*, vol. 2, no. 1-2, pp. 83–97, 1955.

[36] N. Lageman, E. D. Kilmer, R. J. Walls, and P. D. McDaniel, "Bindnn: Resilient function matching using deep learning," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2016, pp. 517–537.

[37] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International Conference on Machine Learning*, 2014, pp. 1188–1196.

[38] Y. Li, W. Xu, Y. Tang, X. Mi, and B. Wang, "Semhunt: Identifying vulnerability type with double validation in binary code." in *SEKE*, 2017, pp. 491–494.

[39] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 667–678.

[40] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "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, 2014, pp. 389–400.

[41] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019, pp. 309–329.

[42] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[43] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with interprocedural control flow," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[44] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[45] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *IFIP International Information Security Conference*. Springer, 2015, pp. 416–430.

[46] E. W. Myers, "Ano (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[47] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu, "Asymmetric transitivity preserving graph embedding," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 1105–1114.

[48] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 701–710.

[49] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.

[50] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[51] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 659–673.

[52] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2016, pp. 1225–1234.

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

[54] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "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, 2017, pp. 363–376.

[55] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 462–472.

[56] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang, "Network representation learning with rich text information." in *IJCAI*, 2015, pp. 2111–2117.

[57] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Parallel matrix factorization for recommender systems," *Knowledge and Information Systems*, vol. 41, no. 3, pp. 793–819, 2014.

[58] F. Zuo, X. Li, Z. Zhang, P. Young, L. Luo, and Q. Zeng, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *NDSS*, 2019.