2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-1594-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSRE59848.2023.00013

OPTango: Multi-central Representation Learning against Innumerable Compiler Optimization for Binary Diffing

Hongna Geng
SKLP, ICT, CAS
UCAS
Beijing, China
genghongna@ict.ac.cn

Ming Zhong
SKLP, ICT, CAS
UCAS
Beijing, China
zhongming21s@ict.ac.cn

Peihua Zhang
SKLP, ICT, CAS
UCAS
Beijing, China
zhangpeihua@ict.ac.cn

Fang Lv
SKLP, ICT, CAS
Beijing, China
flv@ict.ac.cn

Xiaobing Feng
SKLP, ICT, CAS
UCAS
Beijing, China
fxb@ict.ac.cn

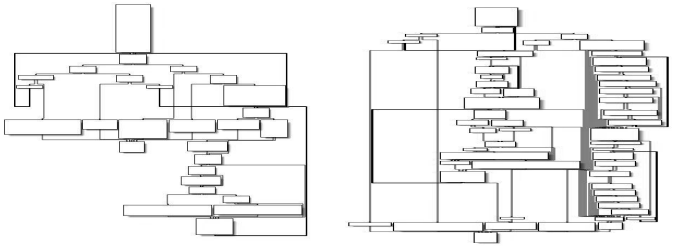
***Abstract*—Binary diffing, which quantitatively measures the difference between given binaries, has been broadly used in critical security areas. Previous studies have been tackling the challenge of *default* compiler optimization, as it can affect binary representation but overlooked the exploration of *non-default* optimization settings, which can also significantly affect the accuracy of diffing. Recent research indicates a growing trend of compiling applications with *non-default* optimization settings to magnify binary code discrepancies, enabling them to evade detection by binary diffing tools. This paper takes the first step to systematically studying the resistance of compiler optimization (including *default* and *non-default* optimization settings) on binary diffing tasks. To this end, we construct a diverse and unique dataset, OPTBinary, with 3.6 million functions compiled from 514 optimization settings. Then, we propose OPTango, an innovative transformer-based multi-central representation learning approach, exploring the solution to build a compiler optimization-agnostic binary diffing tool. We conduct extensive experiments and benchmark OPTango with state-of-the-art binary diffing approaches. Evaluation results show that OPTango is more robust and significantly outperforms existing methods against both *default* and *non-default* compiler optimization.**

***Index Terms*—Binary diffing, compiler optimization, representation learning**

I. Introduction

Binary code pervades various domains, encompassing traditional PC software, emerging IoT device firmware, and numerous malicious software. Researching software security issues based solely on binary code presents a challenging yet immensely valuable undertaking due to the absence of high-level language information like data structures and types [1]. Binary diffing has attracted considerable attention in recent years due to its broad applications and utility in diverse domains, including vulnerability discovery, malware detection, software plagiarism detection, patch analysis, and software supply chain analysis [2–8]. As deep learning continues to advance, deep learning methods have been increasingly adopted in mainstream binary diffing tasks, resulting in significant enhancements in accuracy.

Compiler optimization can significantly alter the structure of binary code, leading to notable distinctions. Figure 1 illustrates the contrasting code layout between binary code compiled from the same source code of the `help_main()` function in OpenSSL-1.1.1 using GCC -O0 option and GCC -O3 option. Consequently, previous studies have recognised compiler optimisation as a crucial challenge in binary diffing. For instance, Zeek [9] and jTrans [8] emphasize the need to evaluate the resilience of their methods against various compiler optimization levels (such as -O0, -O1, -O2, -O3, and -Os) as a significant experimental setup.



(a) -O0 (b) -O3

Fig. 1. Different binary code compiled from the same source code of help_main in Openssl-1.1.1. (a). Compiled with GCC -O0 option (b). Compiled with GCC -O3 option.

However, previous studies only concentrate on *default* optimization levels, neglecting in-depth exploration of the more intricate *non-default* optimization settings which are challenging to analyze comprehensively. For instance, GCC 11.2 offers more than 200 compiler optimization flags [10], which could be combined arbitrarily to compile the source code, making it impractical to exhaustively enumerate binary variants.

The influence of *non-default* optimization options on binary diffing tasks should not be ignored but rather treated with utmost importance. As revealed by recent research, by fine-tuning compiler options alone, the binary code differences can be up to 90%. Moreover, an increasing number of applications are compiled using *non-default* compiler options to amplify binary code disparities, effectively evading binary detection tools [11]. To gain a deeper understanding of the impact of *non-default* options on binary diffing tasks, we examined their resilience to both *default* and *non-default* compiler options using five state-of-the-art binary diffing tools. As Figure 2 shows, while these binary diffing tools achieve acceptable diffing accuracy on *default* options, there is a significant

The work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320000 and XDA0320300.

2332-6549/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSRE59848.2023.00013

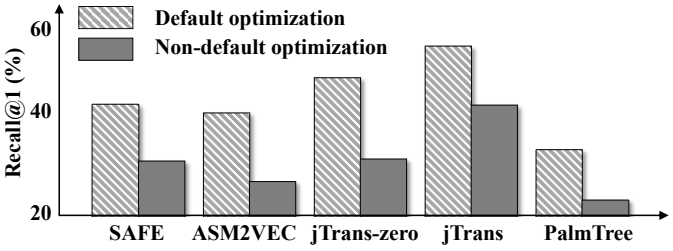


Fig. 2. Recall@1 scores of SOTA binary diffing tools on binaries compiled from SPEC CPU2006/2017 benchmarks with *default* and *non-default* compiler optimization settings. Each function in the index pool contains variants compiled from -O0 and -O2 options. The query functions in the *default* set are compiled from the -O0 and -O3 options, while the *non-default* set encompasses over 500 different optimization settings.

decrease in diffing accuracy when encountering *non-default* options. Consequently, addressing the effects of these *non-default* options on binary diffing becomes increasingly crucial.

Binary diffing is typically modelled as a retrieval task, aiming to search for a query binary function within an index pool consisting of multiple binary functions [1]. Since the binary representations of functions can be easily changed by compiler optimization, simply comparing the binary representations can hardly yield satisfactory accuracy. Many binary diffing works have been exploring the representations of functions [3,6,7,12–15], including the instruction statistics [14], encoded CFG [3,7], etc. Recently, there has been a growing trend in diffing research to leverage machine learning techniques for representation learning, enabling the formation of feature embedding for functions. Existing binary diffing works [2,8,16,17] commonly focus on aligning the feature embedding of several optimization variants towards a central direction. The objective is to learn similar feature embedding for different variants of a specific function. By bringing the feature embedding of different function variants close to each other, these methods attempt to adapt to changes in the binary function representation.

However, the above experiment has proved that the one-central diffing way is clearly insufficient when considering the significant differences among various compiler optimization variants. Even with *default* compiler optimization, the diffing results are also limited. For example, the diffing accuracy is less than 60% when comparing a query function compiled with the -O3 option against an index function compiled with the -O0 option for jTrans [8]. Once the non-default optimization options, which have much larger combination space and more significant differences, are taken into account, the diffing result is far from satisfactory.

This paper systematically researches the interference of compiler optimization (including both *default* and *non-default* options) on binary diffing tasks for the first time. To address the diversity of variants resulting from different compiler optimization, our main idea is to enrich the index pool by increasing multiple representative optimization variants for each function (refer to multi-variant index diffing in the subsequent text). Then, a multi-central representation learning method is devised to release the potential of the multi-variant index diffing policy. This approach helps alleviate the impact of diverse optimization variants and enhances the accuracy of the retrieval process.

We propose OPTango, an innovative transformer-based multi-central representation learning approach, exploring the solution to build a compiler optimization-agnostic binary diffing tool. Considering there are multiple (T) representative optimization (e.g., -O0 and -O2) variants for each target function in the index pool, instead of blindly aligning the embedding over all optimization variants, we align the embedding of each variant of the target function to the most suitable representative variant in the index pool. Specifically, all variants are automatically divided into T groups based on their similarity to the T variants in the index pool. Then, we align the embedding of variants within each group. Compared to previous approaches of treating all variants as a single group, our method considers all variants as multiple automatically generated groups. This reduces the variance within each group and is more conducive to representation learning. As a result, we have achieved a significant improvement in diffing accuracy.

Additionally, although multi-variant index diffing provides better accuracy than single-variant index diffing, it also increases query time proportional to the number of variants. To address this issue, we introduce a lightweight module named group predictor. For a given function, this module can explicitly predict its closest optimization setting among the settings used by the T variant groups before similarity diffing. Only the functions from the predicted variant group in the index pool are used for diffing, thereby reducing the time cost. The time overhead of this module is negligible compared to the entire representation extraction process, and its complexity remains unaffected by the size of the index pool. Experimental results (subsection V-D) demonstrate that using the group predictor module, OPTango achieves comparable accuracy to multi-variant index diffing while maintaining nearly the same time overhead as single-variant index diffing.

In addition to our novel approach, we introduce a diverse and unique dataset called OPTBinary. OPTBinary is compiled from SPEC CPU2006 and SPEC CPU 2017, which are popular utilities in binary diffing evaluations [18–20] using common *default* optimization levels (including -O0, -O1, -O2 and -O3), as well as rich *non-default* optimization settings searched through BinTuner [11]. This compilation pipeline produces binary files with a wide range of optimization variants, resulting in a total of 3.6 million functions over 514 optimization settings. Our newly created dataset allows us to fully explore the representation of the same functions under different compiler optimization configurations. It also serves as a foundation for systematically addressing the resistance of compilation optimization on binary diffing tasks in the future. To the best of our knowledge, this is the first dataset that incorporates *non-default* optimization for binary diffing tasks.

In summary, we have made the following contributions:

- We firstly systematically study the resistance of compiler optimization in binary diffing tasks by incorporating and addressing both *default* and *non-default* optimization.

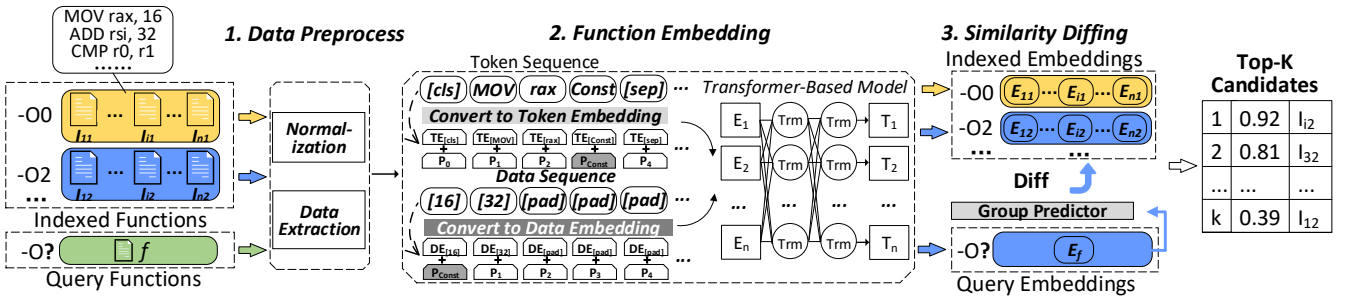


Fig. 3. The overall workflow of OPTango consists of three components: 1) Preprocess input assembly files, additionally extracting data information. 2). Function embedding with a multi-central representation learning method. 3). Similarity diffing to obtain the top-k candidate functions. Provide a group predictor to maintain the benefits of multi-variant index diffing without incurring additional time overhead.

- We propose OPTango, an innovative transformer-based multi-central representation learning approach, exploring the solution to build a compiler optimization-agnostic binary diffing tool. Additionally, we introduce a group predictor to maintain the benefits of multi-variant index diffing without incurring additional time overhead.
- We have implemented a prototype of OPTango and evaluated it in four scenarios: diffing with *default* optimization, diffing with *non-default* optimization, diffing with group predictor and real-world CVE search tasks. The results demonstrate that OPTango can significantly outperform the state-of-the-art methods.
- We construct a newly-created dataset, OPTBinary, with 3.6 million functions over 514 optimization settings.
- The model and dataset are publicly available[1].

II. Problem Definition

Binary diffing is a crucial task in identifying the similarity of two or more code snippets (for example, an entire program, function, basic block, or even instruction) when source code is unavailable. In this paper, we follow the mainstream approach [1] and adopt a function-level diffing method. There are generally three comparison scenarios [21] as follows: (1) One-to-One (OO), which provides a similarity score between a query function and a target function; (2) One-to-Many (OM), which ranks a pool of target functions based on their similarity scores with a query function; (3) Many-to-Many (MM), which treats all input pieces as equal and compares them against each other without distinguishing between query and target. As discussed in previous work [8], by appropriately adjusting the settings for the OM problem, it can be transformed into both the OO problem and the MM problem. Therefore, this paper primarily focuses on the OM problem. Given an assembly function, our objective is to diff for functions with similar semantics from a function index pool. We formally define the task of binary diffing as follows: Given a query function f_q and an index pool F, the task is to retrieve the top k functions in F with the highest similarity.

III. Overall Workflow

Figure 3 depicts the overall workflow of OPTango, which consists of three steps as follows:

[1]https://github.com/orangehn/OPTango.

Firstly, we preprocess the input assembly files by additionally incorporating data information compared with the existing approaches. The inclusion of data information enhances the model's ability to recover data flow and understand semantics.

Secondly, in the function representation learning phase, we vectorize the normalized token sequence and the data sequence. These vectorized tokens are then fed into a transformer-based multi-central representation learning model to generate the function embedding for similarity diffing.

Finally, we diff the embedding vector of query function f against the other embedding vectors in the index pool and retrieve the tok-k candidates as results. Before performing the diffing process, we can utilize the group predictor to estimate the closest optimization group for diffing, thereby reducing the time required for multi-variant index diffing.

IV. Method

In this section, we begin by introducing the process of representing an assembly function as a vector in subsection IV-A. Then, we present the core method of function representation learning, which is multi-central representation learning in subsection IV-B. It involves two mutually iterative processes: group assignment and group learning. Following that, we discuss the group predictor module, which plays a crucial role in reducing the time required for diffing in subsection IV-C. Finally, we provide a concise overview of the construction method employed for OPTBinary in subsection IV-D.

A. Function Embedding

BERT [22] is a pre-trained language model that uses transformer architecture to understand and generate natural language representations. OPTango follows a similar approach to BERT for text modelling, creating token embeddings and using the attention mechanism to model binary code.

Data preprocess. To normalize a disassembled function, the following preprocessing steps are applied:

1) Use the mnemonics and operands as tokens.
2) Replace string literals with a special token ⟨str⟩.
3) Keep external function calls' names as tokens and replace internal function call's names with ⟨function⟩.
4) Replace data token (e.g., "16" in the "mov rax, 16") with ⟨const⟩, while preserving the data information for data embedding.

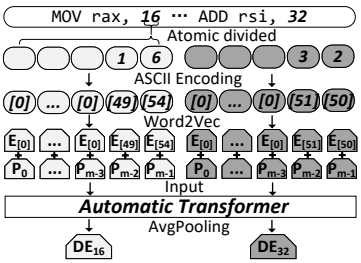


Fig. 4. Data embedding diagram: Each data item is divided into a fixed number of atomic tokens represented in ASCII code format; Then use a lightweight model known as the atomic transformer to enhance the embedding of each atomic token vectorized by word2vec; Finally, average these enhanced atomic embeddings to obtain the embedding for the data token, effectively addressing the OOV problem.

As a result, a given function f can be represented as $([t_1^N, t_2^N, ..., t_n^N], [t_1^D, t_2^D, ..., t_m^D])$ $(m < n)$, where t_i^N represents the i-th normalized token, and t_i^D represents the i-th reserved data token in f.

Normalized Token Embedding. Given a function $f = ([t_1^N, t_2^N, ..., t_n^N], [t_1^D, t_2^D, ...t_m^D])$, the normalized tokens are transformed into vectors $[TE(t_1^N)), TE(t_2^N)), ..., TE(t_n^N))]$. Each token embedding $TE(t_i^N))$ is obtained by adding the token embedding to its corresponding position embedding P_i.

Data Sensitive Embedding. Data flow analysis, as a widely used technique in compiler optimization, helps in understanding variable dependencies and enables optimizations such as loop optimization and constant propagation. In an assembly function, a data token such as the number 16 in the "mov rax, 16" instruction carries crucial semantic information for the program. However, due to the out-of-vocabulary (OOV) problem caused by a large number of numeric combinations, existing approaches [2,8] often address this issue by replacing all data tokens in an assembly function with a single token during preprocessing, which limits the ability to learn the semantics of the function. In this paper, we introduce a data embedding strategy to tackle the OOV problem. This strategy not only resolves the OOV challenge but also utilizes data information to improve the representation of data flow, resulting in enhanced function understanding.

In a given function $f = ([t_1^N, t_2^N, ..., t_n^N], [t_1^D, t_2^D, ...t_m^D])$, t_i^D represents the extracted data token. As depicted in Figure 4, each data item, such as t_i^D, is divided into a fixed number of atomic tokens (e.g., "16" is divided into "1" and "6"). The j-th atomic token is then tokenized by mapping it to its corresponding ASCII code t_{ij}^D. To ensure a fixed number of divided atomic tokens, we pad the atomic token sequence of t_i^D with ASCII 0, as required by the transformer. Next, we utilize the word2vec method to obtain an embedding vector for each atomic token t_{ij}^D, incorporating the corresponding atomic positional embedding P_j^D. This results in the embedding $DE(t_{ij})$ for an atomic token t_{ij}^D.

Subsequently, the atomic token embeddings are improved using a lightweight transformer model known as the atomic transformer. Finally, these enhanced atomic embeddings are averaged to obtain the embedding $DE(t_i^D)$ for the token t_i^D. By employing the atomic embedding approach, any data token can be represented as a vector using a finite number of parameters, effectively addressing the OOV problem. The consolidated representation, as depicted in Figure 3, is fed into a transformer-based model to learn semantic representations.

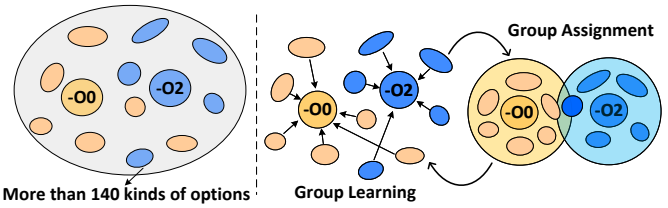


Fig. 5. Multi-central learning diagram: Group assignment determines which optimization variant group each function variant should be assigned to. Group learning clusters all the function variants within one group towards the direction of its central function variant representation to minimize the distance between them.

Function Embedding. Both the embeddings of normalized tokens $TE = [TE(t_1^N), TE(t_2^N), ..., TE(t_n^N)]$ and data tokens $DE = [DE(t_1^D), DE(t_2^D), ..., DE(t_m^D)]$ are inputted into the transformer-based model. This model utilizes the multi-head self-attention mechanism [23] to dynamically assign attention weights to these tokens, considering their relevance to other positions. This enables the model to capture the function semantic information more effectively.

To simplify the formalization of the transformer [24] for better understanding, we use the terms $Transformer$ and θ_T to represent the transformer layers and their parameters, respectively. The whole parameters in model are denoted as $\theta = \{\theta_{TE}, \theta_{DE}, \theta_T\}$. The most common way [25–27] to combine the transformed normalized embedding E_f^N and data embedding E_f^D is simply adding them or concatenate them, since this is not the focus of this paper, we take the average of them to produce the function embedding E_f as follows:

$$\begin{aligned} TE = TE(f; \theta_{TE}), DE = DE(f; \theta_{DE}); \\ E_f^N, E_f^D = \text{Transformer}(TE, DE; \theta_T); \\ E_f = (E_f^N + E_f^D)/2 \end{aligned} \tag{1}$$

The following section will explain how we utilize the multi-central representation learning method to learn robust function representations against diverse optimization variants for binary diffing tasks.

B. Multi-central Representation Learning

To train OPTango, we collect the source code of N functions and then compile them with M compiler optimization settings to obtain N sets of functions, forming a dataset denoted as $F = \{F_1, \ldots, F_n, \ldots, F_N\}$, where function set $F_n \in F$ consists of M variants. It is worth noting that the T optimization settings of the index pool should be included in the M optimization settings, where the corresponding central variants from the T settings are denoted as $G_n = \{g_1, ...g_t, ..., g_T\} \subseteq F_n$. As Figure 5 shows, due to there being multiple (T) optimization (e.g., -O0 and -O2, means T=2) variants for each target function in the index pool, we align the embedding of each variant to the most suitable variant in the index pool (right part of Figure 5) instead of blindly aligning the

embedding over all compiler optimization variants (left part of Figure 5). To achieve this, we design an EM-based approach that iteratively performs group assignment and group learning.

1) Group Assignment: In the group assignment stage, the model's parameters θ are fixed. For each Fn, all variants are automatically divided into T groups based on their similarity to the T central variant in the embedding space. For a binary function $f_i \in F_n$, its function embedding, denoted as E_{f_i}, can be obtained as described in subsection IV-A. Then, the grouping scores are calculated by measuring the cosine similarity between E_{f_i} and the function embedding of the corresponding T centre functions in G_n, as follows:

$$s_t = cos(E_{f_i}, E_{g_t}), \text{ for each } g_t \in G_n \tag{2}$$

Based on the grouping scores, function f_i is assigned to the tid_i-th group through a particular strategy. In this paper, we have explored two representative strategies:

(a) **Max**. The simplest strategy is to select the central variant with the highest grouping score (Equation 3), which means assigning f_i to the group that contains the central variant with the closest proximity in the function embedding space.

$$tid_i = argmax_t \ s_t \tag{3}$$

(b) **Balance**. However, grouping solely based on the highest score may lead to an imbalance problem, where the number of functions in each group can vary significantly. The imbalance problem poses significant challenges for model optimization. To address this issue, we adopt a strategy where we select the top 1/T closest instances for each central variant, forming a sample for that particular group. This helps mitigate the imbalance problem and ensures a more balanced distribution of functions within each group.

2) Group Learning: In the group learning stage, we update model parameters θ by aligning the embedding of variants within each group. We consider compiler optimization flags as latent variables [28], and the process of function representation learning also involves modelling these latent variables. By incorporating latent variable modelling, even when facing function variants compiled with uncovered combinations of optimization flags, we can classify them into appropriate central groups. This enables binary diffing tasks to effectively handle the impact of innumerable compiler optimizations, ensuring robustness and adaptability.

Giving a pair of functions $f_i \in F_{n_1}$ and $f_j \in F_{n_2}$ ($i \neq j$), We first measure their dissimilarity by calculating the normalized Euclidean distance D_{ij} between their function embedding as follows:

$$D_{ij} = ||\frac{E_{f_i}}{||E_{f_i}||_2} - \frac{E_{f_j}}{||E_{f_j}||_2}||_2 \tag{4}$$

where $||*||_2$ is Euclidean norm of the given vector. This distance metric, along with negative cosine similarity, achieves similar effects, while the former is more conducive to optimizing the model. Then, all possible function pairs are divided into three sets: Pos, Nau and Neg, based on their function sets F_{n_1}, F_{n_2} and assigned group tid_i, tid_j (Equation 5). For example, if f_i and f_j are two variants of the same function set and they are assigned to the same group, we categorize them as positive pair in Pos.

$$a_{ij} \in \begin{cases} Pos, & \text{if } F_{n_1} = F_{n_2} \ \& \ tid_i = tid_j \\ Nau, & \text{if } F_{n_1} = F_{n_2} \ \& \ tid_i \neq tid_j \\ Neg, & \text{otherwise} \end{cases} \tag{5}$$

We aim to minimize the distance in the feature space for function pairs in Pos while maximizing the distance for function pairs in Neg. For function pairs in Nau, We impose soft constraints or even no constraints on them, allowing them to adjust automatically along with the model. The loss function can be defined as follows:

$$L = log(\sigma(\sum_{a_{ij} \in Pos} \frac{D_{ij}}{|Pos|} + \alpha \sum_{a_{ij} \in Nau} \frac{D_{ij}}{|Nau|} - \sum_{a_{ij} \in Neg} \frac{D_{ij}}{|Neg|})) \tag{6}$$

where $\sigma(*) = 1/(1 + e^{-*})$ and $|*|$ is number of elements in set. α is the soft loss weight whose value can range between 0 and 1. However, in this paper, we simply set it as 0.

In order to improve the performance and generalization ability of the model, we employ the technique of hard example mining [29], which involves identifying and utilizing challenging samples to optimize the model. The corresponding loss function is accordingly modified as follows:

$$L = log(\sigma(\max_{a_{ij} \in Pos} D_{ij} + \alpha \max_{a_{ij} \in Nau} D_{ij} - \min_{a_{ij} \in Neg} D_{ij})) \tag{7}$$

During the group learning phase, the model parameters θ are optimized by minimizing the loss function L.

Based on the model with updated parameters θ, the function embeddings of all variants in the dataset are also updated accordingly. Subsequently, a new round of group assignment is performed. We iterate through the process of group assignment and group learning until a stable state is reached. The final output consists of the updated model parameters θ and the resulting group assignments tid.

C. Group Predictor

In this subsection, we will discuss the process of multi-variant index diffing and the role of the group predictor in reducing its time cost.

The index pool $I = \{I_{nt} | n = 1, 2, ..., N; t = 1, 2, ...T\}$ consists of N functions compiled from T compiler optimization settings. For multi-variant index diffing, the query function f needs to be compared with all functions in the T variant groups of the index pool to find the best match function $\hat{I_{nt}}$ as follows:

$$\hat{I_{nt}} = argmax_{I_{nt}} \ cos(E_f, E_{nt}), \text{ for each } t \in \{1, 2, ..T\} \tag{8}$$

where E_f and E_{nt} are function embedding extracted with the trained model, E_{nt} is the simplification of $E_{I_{nt}}$. Multi-variant index diffing provides improved accuracy compared to single-variant index diffing. However, it also increases query time proportional to the number of variants, which becomes impractical when the index pool contains numerous functions.

Therefore, we introduce a lightweight module named group predictor, which is supervised by the tid generated from group

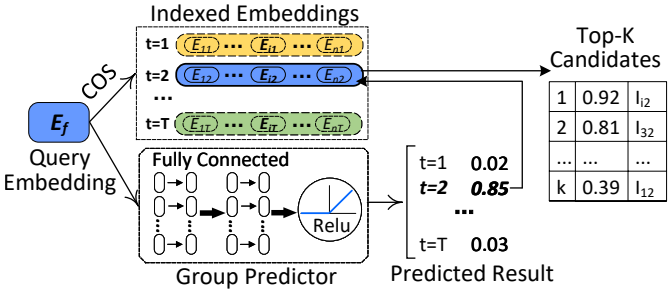


Fig. 6. Group predictor module: automatically predicts the variant group under the closest optimization settings with the query function, maintaining the benefits of multi-variant index diffing and reducing the time cost.

assignment. As shown in Figure 6, for a given query function f, the goal of this module is to automatically select the variant group under the closest optimization setting as f from the index pool. Instead of comparing f with all the variant groups, only the predicted variant group is used for diffing, resulting in a reduction in time cost.

Along with the T optimization settings of the index pool, we model the group predictor GP as a T-class classifier. The module consists of two fully-connected layers followed by an activation function. It takes the embedding E_f of the query function as input and outputs the probability score for each of the T optimization settings:

$$GP(E_f) = [p_1, p_2, \ldots, p_T] \tag{9}$$

Then the variants under the t'-th optimization settings in the index pool are selected for diffing, where the t'-th optimization setting corresponds to the one with the highest score. The best match function $\hat{I_{nt'}}$ is selected as follows:

$$t' = argmax_t\ p_t \tag{10}$$

$$\hat{I_{nt'}} = argmax_{I_{nt'}}\ cos(E_f, E_{nt'}) \tag{11}$$

By focusing on the diffing between f and the variant group under a specific optimization setting, we can effectively decrease the time required for diffing to match that of single-variant index diffing. This optimization is achieved through the utilization of the group predictor.

When training the group predictor, we use the tid_i obtained through group assignment as the supervision for f_i. We employ a standard cross-entropy classification loss to define the training objective, denoted as $\mathcal{L}_{gp}$:

$$\mathcal{L}_{gp} = \sum_{i=1}^{N} -\log(p_{tid_i}) \tag{12}$$

Moreover, the time overhead of this module is minimal compared to the entire process of extracting embedding, and its complexity remains unaffected by the size of the index pool. Experimental results (subsection V-D) demonstrate that by using the group predictor module, OPTango achieves comparable accuracy to multi-variant index diffing while maintaining nearly the same time overhead as single-variant index diffing.

D. Large-scale Optimization Variant Dataset Construction

As previous research methods have not addressed the handling of *non-default* optimization settings, we created a comprehensive dataset to systematically investigate the impact of compiler optimization (including both *default* and *non-default* optimization) on binary diffing tasks. Our dataset is constructed using all C/C++ programs in SPEC CPU 2006 and SPEC CPU 2017, which often served as a complicated evaluation benchmark for binary diffing methods [18–20].

***Default* and *Non-default* optimization variants Generation**: To generate optimization variants of a selected benchmark program, we divide the generation process into two parts. Initially, we generate binary code using 4 *default* optimization levels, specifically the O0−O3 options. Subsequently, we employ Bintuner [11] to generate *non-default* compiler optimization variants. Bintuner iteratively searches near-optimal optimization settings that can maximize the amount of binary code differences. For each benchmark program, we conduct 5 sets of searches using distinct settings, with each set comprising 3 independent searches. Moreover, we take care to ensure that the resulting *non-default* optimization settings are different from one another. Binaries generated in the first four groups (named as Ot_0−Ot_3) are designed to exhibit substantial differences compared to the binaries compiled using the corresponding *default* optimization level (O0−O3). For instance, binaries in Ot_0 group are intended to be notably distinct from those compiled with the -O0 option. In the fifth group (named Ot_4), our optimization objective is to maximize the variability in the generated binaries compared to those compiled from all 4 *default* optimization levels, ensuring that binaries in this group are distinct from any of those compiled with the *default* optimization levels. By the end of the iteration, each benchmark program will encompass 19 = (5*3)+4 compiler optimization variants. And the 15 *non-default* optimization settings searched by Bintuner vary for different benchmark programs. Experimental results in subsection V-C will indicate that the variants discovered through Bintuner pose a greater challenge to binary diffing tools than those compiled with *default* optimization options.

Label Collection: To collect the labels, we need first to obtain the unstripped binary code and the offsets of the functions. We refer to jTrans [8] to obtain the labels from the symbol table.

V. Evaluation

In this section, we compare our method with available state-of-the-art binary diffing approaches. The experiments aim to answer the following four research questions:

- **RQ.1:** How well does OPTango perform when diffing with *default* optimization options?
- **RQ.2:** How accurate is OPTango when diffing with *non-default* optimization options?
- **RQ.3**: How effective is the group predictor in OPTango?
- **RQ.4**: How effective is the data embedding in OPTango?
- **RQ.5**: How effective is OPTango in searching real-world vulnerabilities?

A. Experiment Setup

We implement OPTango using transformers [30] based on Python 3.8.5 and Pytorch 1.12.0. In our experiments, all binary

TABLE I
STATISTICS ON SUBSET PARTITION CRITERIA, COMPILER OPTIMIZATION VARIANTS, BINARIES, AND FUNCTIONS OF THE THREE SUBSETS.

Subset	Criteria: function number in per binary	*Default* compiler options	*Non-default* compiler options	*Non-default* option variant groups	Binary	Function
Small	<200	4	165	Ot_0, Ot_1,Ot_2,Ot_3,Ot_4	209	19,510
Medium	200∼2,000	4	180	Ot_0, Ot_1,Ot_2,Ot_3,Ot_4	228	208,223
Large	>2,000	4	165	Ot_0, Ot_1,Ot_2,Ot_3,Ot_4	209	3,400,213

programs are stripped to remove symbol information. We use IDA Pro 7.5 [31] to disassemble and extract functions from the binaries. All the training and inference are conducted on a Linux server running Ubuntu 16.04, equipped with an Intel Xeon CPU with 56 cores, 503GB RAM, and 8 V100 GPUs.

1) Implementation: In the experiments, we follow the parameter settings and pre-trained method described in jTrans-zero [8] for the Large Language Model. For data embedding, we implement a two-layer transformer encoder. Additionally, during multi-variant index diffing, we selected two representative optimization variants compiled from -O0 and -O2 options as the T variants of the index pool. This is motivated by existing work on compiler chain provenance [32–34], which categorizes compiler optimization effects into two classes: "high" and "low." The low optimization class is denoted as -O0, while the high optimization class is typically represented as -O2. In future studies, we will investigate the influence of different selections and quantities of optimization variant centres on binary diffing tasks.

2) Dataset: We present OPTBinary, a large-scale dataset created to evaluate binary diffing methods under diverse compiler optimization configurations. OPTBinary consists of numerous functions and their variants compiled from all C/C++ benchmarks in SPEC CPU 2006 and SPEC CPU 2017. These benchmarks are commonly used in binary diffing tasks [18–20]. All functions in OPTBinary are compiled with both *default* and *non-default* optimization settings.

In general, the number of functions within a binary impacts the optimization space available to the compiler [35]. As the number of functions increases, the compiler's optimization space expands, leading to greater diversity within the binary code. To evaluate the ability of binary diffing methods to handle different compiler optimization variants and various pooling sizes (a common metric for binary diffing [8,16]), we divided the dataset into three subsets: *small*, *medium*, and *large*, based on the number of functions in each program, as listed in Table I. For example, 619.lbm_s in SPEC CPU2017 has approximately 25 functions and is classified into *small* subset, while 526.blender_r contains around 42,000 functions and is categorized into *large* subset. The *small* subset comprises roughly 20,000 functions, the *medium* subset includes around 21,000 functions, and the *large* subset encompasses about 340,000 functions. In total, the dataset comprises approximately 3,500,000 functions. The number of binaries in each subset is well-balanced, and there is a relatively equal distribution between *default* and *non-default* optimization variants. These *non-default* optimization variants are generated using the algorithm described in subsection IV-D and are organized into five groups: Ot_0, Ot_1, Ot_2, Ot_3, and Ot_4. The entire dataset comprises a total of 514 = 4+(165+180+165) optimization settings.

Among the existing open datasets, BinKit [36] stands out as the largest binary dataset, however, it only covers variants compiled from 6 *default* optimization options ranging from -O0 to -Ofast option, thus unusable in the *non-default* scenario. Similarly, Binarycrop [8] claims to be more diverse than BinKit yet includes only five *default* options from -O0 to -Os option, which is also incompetent. The newly created dataset OPTBinary, along with the trained models, will be made available to the community. It provides a more comprehensive and realistic basis for learning and evaluating the effectiveness of binary diffing methods against the wide range of compiler optimization variants.

3) Baselines: We selected five state-of-the-art binary diffing tools that have considered the impact of compiler optimization as our comparison targets:

- **SAFE** [16] utilizes a word2vec model to generate instruction embeddings and proposes a recurrent neural network for function embedding generation.
- **Asm2Vec** [2] employs random walks on the function CFG to sample instruction sequences and then uses the PV-DM model to jointly learn function and instruction token embedding.
- **jTrans-zero** [8] incorporates control flow information from binary code into transformer-based language models for function embedding.
- **jTrans** [8] fine-tunes the pre-trained model to generate function embedding for the supervised learning task of binary diffing.
- **PalmTree** [17] utilizes pre-trained models to generate instruction embedding, which is then averaged to calculate function embedding in our evaluation.

4) Evaluate Metric: Recall@K

Let there be a query function pool F, where $F = \{f_1, f_2, ..f_i, ...f_N\}$, and its ground truth function index pool GT, where $GT = \{f_1^{gt}, f_2^{gt} ...f_i^{gt}, ...f_n^{gt}\}$. We denote a query function $f_i \in F$ and its corresponding ground truth function $f_i^{gt} \in GT$. In the binary diffing scenario, the main goal is to retrieve the top-k functions in index pool GT, which have the highest similarity to f_i. The returned functions are ranked by a similarity score, $R_{f_i}^{gt}$, which denotes their position in the list of retrieved functions. The diffing accuracy can be evaluated using the following metric:

$$R@k = \frac{1}{|F|}\sum\nolimits_{f_i \in F}(R_{f_i}^{gt} \leq K) \times 100\% \tag{13}$$

TABLE II
RESULTS OF DIFFERENT BINARY DIFFING METHODS WITH *DEFAULT* COMPILER OPTIMIZATION (%)

Models	Small Subset				Medium Subset				Large Subset			
	spec17 619.lbm_s (25)	spec06 401.bzip2 (86)	spec17 508.namd_r (154)	avg (88)	spec06 456.hmmer (507)	spec06 450.soplex (1048)	spec06 453.povray (1720)	avg (825)	spec06 403.gcc (5k)	spec17 623.xalan (18k)	spec17 526.blender_r (42k)	avg (13k)
SAFE	82.4	75.8	35.6	67.9	74.5	50.4	59.2	61.0	61.8	35.9	40.3	40.1
Asm2Vec	91.7	69.8	34.5	68.4	74.3	57.7	62.9	64.1	58.0	33.7	35.9	38.1
jTrans-zero	91.2	82.9	38.8	75.0	82.0	56.0	68.7	69.4	68.1	36.8	46.9	45.1
jTrans	94.1	88.6	43.6	80.8	89.4	66.3	76.8	77.5	74.7	46.3	51.1	51.1
PalmTree	85.3	67.2	35.1	65.4	65.9	47.8	51.7	54.0	48.6	25.4	31.7	31.0
OPTango	**97.1**	**94.3**	**44.9**	**81.4**	**91.3**	**67.8**	**77.9**	**79.2**	**82.8**	**50.5**	**60.2**	**57.8**

TABLE III
RESULTS OF DIFFERENT BINARY DIFFING METHODS WITH *NON-DEFAULT* COMPILER OPTIMIZATION (%)

	Small Subset						Medium Subset						Large Subset					
Models	Ot_0	Ot_1	Ot_2	Ot_3	Ot_4	avg	Ot_0	Ot_1	Ot_2	Ot_3	Ot_4	avg	Ot_0	Ot_1	Ot_2	Ot_3	Ot_4	avg
SAFE	56.6	66.5	76.3	84.7	54.2	68.4	40.5	50.8	67.0	69.2	38.0	54.2	32.2	28.7	28.2	28.4	28.2	29.0
Asm2Vec	52.7	64.2	74.4	77.1	45.9	64.0	39.3	50.5	68.3	70.9	35.3	54.4	24.3	24.1	24.7	26.4	23.6	24.6
jTrans-zero	63.7	69.3	75.2	81.9	60.1	70.5	51.0	51.9	66.0	68.0	46.4	57.3	32.5	27.8	28.5	28.7	28.4	29.1
jTrans	**75.6**	79.7	84.4	86.3	74.1	80.3	66.3	70.2	78.8	80.7	65.1	72.7	41.4	38.2	38.3	39.8	39.0	39.3
PalmTree	53.0	63.2	71.7	76.2	49.6	63.4	32.7	42.7	62.2	64.2	27.7	47.2	20.9	20.8	20.9	22.5	21.0	21.2
OPTango	75.3	**80.2**	**87.8**	**90.3**	**75.1**	**82.1**	**67.2**	**73.6**	**82.0**	**84.2**	**67.0**	**75.4**	**47.0**	**42.7**	**42.8**	**44.6**	**43.2**	**43.9**

In the following evaluation, K is set to 1. Additionally, it is worth noting that all our experimental results underwent cross-validation unless otherwise specified and are weighted average based on the number of functions across all benchmarks.

B. Diffing with Default Compiler Optimization

To assess the superiority of the multi-central representation learning approach over existing binary diffing methods under *default* compiler optimization options, we included both -O0 and -O2 optimization variants for each function in the index pool. We then performed function diffing by sequentially querying with -O1 and -O3 optimization variants. To demonstrate the diffing accuracy of OPTango and the baselines with different pool sizes, we conducted evaluations with the same settings on the *small*, *medium*, and *large* subsets of the dataset.

The comparison results of OPTango and the baselines are shown in Table II. OPTango outperforms all the baselines by considerable margins across all three subset datasets. For example, OPTango achieves a top-1 recall of 94.3% on 401.bzip2 in the *small* subset, followed by jTrans with a score of 88.6%. In the *large* benchmark, OPTango outperforms SAFE with an increase of 44.1%, Asm2Vec with a 51.7% increase, jTrans-zero with a 28.2% increase, jTrans with a 13.1% increase, and PalmTree with a 86.5% increase.

As discussed earlier, the binary diffing process can be regarded as a searching process, where the function number represents the size of the searching space. When comparing the diffing accuracy of each diffing tool horizontally, we observe a decrease in accuracy as the number of functions increases. For example, the average accuracy of Asm2Vec dropped from 68.4% to 64.1% and 38.1% in the *small*, *medium*, and *large* subsets, respectively. This downward trend can be attributed to the increased number of functions in each subset, bringing more optimization opportunities for the compiler, which in turn has a greater impact on the differences among the binaries. The experimental results clearly demonstrate that our multi-central representation strategy outperforms all other methods as the function pool size increases.

Answer to RQ.1: OPTango consistently achieves the top ranking in all pool size diffing tasks, showcasing the robustness of OPTango's multi-central representation learning method against *default* compiler optimization.

C. Diffing with Non-default Compiler Optimization

Given the growing trend of compiling binaries using *non-default* optimization settings in recent years [11], we assess the diffing accuracy under such conditions, which has not been explored in previous methods. Unlike the diffing conducted with several *default* optimization levels, the diffing with *non-default* optimization employs function variants from five sets with over 500 settings, namely Ot_0, Ot_1, Ot_2, Ot_3, Ot_4, to compare function similarity. Table III presents the comparative results of OPTango and other baseline methods.

The results highlight OPTango's superiority over five state-of-the-art approaches in 14 different settings, except the Ot_0, where OPTango's accuracy is slightly lower than the recall@1 score of jTrans in *small* subset. To illustrate the diffing accuracy, we consider the Ot_3 set as an example. In Ot_3, the benchmarks are generated by searching for the maximum normalized compression distance (NCD) between the target binary compiled from *non-default* settings and the O3-based binary. OPTango excels with the highest recall@1 score of 90.3% in the *small* subset and 84.2% in the *medium* subset. Despite an accuracy decrease in the *large* subset, OPTango still outperforms its closest baseline competitor by 12.1%.

By comparing Table II and Table III, a significant decrease in accuracy can be observed under the *non-default* optimization settings compared to the *default* optimization settings for these binary diffing methods, especially in the *large* dataset. This suggests that *non-default* optimization variants take more challenges for binary diffing tasks than *default* optimization variants do. This finding highlights the importance of considering the impact of *non-default* optimization on binary diffing tasks and reinforces the motivation behind OPTango.

Furthermore, conducting a vertical comparison of Table III, we observe that binaries in Ot_4, generated by searching for

TABLE IV
RESULTS OF DIFFING WITH *DEFAULT* OPTIMIZATION WITHIN THE TIME COST OF SINGLE-VARIANT INDEX DIFFNG (%)

Models	Small Subset				Medium Subset				Large Subset			
	spec17 619.lbm_s (25)	spec06 401.bzip2 (86)	spec17 508.namd_r (154)	avg (88)	spec06 456.hmmer (507)	spec06 450.soplex (1048)	spec06 453.povray (1720)	avg (825)	spec06 403.gcc (5k)	spec17 623.xalan (18k)	spec17 526.blender_r (42k)	avg (13k)
Best Recall@1												
SAFE	82.4	75.8	33.6	66.1	73.1	50.2	56.4	59.2	58.8	33.1	34.9	36.7
Asm2Vec	91.7	70.6	34.6	68.0	73.7	57.0	60.6	62.8	55.8	31.0	28.3	33.6
jTrans-zero	91.2	82.9	37.9	74.7	81.9	55.6	68.7	69.3	68.1	36.5	46.9	45.0
jTrans	94.1	87.0	43.2	79.5	87.8	65.6	74.4	75.8	72.0	45.1	49.9	49.7
PalmTree	82.4	68.0	34.4	64.5	65.1	47.4	50.2	52.9	46.7	23.3	27.3	28.4
Mean Recall@1												
SAFE	70.6	59.4	27.6	52.5	57.3	29.9	38.2	41.0	38.1	20.3	20.0	22.0
Asm2Vec	59.8	43.3	22.3	43.1	43.7	30.4	34.3	36.2	31.3	17.8	15.1	18.4
jTrans-zero	70.6	57.3	25.6	51.9	49.8	31.8	40.2	41.4	37.7	21.7	24.7	24.8
jTrans	85.3	83.8	38.8	74.1	83.0	52.7	62.4	65.2	59.7	34.5	36.9	37.9
PalmTree	54.5	43.8	22.0	41.4	36.8	25.5	27.7	29.9	26.0	13.4	14.5	15.5
OPTango	**97.1**	**92.7**	**45.4**	**81.0**	**91.5**	**66.9**	**74.7**	**77.0**	**80.5**	**48.0**	**59.9**	**56.4**

TABLE V
RESULTS OF DIFFING WITH *NON-DEFAULT* OPTIMIZATION WITHIN THE TIME COST OF SINGLE-VARIANT INDEX DIFFNG (%)

	Small Subset						Medium Subset						Large Subset					
Models	Ot_0	Ot_1	Ot_2	Ot_3	Ot_4	avg	Ot_0	Ot_1	Ot_2	Ot_3	Ot_4	avg	Ot_0	Ot_1	Ot_2	Ot_3	Ot_4	avg
Best Recall@1																		
SAFE	36.4	59.1	73.8	82.3	39.3	59.5	20.3	42.7	63.6	67.0	21.7	45.0	22.0	22.1	20.5	20.9	20.8	21.2
Asm2Vec	19.6	53.1	71.0	75.6	25.6	51.6	10.0	39.0	62.7	66.6	11.5	41.2	14.3	17.6	16.5	18.6	16.0	16.7
jTrans-zero	28.5	56.5	71.9	80.4	32.7	55.3	11.6	36.8	62.3	66.8	13.0	40.1	20.2	22.8	21.0	22.7	20.9	21.5
jTrans	67.6	75.8	83.8	86.0	67.6	76.6	50.7	64.2	77.6	80.4	52.0	66.1	33.1	34.9	33.4	36.6	33.6	34.3
PalmTree	19.5	50.7	67.5	72.9	24.9	48.7	6.1	31.5	57.2	60.9	6.7	34.9	13.2	15.8	14.8	16.3	14.7	15.1
Mean Recall@1																		
SAFE	45.7	52.1	58.4	63.1	44.3	53.2	29.7	37.2	46.7	48.1	29.0	38.9	19.2	17.0	16.6	16.4	17.0	17.1
Asm2Vec	35.9	43.0	49.9	51.9	32.8	43.5	24.3	32.6	42.7	44.0	22.8	34.4	12.9	13.2	13.0	14.1	12.9	13.2
jTrans-zero	45.8	47.3	51.0	55.2	42.9	48.6	31.2	33.9	42.5	43.5	29.7	36.6	19.9	17.9	17.7	18.2	18.1	18.3
jTrans	71.3	72.8	75.8	77.0	70.0	73.5	58.0	59.9	65.0	66.3	57.9	61.7	31.3	29.3	29.1	30.3	29.2	29.8
PalmTree	35.7	40.4	45.6	48.3	32.8	40.9	19.1	26.0	37.9	39.1	16.8	28.7	11.4	11.4	11.3	11.9	11.6	11.6
OPTango	**74.7**	**79.9**	**87.6**	**90.1**	**74.1**	**81.6**	**65.6**	**72.1**	**81.0**	**83.2**	**65.4**	**74.1**	**46.1**	**42.3**	**42.3**	**44.0**	**42.7**	**43.4**

the maximum NCD compared to those compiled from all four *default* options, exhibit poorer diffing accuracy contrast with the other four sets of *non-default* variants. For instance, among the baselines, jTrans demonstrates the highest accuracy and achieves a diffing score of 74.1% for Ot_4 in the *small* dataset. However, this is 2.0% lower than Ot_0, 7.0% lower than Ot_1, 12.2% lower than Ot_2, and 14.1% lower than Ot_3. These findings provide further evidence that representation learning for *non-default* optimization variants is more challenging and significantly impacts binary diffing tasks. None of these baselines has addressed the inclusion of *non-default* optimization variants in their methods, while our approach automatically assigns *non-default* optimization variants to the most similar variant centre group, enabling binary diffing tasks to effectively handle interference from different optimization variants. Consequently, OPTango achieves near-optimal diffing accuracy in almost all scenarios.

Answer to RQ.2: OPTango accurately retrieves the best candidates even in the presence of *non-default* optimization.

D. Group Predictor Study

The limitation of the existing binary diffing methods is that only one optimization variant group can be chosen for diffing within the time required for single-variant index diffing. Once there are multiple variant groups in the index pool, it is necessary to perform differential analysis on each variant group to obtain the closest variant in order to improve accuracy for a query function. In our previous evaluations in subsection V-B and subsection V-C, we follow their experimental settings and include several optimization variants for each function in the function index pool to demonstrate the concept of multiple centre points. This setting inevitably increases the diffing time, particularly when dealing with large index pools in practical scenarios. Since there is no group predictor module in these baselines' design, predicting the most suitable group before diffing is not feasible.

To speed up the diffing process, we introduce the group predictor module in our method (described in subsection IV-C) to predict the most suitable centre group for each query function before diffing. This approach enables us to achieve comparable accuracy to multi-variant index diffing while maintaining the same time cost as single-variant index diffing. The experiments are conducted separately for both *default* and *non-default* optimization settings, as depicted in Table IV and Table V, respectively. For these baselines, the average accuracy and the accuracy of the variant group with the best overall recall@1 values are then recorded as Mean Recall@1 and Best Recall@1, respectively. The last row of the table shows the diffing accuracy of OPTango with the group predictor module.

The results demonstrate that OPTango surpasses all baselines for the recall@1 metric in all subsets under both *default* and *non-default* compiler optimization. In the *large* subset from Table IV, OPTango outperforms other models by 13.5% to 98.6% in Best Recall@1 and 48.9% to 263.9% in Mean Recall@1. On the other hand, in the *large* subset from Table V, OPTango consistently outperforms existing methods,

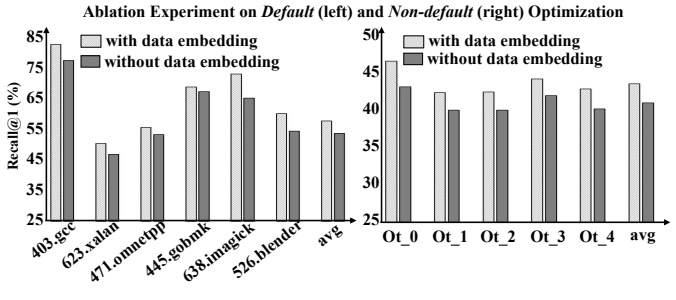


Fig. 7. Ablation experiment on data embedding

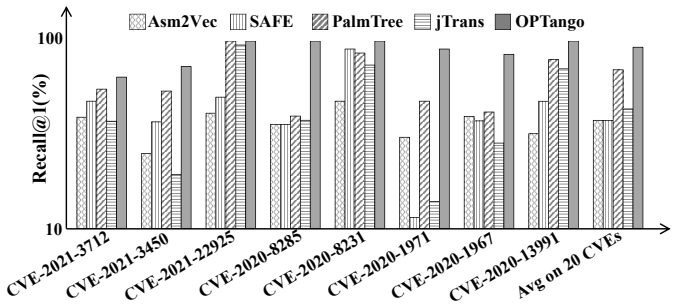


Fig. 8. Recall@1 of real-world vulnerability diffing

with improvements ranging from 26.5% to 274.1% across the *small*, *medium* and *large* subsets in Best/Mean Recall@1. In conclusion, these two sets of experiments demonstrate that OPTango exhibits superior accuracy using the group predictor module compared to existing state-of-the-art works, especially in large-scale benchmarks.

Answer to RQ.3: With the predictor module, OPTango achieves comparable accuracy to multi-variant index diffing within the time spent for single-variant index diffing, whereas methods without the predictor module exhibit significant accuracy degradation.

E. Data Embedding Study

We conducted ablation experiments on the large subsets for data-sensitive embedding in OPTango. As shown in Figure 7, the absence of this component resulted in an average performance reduction of 6.9% and 5.9% under *default* and *non-default* optimization settings(same as TABLE II and TABLE III), respectively. **Answer to RQ.4**: Data embedding plays a significant role under all optimization settings in OPTango.

F. Diffing Vulnerability Functions

Vulnerability detection plays a crucial role in computer security. In this section, we evaluate the accuracy of OPTango and four other baseline approaches in real-world vulnerability searching scenarios. The OpenSSL and Curl projects are widely used in IoT firmware, while JerryScript is a popular IoT JavaScript engine. Therefore, we selected these projects as targets and created a vulnerability repository containing 20 relatively recent Common Vulnerabilities and Exposures (CVE) from the CVE database. To introduce diversity, we generated 50 variants of each function for every CVE by applying different compiler optimization settings. Our evaluation metric remains recall@1. To simulate real-world scenarios, we used all functions within each benchmark as the index pool.

Figure 8 illustrates the recall@1 results for each query, specifically focusing on the latest 8 CVEs out of the total 20. The final set in Figure 8 showcases the average accuracy of each binary diffing method across all 20 CVEs. Clearly, OPTango outperforms the four baselines significantly in the majority of CVE cases. For instance, in CVE-2020-8285 of the libcurl project, our method achieved a remarkable recall@1 of 100%. This implies that it successfully retrieved all 50 variants, while SAFE, Asm2Vec, jTrans, and PalmTree achieved recall@1 values of 60%, 60%, 64%, and 62%, respectively.

Answer to RQ.5: OPTango is highly effective for searching vulnerable functions.

VI. Related Work

Traditional Non-ML Binary Diffing Techniques. Traditional non-ML binary diffing techniques can be categorized into two main groups: static and dynamic methods. Static methods primarily rely on graph isomorphism [4,37], symbolic execution [5,38], and data flow analysis [39,40], etc [41–44]. However, these methods only capture structural and syntactic information, disregarding semantic and inter-instructional relationships, resulting in low precision. On the flip side, dynamic methods for binary diffing involve analyzing the semantics of binaries by executing them [45–50]. However, these approaches are computationally intensive and not suitable for large function repositories.

ML-Based Binary Diffing Techniques. The rapid advancement of machine learning technologies has brought revolutionary developments to the field of binary diffing [6,16,51–54]. For instance, DeepBinDiff [6] and SAFE [16] extract function components such as instructions and basic blocks, transform them into embeddings and calculate similarities between functions. The emergence of transformer-based language models like BERT [22] and GPT [55] has sparked a multitude of ideas and methods in binary diffing tasks. Several assembly language models have been developed to facilitate binary code analysis [8,17,56–58]. Although these machine learning-based approaches are suited for large-scale binary diffing tasks, they only focus on several *default* optimization variants and overlook the impact of innumerable *non-default* compiler optimization, which can also significantly affect diffing accuracy.

VII. Conclusions

We present OPTango, the first try to systematically study the resistance of compiler optimization (including *non-default* and *non-default* optimization) of binary diffing methods, exploring the solution to build a compiler optimization-agnostic binary diffing tools. Our extensive evaluation reveals weaknesses in SOTA methods' and demonstrates OPTango's high accuracy.

VIII. Acknowledgment

This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320000 and XDA0320300.

REFERENCES

[1] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Comput. Surv.*, vol. 54, no. 3, apr 2021. [Online]. Available: https://doi.org/10.1145/3446371

[2] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 472–489.

[3] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 667–678. [Online]. Available: https://doi.org/10.1145/3238147.3238199

[4] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *SSTIC*, vol. 5, 01 2005.

[5] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security*, L. Chen, M. D. Ryan, and G. Wang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 238–255.

[6] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," 01 2020.

[7] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 363–376. [Online]. Available: https://doi.org/10.1145/3133956.3134018

[8] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, "Jtrans: Jump-aware transformer for binary code similarity detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–13. [Online]. Available: https://doi.org/10.1145/3533767.3534367

[9] N. Shalev and N. Partush, "Binary similarity detection using machine learning," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 42–47. [Online]. Available: https://doi.org/10.1145/3264820.3264821

[10] GCC team, "gcc optimize options," 2023. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc/Optimize-Options.html

[11] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: An empirical study," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 142–157. [Online]. Available: https://doi.org/10.1145/3453483.3454035

[12] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla *et al.*, "discovre: Efficient cross-architecture identification of bugs in binary code." in *Ndss*, vol. 52, 2016, pp. 58–79.

[13] H. Flake, "Structural comparison of executable objects," in *Detection of intrusions and malware & vulnerability assessment, GI SIG SIDAR workshop, DIMVA 2004*, U. Flegel and M. Meier, Eds. Bonn: Gesellschaft für Informatik e.V., 2004, pp. 161–173.

[14] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 896–899. [Online]. Available: https://doi.org/10.1145/3238147.3240480

[15] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 461–470.

[16] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, pp. 309–329.

[17] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3236–3251. [Online]. Available: https://doi.org/10.1145/3460120.3484587

[18] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 361–374. [Online]. Available: https://doi.org/10.1145/3564625.3567975

[19] P. Zhang, C. Wu, M. Peng, K. Zeng, D. Yu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, "Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 55–67. [Online]. Available: https://doi.org/10.1145/3579990.3580007

[20] Z. Liu, "Binary code similarity detection," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1056–1060.

[21] I. U. Haq and J. Caballero, "A survey of binary code similarity," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–38, 2021.

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *ArXiv*, vol. abs/1810.04805, 2019.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[24] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[25] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[27] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[28] B. Everett, *An introduction to latent variable models*. Springer Science & Business Media, 2013.

[29] A. Hermans, L. Beyer, and B. Leibe, "In defense of the triplet loss for person re-identification," *arXiv preprint arXiv:1703.07737*, 2017.

[30] Github - Transformers, "Transformers," 2021. [Online]. Available: https://github.com/huggingface/transformers

[31] H. Rays, "Ida pro," 2021. [Online]. Available: https://www.hex-rays.com/products/ida/

[32] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 100–110. [Online]. Available: https://doi.org/10.1145/2001420.2001433

[33] A. Rahimian, P. Shirani, S. Alrbaee, L. Wang, and M. Debbabi, "Bincomp: A stratified approach to compiler provenance attribution," *Digital Investigation*, vol. 14, pp. S146–S155, 2015.

[34] B. Xie, Z. Tian, C. Gao, and L. Chen, "Towards fine-grained compiler identification with neural modeling." in *SEKE*, 2020, pp. 305–310.

[35] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, 2020.

[36] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, 2022.

[37] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *PPREW '13*, 2013.

[38] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, 01 2017.

[39] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *SIGPLAN Not.*, vol. 51, no. 6, p. 266–280, jun 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908126

[40] ——, "Similarity of binaries through re-optimization," *ACM SIGPLAN Notices*, vol. 52, pp. 79–94, 06 2017.
[41] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 461–470. [Online]. Available: https://doi.org/10.1145/2939672.2939719
[42] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 329–338.
[43] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 376–387. [Online]. Available: https://doi.org/10.1145/3395363.3397361
[44] P. Shirani, L. Wang, and M. Debbabi, "Binshape: Scalable and robust binary library function identification using function shape," in *International Conference on Detection of intrusions and malware, and vulnerability assessment*, 2017.
[45] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 303–317.
[46] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 709–724.
[47] C. Mahinthan, Y. Xue, Z. Xu, Y. Liu, C. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," 11 2016, pp. 678–689.
[48] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *USENIX Security Symposium*, 2017.
[49] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17. IEEE Press, 2017, p. 319–330.
[50] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 342–352.
[51] L. Massarelli, G. Luna, F. Petroni, and L. Querzoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," 01 2019.
[52] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, "Codee: A tensor embedding scheme for binary code search," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2224–2244, 2022.
[53] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," 01 2019.
[54] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," 01 2019.
[55] S. Black, L. Gao, P. Wang, C. Leahy, and S. R. Biderman, "Gpt-neo: Large scale autoregressive language modeling with mesh-tensorflow," 2021.
[56] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Trex: Learning execution semantics from micro-traces for binary similarity," 12 2020.
[57] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *AAAI Conference on Artificial Intelligence*, 2020.
[58] H. Koo, S. Park, D. Choi, and T. Kim, "Semantic-aware binary code representation with bert," *ArXiv*, vol. abs/2106.05478, 2021.