Improving Cross-Platform Binary Analysis using Representation Learning via Graph Alignment

Geunwoo Kim
Department of Computer Science
University of California, Irvine
kgw@uci.edu

Sanghyun Hong
Department of Computer Science
Oregon State University
sanghyun.hong@oregonstate.edu

Michael Franz
Department of Computer Science
University of California, Irvine
franz@uci.edu

Dokyung Song*
Department of Computer Science
Yonsei University
dokyungs@yonsei.ac.kr

ABSTRACT

Cross-platform binary analysis requires a common representation of binaries across platforms, on which a specific analysis can be performed. Recent work proposed to learn low-dimensional, numeric vector representations (*i.e.*, embeddings) of disassembled binary code, and perform binary analysis in the embedding space. Unfortunately, however, existing techniques fall short in that they are either (i) specific to a single platform producing embeddings not aligned across platforms, or (ii) not designed to capture the rich contextual information available in a disassembled binary.

We present a novel deep learning-based method, XBA, which addresses the aforementioned problems. To this end, we first abstract binaries as typed graphs, dubbed binary disassembly graphs (BDGs), which encode control-flow and other rich contextual information of different entities found in a disassembled binary, including basic blocks, external functions called, and string literals referenced. We then formulate binary code representation learning as a *graph alignment* problem, *i.e.*, finding the node correspondences between BDGs extracted from two binaries compiled for different platforms. XBA uses graph convolutional networks to learn the semantics of each node, (i) using its rich contextual information encoded in the BDG, and (ii) aligning its embeddings across platforms. Our formulation allows XBA to learn semantic alignments between two BDGs in a semi-supervised manner, requiring only a limited number of node pairs be aligned across platforms for training. Our evaluation shows that XBA can learn semantically-rich embeddings of binaries aligned across platforms without apriori platform-specific knowledge. By training our model only with 50% of the oracle alignments, XBA was able to predict, on average, 75% of the rest. Our case studies further show that the learned embeddings encode knowledge useful for cross-platform binary analysis.

*Corresponding author.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534383

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; • **Theory of computation** → *Program analysis*; • **Computing methodologies** → Knowledge representation and reasoning.

KEYWORDS

Cross-platform, Binary analysis, Graph alignment

ACM Reference Format:
Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. Improving Cross-Platform Binary Analysis using Representation Learning via Graph Alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534383

1 INTRODUCTION

Deep learning has shown great success in many binary analysis tasks in recent years [11, 17, 18, 24, 32, 35, 44, 51, 59, 62, 65], outperforming traditional approaches in terms of analysis accuracy as well as efficiency. Most of prior work studied the problem of learning embeddings of different *entities* found in a binary [17, 18, 35, 38, 62, 65], such as instructions [35, 61], basic blocks [18, 65], and functions [17, 38]. Low-dimensional, numeric vector representations of such entities found in a binary—regardless of how they are obtained, *e.g.*, by training deep neural networks [17, 18], with locality-sensitive hashing [48], etc. [21]—can enhance the accuracy of downstream binary analysis tasks: binary similarity detection [18], code clone or vulnerability search [17], and other semantic binary understanding tasks [11, 44].

This paper considers the problem of learning binary code embeddings that can facilitate *cross-platform* binary analysis [8, 20, 48, 59, 65]. In particular, we aim to learn an embedding model that can produce embeddings of binaries, which are semantically-rich and aligned across different platforms—across different operating systems (OSs) or instruction set architectures (ISAs). Representing binary code compiled for different platforms in a unified vector space enables downstream binary analysis tasks to be performed in a cross-platform manner [48, 59, 65]. The key promise of cross-platform binary analysis is that analysis efforts made in one platform can seamlessly be transferred to other platforms; a bug signature, for example, is defined only once, and bug search can be performed on binaries available in different ISAs [48].

```
if (mprotect(sh.map_result, pgsize,
↪  PROT_NONE) < 0)
  ret = 2;

...
```

```
if (VirtualProtect(sh.map_result,
↪  pgsize, PAGE_NOACCESS,
↪  &flOldProtect) == FALSE)
  ret = 2;
...
```

(a) C source code for Linux (left) and Windows (right).

```
ecp_nistz256_point_add:
...
por %xmm4, %xmm5
pxor %xmm4, %xmm4
por %xmm0, %xmm1
movq $r_ptr, %xmm0
...
lea $Z2sqr(%rsp), $r_ptr
call __ecp_nistz256_sqr_mont$x
...
```

```
ecp_nistz256_point_add:
...
orr $t0,$a0,$a1
orr $t2,$a2,$a3
orr $in2infty,$t0,$t2
cmp $in2infty,#0
csetm $in2infty,ne
add $rp,sp,#$Z2sqr
bl __ecp_nistz256_sqr_mont
...
```

(b) Handwritten assembly code for x86_64 (left) and AArch64 (right).

Figure 1: Semantically-equivalent code in OpenSSL whose implementation is different across OSs or ISAs.

Unfortunately, existing approaches produce binary code embeddings that are not semantically-rich and aligned across platforms to facilitate cross-platform binary analysis. Consider two example source code pairs shown in Figure 1, wherein each pair have the same semantics, but are in either an OS-/ISA-specific form. For these two pairs, state-of-the-art embedding approaches [18, 65] do not produce embeddings well-aligned across platforms. Existing *unsupervised* learning approaches train their models only for a single platform [17, 18], which cannot produce embeddings aligned across platforms. Existing *supervised* learning approaches [59, 65], on the other hand, train their models directly optimizing them for alignment between basic blocks [65] and functions [59] across different platforms. As a training dataset used to supervise their model training, they use compiler-generated cross-platform variants of binary code compiled from the same source code. They use, however, only the features *internal* to each basic block [65] or to each function [59], and not their rich contextual information (*e.g.*, intra- or inter-procedural control flow transfers, or references to string literals) available in disassembled binaries.

Ignoring these contextual information may inadvertently result in models with limited semantic-awareness in a cross-platform setting, considering the way existing supervised approaches [65] construct their labeled training datasets. As compilers "know" how to generate binary code for different platforms having the same semantics, they systematically construct pairwise-similarity-labeled datasets by using compilers. That is, they first compile the same lines of source code written in a platform-independent programming language like C/C++ twice for two different platforms. Then they pairwise-align generated binary code across platforms, if the pair is mapped to the same location in the compiler intermediate representation (IR) produced during compilation [65]. The variety of binary code that can be aligned across platforms by using this method, however, is fundamentally limited for two reasons: First, compilers typically emit only a fraction of instructions available in an ISA. Second, implementations of the same functionality are often platform-specific, as exemplified by the source code shown in Figure 1, and platform-specific source code cannot be compiled for different platforms. Pieces of binary code left unaligned by this method cannot used for training a model in a supervised way, and models trained on a limited dataset may not learn generalizable embeddings that captures rich cross-platform semantics.

Based on our observation that pairwise-alignment labels are only available for a subset of possible binary code, this paper proposes a graph-based, semi-supervised approach to learn an embedding model, which can produce embeddings of binary code that are semantically-rich and aligned across different platforms—across ISAs or OSs. Our key technique is *graph-level alignment* performed on *graph-structured binary code* across platforms: A pair of binaries compiled for different platforms are first abstracted as typed graphs, dubbed binary disassembly graphs (BDGs), which encode rich contextual information. Then, two graph neural networks sharing their parameters are trained, directly optimizing them for alignment between the two graphs, using (i) partial cross-platform alignment information (hence, constituting a *semi*-supervised learning), and (ii) the rich contextual information encoded in each BDG. We employ, in particular, graph convolutional networks (GCNs) [6], inspired by the architecture recently shown to be effective at multilingual knowledge representation learning [56] in the field of natural language processing. Our graph-level alignment approach can enable XBA to predict *potential* alignments between unlabeled entities or even the entities in completely different software. A model trained with our approach can generate richer embeddings that are better-aligned across platforms than existing supervised approaches [59, 65], because they do not leverage the rich contextual information encoded in a BDG as our approach does.

To learn rich cross-platform binary code semantics knowledge beyond those that can be distilled from compilers, we devise unique datasets consisting of a suite of low-level open-source programs and libraries written to be portable across ISAs and OSs, such as GLIBC and OpenSSL. Our approach effectively distills platform-specific knowledge embodied in such cross-platform open-source software, such as ISA-specific handwritten assembly and OS-specific code, and learns an embedding model that can facilitate cross-platform binary analysis. Our evaluation shows that semantically-rich embeddings of binaries aligned across platforms can be learned without any platform-specific knowledge nor manual labeling efforts. By training our model only with 50% of the groundtruth cross-platform alignments, we were able to predict, on average, 75% (or 74% depending on the direction of prediction) of the rest of the alignments. Our case studies show that the embeddings learned via cross-platform BDG alignment encode useful knowledge for cross-platform binary analysis.

Contributions. We make the following contributions:

- We propose a new representation learning approach that can facilitate cross-platform binary analysis. We formulate this task as a graph alignment problem; that is, we train a binary code embedding model with an objective of graph alignment.
- We present a novel deep learning-based method, XBA, which learns binary code embeddings using the graph alignment objective. We develop a format that represents disassembled binary code as a graph, and use graph convolutional networks to capture rich semantic information from the graph.

- To evaluate XBA, we introduce benchmark datasets for cross-platform binary analysis. We show that XBA improves the baseline methods by a large margin, and that XBA can capture semantics of binary code from a small subset of labeled samples.

2 BACKGROUND AND MOTIVATION

2.1 Embedding Binary Code

Recent work has shown that representation learning outperforms traditional, heuristic-based approaches in many binary analysis tasks, such as binary similarity detection [18, 62, 65], function name prediction [14] and function boundary identification [4, 44, 51]. The key to this success is to learn distributed representations of binary code—at the granularity of instructions, basic blocks, or functions—in a vector space, also known as *embeddings*, which can be used for various downstream tasks. Prior work focuses on building pre-trained models that can generate such embeddings, and they either fine-tune those pre-trained models or directly use the obtained embeddings for downstream tasks. We categorize the prior work into two approaches based on how they learn embeddings.

Unsupervised Representation Learning. A line of work on generating embeddings for binary code uses an unsupervised learning approach. Unsupervised approaches, by definition, have the advantage of not requiring any labeled training data; it only requires binaries be disassembled (*i.e.*, instruction and function boundaries can be identified, and inter- and/or intra-procedural control flow graphs can be constructed). DeepBinDiff [18] and Asm2Vec [17] are representative studies in this category. DeepBinDiff uses Word2Vec [42] to embed instructions and basic blocks, and further tunes the basic block embeddings with Text-Associated DeepWalk (TADW) [60] to encode the basic block's contextual location in a program-wide *inter*-procedural control flow graph. Asm2Vec [17], on the other hand, performs random walks on an *intra*-procedural control flow graph of each function. Instruction sequences generated by random walks are used for training their function-level binary code embedding model based on the PV-DM model [33].

Supervised Representation Learning. A separate line of work uses supervised representation learning. The key idea here is to generate embeddings that do well on a particular predictive task, *e.g.*, binary code similarity detection. First proposed by Xu *et al.* [59], most prior work trains a Siamese architecture [10] on *labeled* pairs of binary code. Xu *et al.* (Gemini) represent each function as an augmented control flow graph (ACFG) and use them as inputs to the Structure2Vec [13] model. Two identical Structure2Vec models that share parameters construct a Siamese network, and the network is trained on pairs of ACFGs and their similarity labels. They finally use the outputs of the trained Structure2Vec model as embeddings for functions. Zuo *et al.* [65] (InnerEye) instead represent each basic block as a sequence of instruction tokens and use them as inputs to a Neural Machine Translation (NMT) model to learn binary code embeddings at the granularity of basic blocks. A Siamese network is composed of two identical NMT models and trained on the labeled pairs of basic blocks. By using *pairwise-similarity-labeled* data, supervised approaches learn embeddings where embeddings of similar pieces of binary code are closer and dissimilar ones apart.

2.2 Cross-Platform Binary Analysis

Many software today is written to support multiple OSs and ISAs. Software portable across OSs could contain diverging implementations for the same high-level functionalities to, *e.g.*, comply for an OS-specific application programming interface. Depending on the hardware used for deployment, binaries could also be compiled for various ISAs; such heterogeneity is particularly evident in firmware available in a variety of ISAs [12], *e.g.*, ARMv7, ARMv8, MIPS, etc.

Handling this heterogeneity in binary analysis is an onerous task, which, therefore, has drawn much interest in conducting binary analysis in a platform-independent manner. Prior studies proposed various platform-independent, cross-platform binary analysis techniques, tailored for vulnerability search [48, 49], code reuse exploit development [31], binary similarity detection [59, 65], or binary code search in general [8]. The key promise of cross-platform binary analysis is that analysis efforts made in one platform can seamlessly be transferred to other platforms; a bug signature, for example, is defined only once, and bug search can be performed for binaries available in different ISAs [48]. A line of prior work [8, 19, 45, 48, 52] tackles this problem by *lifting* binaries to OS/ISA-independent representations. In contrast, Gemini [59] and InnerEye [65], the supervised representation learning approaches described in §2.1, use Siamese models to directly learn similarities between binary code compiled for different platforms.

2.3 Embedding for Cross-Platform Analysis

An embedding model that understands the semantic relationship between binaries compiled for different platforms would produce discriminative embedding vectors, where (i) semantically-similar pairs of binary code across platforms are close, even though they appear different across platforms, and (ii) semantically-dissimilar ones apart.

Unsupervised vs. Supervised Learning Approach. Unsupervised learning approaches have the advantage of not requiring any labeled data, but may not produce embeddings that capture semantic relationships between binary code across platforms. Existing unsupervised embedding approaches described in §2.1 rely solely on features available within a single platform [17, 18] during training, without incorporating any cross-platform semantic alignment information. This is less than ideal, because, as we explain below, a labeled dataset, which could provide useful supervision during training, could systematically be constructed by using compilers.

A Siamese architecture, a supervised learning approach used by prior cross-ISA embedding studies [59, 65], has been shown to be effective at training a discriminative embedding model. This approach requires a training dataset whose tuple-level similarities are labeled. Prior studies systematically constructed a pairwise-similarity-labeled training dataset through multiple compiler invocations [59, 65]. The idea is to first compile the same lines of source code written in a platform-independent programming language like C/C++ twice for two different platforms, and map the compiled binary code back to their location in the original source code (or in the platform-independent intermediate representation produced during compilation [65]). Pairs of binary code mapped to the same location constitute semantically-equivalent binary code pairs across platforms. Semantically-different pairs, on the other hand,

Table 1: Number of basic blocks that are 1-to-1 aligned (or unaligned) across platforms or compiler optimization levels. We use compiler-kept debug information for this analysis.

	Configurations	# of Basic Blocks	
	Platform(s) / Opt. Level(s)	1-to-1 Aligned	**Unaligned**
libc.so	x86_64 ↔ ARMv7 / O1	29,626	**11,531**
	x86_64 ↔ ARMv7 / O2	23,274	**14,501**
	x86_64 ↔ ARMv7 / O3	21,106	**14,307**
	x86_64 / O1 ↔ O2	31,880	**6,315**
	x86_64 / O1 ↔ O3	29,166	**7,067**
	x86_64 / O2 ↔ O3	33,260	**1,910**
libcrypto.so	x86_64 ↔ ARMv7 / O0	133,282	**4,621**
	x86_64 ↔ ARMv7 / O1	79,580	**16,836**
	x86_64 ↔ ARMv7 / O2	67,780	**22,595**
	x86_64 ↔ ARMv7 / O3	62,404	**21,837**

Table 2: Number of unique instruction mnemonics in basic blocks that are 1-to-1 aligned (or unaligned) across platforms.

	Platform	1-to-1 Aligned (A)	Unaligned (B)	$A \cup B$	$B - A$
libc.so	x86_64 / O2	125	203	219	**94**
	ARMv7 / O2	235	216	280	**45**
	x86_64 / O2	137	215	235	**98**
	AArch64 / O2	119	128	138	**19**
libcrypto.so	x86_64 / O0	88	225	244	**156**
	ARMv7 / O0	68	87	97	**29**
	x86_64 / O0	87	232	245	**158**
	AArch64 / O0	89	121	149	**60**

are generated through random binary code pair sampling. The combined pairs constitute a labeled training dataset that can be used to supervise metric-based learning with a Siamese architecture.

Limitations of Existing Supervised Approaches. Supervised learning approaches, *e.g.*, training an embedding model using a Siamese architecture, can align labeled pairs, but existing approaches that only considers pairing basic blocks cannot encode the contextual information [59, 65]. The variety of binary code semantically-aligned across multiple platforms that can be constructed with multiple compiler invocations on the same source code for different platforms is limited for the following reasons.

First, as illustrated in Figure 1, implementations of the same functionalities could diverge across platforms. Platform-specific code cannot be compiled for different platforms, so they cannot be semantically-aligned across platforms without manual efforts. Second, compilers typically emit a fraction of all instructions available in an ISA. An ISA may support just too many instructions (*e.g.*, thousands in Intel x86_64) including deprecated ones for backward compatibility, and/or purpose-built instructions (*e.g.*, AES instructions for Intel x86_64 [27] and ARM [3]) that are predominantly used in assembly code handwritten for each ISA. Handwritten assembly code, obviously, cannot be compiled for different ISAs. Last but not least, the binary to source code mapping information used to align binary code could be incomplete. It is not guaranteed that compilers always keep, for a given fragment of binary code, its accurate location in the original source code [16, 25, 36]

Table 3: Design choices of XBA in comparison with prior basic block embedding approaches.

	InnerEye [65]	**DeepBinDiff** [18]	**XBA (Ours)**
Input Format	Blocks	Sequence of Blocks	BDG (§3.1)
Contextual Info.	-	Control-flow only	Full
Cross-Platform	✓	-	✓
Method	NMT [58]	DeepWalk [47]	GCN (§3.3)
Training	Supervised	Unsupervised	Semi-supervised

Table 1 shows the number of unaligned basic blocks when binaries are compiled for different platforms. Depending on the software, 3–40% of basic blocks left unaligned even when we compile the same software for different OSs/ISAs. A closer look into these unalinged basic blocks more clearly reveals the problem. As shown in Table 2, there are, unfortunately, many instruction mnemonics that are only found in these unaligned basic blocks. This is due in part to platform-specific code, *e.g.*, handwritten assembly functions or OS-specific API functions, which cannot be compiled for different platforms. Existing supervised learning approaches [65], as they do not incorporate rich contextual information around these unaligned basic blocks in their model training, cannot capture their semantics.

3 XBA: OUR PROPOSED METHOD

We now present XBA, a novel deep learning-based method that addresses the limitations of prior work, thereby facilitating cross-platform binary analysis. We first discuss our design choices in Table 3, in comparison with the design of prior approaches.

- **Input format:** We develop a format that contains rich contextual information of basic blocks. Prior work considers a basic block either merely as a sequence of instructions without considering its local context [65], or as a node contextually placed within the inter-procedural control flow graph (iCFG) of a given binary [18]. To encode rich basic block contexts beyond program control flow, we propose a new format, Binary Disassembly Graphs (BDG), to represent binaries (§3.1).
- **Method:** We choose a method that can leverage the rich contextual information encoded in our input format. Zuo *et al.* [65] only considers individual blocks without any contextual information. Duan *et al.* [18] use DeepWalk [47], but this method offers only control-flow information explored by randomly taking walks over an iCFG. XBA, on the other hand, uses GCNs, well-known for capturing the contextual information encoded in graphs (BDGs).
- **Training:** We use an objective of graph alignment, a semi-supervised approach, for training our GCNs. Unsupervised methods [18] rely only on structural information available within a single platform; they cannot align binaries across platforms. While supervised learning [65] can align such binaries across platforms, this approach cannot work without alignment labels. Thus, we take a *hybrid* approach that aligns binaries across platforms at a graph level, which only requires a small subset of labeled alignments.

Overview. Figure 2 describes an overview of XBA. On the left side, we show an example of the semantically-equivalent source code

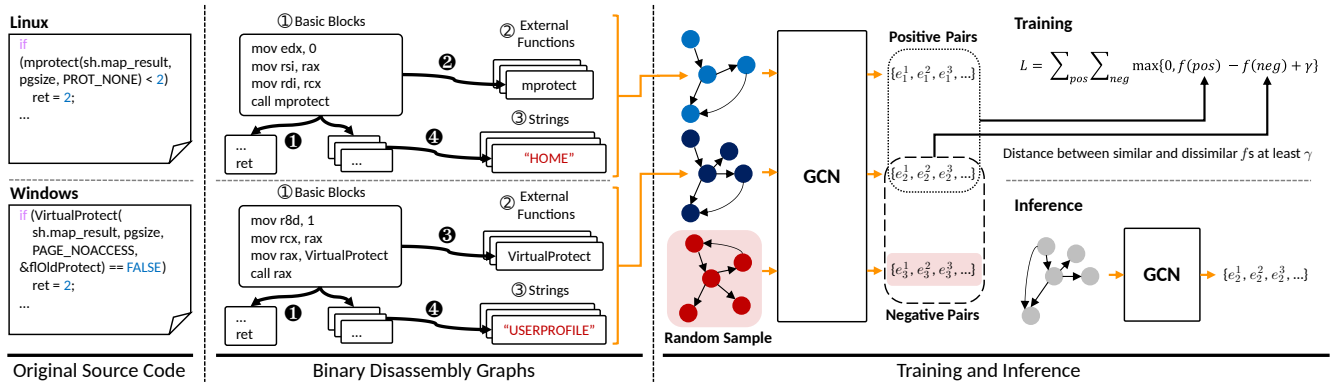


Figure 2: XBA overview. XBA consists of three steps: (i) We first transform binaries into binary disassembled graphs that contain rich contextual information. (ii) We then use Graph Convolutional Networks (GCNs) to generate embeddings for entities, *e.g.,* basic blocks, external functions, or strings. We train GCNs to produce close embeddings between binaries from semantically-equivalent code. (iii) The trained GCNs produce embeddings for the entities useful for diverse downstream tasks.

implemented for Linux and Windows found within the same software, OpenSSL, written to be portable across OSs. XBA receives two binaries compiled for different platforms and learns (or infers) the similarity between them. We first disassemble those binaries and construct their BDGs. They contain basic blocks, external functions, strings, and relations between them. We then process the BDGs with GCNs and generate embeddings for entities (*i.e.*, basic blocks, external functions, or strings). During training, we make the GCNs produce embeddings for the aligned entities close and negatively-aligned entities (artificially constructed via sampling) apart. Once trained, our GCNs produce embeddings for the entities in a BDG (that holds rich contextual information), useful for diverse downstream tasks.

3.1 Binary Disassembly Graph

Binary analysis on an abstract level is analyzing a stream of string tokens obtained from a disassembled binary. In cross-platform analysis, we encounter many platform-specific tokens, and their semantic relationship cannot easily be established across platforms. For example, it is not straightforward to figure, without platform-specific knowledge, that `read` in a Linux binary and `ReadFile` in a Windows binary have similar semantics. To address this issue, we abstract a disassembled binary into a new format, binary disassembled graph (BDG), which encodes rich contextual information from which we can draw the semantics of platform-specific tokens. A BDG is a *typed* graph where we represent a group of tokens found in a disassembled binary as nodes, and the types of their connections to each other as relations:

Nodes. A node in a BDG can be one of the following types: ① basic block, ② external function, and ③ string literal. Here, we choose the granularity of basic blocks for abstracting the instructions found in the code sections of a binary. An alternative is to use the granularity of functions [17, 21, 38, 59]. However, recent work [18, 35, 48, 65] demonstrates that representations learned at a finer-grained granularity could more broadly be useful than those learned at a coarser-grained granularity. Representations learned at a finer-grained granularity (i.e., basic block embeddings) can be used to train those at a coarser-grained granularity (i.e., function embeddings), in a way akin to learning sentence and paragraph embeddings with word embeddings [33], or subgraph- and graph-level embeddings with graph node embeddings [2, 43].

Table 4: Rules used for instruction normalization.

	Type	Normalized into ...
Mnemonic	Call	"icall" (for indirect calls) "call" (for direct calls)
	Others	Unmodified mnenonic
Operand	Register	"reg1", "reg2", "reg4", "reg8", or "reg16"
	Constant value	"imm"
	External function	Function symbol name
	Other references	"ptr1", "ptr2", "ptr4", or "ptr8"

We define the node attributes as follows: ① For each basic block, we use instruction sequences for the node attribute. More specifically, the mnemonic and operands of each instruction found in a given basic block are first normalized per the rule outlined in Table 4, and the sequence of normalized tokens are used as the basic block's node attributes. ② We use their symbol names as the node attribute for external functions, whose instruction sequences reside outside of a disassembled binary. ③ We use the string literal found in a disassembled binary *as-is* for the node attribute.

Relations. Four types of relations can be defined between the nodes of a BDG as follows: ❶ An *intra*-procedural control-flow transfer, defined between two basic blocks, indicates that a program control flows from one basic block to another. ❷ Direct calls (*i.e.*, *inter*-procedural control-flow transfers), are defined between a basic block that contains one or more call sites and their callee(s). We also define ❸ address-taking code-to-code references, when a basic

block contains one or more instructions that take the address of another function. Finally, ❹ code-to-string references are defined between a basic block and a string node, when the basic block contains an instruction that references the string literal.

3.2 Cross-Platform Binary Code Analysis and Graph Alignment Problem

We now detail how cross-platform binary analysis can be formulated as a *graph alignment problem*. In cross-platform binary analysis, the key challenge is to identify basic blocks of a similar (or dissimilar) functionality with *incomplete* knowledge. For example, we cannot assume that two OS-specific external functions, *e.g.*, `read` in Linux and `ReadFile` in Windows perform the same operation. To estimate their functionalities, we need contextual information, such as the same basic block calling those functions or the same argument passed to them. In the example above, if both the functions take `.sqliterc` as an argument, we can estimate that they serve the same purpose, *i.e.*, loading the SQLite configuration file.

Graph Alignment Problem. The above example makes it clear that successful cross-platform binary analysis requires capturing similarities (or dissimilarities) between entities (e.g., basic blocks) found in a binary using their neighborhood information. We can, therefore, formulate this task as a graph alignment problem [9, 34, 54, 56]. Across the BDGs constructed from binaries compiled for different platforms, we aim to find the correspondences between nodes in BDGs based on a small subset of pre-aligned node pairs. Formally, we can rewrite this alignment task as follows:

We denote a BDG by $G = (E, R, A, T^R, T^A)$, where E, R, A are sets of entities, relations, and attributes, respectively. $T^R \subset E \times R \times E$ is the set of relational triples, $T^A \subset E \times A \times V$ are the set of attributional triples, and V represents the set of attribute values. Suppose that two binaries are compiled from the same source code base for different platforms $P1$ and $P2$. We first construct their BDGs G_{P1} and G_{P2}, where $G_{Pi} = (E_i, R_i, A_i, T_i^R, T_i^A)$ for $i \in \{1, 2\}$. Between G_{P1} and G_{P2}, there are m nodes that are pre-aligned, and we identify them from the source code line number and file information kept in the debug section of the binaries. We denote them by $M^+ = \{(e_{1i}, e_{2i}) | e_{1i} \in E_1^{'} \wedge e_{2i} \in E_2^{'}\}_{i=1}^{m}$, where $E_i^{'}$ is the subset of E_i. Finally, our task is to find the node correspondence of all the k unaligned entity pairs in the BDGs.

3.3 Graph-Based Representation Learning

XBA uses deep learning-based *graph embedding* [7] to solve this alignment problem. Here, we first train a model that projects entities and attributes in BDGs onto a unified embedding space. Once trained, XBA uses this pre-trained model to generate embeddings for the entities and attributes in a BDG. In this way, the model can generate quality embedding for the entities unseen during training (§4.3) or embeddings that can improve the performance of many downstream tasks, *e.g.*, cross-platform similarity analysis (§4.4). We use graph convolutional networks (GCNs) [6, 29] for embedding generation. GCNs are well-studied and outperform embedding methods used by prior work [17, 18], such as random walks, in diverse graph alignment tasks (see our related work in §6).

GCN Architecture. We use a multi-layer GCN architecture proposed by Kipf and Welling [29]. It contains a stack of multiple GCN layers. Each layer takes a node embedding matrix $H^{(l)} \in \mathbb{R}^{n \times d^{(l)}}$, where n is the number of nodes, and $d^{(l)}$ is the input dimension of the layer l. The layer's output $H^{(l+1)}$ is computed as follows:

$$H^{(l+1)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)})$$

where σ is an activation function for which we use $ReLU(\cdot)$; A is an $n \times n$ adjacency matrix that encodes the relations of a BDG; $\hat{A} = A + I$, where I is the identity matrix; $\hat{D}$ is the diagonal node holding the degrees of $\hat{A}$; $W^{(l)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is the weight matrix, and $d^{(l+1)}$ is the input dimension of the next layer.

Computing Adjacency Matrix. We require the relations T^R to compute the equation above. We encode T^R as an adjacency matrix A. Following the prior work [56], we define A such that $a_{ij} \in A$ measures a potential influence of the node e_i on the node e_j. We compute the same measures, for each relation type r: functionality, denoted by $fun(r)$, and inverse functionality, denoted by $ifun(r)$. We then compute a_{ij}, the number of relations defined between e_i and e_j weighted by $fun(r)$ and $ifun(r)$, as follows:

$$a_{ij} = \sum_{<e_i, r, e_j> \in G} ifun(r) + \sum_{<e_j, r, e_i> \in G} fun(r)$$

Computing Input Features. The attributional triples T_A defined in a BDG must be encoded as numeric vectors that can be fed into the GCN as the initial input feature vectors $H^{(0)}$. XBA considers the set of all attribute values (e.g., normalized instruction tokens) associated with an entity e, regardless of their type, as the features of e. Numeric vector representations of these features can be computed using any encoding or embedding scheme. Here, we consider the following two schemes: (i) attribute feature vectors encoded in the bag-of-words model [23], and (ii) embeddings obtained from DeepBinDiff and InnerEye [18, 65]. In our evaluation, we show that XBA improves the utility of these embeddings further (§4.3).

3.4 Model Training and Embedding Generation

Remind that we only have a small subset of pre-aligned entities in cross-platform binary analysis. In this scenario, using supervised approaches proposed in prior work [13, 59, 65] cannot learn embeddings for unaligned entities. Hence, we take a *semi-supervised approach*: Those pre-aligned entities serve as supervision signals that allow computing quality embeddings for unaligned entities.

Siamese Architecture. XBA utilizes a Siamese architecture to train our GCNs. Two GCNs with the same architecture and model parameters take the input feature vectors and the adjacency matrices, constructed from two BDGs, as input. It allows us to train our GCNs in a *contrastive* way by using the following loss function.

Training Objective. During training, we minimize the distance f between the embeddings from positively-aligned entity pairs while increasing the distance between the embeddings from negatively-aligned entity pairs. We use a margin-based hinge loss function:

$$\mathcal{L} = \sum_{(e_1, e_2) \in M^+} \sum_{(e_1', e_2') \in M^-_{(e_1, e_2)}} max\Big\{0, f(h(e_1), h(e_2)) - f(h(e_1'), h(e_2')) + \gamma\Big\}$$

Table 5: Dataset statistics. We construct the benchmarks from seven open-source software compiled for different platforms. The number of entities, relations, and pre-aligned entities are shown.

Dataset	Platform	Entities ① / ② / ③	Relations ❶ / ❷ / ❸ / ❹	**1-to-1 Alignments**
SQLITE3	Linux	37.1k / 0.1k / 1.5k	66.9k / 14.3k / 0.2k / 1.5k	24.0k
	Windows	36.1k / 0.2k / 1.6k	66.0k / 14.1k / 0.2k / 1.5k	
OPENSSL	Linux	17.0k / 1.3k / 2.2k	33.0k / 10.6k / 0.2k / 2.2k	10.5k
	Windows	15.8k / 1.1k / 2.1k	31.9k / 9.0k / 0.1k / 2.1k	
CURL	Linux	32.6k / 0.3k / 3.6k	55.0k / 10.0k / 0.1k / 3.6k	18.2k
	Windows	30.5k / 0.3k / 3.6k	59.3k / 2.4k / 0.0k / 3.6k	
HTTPD	Linux	19.2k / 0.4k / 1.5k	32.7k / 6.0k / 0.2k / 1.4k	9.0k
	Windows	27.8k / 0.2k / 1.7k	54.1k / 2.1k / 0.0k / 1.7k	
LIBCRYPTO	Linux	95.1k / 0.1k / 9.1k	186.7k / 54.2k / 1.5k / 9.1k	69.2k
	Windows	93.3k / 0.2k / 9.0k	183.9k / 21.8k / 0.9k / 9.0k	
GLIBC	x86_64	63.7k / 0.0k / 1.1k	106.9k / 9.9k / 0.7k / 1.1k	13.1k
	AArch64	51.7k / 0.0k / 1.1k	87.6k / 8.4k / 0.6k / 1.1k	
LIBCRYPTO	x86_64	95.1k / 0.1k / 9.1k	186.6k / 54.2k / 1.5k / 9.1k	67.8k
	AArch64	95.3k / 0.1k / 9.1k	185.1k / 18.3k / 1.1k / 9.1k	

where M^+ is the set of positive alignments from the pre-aligned entity pairs, $M^-_{(e_1,e_2)}$ is the set of negative alignments for a pre-aligned entity pair (e_1, e_2), f is the L1 distance $f(x, y) = ||x - y||_1$, $h(e)$ is the embedding of an entity e, and γ is the margin hyperparameter that defines the minimum distance between the positive and negative alignments. To compute M^+, we pre-align entities by using the source code line number information kept in the debug section of the given binaries. $M^-_{(e_1,e_2)}$ is obtained by replacing e_i with a node randomly chosen from G_{P1} or G_{P2}.

Embedding Generation. Once trained, our GCN model can generate embeddings for entities e_is found in a given binary. We first construct a BDG of the binary and take the subgraph from it. The subgraph must include the entities e_is and their n-hop neighborhood entities, as well as the relations between them, where n is the number of the GCN layers. We feed the subgraph to our GCN, and it will produce embeddings $h(e_i)$s for the e_is. $h(e_i)$s can be used for many downstream binary analysis tasks, such as computing a similarity score between two pieces of binary code, even they are obtained from binaries compiled for different platforms.

4 EVALUATION

In this section, we answer the following research questions:

- **RQ1.** Does XBA produce similar embeddings for basic blocks that are not pre-aligned, *i.e.*, not included in the seed alignments?
- **RQ2.** Do the binary code embedding vectors learned with XBA encode useful information for cross-platform binary analysis?
- **RQ3.** Can a XBA model generalize across binaries unseen during training?
- **RQ4.** How much of the supervision, *i.e.*, the amount of seed alignments, is required in training a XBA model?
- **RQ5.** How much does each type of relations contribute to the performance of XBA?

4.1 Benchmark Datasets

We construct our own benchmark datasets for evaluating XBA on cross-platform binary analysis. Benchmark datasets used by most prior work on binary code embedding do not always contain OS-specific or ISA-specific components. To learn richer embeddings (thereby enabling richer cross-OS or cross-ISA analysis), we carefully construct our own datasets, which contain many platform-specific components as follows:

- **Cross-OS Datasets:** We choose five different software that natively supports Linux and Windows: (i) CURL, (ii) SQLITE3, (iii) OpenSSL's CLI interface program (OPENSSL), (iv) OpenSSL's cryptographic library (LIBCRYPTO), and (v) Apache HTTPD. We compiled them into binaries (either executables or shared libraries) of Linux ELF and Windows PE32+ with debug information included.
- **Cross-ISA Datasets:** Owing to the layered architecture of modern OSes, most ISA-specific code resides mostly in low-level system libraries. We therefore choose (i) GLIBC, the default C library in many Linux distributions, and (ii) LIBCRYPTO in OpenSSL, a cryptographic library that supports multiple ISAs. Both libraries contain ISA-specific code, such as handwritten assembly code tailored to each supported ISA. We compiled them in Linux for x86_64 and AArch64 into ELF-format shared libraries with debug information included.

We disassembled these binaries with IDA Pro [26]. XBA constructs the corresponding BDGs from the disassembled code. We pre-aligned the entities in the BDGs based on the source code line information (*i.e.*, debug information) kept by compilers. The statistics of our datasets are summarized in Table 5. We can see, for reasons explained in §2.3, there are many entities we were unable to align based on the debug information. However, the BDGs constructed from our datasets *implicitly* contain rich semantic information between entities, defined by relations. XBA captures the semantics of the unaligned entities via semi-supervised learning and can find alignments between them.

4.2 Implementation and Setup

We implemented XBA with Python v3.8 and TensorFlow v2.7.0 [1]. We based our implementation of GCNs on Wang et al.'s [56]. Unless stated otherwise, we use 5-layer GCNs and set the dimension of the hidden units of each layer d to 200, *i.e.*, the output embeddings will have the dimension of 200. In our loss function, we set the distance between similar and dissimilar pairs γ to 0.1, and for each pre-aligned pair, we generate 50 to 100 negative alignment samples, chosen differently for each binary. We ran all of our experiments on a machine equipped with AMD EPYC 7282 CPU with 256GB RAM and two NVIDIA RTX 3090 GPUs.

Baselines. We choose the following binary code embedding (and encoding) approaches as our baselines: InnerEye [65], DeepBinDiff [18], and the bag-of-words (BoW) [23]. In BoW, we use the vocabulary built by using InnerEye's tokenization scheme. InnerEye uses a supervised learning approach with a Siamese architecture, whereas DeepBinDiff takes an unsupervised learning approach. We conservatively evaluate these two baseline approaches: We include all instruction tokens normalized away in the original studies, and

carefully refined the implementation open-sourced by the authors[1]. Both InnerEye and DeepBinDiff models were first trained on our datasets using their default parameters, and use the embeddings produced by the models for evaluating XBA.

Evaluation Metric. We use Hit@K as our evaluation metric, which is the de-facto standard in many similarity detection tasks [17, 34, 38, 56]. A high score means that semantically-similar pairs of binary code across platforms have more similar representations than others in the embedding vector space. We choose K from $\{1, 10, 50, 100\}$.

Model Training. The training details vary depending on the experiments, unless stated otherwise, the baseline and XBA models were trained as follows. We trained two models for InnerEye: one on all of our cross-OS benchmarks and the other on all cross-ISA benchmarks. For DeepBinDiff, as it is a program-wide approach, we trained the DeepBinDiff embeddings individually on each benchmark dataset. XBA was also trained separately for each bianry. For each XBA model, we used as input features (i) features encoded using BoW (shown as BoW Encoding + **XBA**) or (ii) embeddings produced by DeepBinDiff (DeepBinDiff + **XBA**). We early-stopped the training before the model overfits, and the corresponding epoch number varies depending on a binary.

4.3 RQ1. Alignment Accuracy

We first examine how accurately XBA can predict the alignments across platforms. Here, we randomly split the pre-aligned entity pairs (*i.e.*, those with alignment labels) into halves. We use the first half as the datasets for training our 3-layer XBA and InnerEye models and the other half to evaluate the embeddings' utility. We report the prediction accuracy in Hit@K. Table 6 shows the results.

XBA Outperforms the Baselines. In all the cross-OS and cross-ISA evaluations, XBA outperforms the baseline approaches. Compared to the baselines, the accuracy improvements increase as we use more precise metrics (*i.e.*, Hit@100→Hit@1). In the cross-OS alignments, we observe that XBA significantly outperforms the baselines by 1–25% in Hit@1 and Hit@10. Surprisingly, XBA improves the accuracy by 27–75% across all the cross-ISA alignments in Hit@1 and Hit@10; the baseline methods show 2–12% accuracy in the same scenarios. It is particularly surprising XBA improves the BoW Encodings by 7× in Hit@10 (11.34 → 83.41). Our results show that formulating cross-platform binary analysis as a graph alignment problem enables us to generate embeddings that are semantically rich and aligned across platforms.

Comparison with Existing Supervised Methods. InnerEye is a representative work that uses supervised learning for training an embedding model, which, unfortunately, shows the worst performance in cross-platform scenarios. We first find a reason from the data format InnerEye uses. InnerEye expresses basic blocks only with their internal features, *i.e.*, instructions sequences, which do not encode any neighboring information. As we discussed in §3.3, cross-platform analysis requires contextual information outside the scope of basic blocks. On the other hand, XBA, as it attends to both the attributional and rich relational features encoded in BDGs, performs significantly better at aligning entities not included in the training data.

Comparison with Existing Unsupervised Methods. We observe that DeepBinDiff [18], a representative unsupervised learning approach, performs marginally better than InnerEye in cross-OS scenarios. This can be attributed to the embedding method used by DeepBinDiff, which can encode the context of a basic block within a inter-procedural control-flow graph by using Text-Associated DeepWalk [60]. However, DeepBinDiff performs significantly worse (2–12%) in cross-ISA scenarios, where the vocabularies (e.g., instruction mnemonics) used as the node attributes are substantially different across platforms. Their semantic similarities are, because of syntactic differences, not captured by DeepBinDiff, as it uses unsupervised learning; that is, it gives no supervision towards cross-platform alignment during training.

Combining XBA with Existing Methods. We emphasize that XBA is compatible with the existing baseline methods. In our experiments with DeepBinDiff, we use DeepBinDiff embeddings as the input feature vectors of XBA (shown as DeepBinDiff + **XBA**). Our results show that XBA enhances the DeepBinDiff embeddings such that they are better-aligned across platforms; thus, they are better for cross-platform binary analysis. The improvements are consistent across all our experiments with DeepBinDiff. Interestingly, in OPENSSL, DeepBinDiff embeddings enhanced by XBA outperform the XBA embeddings trained with the BoW-encoded embeddings. The results suggest that there could be a synergy between XBA and other embedding approaches.

4.4 RQ2. Predicting New Alignments

We then examine how accurate XBA is in predicting new alignments unseen during training. The experiments shown in §4.3 were designed for the validation of XBA, and those shown in this Section reflect actual testing scenarios. Here, we hypothesize that, as XBA takes a semi-supervised learning approach, and uses BDGs which contain rich contextual information between entities, XBA can predict accurately unseen samples.

Training and Testing Details. We modify our experimental setup for this evaluation. We train XBA on both 100% of the pre-aligned (labeled) pairs. Depending on the datasets, the total number of alignments ranges from 8,964 to 69,213. We use the BoW-Encodings as the input features of XBA. Once trained, we use XBA to produce embeddings for unlabeled entity pairs and analyze their alignments. Table 7 shows our case study results.

Cross-OS: OS-Specific API Function Matching. We first study whether XBA is able to predict new cross-OS alignments between OS-specific entities that have similar semantics but appear different across platforms. The top rows in Table 7 show that XBA places many semantically-similar OS-specific API functions at high ranks. We use Ⓐ as a source, and Ⓑ are the predicted entity in a different platform. We perform the prediction once from Ⓐ to Ⓑ and then another from Ⓑ to Ⓐ. For example, we find that XBA aligns two string literals "`HOME`" and "`USERPROFILE`" used to specify a user's home directory in Linux and Windows, respectively. The results show that XBA's embeddings can effectively capture the similarities

[1]Starting from the implementation of these baseline models as was open-sourced by their authors, we carefully refined the implementation as is described in the corresponding papers. These refined models were trained using our dataset from scratch for a comparison of XBA over strong baselines. Specifically, we (i) modified DeepBinDiff in their favor such that it uses IDA Pro's more precise control flow graph instead of Angr's [52], and (ii) implemented InnerEye's training model based on the inference model open-sourced by the authors.

Table 6: Prediction accuracy of basic block alignment tasks across platforms. XBA outperforms the baselines (InnerEye, DeepBinDiff, and BoW). All the models are trained on 50% of the pre-aligned basic block pairs and evaluated on the other 50%.

Cross-OS		**Linux→Windows**				**Windows→Linux**			
Configuration	Dataset	Hits@1	Hits@10	Hits@50	Hits@100	Hits@1	Hits@10	Hits@50	Hits@100
InnerEye	SQLITE3	70.38	79.27	85.90	90.10	71.52	80.06	86.78	89.76
BoW Encoding	SQLITE3	84.13	89.60	92.83	96.94	85.80	91.59	94.10	94.57
BoW Encoding + **XBA**	SQLITE3	97	99.32	99.65	99.71	97.01	99.41	99.75	99.82
DeepBinDiff	SQLITE3	78.14	86.14	90.07	91.45	80.13	87.32	90.80	92.35
DeepBinDiff + **XBA**	SQLITE3	94	98.14	99.31	99.54	94.57	98.64	99.32	99.54
InnerEye	OPENSSL	59.10	70.86	81.04	86.20	63.19	73.80	82.48	85.17
BoW Encoding	OPENSSL	70.47	80.80	87.96	89.28	78.59	85.08	90.66	92.38
BoW Encoding + **XBA**	OPENSSL	88.17	99.05	99.65	99.79	90.33	99.10	99.69	99.82
DeepBinDiff	OPENSSL	63.48	79.12	86	88.50	84.65	94.66	96.62	97.14
DeepBinDiff + **XBA**	OPENSSL	88.23	97.81	99.22	99.43	90.24	98.46	99.39	99.62
InnerEye	CURL	73.56	81.06	90.20	92.87	74.87	83.15	89.47	91.47
BoW Encoding	CURL	85.21	90.99	97.17	97.76	87.63	92.15	94.34	98.33
BoW Encoding + **XBA**	CURL	88.59	97.83	99.17	99.35	88.08	96.82	98.22	98.53
DeepBinDiff	CURL	76.57	85.46	89.96	91.84	79.17	87.44	90.95	92.55
DeepBinDiff + **XBA**	CURL	88.27	98.08	99.39	99.62	89.05	97.75	98.68	98.97
InnerEye	HTTPD	72.27	83.98	95.09	96.27	73.43	84.38	91.57	92.99
BoW Encoding	HTTPD	84.02	90.79	97.52	98.24	86.61	91.08	93.60	97.89
BoW Encoding + **XBA**	HTTPD	87.95	98.43	99.44	99.64	87.46	97.65	98.88	99.24
DeepBinDiff	HTTPD	68.56	83.14	90.60	92.77	65.22	79.78	87.19	90.76
DeepBinDiff + **XBA**	HTTPD	81.57	95.89	98.19	98.92	81.82	95.22	97.59	98.48
InnerEye	LIBCRYPTO	68.96	73.50	78.26	80.42	68.15	73.01	77.11	79.92
BoW Encoding	LIBCRYPTO	78.51	83.99	87.02	88.80	77.81	82.75	86.29	88.03
BoW Encoding + **XBA**	LIBCRYPTO	79.03	94.28	97.01	97.81	81.33	95.06	97.38	97.92
DeepBinDiff	LIBCRYPTO	68.47	79.78	83.71	85.32	67.79	78.80	82.47	84.25
DeepBinDiff + **XBA**	LIBCRYPTO	76.34	90.40	94.47	95.65	78.56	91.37	94.81	95.73
Cross-ISA		**x86_64→AArch64**				**AArch64→x86_64**			
Configuration	Dataset	Hits@1	Hits@10	Hits@50	Hits@100	Hits@1	Hits@10	Hits@50	Hits@100
InnerEye	GLIBC	1.58	2.37	6.04	9.49	1.52	3.05	6.65	9.89
BoW Encoding	GLIBC	11.02	12.50	13.87	15.47	3.64	8.95	10.46	11.75
BoW Encoding + **XBA**	GLIBC	49.92	80.86	91.52	93.83	51.50	81.76	91.44	94.18
DeepBinDiff	GLIBC	2.96	5.41	8.74	11.09	2.74	5.44	8.34	10.26
DeepBinDiff + **XBA**	GLIBC	29.92	62.77	81.40	87.35	30.63	62.54	81.65	87.56
InnerEye	LIBCRYPTO	4.07	9.15	14	14.52	2.77	3.03	3.51	10.72
BoW Encoding	LIBCRYPTO	6.04	11.34	11.48	11.48	7.46	11.63	11.77	11.77
BoW Encoding + **XBA**	LIBCRYPTO	55.32	83.41	92.48	94.75	59.78	87.48	94.35	95.86
DeepBinDiff	LIBCRYPTO	1.82	3.45	5.92	7.07	1.85	3.51	4.90	5.84
DeepBinDiff + **XBA**	LIBCRYPTO	28.76	61.70	79.79	85.65	30.71	64.22	81.73	87.27

between OS-specific API functions, and even OS-specific string literals whose semantics are embodied in their usages in software written to be portable across different OSes.

Cross-ISA: Handwritten Assembly Function Matching. We now turn our attention to cross-ISA scenarios. Here, we manually align semantically-similar functions found in LIBCRYPTO across platforms, where (i) none of their basic blocks are included in the pre-aligned pairs, and (ii) at least one side of the aligned pair is an ISA-specific handwritten assembly function. We then run XBA to generate embeddings of all basic blocks comprising the two functions in each function pair, and, for every possible pair of their basic blocks across platforms, we compute the rank of each other. In the bottom rows of Table 7, we report the prediction with the highest rank. We observe that XBA places these unlabeled functions tailored for each ISA at high ranks, *e.g.*, ISA-specific functions that use purpose-built instructions such as Intel AES NI [27].

Cross-ISA: Vulnerable Function Matching. We also study, as an application of XBA embeddings, a known potential side channel issue[2] found in LIBCRYPTO. This vulnerability affects all ISA-specific versions of a function called `ecp_nistz256_point_add`, which implements a low-level Elliptic Curve operation with ISA-specific handwritten assembly. There are six different versions of

[2] https://github.com/openssl/openssl/pull/9239

Table 7: XBA's alignment prediction results of either OS-specific (Linux ⇄ Windows) or ISA-specific (x86_64 ⇄ AArch64) entities not included in the training set. All entities are either external or internal functions, except for two OS-specific strings "HOME" and "USERPROFILE". Ranks are zero-indexed, and all ranks translate to top 0.88% or higher.

		Rank (Top %)	
Ⓐ Linux	Ⓑ Windows	Ⓐ→Ⓑ	Ⓑ→Ⓐ
read	ReadFile	104 (**0.53%**)	72 (**0.32%**)
fileno	GetStdHandle	1 (**0.0051%**)	2 (**0.0091%**)
strcasecmp	stricmp	2 (**0.010%**)	5 (**0.022%**)
gettimeofday	GetSystemTimeAsFileTime	355 (**0.88%**)	25 (**0.060%**)
chdir	SetCurrentDirectory	157 (**0.39%**)	96 (**0.23%**)
"HOME"	"USERPROFILE"	84 (**0.20%**)	33 (**0.080%**)
mmap	VirtualAlloc	110 (**0.10%**)	2 (**0.0018%**)
mprotect	VirtualProtect	10 (**0.0091%**)	9 (**0.0081%**)
Ⓐ x86_64	Ⓑ AArch64	Ⓐ→Ⓑ	Ⓑ→Ⓐ
aesni_gcm_encrypt	aes_v8_gcm_encrypt	39 (**0.035%**)	137 (**0.12%**)
aesni_gcm_decrypt	aes_v8_gcm_decrypt	9 (**0.0081%**)	14 (**0.012%**)
gcm_init_avx	gcm_init_v8	10 (**0.0090%**)	1 (**0.00090%**)
aesni_gcm_initkey	armv8_aes_gcm_initkey	17 (**0.015%**)	36 (**0.032%**)
aesni_set_encrypt_key	aes_v8_set_encrypt_key	260 (**0.23%**)	105 (**0.094%**)
sha256_block_data_order_shaext	sha256_block_armv8	348 (**0.31%**)	38 (**0.034%**)
Camellia_Ekeygen	Camellia_Ekeygen	77 (**0.069%**)	89 (**0.080%**)
ecp_nistz256_point_add	ecp_nistz256_point_add	14 (**0.0012%**)	107 (**0.096%**)

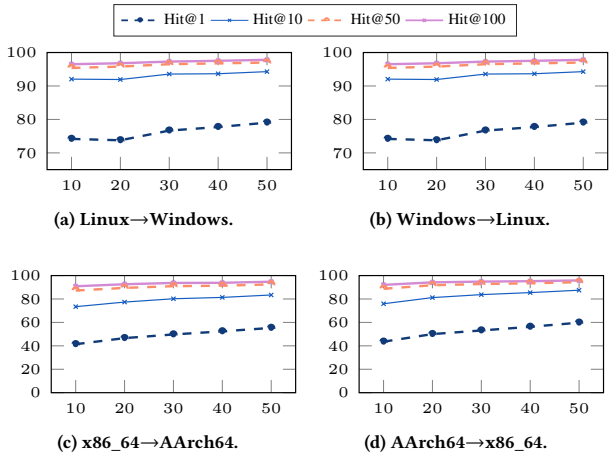
Table 8: XBA outperforms in predicting OOD alignments. We train XBA on all our cross-OS datasets except SQLITE3. We compare our results with the BoW-Encoded embeddings.

Linux→Windows	Hit@1	Hit@5	Hit@10	Hit@100
BoW Encoding	84.13	89.60	92.83	96.94
BoW Encoding + **XBA**	**91.55**	**97.40**	**98.77**	**99.09**
Windows→Linux	Hit@1	Hit@5	Hit@10	Hit@100
BoW Encoding	85.80	91.59	94.10	94.57
BoW Encoding + **XBA**	**91.21**	**97.23**	**98.67**	**99.01**

this function, each written for x86_64 or AArch64. We ensure that all these versions do not appear in the aligned entity pairs. XBA was able to predict the corresponding entities successfully in both directions (*i.e.*, from x86_64 to AArch64 and vice versa), as shown in the last row of Table 7. This case study sheds light on an interesting security application. If a specific vulnerability is pervasive across different platforms and the vulnerability can be defined in the embedding space of XBA, we could perform a vulnerability search in a platform-agnostic manner using XBA.

4.5 RQ3. Predicting OOD Alignments

Another interesting research question is how accurate XBA is for predicting out-of-distribution (OOD) alignments. Here, we investigate whether XBA produces the embeddings for binaries that do not appear in the training data nor the testing data. The baselines may find it difficult to perform well on such OOD inputs as they *strictly* learn embeddings from labeled data, or their methods are hard to encode rich semantics into embeddings. However, as we learn embeddings from large unaligned entities from BDGs, XBA could find several *generalizable* features. Table 8 shows our results.



(a) Linux→Windows. (b) Windows→Linux.

(c) x86_64→AArch64. (d) AArch64→x86_64.

Figure 3: Cross-platform alignment results on LIBCRYPTO using BoW-encoded feature vectors with 5 different training/test dataset split ratios (x-axis). 10% means 10% of the groundtruth alignments were used as a training dataset, and the rest 90% as a test dataset. The alignment accuracy (y-axis) was measured under the Hit@K metric with varying Ks.

Training and Testing Details. Here, we train XBA on all our cross-OS datasets except SQLITE3. We use the BoW-encoded feature vectors as the input feature vectors of XBA, and compare our results with the case of not using XBA. We compare our results with the BoW-Encoding as this method performs better than InnerEye and DeepBinDiff on both cross-OS and cross-ISA. We use the trained XBA to predict the alignments in BDGs constructed from SQLITE3.

XBA Outperforms the BoW-Encoding in OOD Alignments. Across the border, we observe that XBA improves the BoW-Encoding in OOD alignment predictions. In both Linux to Windows and Windows to Linux scenarios, XBA achieves a 3–8% better accuracy than the BoW-Encoding. Similar to §4.3, we find that the improvement increases when our analysis is needed to be precise (Hit@1).

4.6 RQ4. Sensitivity to Labeled Data Size

Another potential concern is that XBA's performance can be sensitive to the size of pre-aligned entity pairs in the training data. As XBA takes a semi-supervised approach, less supervision signals may lead to lower-quality embeddings for cross-platform analysis. We evaluate the scenario in §4.3, but using different amounts of pre-aligned entities {10%, 20%, 30%, 40%, 50%} as the training dataset. We conduct this experiment on LIBCRYPTO, the largest binary in our dataset, for both cross-OS and cross-ISA alignment prediction.

XBA Is Not Sensitive to the Size of Pre-aligned Data. Figure 3 shows the results. We observe that, in all four directions of the cross-platform alignment tasks, the prediction accuracy of XBA is consistent across the board. The accuracy difference between the 10% and 50% cases is less than 6%. The results suggest that XBA effectively captures contextual information available in BDGs even with a small amount of pre-aligned entity pairs.

Table 9: XBA performance after ablating each relation type in BDGs measured using the CURL dataset.

	Hit@K (Averaged over $K = \{1, 10, 50, 100\}$)	
Ablation	**Linux→Windows**	**Windows→Linux**
XBA	96.23	95.42
without ❸	96.07	95.50
without ❹	95.45	95.09
without ❷	92.46	91.25
without ❶	73.06	80.06

4.7 RQ5. Relation Type Ablation

We investigate the importance of each type of relations in generating semantically rich and aligned binary code embeddings. We ablate each relation type by setting the weights of all relations of that type in the adjacency matrix to zero during alignment inference. This effectively prevents messages from being passed through any edge of that type, meaning that the embeddings are generated only with the contextual information propagated through edges of other types. We followed the same experimental methodology described in §4.3 on CURL with varying Ks, and report the average in Table 9. We can observe that XBA draws most of the semantics from control-flow transfer relations (both ❶ and ❷), which are the two most common types of relations. Whereas the relations defined by address-taking code-to-code references, the least common type of relations in our datasets, (❸) rarely affect the accuracy for the given alignment task. We speculate, however, that these relations could still be useful in aligning the unlabeled pairs, *e.g.*, when they constitute the only path from an unlabeled node to the labeled ones.

5 DISCUSSION

Threats to Validity. Deep learning approaches, in general, are sensitive to hyperparameter settings. While XBA outperforms the baseline models in most cases, we found that the quality of embeddings is highly sensitive to a few hyperparameters such as the number of negative samples. To mitigate this threat, we conducted the whole experiments multiple times with a few different settings. We believe XBA can be further improved by more thorough hyperparameter tuning. We implemented the baseline models by using the default hyperparameters either as described in the corresponding papers or as implemented in the publicly available prototype. To ensure the validity of the baseline model implementations, we reproduced part of the experiments described in the papers. Their models, however, were trained on our own benchmark dataset, which might have caused some performance degradation. Also, for XBA to learn useful platform-specific knowledge, benchmarks must have (i) multi-platform support and (ii) platform-specific code. We chose our benchmarks from the top GitHub repositories; that is, they are widely-used open-source software that satisfy the criteria. We acknowledge, however, that they may not be representative enough. Another threat can be found in the selection process of manually aligned binary code pairs in Table 7. Though we showed the ranks of many OS-specific and ISA-specific entities, there will be many more, and XBA may not be able to align all of such entities.

Oversmoothing Effects. An N-layer message passing GNNs such as GCNs, by design, are not able to collect information embedded in the neighbor nodes that are more than N-hops away. This means that XBA, as it uses N-layer GCNs, may not align unlabeled node pairs that are more than N-hops away from the labeled ones in BDGs. For instance, XBA failed to align the embeddings of `mlock` and `VirtualLock` found in the LIBCRYPTO binaries compiled for Linux and Windows, respectively. The two functions have the same semantics—*i.e.*, locking part of the virtual address space into the physical memory. Their neighbor nodes including the ones that called the two functions, unfortunately, are all OS-specific, and there are no labeled nodes within 5-hops. We attempted to increase the number of GCN layers to more than five, which, unfortunately, lowered the overall performance of XBA due to the oversmoothing effect. Replacing GCNs used in XBA with more advanced GNNs [50, 64] could potentially mitigate oversmoothing effects even with a large N, and we leave this as future work.

Node Feature Selection. XBA requires the base node features to be fed into the first layer of the GCNs, for which we used BoW-encoded features and pretrained DeepBinDiff embeddings in our evaluation. The results of our evaluation show that the selection of input features can have a significant impact on the overall accuracy. Future work can explore different ways to construct node features, and investigate their effectiveness in improving the performance of XBA. We note that XBA does not restrict the dimensionality of the input features, and do not require the features be aligned across various platforms.

6 RELATED WORK

Graph Alignment with Graph Embedding. Diverse methods have been proposed to solve the graph alignment problem in various domains [5, 30, 53], such as spectral methods, clustering, decision trees, and distributed belief propagation. A well-studied method is to use graph embedding [7]. It first learns to transform graph entities, *e.g.*, nodes, attributes, or relations, into embeddings and use them to quantify the alignment between two (sub-)graphs. Early work like node2vec [22] and DeepWalk [47] used random walks to encode structural information in node-level embeddings. While this approach could work in matching similar graphs, they are ineffective when graphs being matched contain different entities or have different structures. Recent work proposed techniques that uses GNNs to learn embeddings aligned across dissimilar graphs, such as cross-lingual [56] or heterogeneous [57] knowledge graph alignment. Inspired by this success, our work also uses GNNs to generate embeddings more suitable for cross-platform binary analysis.

Using GNNs for Binary Code Simialrity. For binary code similarity analysis, a number of approaches (*e.g.*, n-gram, fuzzy hash, dynamic analysis, optimization) have been proposed. A recent study by Marcelli et al. [39] shows that, GNN models outperformed other deep learning models in most experimental settings. The pioneering work that uses GNNs is Gemini [59], which uses a Structure2vec [13] to generate function-level embeddings from an Attributed Control-Flow Graphs (ACFGs) whose node attributes are manually engineered. Li et al. later proposed a similar approach [37], which used a graph matching model that can measure the similarity between two distinct graphs. The authors, however, explored the

binary similarity problem as one of the downstream tasks, and, thus, the experiments are somewhat limited. In comparison with prior work that uses manually crafted node features, Massarelli et al. [40] proposed instruction2vec, an unsupervised learning approach, to learn the basic block level features that are fed into the same Structure2vec of Gemini [59]. This work combined a Natural Language Processing technique (NLP) (*e.g.*, Word2Vec) for node base features and graph embedding model.

NLP Techniques for Binary Code Embedding. Many attempts have been made to generating rich binary code embeddings without manually-engineered features by using techniques proposed in the NLP field. The basic idea is to treat the assembly language as a sequence of sentences and instructions as words. A model commonly used by prior work is Word2Vec [41, 42] that can extract the structural information of words without labeled data, which is empolyed in Asm2Vec [17] and DeepBinDiff [18]. In contrast to these unsupervised approaches, which cannot produce embeddings aligned across platforms, InnerEye [65] and CodeCMR [63] employ a Seq2Seq encoder-decoder model [55] to capture semantic alignments between different platforms. More recently, solutions [46, 62] based on BERT [15] are proposed with various language models for pre-training. XBA could further be improved by using a more advanced NLP technique, which we leave as future work.

7 CONCLUSION

This paper presents a new representation learning approach that can facilitate cross-platform binary analysis. We formulate representation learning as a *graph alignment problem*, *i.e.*, finding semantic alignments between nodes in graph-structured binary code across platforms. To solve this problem, we propose XBA, a deep learning-based method that uses graph convolutional networks (GCNs) to learn embeddings for different entities found in a disassembled binary, such as basic blocks, external functions, and strings. XBA builds binary disassembly graphs that contain rich contextual information of these entities, which are then captured by GCNs in learning their embeddings. XBA, as it is trained with the objective of graph alignment, learns embeddings of the entities in a semi-supervised manner, requiring only a limited number of cross-platform alignment labels to predict new potential alignments. Our evaluation shows that XBA can, for cross-platform binary similarity analysis, improve existing embedding methods by a significant margin. Our case studies show that embeddings produced by XBA encode knowledge useful for cross-platform binary analysis.

DATA-AVAILABILITY STATEMENT

In an effort to address recent concerns raised on the reproducibility of machine learning approaches to the binary code similarity problem [39], we release our dataset and source code that can reproduce all of our experiments at https://github.com/yonsei-cysec/XBA and also on Zenodo [28]. Our experiment results can also be found at https://sites.google.com/view/xba-intro.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This material is based upon work partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) under awards 2021R1F1A1057436 and 2022R1C1C1003551, the Office of Naval Research (ONR) under awards N00014-21-1-2409 and N00014-17-1-2232, the Defense Advanced Research Projects Agency (DARPA) under award N66001-20-C-4027, and the Defense Advanced Research Projects Agency (DARPA) Small Business Technology Transfer (STTR) Program Office under contracts W31P4Q-20-C-0052 and W912CG-21-C-0020. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NRF, MSIT, ONR, DARPA, the DARPA STTR Program Office, or any other South Korea and U.S. government agency. We also gratefully acknowledge an "Endeavor" research award from the Donald Bren School of Information and Computer Sciences at UC Irvine, and an award from the Yonsei University Research Fund of 2021 (2021-22-0039).

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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[2] Emily Alsentzer, Samuel G Finlayson, Michelle M Li, and Marinka Zitnik. 2020. Subgraph Neural Networks. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.

[3] Arm Limited. [n. d.]. Arm A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0596/2021-12.

[4] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *Proceedings of the USENIX Security Symposium*.

[5] Mohsen Bayati, David F Gleich, Amin Saberi, and Ying Wang. 2013. Message-passing algorithms for sparse network alignment. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 7, 1 (2013), 1–31.

[6] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2013. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203* (2013).

[7] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering* 30, 9 (2018), 1616–1637.

[8] 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 ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*.

[9] Muhao Chen, Yingtao Tian, Mohan Yang, and Carlo Zaniolo. 2016. Multilingual knowledge graph embeddings for cross-lingual knowledge alignment. *arXiv preprint arXiv:1611.03954* (2016).

[10] Sumit Chopra, Raia Hadsell, and Yann LeCun. 2005. Learning a similarity metric discriminatively, with application to face verification. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *Proceedings of the USENIX Security Symposium*.

[12] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the USENIX Security Symposium*.

[13] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative Embeddings of Latent Variable Models for Structured Data. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[14] Yaniv David, Uri Alon, and Eran Yahav. 2020. Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 225 (Nov. 2020), 28 pages.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional Transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[16] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's debugging the debuggers? Exposing debug information bugs in optimized binaries. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[17] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the IEEE Symposium on*

Security and Privacy (IEEE S&P).
[18] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning program-wide code representations for binary diffing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*
[19] Thomas Dullien and Sebastian Porst. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest* (2009).
[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*
[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*
[22] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD).*
[23] Zellig S Harris. 1954. Distributional structure. *Word* 10, 2-3 (1954), 146–162.
[24] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*
[25] John Hennessy. 1982. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 323–344.
[26] hex-rays. [n. d.]. IDA Pro: A powerful disassembler and a versatile debugger. https://hex-rays.com/ida-pro.
[27] Intel. [n. d.]. Intel Advanced Encryption Standard (AES) New Instructions Set. https://www.intel.com/content/dam/develop/external/us/en/documents/aes-wp-2012-09-22-v01-165683.pdf.
[28] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. XBA. Zenodo. https://doi.org/10.5281/zenodo.6579248
[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[30] Gunnar W Klau. 2009. A new graph-based method for pairwise global network alignment. *BMC bioinformatics* 10, 1 (2009), 1–9.
[31] Yonghwi Kwon, Weihang Wang, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. CPR: Cross platform binary code reuse via platform independent trace program. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).*
[32] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).*
[33] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the International Conference on Machine Learning (ICML).*
[34] Chengjiang Li, Yixin Cao, Lei Hou, Jiaxin Shi, Juanzi Li, and Tat-Seng Chua. 2019. Semi-supervised entity alignment via joint knowledge embedding model and cross-graph model. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing and the International Joint Conference on Natural Language Processing (EMNLP-IJCNLP).*
[35] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*
[36] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug information validation for optimized code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*
[37] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *Proceedings of the International Conference on Machine Learning (ICML).*
[38] 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 IEEE/ACM International Conference on Automated Software Engineering (ASE).*
[39] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *Proceedings of the USENIX Security Symposium.*
[40] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR).*
[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[42] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS).*
[43] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005* (2017).
[44] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2020. XDA: Accurate, robust disassembly with transfer learning. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*
[45] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs?. In *Proceedings of the International Conference on Machine Learning (ICML).*
[46] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs?. In *Proceedings of the International Conference on Machine Learning (ICML).*
[47] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD).*
[48] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*
[49] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC).*
[50] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *Proceedings of the International Conference on Learning Representations (ICLR).*
[51] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *Proceedings of the USENIX Security Symposium.*
[52] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*
[53] Rohit Singh, Jinbo Xu, and Bonnie Berger. 2008. Global alignment of multiple protein interaction networks with application to functional orthology detection. *Proceedings of the National Academy of Sciences* 105, 35 (2008), 12763–12768.
[54] Zequn Sun, Wei Hu, and Chengkai Li. 2017. Cross-lingual entity alignment via joint attribute-preserving embedding. In *International Semantic Web Conference.* Springer, 628–644.
[55] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS).*
[56] Zhichun Wang, Qingsong Lv, Xiaohan Lan, and Yu Zhang. 2018. Cross-lingual knowledge graph alignment via graph convolutional networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP).*
[57] Yuting Wu, Xiao Liu, Yansong Feng, Zheng Wang, Rui Yan, and Dongyan Zhao. 2019. Relation-aware entity alignment for heterogeneous knowledge graphs. *arXiv preprint arXiv:1908.08210* (2019).
[58] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).
[59] 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 ACM Conference on Computer and Communications Security (CCS).*
[60] Cheng Yang, Zhiyuan Liu, Deli Zhao, Maosong Sun, and Edward Chang. 2015. Network representation learning with rich text information. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI).*
[61] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *Proceedings of the USENIX Security Symposium.*
[62] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI).*
[63] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-modal retrieval for function-level binary source code matching. *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS).*
[64] Lingxiao Zhao and Leman Akoglu. 2020. PairNorm: Tackling Oversmoothing in GNNs. In *Proceedings of the International Conference on Learning Representations (ICLR).*
[65] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*