Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning

Anonymous Author(s)*

ABSTRACT

Binary code similarity detection serves as a basis for a wide spectrum of applications, including software plagiarism, malware classification, and known vulnerability discovery. However, the inference of contextual meanings of a binary is challenging due to the absence of semantic information available in source codes. Recent advances leverage the benefits of a deep learning architecture into a better understanding of underlying code semantics and the advantages of the Siamese architecture into better code similarity detection.

In this paper, we propose BinShot, a BERT-based similarity learning architecture that is highly transferable for effective binary code similarity detection. We tackle the problem of detecting code similarity with one-shot learning (a special case of few-shot learning). To this end, we adopt a *weighted distance vector* with a *binary cross entropy* as a loss function on top of BERT. With the prototype implementation of BinShot, our experimental results demonstrate the effectiveness, transferability, and practicality of BinShot, which is robust to detecting the similarity of previously unseen functions. We show that BinShot outperforms the previous state-of-the-art approaches for binary code similarity detection.

1 INTRODUCTION

Today, software arguably plays a pivotal role in operating billions of computing systems around us, including desktops, laptops, workstations, mobile phones, cloud servers, and IoT (Internet of Things) devices. Each program comprises a collection of instructions, routines, and data associated with a certain task, most of which (*e.g.*, off-the-shelf applications) are distributed as an executable binary form. Albeit the surge in programs' complexity and size, software development has been evolving in a simplified manner with a myriad of well-designed libraries, toolkits, modules, and frameworks available. While code reuse benefits from high productivity and handy code maintenance, it comes along with unforeseen threats such as copyright infringement [87], the prevalence of malware variants [6], and vulnerable code reproduction [43].

Binary code similarity detection is the task of determining whether a pair of code snippets (*i.e.*, assembly function) is similar when the original source code is unavailable. The task serves as a basis for a wide range of applications, including code clone detection [20, 21, 24, 26, 40, 60, 62, 76, 85, 92–95, 98], malware detection [7, 9, 44, 51, 96], malware family classification [38, 56], known vulnerability discovery [11, 78, 89], and code patching [29, 39, 90]. However, the inference of contextually underlying meanings from a stripped binary is non-trivial because ① a complex compilation process eliminates lots of semantic information (*e.g.*, variable name, structure, type, class hierarchy) with massive transformations and optimizations, ② the form of (semantically equivalent) binaries can vary depending on compilers, compiler versions, compiler optimizations, architectures, or even obfuscation techniques, and ③ it is undecidable to prove that two arbitrary programs are functionally equivalent (*i.e.*, Halting Problem [80]).

To tackle a code similarity detection problem from such dynamic nature of binary code generation, early works adopt static approaches with diverse algorithms, including graph isomorphism detection [4, 25, 99], symbolic execution [30, 60], or data flow analysis [20, 21, 71], which helps extract useful information such as instructions, control flow, and data values. Another direction utilizes dynamic approaches [10, 26, 45, 63, 64, 70, 86] by observing a binary execution to understand the code semantics better; however, it often suffers from poor scalability and incomplete code coverage.

Meanwhile, with the rise of machine learning (ML) techniques, various ML-based approaches have been proposed [24, 28, 57, 61, 62, 81, 91–93, 98] in the field of binary code similarity detection. Moreover, recent advancements (*e.g.*, deep learning) in the area of natural language processing (NLP) inspire the inference of machine-interpretable code [24, 62, 93, 98], which repeatedly demonstrates noticeable results. For example, InnerEye [98] views an assembly instruction, a basic block, and a function as a word, a sentence, and a paragraph, respectively. Lately, the emergence of the BERT architecture [23] and its descendants [52, 58, 75] assists in better deduction of the underlying context. A handful of approaches [49, 54, 70, 85, 94] leverage BERT into various downstream tasks, including binary code similarity detection.

The essence of a binary code similarity detection problem with ML-based approaches is to learn a similarity metric from data so that a model appropriately determines the similarity of a (previously unseen) new sample. As the number of semantically equivalent binary codes is limited to learn, learning a similarity metric is a suitable approach (*e.g.*, a Siamese neural network) for an N-way k-shot classification problem (*i.e.*, N classes with k examples of each). A number of prior approaches [11, 54, 57, 61, 62, 70, 81, 85, 91, 92, 94, 98] utilize a Siamese neural network [5, 12], which computes a similarity score with a distance function that represents the proximity of two vectors (*e.g.*, function embeddings). Taking it to the next level, we view a task of identifying semantically analogous code as an *N-way one-shot classification problem* (*i.e.*, one-shot learning), a special case ($k = 1$) of an N-way k-shot classification problem (*i.e.*, few-shot learning in §2). In particular, it fits into a practical scenario that seeks the presence of a particular function in a pre-built database.

In this paper, we propose BinShot, a BERT-based transferable similarity learning architecture (with a Siamese neural network) for effective binary code similarity detection. The main difference is to learn a *weighted distance vector* with a *binary cross entropy loss function* (suggested by Koch et al. [48]), while the existing models (for both BERT-based and non-BERT-based ones) learn a *distance itself*. Leveraging BERT, we take two-phase training: pre-training for building a generic model with assembly code, followed by fine-tuning for building a downstream model for a binary

similarity detection task. Note that one of the popular loss functions is a contrastive loss [12] used in previous learning-based approaches [11, 54, 57, 62, 92, 98].

We implemented a full-fledged prototype of BinShot that consists of four components: ① pre-processor for learning preparation, ② pre-trainer for building a machine code model, ③ fine-tuner for building a code similarity model, and ④ predictor for detecting binary similarity with a given code snippet (*i.e.*, function). We build 1,400 binaries from varying source code, which contains around 1.77 million functions. To the best of our knowledge, we first conduct an experiment for transfer learning in the field of binary similarity detection. Our evaluations for both effectiveness and transferability demonstrate that BinShot surpasses previous state-of-the-art models for detecting binary code similarity, including Gemini [92], Asm2Vec [24], PalmTree [54], and DeepSemantic [49]. Further, we set up a realistic scenario that detects vulnerable functions. Surprisingly, previous approaches [24, 49, 54, 92] reveal a quite low accuracy (< 15%) due to large false positives, remaining their practicality questionable. Finally, we visualize how BinShot can distinguish a binary function from a (dis)similar one.

The following summarizes the contribution of our paper:

- We propose BinShot that learns a weighted distance vector with a binary cross entropy atop the BERT-based Siamese architecture for binary code similarity detection.
- We design and implement the prototype of BinShot. We plan to open-source BinShot[1] to foster the area of binary similarity detection in the future.
- We evaluate BinShot to present its effectiveness, transferability, and practicality by comparing it with state-of-the-art baseline models.
- We demonstrate that BinShot represents similar embeddings well in a vector space via visualization.
- We investigate interesting (but easily overlooking) cases, providing an in-depth study and discussion.

2 BACKGROUND

Bidirectional Encoder Representations from Transformers. BERT [23] is a transformer-based machine learning architecture that originates from the domain of NLP. The Transformer [83] architecture gains high popularity in text processing tasks with noticeable results. It adopts ① the self-attention mechanism that aims to infer contextual information per each word (considering a position) within the input sentence, and ② the design that allows for parallel processing during training on a large volume of a dataset, which reduces training time over sequential processing like Long Short-Term Memory (LSTM) [37]. The BERT architecture leverages a careful design choice from the Transformer's encoder (*e.g.*, multi-head self-attention) into various downstream applications with two major phases (*i.e.*, pre-training and fine-tuning). First, the pre-training of BERT aims to capture both token-level and sentence-level contextual information, building a generic model that contains vectors (*i.e.*, embeddings) of each token. A final input embedding representation considers a position (*e.g.*, location of a word in a sentence) and a segment (*e.g.*, a sentence that a word belongs to) as well as a token itself. For capturing hidden context

[1]https://anonymous.4open.science/r/binshot-C824/

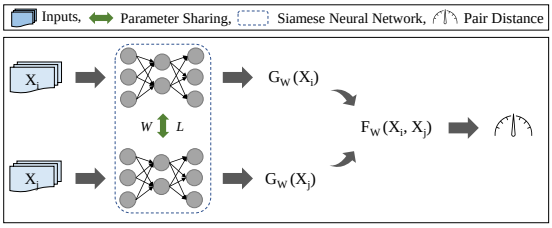


Figure 1: A popular loss function in a Siamese neural network (NN): a contrastive loss (L_c) using a twin NN. G, F, X, and W denotes an output vector, similarity metric, input, and weight, respectively.

from a sentence, pre-training performs two sub-tasks: masked language modeling (MLM) and next sentence prediction (NSP). MLM randomly masks a certain portion of tokens in a given sentence (*e.g.*, 15% in the original BERT scheme), exploiting unlabeled data (*i.e.*, masked positions) to yield labels (*i.e.*, original tokens). The special token, [CLS], is placed at the beginning of every input, where [SEP] represents a token separator so that two sentences can be concatenated as a single input. Meanwhile, NSP predicts if the next sentence pertains to the current one. Note that pre-training does not require a laborious labeling process (*i.e.*, unsupervised learning), creating a generic model to be served as a base knowledge layer to build from. Second, the fine-tuning phase of BERT improves a generic model by retraining weights with another dataset for a specialized user-defined task (*i.e.*, supervised learning), which allows for being adjusted to task-oriented contextual representations. In this paper, we build a pre-trained BERT model for binary code representation and a fine-tuned model for binary code similarity detection as a downstream task.

Siamese Neural Network. A Siamese neural network [5] was first introduced for verifying signatures, in which multiple identical subnetworks extract features from multiple inputs while training. The gist of the Siamese neural network is to compute a *distance function* that represents how two feature vectors are close together; that is, a set of similar inputs should be encoded as adjacent as possible, whereas that of dissimilar inputs should be placed as far as possible. Unlike traditional approaches (*e.g.*, support vector machines) that require known labels ahead of training time for a classification problem, the architecture is suitable for which there are a large number of unknown categories with a limited number of datasets at the time of training [47]. More precisely, we can formally define a *similarity metric* (F_W) between two inputs (X_i and X_j) with a distance function (D), where $G_W(X)$ represents an output vector in a neural network model that is parameterized by W (*i.e.*, weight) as follow:

$$F_W(X_i, X_j) = D(G_W(X_i), G_W(X_j)) \quad (1)$$

The distance function in Equation 1 between two n-dimensional vectors may vary [65], including an L1 norm (*i.e.*, Manhattan distance), L2 norm (*i.e.*, Euclidean distance), inner product (*i.e.*, scholar or dot product), or cosine similarity (*i.e.*, inner product divided by the product of two vectors' lengths) in a space. The similarity metric can be learned by seeking W that minimizes a loss function over a training set. With a distance function, a Siamese neural network regards the resulting distances as the degree of two inputs' similarity. A Siamese neural network relies on a pre-defined loss function

as well as a distance function that one chooses from. The following exemplifies one of the popular loss functions (Figure 1), contrastive loss (L_c in Equation 2). A contrastive loss minimizes the distance of positive pairs while maximizing the distance of negative pairs. Note that Y represents a label (0 or 1) for the pair of inputs (X_i, X_j).

$$L_c(W, Y, X_i, X_j) = (1 - Y)\frac{1}{2}(F_W)^2 + Y\frac{1}{2}(max(0, F_W))^2 \quad (2)$$

Few-shot Learning. Few-shot learning differs from classical supervised learning in that it aims to *learn how to learn* (*e.g.*, two objects are alike) instead of letting a model directly recognize a label (*e.g.*, an object is a dog). The main idea [5, 12, 48, 77] stems from computer vision tasks, which are inspired by a human capability that learns a new object based on previously acquired information with a limited number of instances. Few-shot learning is particularly fruitful when generalizing prior knowledge for data-hungry applications. In general, classifying N classes with k samples per each category (*i.e.*, few samples) is known as an N-way k-shot classification problem[2]. Simply put, a dataset is divided into two mutually exclusive sets: a support set ($N \times k$) for training and a query set for testing, followed by learning similarity between feature vectors. Signature verification [5] and face verification [12, 77] are representative examples of few-shot learning applications. A Siamese neural network is one of the few-shot learning enablers, which fits well into binary similarity detection. Ideally, a well-learned model from other function pairs can successfully identify a distance from a given (possibly unseen) function pair with the model.

3 BINARY CODE SIMILARITY DETECTION

This section briefly describes the definition, scope, and challenges of a binary code similarity detection problem.

Problem Definition and Scope. A task of detecting binary code similarity is defined as identifying the proximity of two or more machine codes (*i.e.*, assembly) when source code is unavailable. The task may vary depending on different aspects of code comparison. For instance, the comparison granularity of a code snippet can be an entire program, function, basic block, or even instruction. One may view the detection task as a code search problem, in which the number of code pieces can be one-to-one, one-to-many, or many-to-many. Furthermore, binary code may hold equivalent semantics with diverse code transformations from different compilers, different compiler options, different compiler versions, different compiler optimization levels, different architectures, or different obfuscations. Meanwhile, it may represent similar semantics with distinct updates from different source code versions or bug fixes (patches). In this work, we deal with detecting similar code snippets whose transformation solely originates from *cross-compiler* and *cross-optimization-levels* for the `x86_64` architecture. In particular, we focus on seeking a similar binary *function pair* from a prepared function corpus (*one-to-many* setting) in practice.

Open Challenges. The dynamic nature of diversified binary code generation inevitably makes code similarity detection quite challenging. It is undecidable to prove that two arbitrary programs are functionally and semantically equivalent (*i.e.*, Halting Problem [80]). Note that our methodology is agnostic to other transformations from compiler options or versions; however, we deliberately do not consider the differences in code semantics from cross-source-versions (*e.g.*, updates, bug repair) due to the absence of quantitative metrics for the semantic gap. Another challenge arises from code obfuscation techniques that often blur the original semantics with packing, random code insertion, customized virtual machines, etc.

[2]Few-shot learning is an example of meta-learning, a promising direction without harnessing an intensive dataset.

4 BINSHOT DESIGN

4.1 BinShot Overview

Figure 2 illustrates the whole workflow of BinShot, which mainly consists of four components as follows:

Pre-processor. With a binary corpus, we prepare a dataset that can be readily fed into a BERT model for training (①). Using a reverse engineering tool, we obtain all disassembled instructions. Next, we convert a naïve instruction into a concise form (Table 4 in Appendix A), building the database of normalized functions (NFs) with normalization rules proposed by DeepSemantic [49].

Pre-trainer. Once all NFs are ready, we build a generic BERT model with pre-training, being able to emit an embedding per function (②). Akin to NLP's pre-trained language models, this model for machine instructions allows for repurposing it to varying downstream tasks.

Fine-tuner. Next, we re-train the pre-trained model (*i.e.*, fine-tuning) to meet our purpose; binary similarity detection (③). To this end, we prepare another dataset with function pairs and their labels for a binary classification problem. The fine-tuned model predicts if a function is analogous to another.

Predictor. As a final step, using the fine-tuned BERT, we build a database that contains all function vectors of our interest for further comparison (④). A downstream model takes a function pair as an input, seeking any similar function with a series of predictions.

4.2 Preprocessor: Learning Preparation

A pre-processing step encompasses a conversion needed to feed a wide range of collected executable binaries (See Table 1) into a neural network, including disassembly and normalization.

Disassembly Process. We first disassemble every instruction in a code region of each executable with a state-of-the-art reversing tool (*e.g.*, IDA Pro [35]). We generated the whole binary corpus with debugging information available (*e.g.*, obtaining function boundaries) for training a better model with accurate disassembly. Note that we assume a stripped binary for further testing.

Well-balanced Instruction Normalization. Feeding machine instructions as they are into a neural network raises an out-of-vocabulary (OOV) problem because, unlike NLP, the number of possible tokens (*i.e.*, an opcode, operands, or a combination of both) would be prohibitively huge. For example, a four-byte immediate value or a relative address as an operand could hold $2^{32} - 1$ (*i.e.*, around four billion) different tokens. A large number of OOV may end up incurring the failure of meaningful input embedding generation. In this regard, we adopt a strategy of well-balanced instruction normalization that is introduced by DeepSemantic [49], which strikes a balance between expressing binary code representation that preserves the original semantics while maintaining a reasonable amount of tokens to avoid OOV (Interested readers refer to Appendix A).

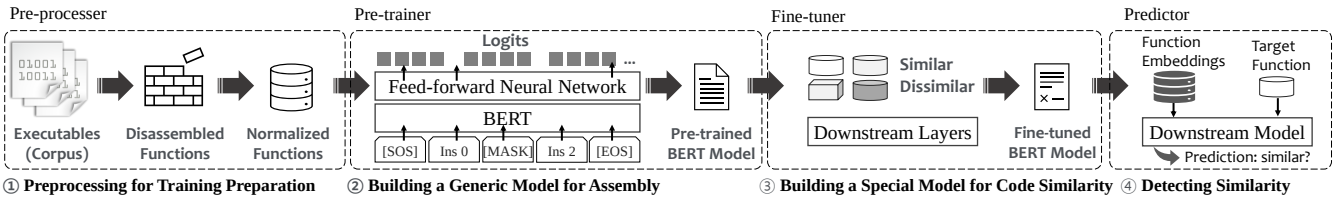


Figure 2: The overall workflow of BinShot that consists of four components: ① a pre-processor (§4.2) that generates a dataset in preparation for training, ② a pre-trainer (§4.3) that builds an assembly representation model (*i.e.*, pre-trained BERT), ③ a fine-tuner (§4.4) that constructs a code similarity detection model (*i.e.*, fine-tuned BERT), and ④ a predictor (§4.5) that judges the similarity of a target function.

4.3 Pre-trainer: Model for Assembly

BinShot adopts a BERT-based model (but NSP) that conforms to two-step training; the first one is pre-training (② in Figure 2) that plays a role to train a generic model for assembly codes.

MLM task. We take the identical strategy with the original BERT, replacing 15% of input tokens (instructions) with a mask symbol (*i.e.*, [MASK] token). The parameters of MLM, θ_m, are optimized by solving the following optimization problem:

$$\theta_m = \underset{\theta_m}{\operatorname{argmin}} \sum_{t \in T} p(t|y) \log p(t|\hat{y}) \tag{3}$$

$$\hat{y} = Softmax(G_m(X)) \tag{4}$$

where t, T, y, and $\hat{y}$ denote a token, a set of tokens, an original token before masking, and a predicted token for MLM, respectively. $G_m(X)$ represents the output vector of a fully connected layer from an MLM classifier with an input function X. At a high level, MLM learns to predict an appropriate instruction in place of a masked token, aiding in capturing the context between individual instructions.

Alternatives to an NSP task. We exclude NSP from the original BERT architecture because, unlike a neighboring sentence in NLP, a function does not necessarily have a meaningful relationship with an adjacent one. In other words, the relationship between functions is determined by a function invocation rather than their locations, rendering the next function prediction (*i.e.*, NSP) pointless. Although we do not directly take a function call (*i.e.*, caller and callee relationship) into account, well-balanced instruction normalization implicitly deals with a significant `libc` call (*e.g.*, system calls) by defining it as a separate word. Otherwise, the normalization process recognizes either external library calls (*e.g.*, throughout PLT; procedure linkage table, and GOT; global offset table) or internal function calls (*e.g.*, other functions defined within an executable). Besides, it views an indirect call (*e.g.*, `call reg8`) or recursive call (*e.g.*, `call self`) as a distinguishing instruction for better contextual inference as well.

4.4 Fine-tuner: Model for Code Similarity

Based on a generic BERT model with a large swath of binaries, we define a downstream task (③ in Figure 2); code similarity detection. To this end, we leverage a Siamese neural network into a classifier, learning a weighted distance from a labeled dataset (*i.e.*, (NF_1, NF_2, {0,1}) where 1 for a similar pair and 0 for a dissimilar one). Figure 3 illustrates the Siamese architecture for the fine-tuner in BinShot. The following equation shows how to compute a distance vector

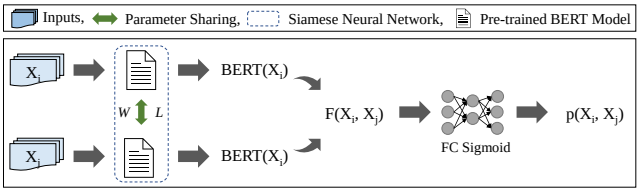


Figure 3: Siamese neural network for building a code similarity detection model. Our model learns a weighted distance vector from a labeled dataset (*i.e.*, a set of two functions and a label). X, W, L, F, FC, and p represent an input, weights, loss distance vectors, a fully connected layer, and the probability of code similarity.

($F(X_i, X_j)$) between two function embeddings based on the pre-trained BERT model ($BERT(X)$) where D represents a distance function such as an L1 or L2 norm.

$$F(X_i, X_j) = D(BERT(X_i), BERT(X_j)) \tag{5}$$

Then, our binary classifier learns a *weighted distance vector* with the following *binary cross entropy loss function*:

$$L = Y \log p(X_i, X_j) + (1 - Y) \log(1 - p(X_i, X_j)) \tag{6}$$

$$p(X_i, X_j) = \sigma(FC(F(X_i, X_j))) \tag{7}$$

where Y is a label for (X_i, X_j). $FC(X) = WX + b$ represents the output vector of a fully connected layer with weights (W), and σ means a Sigmoid function ($\sigma(z) = \frac{1}{1+e^z}$) as an activation function, ranging from 0 to 1 for the probability of the code similarity with a function pair ($p(X_i, X_j)$).

Loss Function. It is noteworthy mentioning that we carefully choose an appropriate loss function for a binary similarity detection task. One of the common scenarios (*e.g.*, code clone detection against copyright infringement) is to search for a certain function or a series of particular functions of one's interest. Assuming that each function has a single sample, the task of identifying semantically analogous code is aligned with an N-way one-shot classification problem, which is a special case ($k = 1$) of an N-way k-shot classification problem (See §2). In essence, we employ *a weighted distance learning with the Siamese architecture* that has been proposed by one-shot learning [48], which defines a loss function as a binary cross entropy for binary classification (*i.e.*, similar or not). In the same vein, we solve an optimization problem with the loss function in Equation 6, seeking the parameters (θ_d) that minimize the loss for a downstream model. Meanwhile, previous learning-based binary code similarity models [11, 54, 57, 62, 92, 98] often employ a contrastive loss L_c [12]. We empirically confirm that BinShot

equipped with our loss performs better for the inference of an unseen (new) function (See §6) than other state-of-the-art binary code similarity approaches.

4.5 Predictor for Code Similarity Detection

With the fine-tuned model for code similarity detection, a predictor (④ in Figure 2) takes a function pair as an input; one as a target function to compare with, and the other from a database. Note that a look-up process could speed up by i) pre-computing function embeddings in the database and ii) inserting unseen function vectors into the database. Once the predictor acquires two function vectors with $BERT(X_i)$ and $BERT(X_j)$, it computes a probability according to Equation 5 and Equation 7. It is noted that we set $p = 0.5$ as a threshold to determine if the two functions are similar.

5 IMPLEMENTATION

Static Binary Analysis. We leverage one of the state-of-the-art binary reverse engineering tools, IDA Pro 7.6 [35], into disassembling binaries and extracting fruitful static binary information. We wrote a script with built-in IDAPython APIs [34] that can extract a list of assembly functions, cross references (*i.e.*, call graph), section names, string references, and external library calls, facilitating further instruction normalization. Other binary reverse engineering tools such as angr [3], Ghidra [67], or radare2 [74] would suffice for our static analysis purpose.

BinShot Implementation. We develop BinShot with PyTorch [73], one of the most popular frameworks for machine learning. At the heart of BinShot, we implement the prototype using BERT [23, 32, 41, 88] and a Siamese neural network [48]. For a distance function described in §2, we adopt an L2 norm that generates a distance vector as an output because our experiment shows that overall performance with the L2 norm was slightly better than that with the L1 norm. BinShot adopts the following hyperparameters: the 256 dimensions for instruction embeddings, 128 hidden layers, eight attention layers and heads (for the Transformer structure), 256 maximum length of tokens, and three token encoder layers. We use the Adam optimizer [46] with a learning rate of 0.0005 and the dropout algorithm [36] for both pre-training and fine-tuning. The dropout rate is 0.2 for token encoder layers and 0.1 for the other layers. We employ a linear learning rate warm-up strategy [59] and a gradient clipping strategy [69] for stable training results. The number of parameters for building the BinShot model is 6,815,362. We train BinShot for 20 epochs with a batch size of 32.

6 EVALUATION

Prior binary similarity detection studies with machine learning approaches predominately focus on effectiveness (*i.e.*, accuracy, F1 score) and efficiency (*i.e.*, runtime performance). In this paper, we attempt to evaluate BinShot in terms of practicality by reasoning dataset generation and usefulness of a result by closely looking into interesting cases as well as previous metrics. To this end, we define two different test sets to compare BinShot with baseline models. Additionally, we demonstrate that a group of similar function embedding vectors are indeed close together in a space with a visualization technique (*i.e.*, t-SNE [82]).

Table 1: Our binary corpus. We build 1.4K binaries that contain 1.77M functions, obtaining 18K unique tokens after the well-balanced instruction normalization process.

Dataset	Binaries	Functions	Basic blocks	Instructions	Tokens
GNU utilities	1,000	439,036	3,965,532	22,137,920	1,244
SPEC2006	176	407,277	4,661,761	28,307,441	9,631
SPEC2017	120	755,297	9,336,587	52,940,593	11,932
Real-world programs	104	169,065	2,422,651	14,269,468	8,892
Total	1,400	1,770,675	20,386,531	117,655,422	18,449

Environment. We evaluate BinShot on the server equipped with two Intel Xeon Gold 6226R CPUs (with 32 cores in total) running at 2.90 GHz, 256 GB RAM, and two NVIDIA RTX A6000 GPU cards. Note that we had a single GPU card for performance assessment.

Evaluation Metrics. We use the following metrics for BinShot: a precision (P), recall (R), F1 score (Equation 8), and accuracy (Equation 9) where TP, FP, TN, and FN are the number of true positive, false positive, true negative, and false negative cases, respectively.

$$\mathrm{P} = \frac{TP}{TP+FP},\ \mathrm{R} = \frac{TP}{TP+FN},\ \mathrm{F1} = \frac{2\times \mathrm{P}\times \mathrm{R}}{\mathrm{P}+\mathrm{R}} \tag{8}$$

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FN+FP} \tag{9}$$

6.1 Dataset

Binary Corpus. Table 1 summarizes our corpus for evaluation. We collect varying applications in our corpus: executable binaries from GNU utilities, SPEC2006, SPEC2017, and real-world programs of our choice. GNU utilities include `binutils` (v2.26), `coreutils` (v8.30), `diffutils` (v2.8), and `findutils` (v4.7.0) that have been extensively evaluated in previous works [4, 10, 20–22, 24, 26, 28, 49, 54, 57, 62, 70, 71, 86, 89, 93, 98]. SPEC2006 and SPEC2017 provide a series of standardized CPU-intensive benchmark suites across different systems, different architectures, and different configurations. We also gather 11 real-world programs that are available as popular open-source projects from Github, including BusyBox (v1.34.1) [8], Libgmp (v6.2.1) [31], ImageMagick (v7.0.10) [42], Libcurl (v7.78.0) [19], LibTomCrypt (v1.18.2) [55], OpenSSL (v1.1.1f) [68], SQLite (v3.30.1) [79], zlib (v1.2.8) [97], PuTTYgen (v0.76) [72], Nginx (v1.16.1) [66], and vsftpd (v3.0.3) [84]. With two different compilers (*i.e.*, GCC v5.4, Clang v6.0.1) and four different compiler optimization levels (*i.e.*, O0–O3), we generated 1,400 executable binaries as a whole.

Refining Dataset. We refine our dataset (③ in Figure 4) before creating similar and dissimilar function pairs by filtering out certain functions depending on the size of a function (*i.e.*, the number of instructions within). Namely, we set up a threshold of two outliers: too small function where the number of instructions is less than or equal to m, and too large function where that of instructions is greater than n. The rationale behind this decision is that i) too small function may not often return a semantically meaningful result, ii) chances are slim that a target function (*e.g.*, containing a vulnerability) is too trivial, and iii) BERT has a limitation to handle too long sequences (*i.e.*, large function) while training. For our experiment, we set $m = 5$ and $n = 250$ in BinShot, which can be

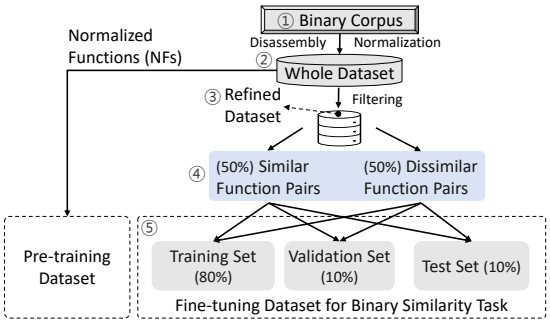


Figure 4: Dataset generation for similarity learning. All normalized functions from the raw binary corpus (①) constitute the whole dataset (②) for pre-training. We refine a dataset (③) by filtering out too small/large functions, followed by generating function pairs (④), splitting them into a training, validation, and test set (⑤).

adjustable. In our dataset, the proportions of too large functions and too small functions are 4.66% and 16.47%, respectively.

Dataset Generation for Learning. Figure 4 depicts an overview of generating the whole dataset (②) from a binary corpus (①), which can be fed into the BERT architecture. Borrowing the well-balanced instruction normalization strategy [49], the whole 117,655,422 instructions (from 1,770,675 functions) result in 18,449 vocabularies (§4.2). Because each vocabulary is dealt with as a token, we obtain 18,454 tokens, including five special ones for BERT: [SOS]; start of a function, [EOS]; end of a function, [UNK]; unknown token, [MASK]; mask symbol, and [PAD]; padding symbol. Recall that a pre-trainer builds a generic BERT model with the pre-processed dataset (*i.e.*, NFs). Once the instruction normalization step is complete, we create a set of similar and dissimilar function pairs for fine-tuning. We define a function pair to be similar when two function names are identical from a binary built with either a different compiler or optimization level. We exclude the cases where a pair of function bodies are identical after instruction normalization because it causes a distance of 0 in the Siamese neural network, which hinders the training process (*e.g.*, Figure 8 in Appendix A). For example, the function `openat_safer` (from `mkdir`) compiled with `gcc` and `O1` and the one compiled with `clang` and `O2` are similar as long as two function bodies after normalization are disparate. We adopt negative sampling (with the ratio of 1:1) for dissimilar function pairs, selecting different functions from different binaries. In our experiment, we create 5, 259, 310 function pairs in total (④ in Figure 4). We split the whole pairs into three disjoint groups for training, validation, and test dataset with the ratio of 8:1:1 (*i.e.*, 4, 207, 446 : 525, 932 : 525, 932), while maintaining half and half for similar and dissimilar pairs.

OOV. In our setting, there are unique 18, 280 tokens in a training set and 16, 166 tokens in a test dataset. The 64 tokens were merely discovered in a test set (*i.e.*, missing them in a training set), in which OOV accounts for 0.40% of the whole tokens.

6.2 Effectiveness

In this section, we evaluate the effectiveness of BinShot with previous binary similarity detection approaches using deep learning techniques. Table 2 summarizes a distance function, loss function, and architecture for each approach, which largely impacts performances.

Table 2: Summary of a distance function, loss function, and architecture for baseline models, BinShot, and its variant (*). GNN and PV-DM represent Graph Neural Network and Distributed Memory Model of Paragraph Vectors.

Model	Distance Function	Loss Function	Architecture
Gemini	Cosine distance	Contrastive loss	GNN, Siamese NN
Asm2Vec	Cosine distance	Log probability	PV-DM
PalmTree	Cosine distance	Contrastive loss	BERT, GNN, Siamese NN
DeepSemantic	None	Cross entropy	BERT, Softmax classifier
BinShot-CTR*	L2 norm	Contrastive loss	BERT, Siamese NN
BinShot	Weighted L2 norm	Binary cross entropy	BERT, Siamese NN

Baseline. As a baseline, we assess varying state-of-the-art binary similarity detection models, including Gemini [92], Asm2Vec [24] [3], PalmTree [54], and DeepSemantic [49]. In the case of PalmTree, we leverage the pre-trained model that is publicly available [53] to build a downstream model with our datasets for comparison. Note that we follow the hyperparameters from the original implementations unless otherwise stated.

BinShot Variant. To show the effectiveness of a loss function, we additionally generate a BinShot variant, BinShot-CTR, which adopts a contrastive loss (Equation 2) with an L2 norm as a distance function (Equation 1).

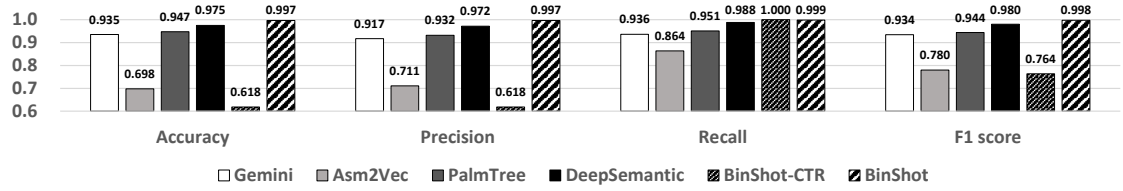
Similarity Threshold. We choose a threshold for similarity prediction that maximizes an F1 score in the case of Gemini [92], Asm2Vec [24], PalmTree [54], and BinShot-CTR (Appendix B).

Test Set Evaluation. Recall that the test set (§6.1) contains the same rate of similar and dissimilar function pairs. Figure 5 demonstrates the effectiveness of BinShot, compared with four baseline models (Gemini [92], Asm2Vec [24], PalmTree [54], DeepSemantic [49]) and the BinShot variant model (BinShot-CTR). We measure an accuracy, precision, recall, and F1 score with i) the whole dataset and ii) the datasets that are separated with all 36 combinations across different compilers and optimization levels (*e.g.*, (gcc-O0, gcc-O0), (gcc-O0, clang-O0), …, (clang-O3, clang-O3)). Note that the latter dataset is designed for a better understanding of the impact of different configurations (*i.e.*, cross-compilers, cross-optimization-levels) to deduce code similarity. Figure 5a clearly shows that BinShot outperforms all other state-of-the-art models for binary code similarity detection. Likewise, Figure 5b supports that the performance of BinShot stays stable with a low variance regardless of any combination of a compiler and optimization level.

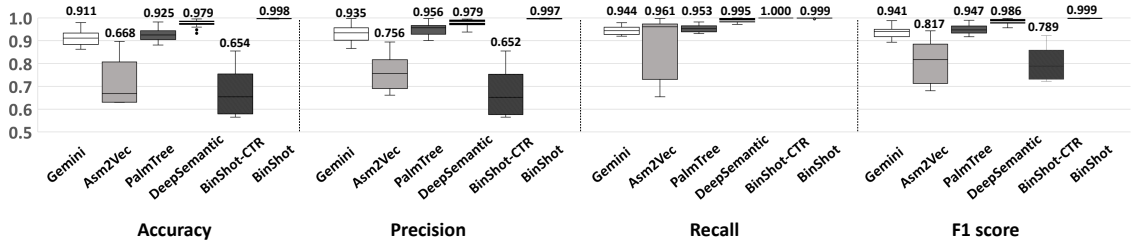
6.3 Transferability

To the best of our knowledge, our work is the first attempt that applies transferable learning in the field of binary similarity detection. This section delineates empirical results of model transferability; specifically, we raise the following research questions: ① how well is the model learned from a dataset X capable of inferring binary similarity in another dataset Y? and ② which state-of-the-art model performs the best? The higher transferability means that a model is more generalizable or scalable.

[3] As the original implementation [24] runs on the Windows platform, we adopt a version from https://github.com/oalieno/asm2vec-pytorch for handy comparison.

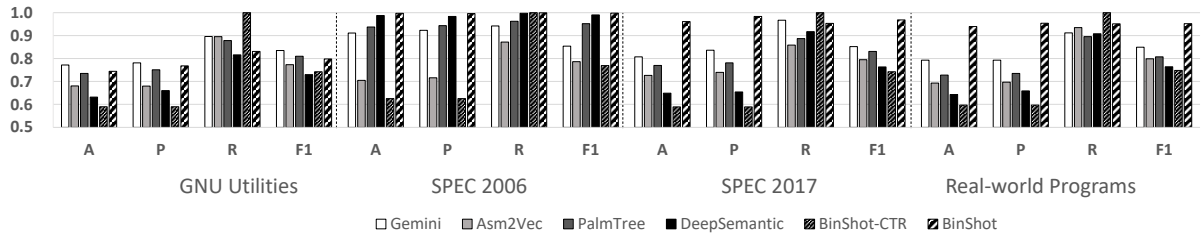


(a) Accuracy, precision, recall, and F1 score of six approaches for the whole test set.

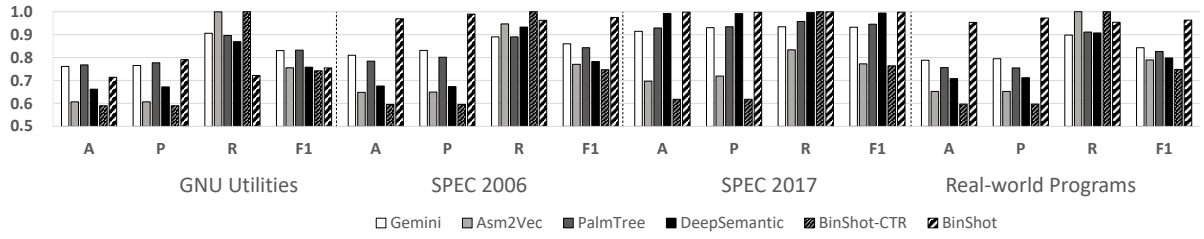


(b) Accuracy, precision, recall, and F1 score of six approaches across all combinations of a compiler and optimization level. We separate the dataset with a different set of compilers or optimization levels to confirm the effectiveness of BinShot. The numbers represent median values.

Figure 5: Results of code similarity detection comparison with the test set. BinShot surpasses other approaches with the lowest variations in accuracy and F1 metrics.



(a) Results of inferences from the models with the SPEC2006 dataset



(b) Results of inferences from the models with the SPEC2017 dataset

Figure 6: Results of transferability experiments. As expected, the inferences to the same dataset (*e.g.*, SPEC2006 → SPEC2006, SPEC2017 → SPEC2017) are the highest values across all metrics. Empirically, BinShot demonstrates the best generalizable capability when applying its model to other datasets. We explain why BinShot shows poor performance on GNU utilities in §6.3.

Experimental Setup. We conduct another experiment that utilizes a binary corpus from SPEC2006 and SPEC2017 (Table 1), building two fine-tuned models with BinShot because they contain the largest number of functions that could be further generalizable. Likewise, using Gemini [92], Asm2Vec [24], PlamTree [54], DeepSemantic [49], and BinShot-CTR, we build twelve different models with the same dataset (*i.e.*, SPEC2006, SPEC2017). Then, we have each model infer similarity with unseen datasets (*i.e.*, GNU utilities, real-world programs). Figure 6 briefly shows the results of inference across different models, including accuracy, precision, recall, and F1 score. As seen in both Figure 6a and Figure 6b, we observe that BinShot mostly achieves better performance than others but GNU utilities. Namely, the accuracy and F1 score with the BinShot model are higher than those with four baseline models and the BinShot variant; for example, the F1 score for real-world programs (rightmost bar in Figure 6b) when applying transfer learning from SPEC2017 indicates the best performance of all.

Results. We obtain the following insights with our experimental outcomes: ① a fine-tuned model with BinShot could be applicable to other (unseen) datasets in general, indicating that a weighted distance learning with a Siamese neural network appropriately works as intended, ② training and test set have evenly distributed based on Figure 6a and Figure 6b that are similar to Figure 5b, and ③ functions from SPEC2006 do not have largely those from SPEC2017 in common because the mutual inferences (*e.g.*, SPEC2006-based model toward SPEC2017 inferences, SPEC2017-based model toward SPEC2006 inferences) do not have much difference from the ones for real-world programs. Last but not least, BinShot shows relatively poor performance on GNU utilities. With a careful investigation, we found that GNU utilities have quite a few sharing functions across different binaries. This is because, `coreutils` have a static library (*i.e.*, `libcore.a`) in common when compiling various utility programs, which inherently contain identical (normalized) functions in multiple binaries (See Appendix D). Note that our finding is aligned with the observation from Koo et al. [50].

6.4 Vulnerable Function Detection

In this section, we set up a realistic scenario that detects a vulnerable function in a binary to demonstrate the practicality of BinShot.

Scenario Setup. Table 3 displays three programs that contain nine vulnerable functions along with six common vulnerabilities and exposures (CVEs). Each program has eight variants generated by two different compilers (*i.e.*, GCC v5.4, Clang v6.0.1) and four different compiler optimization levels (*i.e.*, O0–O3). We arrange the following three assumptions close to a real setting: ① the database of vulnerable function embeddings (compiled by GCC) of our interest has been prepared; ② the query binary is stripped, and compiled with Clang; ③ one attempts to find a vulnerable function in a query binary. Note that previous approaches [10, 11, 22, 24, 26–28, 57, 62, 70, 71, 89, 91, 92, 98] solely query a function to determine if it contains a vulnerability. However, querying a vulnerable function could suffer from accuracy in case of a number of false positive responses (*e.g.*, say all functions are vulnerable). Instead, we allow one to find all vulnerable functions by querying an entire binary itself for a precise result. Moreover, a single CVE may entail multiple functions in practice.

Evaluation Metric. In the above scenario, similar function pairs are much rarer than dissimilar ones. Unavoidably, this imbalance may distort a few metrics even for a highly accurate model, such as a recall and F1 score. Suppose that we have S positive samples and D negative samples, and a model reaches up to $TP = 0.99 \times S$ and $FP = 0.01 \times D$ where $S = TP + FN$ and $D = TN + FP$. Also, assume that $S \simeq 0.003 \times D$. Then, the precision (P), recall (R), and F1 score ($F1$) of this model can be calculated as follows:

$$P = \frac{TP}{TP + FP} \simeq \frac{0.99 \cdot 0.003D}{0.99 \cdot 0.003D + 0.01D} \simeq 0.22899 \tag{10}$$

$$R = \frac{TP}{TP + FN} = \frac{0.99S}{0.99S + 0.01S} = 0.99 \tag{11}$$

$$F1 = \frac{2PR}{P + R} \simeq \frac{2 \times 0.22899 \times 0.99}{0.22899 + 0.99} \simeq 0.37195 \tag{12}$$

As shown, a large volume of negative samples can distort a precision and F1 score even for an accurate model, so we merely include accuracy and recall for the evaluation with a query binary.

Results. We assess four baselines, BinShot and BinShot-CTR with 12 query binaries from three programs compiled with Clang O0-O3. Table 3 summarizes the results of detecting a vulnerable function along with functions in our database. Note that we rule out a few vulnerable functions (*e.g.*, `crypto_recv`, `ctl_putdata-O3`, `configure-O1`) because a reversing tool (*i.e.*, IDA [35]) failed to recognize their boundaries in a stripped binary. Interestingly, with the thresholds in §6.2, all models but DeepSemantic [49] discover the whole vulnerable functions in the query binaries. However, only BinShot achieves the high accuracy (*i.e.*, 88.2% on average) while all the others show the quite poor accuracy (*i.e.*, less than 50%), which remains their effectiveness questionable when applying them to a real scenario in practice. This is mainly because even a single false positive (of all comparisons in the database) returns the result as a vulnerable function, which decreases the overall accuracy. Besides, we investigate the case of NTP in Table 3; the boundary of a function was different (incorrect) from the ground truth, resulting in the relatively poor accuracy (79.4%).

6.5 Function Embedding Visualization

One of the popular means to comprehend a classification model on vast amounts of data is through visualization. Because our model relies on binary classification, it requires checking if the embeddings for a similar function pair are close enough while those of a dissimilar one is sufficiently distant. However, representing each vector in a high dimensional space (*e.g.*, 128 dimensions in our experiment) is difficult. We utilize a t-distributed Stochastic Neighbor Embedding (t-SNE) method [82] that offers non-linear dimensionality reduction where similar data are pointed close together in a lower-dimensional space. As in Figure 7, we select five different functions that have been wrongly classified (*i.e.*, false negative) by baseline models in §6.2, which BinShot has correctly classified (*i.e.*, true positive). Note that Figure 7 shows eight different embeddings per assembly function (with a unique symbol) because each function has been compiled with a combination of two compilers and four optimization levels. Hence, the vectors representing similar functions are expected to be located nearby. Figure 7 intuitively visualizes the vectors representing similar functions generated by BinShot, which are indeed well clustered together. Interestingly, visualized functions other than `hmac_key` (green X mark) have two disparate groups: one comes from `gcc`, and the other from `clang`.

6.6 Runtime Efficiency

In this section, we conduct two experiments to assess runtime efficiencies of BinShot and other baseline models for inferring binary similarity detection (*i.e.*, binary classification of a given function pair). First, we sampled 16, 595 function pairs (0.28%; 99% confidence level) from all 5, 884, 690 pairs that include training, validation, and test sets (§6.1). As a result, Gemini [92], Asm2Vec [24], PalmTree [54], DeepSemantic [49], BinShot-CTR, and BinShot took 0.10, 81.94, 1.33, 1.34, 1.30, and 1.32 ms (millisecond) per each function pair, respectively. Other than Gemini (the fastest run) and Asm2Vec (the slowest run), all the others are similar. Next, we assume the scenario (§4.5) where one attempts to seek if any function in a target binary has a similar one in a database that contains 100 function embeddings of our interest. We pre-build the database that

Table 3: Results of detecting vulnerable functions with BinShot. An asterisk (*) indicates that our model includes a program (but a different version) during training. Each symbol of ✓and ✗ denotes detection success and failure, while - denotes a case that a reversing tool fails to discover a function boundary. BinShot achieves the best performance in the accuracy (A) and recall (R) while others show the poor accuracy.

Program	CVE	Vulnerable function	Gemini		Asm2Vec		PalmTree		DeepSemantic		BinShot-CTR		BinShot	
			O0–O3	A/R	O0–O3	A/R	O0–O3	A/R	O0–O3	A/R	O0–O3	A/R	O0–O3	A/R
OpenSSL v1.0.1g*	2014-0160 [14]	tls1_process_heartbeat	✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✓✓✓	
		dtls1_process_heartbeat	✓✓✓✓	0.0063/	✓✓✓✓	0.0987/	✓✓✓✓	0.0101/	✗✓✗✓	0.2604/	✓✓✓✓	0.0063/	✓✓✓✓	0.9021/
	2014-0221 [15]	dtls1_get_message_fragment	✓✓✓✓	1.0000	✓✓✓✓	1.0000	✓✓✓✓	1.0000	✓✓✓✓	0.7000	✓✓✓✓	1.0000	✓✓✓✓	1.0000
	2014-3508 [16]	OBJ_obj2txt	✓✓✓✓		✓✓✓✓		✓✓✓✓		✗✓✗✓		✓✓✓✓		✓✓✓✓	
	2015-1791 [18]	ssl3_get_new_session_ticket	✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✗✓✗		✓✓✓✓		✓✓✓✓	
NTP v4.2.7p10	2014-9295 [17]	crypto_recv	- - - -		- - - -		- - - -		- - - -		- - - -		- - - -	
		ctl_putdata	✓✓✓ -	0.0055/	✓✓✓ -	0.1588/	✓✓✓ -	0.0083/	✓✓✓ -	0.4505/	✓✓✓ -	0.0064/	✓✓✓ -	0.7940/
		configure	✓ -✓✓	1.0000	✓ -✓✓	1.0000	✓ -✓✓	1.0000	✓ -✓✓	1.0000	✓ -✓✓	1.0000	✓ -✓✓	1.0000
libav v0.8.3	2012-2776 [13]	decode_cell_data	✓✓✓✓	0.0007/ 1.0000	✓✓✓✓	0.1215/ 1.0000	✓✓✓✓	0.0065/ 1.0000	✗✓✗✓	0.0003/ 0.5000	✓✓✓✓	0.0007/ 1.0000	✓✓✓✓	0.9497/ 1.0000

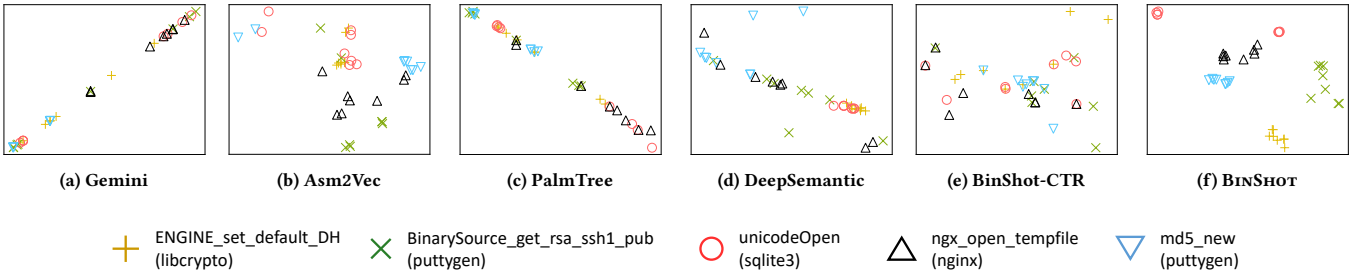


Figure 7: t-SNE visualization [82] of six different models in a two-dimensional space. Each symbol (*e.g.*, +, x, o) denotes five function embeddings where each function has eight similar versions (*i.e.*, functions with two compilers and four optimization levels). BinShot has the clearest boundary so that similar functions can group together.

stores a series of (NF, NF embedding) as a (key, value) pair for a quick look-up. Then, we collect 823 functions (after refinement as §6.1) from the OpenSSL library as a target. With the above settings, we measure the running time for both function embedding generation (*i.e.*, 823) and similarity inferences (*i.e.*, $82,300 = 100 \times 823$) with five state-of-the-art approaches, including BinShot. It is noted that generating embeddings dominates runtime overheads over comparisons by look-ups. As a result, Gemini [92], Asm2Vec [24], PalmTree [54], DeepSemantic [49], BinShot-CTR, and BinShot took 1.16, 6, 743.66, 29.03, 1.51, 1.45, and 1.54 seconds, respectively. While Gemini [92] ranks first, BinShot shows a comparable efficiency.

7 DISCUSSIONS AND LIMITATIONS

Mangled Names. Name mangling (*i.e.*, name decoration) is a technique that allows one to employ unique entity names (*e.g.*, function name), which has been widely adopted by modern programming languages. Simply put, one can declare multiple entity identifiers using a different namespace (with a structure, class, or module) or function overloading. Then, a compiler internally creates a distinguishing name that utilizes class hierarchy, arguments, and type information. Since the current labeling process endorses the original mangled names, it may give rise to incorrect labels (See Appendix D).

Code Obfuscation and Other Code Constructs. BinShot mainly focuses on binary code similarity detection where obfuscation techniques have not been applied. Asm2Vec [24] and Luo et al. [60] introduce a technique for obfuscation-resilient binary code similarity comparison. InnerEye [98] proposes a similar function detection technique across different architectures (*e.g.*, Intel and ARM instruction set architecture). DeepBinDiff [93] seeks similar functions across different versions of source code. Although the current form of BinShot may not be directly applicable to obfuscated binaries or other code constructs, we plan to expand BinShot in the future.

Function Inlining. Function inlining is one of the well-known compiler optimization techniques[4] which aim to reduce the runtime overheads that come from entering and exiting a function. Oftentimes, an inlining process involves either a caller or callee becoming part of another, leading to drastic changes in resulting assembly; *e.g.*, an inlined function disappears while an inlining function includes an additional functionality with the original name. This might be problematic for labeling with a function name because semantically similar functions could be classified as dissimilar. Previous approaches [10, 24, 40] propose a selective callee expansion technique to handle it. We hypothesize that the impact of function inlining seems limited for a binary similarity detection task considering the performance of BinShot. Interested readers refer to our case study in Appendix D.

Rarely Appeared Instructions. A pre-training model may encounter rarely appeared instructions, which results in the embedding of those instructions that could stay close to randomly initialized values. On the one hand, one can increase a pre-training dataset to relax the issue of hardly-updated instructions; however,

[4] A programming language like C++ offers an `inline` specifier as a keyword on demand; however, a compiler fully controls an inlining operation.

further investigation is needed. On the other hand, it is possible to leverage the presence of a particular instruction as a birthmark to binary similarity detection because it could be the unique property of a code snippet.

8 RELATED WORK

The related work in the domain of binary code similarity is vast [33]. We classify prior work into three categories: static [4, 20–22, 25, 27, 30, 60, 71, 89, 99], dynamic [10, 26, 45, 63, 64, 70, 86], and ML-based [11, 24, 28, 49, 54, 57, 61, 62, 70, 81, 91–94, 98] approaches.

Static Approaches. In the era of non-ML techniques, the main static approaches in the field of binary code similarity detection utilize graph isomorphism detection [4, 25, 99], symbolic execution [30, 60], data flow analysis [20, 21, 71], or other techniques [22, 27, 89]. BinDiff [25, 99] uses graph isomorphism on both call graph and control flow graph (CFG), and later BinSlayer [4] improves BinDiff by augmenting the Hungarian algorithm for bipartite graph matching. BinHunt [30] and CoP [60] employ static symbolic execution to understand the semantics of binaries. In particular, BinHunt utilizes theorem proving for extracting semantics of binaries, whereas CoP searches the longest common subsequence to compute the similarity. Tracelet [22] divides the CFG of a function into a fixed length of partial traces of an execution to measure similarity with those traces. Pewny et al. [71] propose a tool that collects input/output (I/O) pairs by providing random inputs to a code snippet. With these I/O pairs, it obtains a hash value that represents a basic block, followed by performing graph matching with the represented values. discovRE [27] introduces a fast similarity comparison method by filtering numeric and structural features from pre-generated features before function matching. ESH [20] and GitZ [21] decompose a target code into smaller fragments via data flow analysis, applying statistical reasoning for further code similarity detection. BinXary [89] analyzes the signature of a patch by comparing a vulnerable program with a patched program.

Dynamic Approaches. Dynamic approaches for binary code similarity detection aim to analyze the semantics of binaries by running executables [10, 26, 45, 63, 64, 70, 86]. Varying techniques have been introduced, such as comparing I/O pairs [10, 26, 45, 86], dynamic symbolic execution [63, 64], and learning semantics of execution paths with a neural network [70]. BLEX [26] executes two target functions with randomly sampled input values to compare their behaviors provoked by the inputs. BinSim [64] utilizes system calls or API calls to slice out corresponding code segments and then check their equivalence with symbolic execution. TREX [70] employs a neural network model, training with diverse execution traces and assembly codes. However, dynamic approaches require a time-consuming and cumbersome task to execute binaries multiple times, limiting them to be scalable.

ML-based Static Approaches. Over the last decade, as the ML field has been remarkably advanced, applying ML-based techniques to binary code similarity detection [11, 24, 28, 57, 61, 62, 81, 91–93, 98] has become common. To learn the features of a function from a CFG, Genius [28] employs a genetic algorithm and spectral clustering. On the other hand, Gemini [92] and VulSeeker [11] employ a graph embedding network-based Siamese architecture. Meanwhile, Asteria [91] utilizes a target function's abstract syntax tree (AST) from decompilation, followed by training a Tree-LSTM-based model based on a Siamese architecture. With the model and a set of callees from the target, Asteria obtains a similarity score. αDiff [57] takes another approach that uses raw bytes of functions to train a convolution neural network-based Siamese architecture. Recently, advances in NLP techniques have triggered applications [24, 62, 93, 98] in the computer-language domain that leverage various techniques into training assembly language. Asm2Vec [24] and DeepBinDiff [93] conduct unsupervised learning by training the context of instructions. InnerEye [98] and SAFE [62] employ an NLP-based Siamese network. The closest work to BinShot would be Luo et al. [61] and $\text{BinDiff}_{\text{NN}}$ [81], which takes an approach of learning a weighted distance vector. However, Luo et al. [61] manually extract features for function embeddings, whereas $\text{BinDiff}_{\text{NN}}$ [81] employs an attention-based neural network. Moreover, unlike ours, they do not thoroughly discuss the difference between their Siamese architectures with previous works.

BERT Applications. With the emergence of BERT [23], there is a prominent research trend in the field of NLP with transformer-based language models [52, 58, 75]. The popularity of BERT penetrates the domain of binary code analysis by building a representative assembly language model [49, 54, 70, 85, 94]. Yu et al. [94], TREX [70], and jTrans [85] fully focus on binary code similarity detection, whereas others [49, 54] utilize a BERT-based model for various downstream tasks. Yu et al. [94] obtain a pre-trained BERT model with the information extracted from a CFG, generating semantic-aware token embeddings and block embeddings, followed by predicting if two binaries are similar. TREX [70] employs MLM to learn dynamic values with micro-traces (a form of under-constrained dynamic traces) of a function and then fine-tune the downstream model for a binary function similarity task. jTrans [85] proposes modified BERT to represent jump instructions to be aware of their jump target. The authors use a triplet loss [77], another popular loss function in Siamese neural network, to train their downstream model. Meanwhile, DeepSemantic [49] and PalmTree [54] apply a generic model for assembly language to other tasks. DeepSemantic [49] demonstrates two downstream tasks: binary code similarity detection and classification for a compiler or optimization level. Likewise, PalmTree [54] applies a generic model to five downstream tasks: outlier detection, basic block search, binary code similarity detection, function type signature inference, and value set analysis. We adopt a BERT-based model for BinShot; however, we focus more on how to train a Siamese neural network for better code similarity detection.

9 CONCLUSION

In this paper, we propose BinShot that learns a weighted distance vector with a BERT-based Siamese architecture for binary code similarity detection. We adopt BERT to be able to learn the semantics of an assembly language and harness a binary cross entropy as a loss function. We implemented the prototype of BinShot, which shows that BinShot is more robust than existing state-of-the-art models for binary code similarity detection. Our experimental results show that the performance of BinShot surpasses that of the cutting-edge approaches, demonstrating its effectiveness and practicality.

REFERENCES

[1] 2017. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana-Champaign, IL.

[2] 2020. *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI)*. New York, New York.

[3] Angr. 2016. Python Framework for Analyzing Binaries. https://angr.io/.

[4] Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*. Rome.

[5] Jane Bromley, Isabelle M Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature Verification using a Siamese Time Delay Neural Network. In *Proceedings of the 6th Conference on Neural Information Processing Systems (NIPS)*. Denver, CO.

[6] Tom Brosch and Maik Morgenstern. 2006. Runtime packers: The hidden problem. *Black Hat USA* (2006).

[7] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-flow Graph Matching. In *Proceedings of the 3rd Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Berlin.

[8] BusyBox. 2022. The Swiss Army Knife of Embedded Linux. https://busybox.net.

[9] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 11, 4 (2013), 307–317.

[10] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Tan Hee Beng Kuan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Seattle, WA.

[11] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Tan Hee Beng Kuan. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier.

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

[13] The MITRE Corporation. 2012. CVE-2012-2776 Detail. https://www.cve.org/CVERecord?id=CVE-2012-2776/.

[14] The MITRE Corporation. 2014. CVE-2014-0160 Detail. https://www.cve.org/CVERecord?id=CVE-2014-0160/.

[15] The MITRE Corporation. 2014. CVE-2014-0221 Detail. https://www.cve.org/CVERecord?id=CVE-2014-0221/.

[16] The MITRE Corporation. 2014. CVE-2014-3508 Detail. https://www.cve.org/CVERecord?id=CVE-2014-3508/.

[17] The MITRE Corporation. 2014. CVE-2014-9295 Detail. https://www.cve.org/CVERecord?id=CVE-2014-9295/.

[18] The MITRE Corporation. 2015. CVE-2015-1791 Detail. https://www.cve.org/CVERecord?id=CVE-2015-1791/.

[19] Curl. 2022. libcurl - the multiprotocol file transfer library. https://curl.se/libcurl.

[20] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Santa Barbara, CA.

[21] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona.

[22] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices* 49, 6 (2014), 349–360.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the Association for Computational Linguistics (NAACL)*. Minneapolis, Minnesota.

[24] Steven H. H. Dinga, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[25] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (english version). *Sstic* 5, 1 (2005), 3.

[26] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium*. San Diego, CA.

[27] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[28] 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 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna.

[29] Halvar Flake. 2004. Structural Comparison of Executable Objects. In *Proceedings of the 1st Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Dortmund.

[30] Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS)*. Birmingham.

[31] GNU. 2022. Libbgmp: The GNU Multiple Precision Arithmetic Library. https://gmplib.org.

[32] Google. 2020. Release of BERT Models. https://github.com/google-research/bert.

[33] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–38.

[34] Hex-rays. 2019. IDAPython Documentation. https://www.hex-rays.com/products/ida/support/idapython_docs/.

[35] Hex-Rays. 2022. IDA Pro Disassembler. https://www.hex-rays.com/products/ida/.

[36] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* (2012).

[37] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[38] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. 2013. MutantX-S: Scalable Malware Clustering Based on Static Features. In *Proceedings of the 22th USENIX Security Symposium*. Washington, DC.

[39] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture Binary Semantics Understanding via Similar Code Comparison. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita, Osaka.

[40] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC)*. Buenos Aires.

[41] huanghonggit. 2019. BERT MLM with Pytorch. https://github.com/huanghonggit/Mask-Language-Model.

[42] ImageMagick. 2022. ImageMagick. https://imagemagick.org.

[43] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[44] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. 2019. Neurlux: Dynamic Malware Analysis Without Feature Engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. San Juan.

[45] Ulf Kargén and Nahid Shahmehri. 2017. Towards robust instruction-level trace alignment of binary code, See [1].

[46] Diederik P Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. San Diego, CA.

[47] Gregory Koch. 2015. *Siamese Neural Networks for One-Shot Image Recognition*. Ph. D. Dissertation. University of Toronto, Toronto. http://www.cs.toronto.edu/~gkoch/files/msc-thesis.pdf.

[48] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. 2015. Siamese Neural Networks for One-shot Image Recognition. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. Lille.

[49] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. 2021. Semantic-aware Binary Code Representation with BERT. *arXiv preprint arXiv:2106.05478* (2021).

[50] Hyungjoon Koo, Soyeon Park, and Taesoo Kim. 2021. A Look Back on a Function Identification Problem. In *Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC)*. 158–168.

[51] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Seattle, Washington.

[52] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. In *Proceedings of the 7th International Conference on Learning Representations (ICLR)*. New Orleans, LA.

[53] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. Offical Implementation for PalmTree. https://github.com/palmtreemodel/PalmTree.

[54] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. *arXiv preprint arXiv:2103.03809* (2021).

[55] LibTom. 2022. LibTomCrypt. https://www.libtom.net/LibTomCrypt.

[56] Marina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of Malicious Code: Insights Into the Malicious Software Industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. Orlando, Florida.

[57] 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 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). Montpellier.
[58] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).
[59] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).
[60] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Hong Kong.
[61] Zhengping Luo, Tao Hou, Xiangrong Zhou, Hui Zeng, and Zhuo Lu. 2021. Binary Code Similarity Detection through LSTM and Siamese Neural Network. *EAI Endorsed Transactions on Security and Safety* 8, 29 (2021).
[62] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proceedings of the 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Gothenburg.
[63] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology (ISISC)*. Seoul.
[64] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium*. Vancouver, Canada.
[65] Abhilash Nandy, Sushovan Haldar, Subhashis Banerjee, and Sushmita Mitra. 2020. A Survey on Applications of Siamese Neural Networks in Computer Vision. In *Proceedings of the 2020 International Conference for Emerging Technology (INCET)*. GOA.
[66] Nginx. 2020. High Performance Load-balancer and Web Server. https://nginx.com.
[67] National Security Agency (NSA). 2019. Software Reverse Engineering (SRE) Suite of Tools. https://ghidra-sre.org/.
[68] OpenSSL. 2022. Cryptography and SSL/TLS Toolkit. https://www.openssl.org.
[69] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning (ICML)*. Atlanta, GA.
[70] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv preprint arXiv:2012.08680* (2020).
[71] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[72] PuTTygen. 2022. Download PuTTYgen - Putty key generator. https://www.puttygen.com.
[73] PyTorch. 2019. Open Source Machine Learning Framework. https://pytorch.org/.
[74] Radare2. 2019. Libre and Portable Reverse Engineering Framework. https://rada.re/n/.
[75] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018). https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf.
[76] Kimberly Redmond, Lannan Luo, and Qiang Zeng. 2019. A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*. San Diego, CA.
[77] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the 2015 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA.
[78] Paria Shirani, Leo Collard, Basile L. Agba, Bernard Lebel, Mourad Debbabi, Lingyu Wang, and Aiman Hanna. 2018. BinArm: Scalable and Efficient Detection of Vulnerabilities in Firmware Images of Intelligent Electronic Device. In *Proceedings of the 15th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Paris.
[79] SQLite. 2022. SQLite. https://www.sqlite.org.
[80] Alan Mathison Turing et al. 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math* 58, 345-363 (1936), 5.
[81] Sami Ullah and Heekuck Oh. 2021. BinDiff NN: Learning Distributed Representation of Assembly for Robust Binary Diffing against Semantic Differences. *IEEE Transactions on Software Engineering* (2021).
[82] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).
[83] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS)*. Long Beach, CA.
[84] vsftpd. 2022. Probably the Most Secure and Fastest FTP server. https://security.appspot.com/vsftpd.html.
[85] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-Aware Transformer for Binary Code Similarity Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Virtual Event.
[86] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis, See [1].
[87] Harald Welte. 2012. Current developments in GPL compliance. (2012). http://taipei.freedomhec.org/dlfile/gpl_compliance.pdf
[88] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFaceś Transformers: State-of-the-art Natural Language Processing. *ArXiv* (2019).
[89] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Virtual Event.
[90] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. Buenos Aires.
[91] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. 2021. Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Taipei.
[92] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.
[93] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[94] 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, See [2].
[95] Xiaochuan Zhang, Wenjie Sun, Jianmin Pang, Fudong Liu, and Zhen Ma. 2020. Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture. In *Proceedings of the 3rd Workshop on Binary Analysis Research (BAR)*. San Diego, CA.
[96] Zhaoqi Zhang, Panpan Qi, and Wei Wang. 2020. Dynamic Malware Analysis with Feature Engineering and Feature Learning, See [2].
[97] Zlib. 2022. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://zlib.net.
[98] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, and Zhexin Zeng, Qiang andZhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison Beyond Function Pairs. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[99] Zynamics. 2022. zynamics BinDiff. https://www.zynamics.com/bindiff.html.

A NORMALIZED FUNCTIONS

Table 4 shows the primary rules for processing instruction normalization suggested by [49], which apply an operand that represents an immediate, a register, or a pointer. After the normalization, two different functions may be mapped into the same instruction sequence as shown in Figure 8. Note that we exclude such cases for stable training.

B MODEL THRESHOLDS

A different model adopts its own distance function such as a cosine distance or L2 norm (Table 2). Moreover, a model takes its own similarity threshold of a computed distance (or probability) to determine the similarity of a given pair. For fair comparisons, we choose a similarity threshold that maximizes an F1 score per baseline approach. As an example, Gemini [92] utilizes a cosine distance that ranges from −1 to 1, resulting in a drastic change

Table 4: Illustrative rules on an operand for the well-balanced instruction normalization [49] and examples accordingly. [I], [R], and [P] represent a category for an immediate, register, and pointer.

Rule	Notation	Example
[I] libc library call	libc[name]	call_libcprintf
[I] recursive call	self	call_self
[I] call within a binary	innerfunc	call_innerfunc
[I] call out of a binary	externfunc	call_externfunc
[I] jump to a destination	jmpdst	jmp_jmpdst
[I] referring a string	dispstr	mov_reg4_dispstr
[I] referring a static variable	dispbss	mov_reg8_dispbss
[I] referring non-string data	dispdata	movabs_reg8_dispdata
[I] other immediates	immval	sar_reg8_immval
[R] register size	reg[1|2|4|8]	rax → reg8, ebx → reg4
[R] stack register	rsp	rsp → sp8
[R] base register	rbp	rbp → bp8
[R] instruction register	rip	rip → ip8
[P] direct pointer	[byte|word |dword|qword]ptr	mov_qwordptr [reg8-8]_reg8
[P] indirect pointer	[base_index*scale +displacement]	or_dwordptr [reg8+disp]+immval

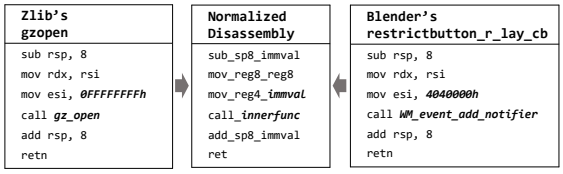


Figure 8: Example of two identical functions after normalization. Here the function `gzopen` from `Zlib` and `restrictbutton_r_lay_cb` from `Blender` embody the same sequence of tokens. Both are compiled with `gcc` with `-O1`. We exclude this case from our dissimilar function pair set.

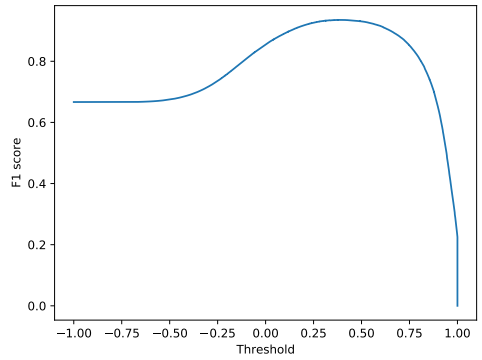


Figure 9: F1 scores by similarity thresholds with the Gemini model (§6.2). The cosine distance threshold of 0.383 maximizes an F1 score.

in F1 scores. Figure 9 depicts that a threshold of 0.383 maximizes Gemini's performance. A too high threshold lowers a prediction of positive samples, resulting in the recall (and F1) decrease accordingly. In the same vein, we choose the thresholds of 0.037 and 0.394 for Asm2Vec [24] and PalmTree [54]. BinShot-CTR utilizes the L2 norm as a distance function, having a threshold of 0.485. It is worth noting that a threshold could vary empirically depending on a distance function, loss function, and architecture across different approaches.

Table 5: Results of code similarity detection for large functions with BinShot.

Accuracy	Precision	Recall	F1 score
0.978	0.995	0.966	0.980

C LARGE FUNCTION EVALUATION

Recall that we exclude large functions (4.66%) when training the current model for BinShot. We conduct an additional experiment to demonstrate the capability of BinShot even for such large functions without training them at all. We adopt a naïve approach that truncates the number of tokens when exceeding the limit that BERT can take as an input. Table 5 shows accuracy, precision, recall, and F1 score, which shows little performance degradation.

D CASE STUDY

We analyze several cases that make code similarity detection challenging, which has been discussed in §7.

```
1 static void emDM_drawEdges(DerivedMesh *dm,
2 int UNUSED(drawLooseEdges),
3 int UNUSED(drawAllEdges))
4 {
5   emDM_drawMappedEdges(dm, NULL, NULL);
6 }
```

Listing 1: Example of a function inlining. `emDM_drawMappedEdges` is inlined at the optimization level 3.

(Case 1) False Negative: Function Inlining. The code snippet in Listing 1 shows the definition of the function, `emDM_drawEdges`, in the `blender_r` binary from the SPEC2017 suite. As it contains a single function call, a modern compiler often embeds (copies) it into another function (*i.e.*, function inlining) as part of optimization to eliminate a call-linkage overhead like a function prologue or epilogue. For instance, `emDM_drawEdges-O0` (*i.e.*, compiled with the optimization level of -O0) contains a single call with 13 instructions, whereas `emDM_drawEdges-O3` (*i.e.*, compiled with the optimization level of -O3) contains seven calls with 56 instructions because the compiler inserts `emDM_drawMappedEdges` into `emDM_drawEdges`. As a label for ground truth is based on a function symbol name (from binary debugging information) irrespective of such an inlining optimization, our dataset may erroneously claim that `emDM_drawEdges-O0` and `emDM_drawEdges-O3` are a similar function pair. Strictly speaking, this case must be regarded as a true negative case because those functions are contextually not identical.

(Case 2) False Positive: Mangled Names. In our dataset, the function named `_M_get_Tp_allocator` at `ld.gold` (from `binutils`) and the one at `parest_r` (from SPEC 2017) turn out to be identical because both originate from the same member function in the standard template library (STL). In this case, the label marks "dissimilar" due to different namespaces, leading to a false positive, although the

decision must be correct. In general, polymorphic techniques (for inheritance), such as function overloading and overriding, make a decision more complicated because it is challenging to be aware of one's intention; *e.g.*, the implementation of an overridden function may or may not be different.

(Case 3) False Positive: Identical Functions in Different Binaries. Another interesting case arises from a traditional linking process. It is possible to have literally identical functions (except adjustable fixups for relocation) across different executable binaries when there is a statically shared library in common during compilation. The static library (*e.g.*, shared objects with an `a` file extension in a *NIX system) holds a number of object files, each of which contains one or more functions within. If a binary borrows any function in a certain object file, a linker consolidates all (binary) functions in that object file by default. Those functions would preserve exactly the same function bodies across other binaries that import an arbitrary function in that object file because an offset (*e.g.*, relative address to call or jump) as an operand would be ignored during instruction normalization. A good example would be the case of GNU utilities; *e.g.*, `quotearg_style_mem` is present at both `find` (from `findutils`) and `stat` (from `coreutils`). With the current labeling scheme, they are a dissimilar function pair; however, the truth is that they should be served as similar because those functions stem from the same source code.