BINENHANCE: An Enhancement Framework Based on External Environment Semantics for Binary Code Search

Yongpan Wang[†‡], Hong Li[†‡*], Xiaojie Zhu[§], Siyuan Li[†‡], Chaopeng Dong[†‡], Shouguo Yang[¶], Kangyuan Qin[†‡]

[†]Institute of Information Engineering, Chinese Academy of Sciences, China
[‡]School of Cyber Security, University of Chinese Academy of Sciences, China
[§]King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
[¶]Zhongguancun Laboratory, Beijing, China
ncepu_wyp@163.com, {lihong, lisiyuan, dongchaopeng, qinkangyuan}@iie.ac.cn
xiaojie.zhu@kaust.edu.sa, yangshouguo@outlook.com

Abstract—Binary code search plays a crucial role in applications like software reuse detection, and vulnerability identification. Currently, existing models are typically based on either internal code semantics or a combination of function call graphs (CG) and internal code semantics. However, these models have limitations. Internal code semantic models only consider the semantics within the function, ignoring the inter-function semantics, making it difficult to handle situations such as function inlining. The combination of CG and internal code semantics is insufficient for addressing complex real-world scenarios. To address these limitations, we propose BINENHANCE, a novel framework designed to leverage the inter-function semantics to enhance the expression of internal code semantics for binary code search. Specifically, BINENHANCE constructs an External Environment Semantic Graph (EESG), which establishes a stable and analogous external environment for homologous functions by using different inter-function semantic relations (*e.g.*, *call*, *location*, *data-co-use*). After the construction of EESG, we utilize the embeddings generated by existing internal code semantic models to initialize EESG nodes. Finally, we design a Semantic Enhancement Model (SEM) that uses Relational Graph Convolutional Networks (RGCNs) and a residual block to learn valuable external semantics on the EESG for generating the enhanced semantics embedding. In addition, BINENHANCE utilizes data feature similarity to refine the cosine similarity of semantic embeddings. We conduct experiments under six different tasks (*e.g.*, under *function inlining* scenario) and the results illustrate the performance and robustness of BINENHANCE. The application of BINENHANCE to HermesSim, Asm2vec, TREX, Gemini, and Asteria on two public datasets results in an improvement of Mean Average Precision (MAP) from 53.6% to 69.7%. Moreover, the efficiency increases fourfold.

I. INTRODUCTION

Software development often reuses open-source code to reduce costs. However, this trend inadvertently propagates vulnerabilities of open-source code into billions of software systems [2] [14], including industrial software and firmware. The extensive workload of code auditing and the complexity of recursive code reuse results in substantial delays in vulnerability patching, with some systems experiencing an average delay of 352 days [27]. In 2023, a review [20] conducted by Synopsys on 1703 software projects exposed a notable problem: 96% of these projects utilize open-source code, and 84% of them have at least one known vulnerability. In reaction to this widespread security risk, the emergence of binary code search has been proven to be a powerful method for automating the detection of insecure software components [43]. This facilitates the prompt patching of identified vulnerabilities.

Binary code search entails the meticulous analysis of numerous binary codes to identify the most similar ones. Hence, its applications span a wide range, including software reuse detection [21, 42, 51], vulnerability search [33, 40, 43, 46], firmware security analysis [59, 63], and patch presence testing [31, 36, 57]. Developing a general and effective binary code search solution is extremely challenging. The syntactic structure of binary code can vary dramatically due to different compiler settings, such as optimization options. Additionally, binary codes with similar syntactic structures may have different semantics [30]. Therefore, a comprehensive understanding of the semantics of binary code is key to addressing this task. With the rapid development of deep learning technology, current solutions for binary code search are to convert binary code into embeddings and obtain the similarity of binary code by calculating the similarity between embeddings. Based on the definition of Section II-A, we classify them into two categories: **internal code semantic models** and **external environment semantic models**.

Current solutions predominantly focus on learning the internal code features of functions. Gemini [56] and Asm2vec [26] employ neural networks to perform semantic encoding within program structures, specifically control flow graphs. Similarly, TREX [44], Palmtree [39], and others [53] [60] utilize Transformer [52] models for assembly code representation. Other studies, such as HermesSim [30], Asteria [59], VulHawk [43] and XLIR [28], utilize deep neural networks to convert intermediate representations, such as pseudocode and toy IR, into embeddings. Furthermore, recent so-

lutions integrate function call relationships with internal code features. αdiff [41] and BMM [29] utilize function call graph to aid in calculating similarity. Despite notable performance improvements in binary code search, these solutions have exhibited some limitations (example in Section II-B), particularly as the number of binary codes requiring comparison increases.

Firstly, the internal code semantics of functions may exhibit substantial variations due to different compilation settings, encompassing factors like function inlining and splitting (identified as **P1**). Function inlining, an optimization strategy during compilation, introduces the actual codes of other functions called by the function at the corresponding address (as shown in Figure 2.(a)). Function splitting, conversely, divides a single function into multiple sub-functions. Study [32] demonstrates that compiler-caused function inlining can reach up to 70%, leading to a significant decrease in the performance of binary code search. Moreover, the selective inlining method [23] is lack of flexibility. This limitation underscores the inadequacy of relying solely on internal code semantics while overlooking the significance of external environment semantics.

Secondly, exclusive reliance on function call graphs (CG) for assistance is insufficient for addressing complex real-world scenarios (**P2**). Relying solely on the external environment semantics of function calls introduces two primary issues that impact the effectiveness of binary code search: missing calls and similar calls. Missing calls occur when function call dependencies are absent, potentially eliminated during code reuse or optimization processes, such as strategies like function inlining. Similar calls, meanwhile, involve similar function call relationships between non-homologous functions, leading to imprecise similarity assessments and false positives.

Thirdly, current solutions exhibit limited scalability and struggle to cope with large-scale function search tasks (**P3**). The most recent Transformer-based methods, such as TREX [44], typically generate function embeddings with a dimension of 768. However, real-world scenarios often involve millions of functions, necessitating rapid comparison speeds. Our experiments (in Section IV-F) revealed that in a pool of 10,000 functions, the time required to recall all the homologous functions of a binary file containing around 2500 functions is around 22 minutes using TREX's 768-dimension embeddings. The time increases exponentially with the growth of the function pool and the binary file size. In contrast, 128-dimension embeddings only require 5 minutes, emphasizing the efficiency challenges in large-scale function search. Moreover, simply reducing the dimensionality for retraining usually leads to substantial performance loss [39].

Our central insight focuses on comprehensively utilizing external environment semantics (definition in Section II-A) to enhance the internal code semantic models. To achieve it, several challenges must be addressed. First, it is crucial to carefully select environment features that can serve external environment semantics for a function within a complex environment. These selected features should positively contribute to the performance of binary code search tasks and exhibit robustness. Second, the integration of various external environment features, each representing distinct structures and semantic spaces, presents a challenge. Determining an effective method for combining these diverse features to improve search tasks is crucial. Finally, an approach needs to be devised to
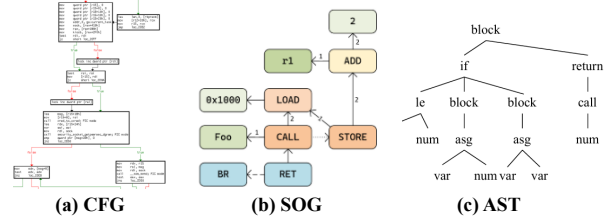

Fig. 1: Examples of Internal Code Feature

appropriately apply the external environment semantics of the function, significantly enhancing its internal code semantics.

In this paper, we introduce BINENHANCE, a framework based on external environment semantics, specifically designed to tackle the aforementioned challenges. BINENHANCE differentiates itself from prior approaches by not solely relying on function call graph (**P2**). Instead, we have designed a novel External Environment Semantic Graph (EESG), which constructs four edges (CD, DCU, AA, and SU) among functions to model a function's external environment since these semantic edges are robust among different scenarios when they coexist. For **P3**, we employ whitening transformations [49] to reduce the dimension of node initial embeddings generated by existing internal code semantics models [30] [59] [56] [44] [26] and MPNET [47], improving search task efficiency. To address **P1**, after initializing the nodes of EESG, BINENHANCE proposes a Semantic Enhancement Model (SEM), which introduces external environment semantics into internal code semantic embedding. SEM utilizes Relational Graph Convolutional Networks (RGCNs) [45] for updating external environment semantic embeddings in EESG and merges the dual embeddings of the function via a residual block to obtain an enhanced semantic embedding. Lastly, BINENHANCE includes a similarity combination module, leveraging data feature similarity to fine-tune semantic embedding similarity. In summary, we have made the following contributions:

- **Perspective.** This paper presents a novel perspective on binary code search by incorporating external environment semantics into internal embeddings. To the best of our knowledge, it is also the first work to demonstrate that this integration can significantly improve the identification of homologous functions. Our approach enriches the semantics of internal embeddings and addresses the problem of false filtering of those approaches based on function calls. Additionally, it can be easily extended to other internal code semantic methods. Moreover, this paper demonstrates the promising potential of designing new external features to enhance internal representations, rather than solely focusing on developing new features for internal code.
- **Technique.** We implement BINENHANCE[1], a general framework that proposes four novel external semantic edges of our EESG to model stable external contexts in complex external environments and introduce an SEM to incorporate external semantics into internal embeddings to improve its representation ability.
- **Study.** We conduct comprehensive experiments to evaluate the effectiveness of the BINENHANCE, including dif-

---

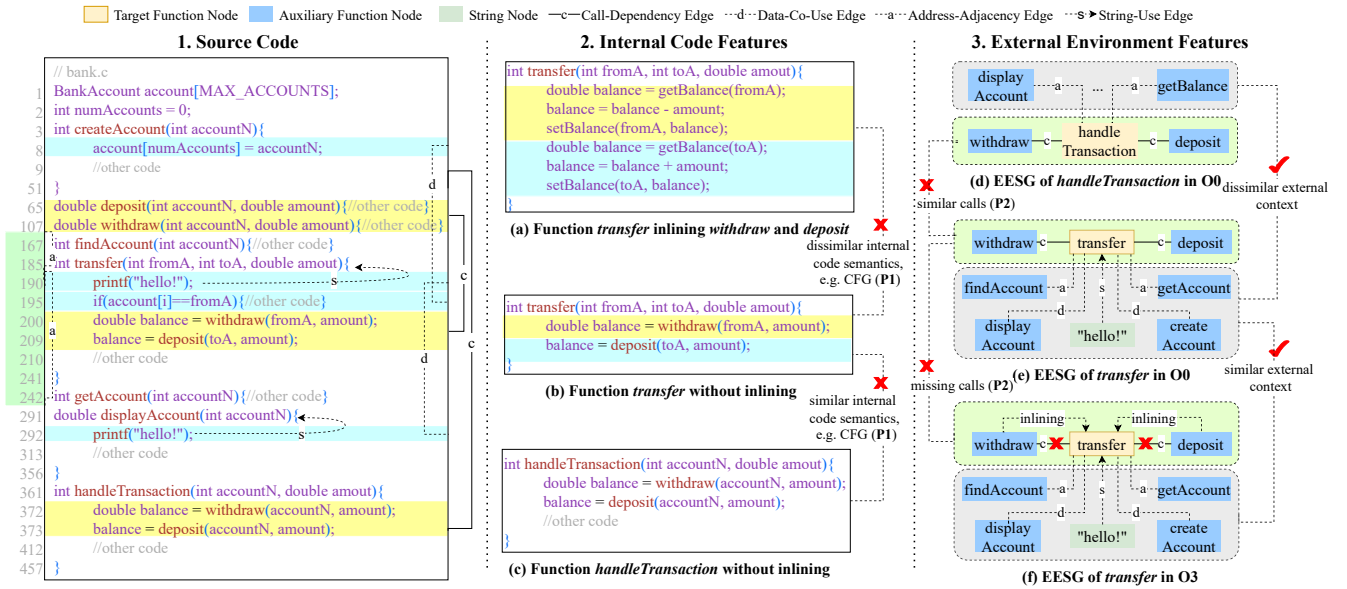[1]https://github.com/wang-yongpan/BinEnhance

Fig. 2: A Motivation Example of external environment semantics.

ferent function pool sizes, cross-architectures, and cross-optimization options, as well as scenarios such as function inlining and real-world vulnerability detection. Our results show that all existing binary code search methods overlook the importance of external environment semantics, leading to suboptimal performance.

II. PRELIMINARIES

A. Definition

In this section, we introduce the terms applied in this paper. In the field of binary code search, **homologous functions** refer to the functions compiled from the same source code with various compilation settings. The set of functions with homologous functions and non-homologous functions is termed a **function pool**. Furthermore, this paper categorizes binary code search methods into two types: **Internal Code Semantic model** and **External Environment Semantic model**.

- **Internal Code Semantics.** It focuses on the target function itself and is derived from both the binary code embedded within the target function or from its derivatives, such as Control Flow Graphs (CFG) and Data Flow Graphs (DFG), which are constructed through code analysis of the binary function code. Figure 1 shows some examples of internal code features to represent internal code semantics.
- **External Environment Semantics.** It focuses on inter-function and is inferred from other supplementary functions (in the code segment) or any data (in the data segment) pertaining to the target function within the same binary, such as the function call relationships. In this paper, as shown in Figure 2, we present Call-Dependency, Data-Co-Use, Address-Adjacency, and String-Use as novel external environment semantics in our work (in Section II-B and III-C2).

B. Motivation Example

We elucidate our motivation with an example depicted in Figure 2. This figure showcases how the external semantic exhibits superior performance than solely internal semantics in binary code similarity detection tasks. It indicates the variance compiled from the same source code (**bank.c**) under different compilation settings (**O0** and **O3**).

- **False Negative.** Figure 2.(a) and Figure 2.(b) illustrate the potential for false negative search results caused by function inlining during compilation. In the figure, the **transfer** function compiled under **O3** incorporates the **withdraw** and **deposit** functions, through inlining, significantly altering its internal code semantics (**P1**). In contrast, Figure 2.(b) shows the result of the code compiled under **O0** setting. There is no aforementioned function inlining, the dissimilar internal code semantics indicating that relying solely on it would fail to accurately match **transfer** function compiled under **O3** with its version compiled under **O0** setting. Additionally, as depicted in Figure 2.(e) and Figure 2.(f), they highlight the problem (**missing calls in P2**), where **transfer** loses call dependency due to function inlining, thus undermining the reliability of call-based approaches.
- **False Positive.** Figures 2.(d) and 2.(e) demonstrate the potential for false positive search results induced by similar function structures. In these two figures, the **handleTransaction** and **transfer** functions, both highlighted in green, call **withdraw** and **deposit** functions, but they are not homologous functions. This scenario exemplifies a potential challenge (**similar calls in P2**), where solely using call dependencies might erroneously associate **transfer** under **O0** with **handleTransaction** under **O0** as homologous functions, impacting the accuracy of internal code semantics. Moreover, Figure 2.(b) and Figure 2.(c) indicates the similar internal code semantics of the **handleTransaction** and **transfer** functions, resulting in false positive.

To tackle these problems, we designed EESG to establish the external environment of the function. EESG is characterized by four external semantic edges and two types of nodes (in Section III-C2). It ensures that even after compilation optimizations like function inlining, the EESG of *transfer* remains largely unchanged (shown in grey in Figure 2.(e) and 2.(f)) and significantly different from the EESG of ***handleTransaction*** (shown in grey in Figure 2.(d) and 2.(e)). This stability is crucial for effectively recalling homologous functions while minimizing interference from non-homologous ones.

III. DESIGN OF BINENHANCE

A. Overview

BINENHANCE seeks to utilize external function context and data features to assist in identifying homologous functions, especially when target functions lack clearly distinguishing internal code features. To achieve this, we propose a robust external environment to aid in the identification of homologous functions. The enhancement provided by the designed external environment is pivotal for enhancing the recall performance of homologous functions.

Figure 3 illustrates the workflow of BINENHANCE, comprising three stages: *Node Initial Embedding Generation*, *Function Embedding Enhancement*, and *Similarity Combination*. In the Node Initial Embedding Generation Stage, BINENHANCE employs the internal code semantic models (*e.g.*, HermesSim [30]) to generate the initial embeddings of functions, which are utilized for initializing nodes in the second stage. Specifically, we utilize official code to extract the internal code features and train the embedding model for generating all the function embeddings. In addition, we extract the readable data features from the data segments of binaries to refine the similarity score post the second stage and utilize MPNET [47] to encode strings. In the Function Embedding Enhancement stage, BINENHANCE constructs an External Environment Semantic Graph (EESG) acting as a supplementary source for the internal code semantics of functions. Subsequently, the embedding of string and function nodes are harmonized using Whitening Transformation [49] to ensure the uniformity of embedding dimension in the encoded EESG. Finally, the encoded EESGs of the target functions are updated by SEM (Semantic Enhancement Model), which consists of multi-layer RGCN [45] and a residual block, to enhance their nodes embedding. In the Similarity Combination stage, BINENHANCE computes the Jaccard similarity of the data features and utilizes this score to adjust the cosine similarity output from stage two. This computation is implemented through a feed-forward neural network. The resulting similarity score is then considered as the final measure of similarity between the two input functions.

B. Node Initial Embedding Generation

In this section, we introduce BINENHANCE how to utilize the function embeddings generated by HermesSim and other internal code semantic models, as well as how to initialize the nodes embedding of EESG.

Section III-C2 introduces the two types of nodes in the EESG: function nodes and string nodes, necessitating distinct methods for embedding generation. We observe that existing internal code semantic models, such as HermesSim [30], are well-suited for function node embeddings. Additionally, HermesSim can be seamlessly replaced with other more advanced models as needed. In this paper, we select five existing internal code semantic models to generate function embeddings and respectively enhance their performance using BINENHANCE (in Section IV-A3). Specifically, we utilize the official code provided by each model to extract the required internal features, such as the control flow graph (CFG) by Gemini [56], the abstract syntax tree (AST) used by Asteria [59], and the semantic-oriented graph (SOG) used by HermesSim [30], and so on. Following the extraction process, we retrain the models using the official training codes and hyperparameters to prevent performance discrepancies arising from varying dataset distributions. Subsequently, we employ the retrained internal embedding models to generate function embeddings within the target binary.

For string nodes, we utilize the widely recognized MP-NET [47], a sentence-transformer and pre-trained English language model, to transform English strings into embeddings, ensuring rich linguistic representation. In addition, based on the existing research [61], there may be some consistent data features within the $.rdata, .data, .bss, .idata$, and other data segments of binary files. In binary files readable strings predominantly reside in these segments or symbol tables, and those residing in the symbol table are often stripped, thus prioritizing data segment strings for stability. Global data, such as global constant arrays and integer constants, are shared across multiple functions. Like OSLDetector [62], we discover that most strings with less than 5 (e.g., $time$) are more likely to be reused, with lower specificity, and thus we filter them. B2Sfinder [61] demonstrates that these features are helpful and typically not stripped off. We extract these data features for the third stage. While extracting each data feature, we concurrently save a list of addresses for all functions that use this data within the code segment. This facilitates the subsequent construction of the EESG (in Section III-C2).

C. Function Embedding Enhancement

In this section, we introduce the details construction process of EESG and how to use SEM to learn the external semantics of EESG.

BINENHANCE employs EESG to model the semantic information of the functions' external contexts. This is in contrast to previous approaches that relies solely on function call relationships within the function's call graph (CG). After generating the initial embeddings of the two types of nodes on EESG, two problems arise from different initialization methods. Firstly, there's a discrepancy in the embedding dimensions between string nodes and function nodes (the former is fixed at 768 dimensions by MPNET, while the latter varies depending on the internal code semantic models applied). Secondly, high-dimensional embeddings, such as 768 dimensions, impose significant computational and memory burdens, especially in extensive binary function search tasks.

1) Whitening Transformation: To address these problems, we apply the whitening transformation [49] for dimensionality reduction, enabling two nodes to encode into a unified vector space. In Algorithm 1, we elaborate on the details of applying
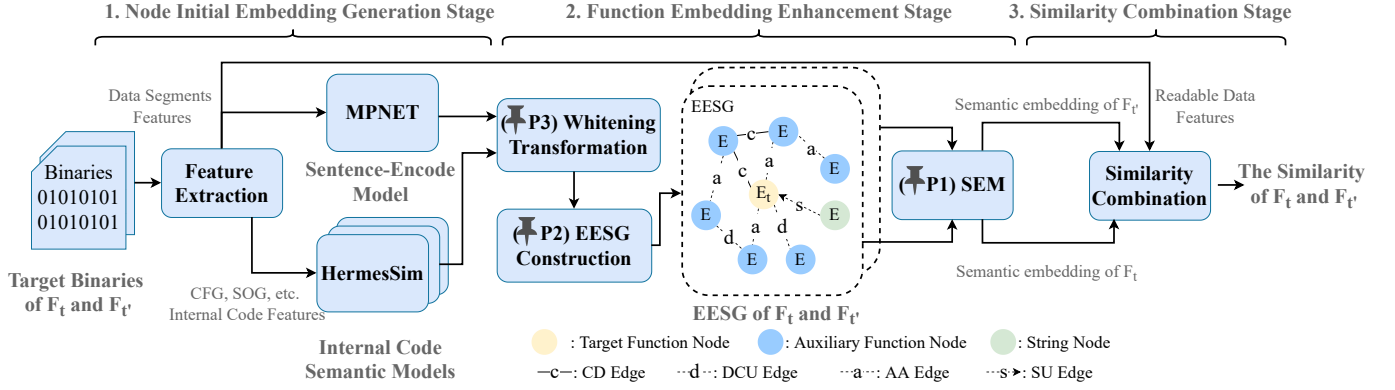
Fig. 3: Workflow of BINENHANCE. BINENHANCE aims to calculate the similarity of functions F_t and $F_{t'}$.

Algorithm 1: Whitening Transformation Method

Input: Node Embeddings X, Target Dimension d_t
Output: Whitening Embeddings E

1 **Function** *Whitening_Transformation(X, d_t)*:
2    $\mu = \frac{1}{n}\sum_{i=1}^{n} X_i$;
3    $Cov = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \mu)(X_i - \mu)^T$;
4    $V, D, V^T = SVD(Cov)$;
5    $W = VD^{-\frac{1}{2}}V^T[:,:d_t]$;
6   **for** $i = 1$ *to* n **do**
7      $E_i = (X_i - \mu)W$;
8      $E_i = \frac{E_i}{\sqrt{\sum_{j=1}^{d_t} E_{i,j}^2}}$;
9   return E;
10 **end**

the whitening transformation method for linear dimensionality reduction. Specifically, the input $X \in \mathbb{R}^{n \times d}$ represents the node embedding matrix of the EESG, where n is the total number of nodes and d is the initial dimension of embeddings. The algorithm's output, $E \in \mathbb{R}^{n \times d_t}$, is the transformed node embedding after whitening, where d_t signifies the reduced dimension of nodes. The algorithm initiates by calculating the mean and covariance matrix of X (lines 2 and 3). Subsequently, an eigenvalue decomposition of the covariance matrix is executed using Singular Value Decomposition (SVD) (line 4). Here, D is defined as a diagonal matrix containing the eigenvalues, and V is an orthogonal matrix with its columns representing the corresponding eigenvectors. The whitening transformation is then facilitated through the construction of a whitening matrix W, derived from V and D (line 5). This matrix is applied to the initial embeddings to yield the whitening embeddings (line 7). In the final step, normalization of the whitening embeddings is achieved by first calculating the $L2$ norm for each embedding E_i, followed by division of each embedding by its respective $L2$ norm to complete normalization. This results in a set of standardized, dimensionally-reduced node embeddings suitable for further semantic learning.

2) Details of EESG and its Construction: EESG construction focuses on constructing knowledge from two distinct perspectives: nodes and edges. Specifically, EESG is represented as a directed graph, denoted as $G = (N, E, N_e, E_t)$, where N represents the set of different types of nodes, $E \subseteq N \times N$ is the set of edges, N_e is the set of node embeddings, and E_t is the set of edge types.

Nodes. In the EESG, each node (N_i) represents one of two distinct entities: ❶ **Functions within code segments** and ❷ **Readable strings within data segments**. Function nodes are further categorized into two types: the target function (F_t) and the auxiliary function (F_a). The auxiliary function F_a refers to all functions that have a direct or indirect semantic relationship with the target function. The purpose is to construct complete semantic information about the external environment of the target function (F_t). Readable string nodes, extracted from the data segment, represent strings used in function nodes. After the whitening transformation, the embedding dimensions of the two types of nodes become consistent, preparing them for subsequent SEM to update the embedding of nodes.

Edges. Edges express not only the semantic information of target and auxiliary nodes but also the function nodes and readable string nodes. Building upon existing research [35], we categorize edges within the EESG into the following four distinct types. ❶ **Call-Dependency** (CD) **edge**: Within the x86 instruction set, a function F_i calling another function F_j via the $CALL$ instruction denotes a CD edge. For instance, considering a compiler adopting function inlining strategy, a function F_j is inlined into another function F_i, resulting in a modified function F_i' of F_i. From the view of static analysis, F_i' exhibits substantial deviations in semantic content compared to the original F_i. To mitigate this, establishing an edge between F_i and F_j is proposed. This connection allows F_i to supplement and access F_j's semantic content, thereby aligning F_i more closely with the modified semantic context of F_i'. Nevertheless, relying exclusively on CD edges presents certain constraints (in Section I). To address these, three additional types of edges are introduced below, designed to enrich the semantic representation and alleviate the impact introduced by optimization strategies, including inlining. ❷ **Data-Co-Use** (DCU) **edge**: Binary files typically contain data in segments like *.data*, *.bss*, *.rodata*, and *.idata*. We focus on constant readable strings and global data within these seg-

5

ments. We create bidirectional DCU edges of string or global data by linking two functions using the same constant readable string or global data. These edges introduce data semantics into function semantics and counteract false positives due to structural similarities introduced by CD edges in the same call graph. Notably, even when the symbol names of global data are stripped, we can establish these edges based on data addresses. ❸ **Address-Adjacency (AA) edge**: Inspired by BinaryAI [34] and LibDX [50], we observed that after the compiler compiles all source functions within the same source file, the order of function addresses in the binary file may differ from that in the source code due to different compilation settings. However, the relative locality of functions in the code segment remains unchanged. Therefore, we establish a directional AA edge based on the starting address and size of each function. Specifically, when the address of function F_i exceeds that of F_j, an AA edge is created from F_i to F_j. This edge signifies that F_i precedes F_j in the address sequence. Conversely, an edge from F_j to F_i indicates that F_i follows F_j. This methodology is particularly effective in capturing the address semantics of functions, a critical aspect of open-source code reuse. Open-source functions are predominantly reused as modular units and are stored adjacently [50, 58]. This adjacency is preserved during compilation, allowing our AA edges to represent the address semantics between functions. Additionally, this approach addresses the challenge of enhancing semantic representation in scenarios where the above CD and DCU edges are insufficient or absent. ❹ **String-Use (SU) edge**: Prior research [56] indicates the stability of strings used in functions across different compilation environments. Therefore, we establish edges from readable string nodes to function nodes, aiming to augment the ability of a function node to identify homologous nodes based on the semantic information in readable strings.

EESG Construction Algorithm. EESG construction initiates from the function node F_t and is gradually completed recursively by obtaining the auxiliary function nodes F_a based on four types of edges. As detailed in Algorithm 2, the process begins with the initialization of the EESG, denoted as $eesg$. Because of the existing multiple identical edges between two nodes (*e.g.*, multiple calls), this step is followed by creating two lists: ns for minimizing redundant nodes, and es for eliminating duplicate edges. Considering that the auxiliary functions staying far away from the function node F_t may have a weaker effect on its semantic enhancement, we set a max depth for recursive execution and to limit the size of $eesg$, marked as md. Empirical evidence from our experiments suggests that a maximum depth of 4 strikes an optimal balance. Algorithm 2 systematically constructs the EESG, layer by layer, commencing with the function node F_t. It identifies non-repetitive function nodes in successive layers, represented as nes for the next layer and nss for the current one. By traversing each node in nss, we ascertain the subsequent layer's function nodes, nes, through three edge types: call-dependency (cd), data-co-use (dcu), and address-adjacency (aa). The algorithm establishes bi-directional edges between function nodes, with 0/1 indicating call/be-called dependency edges, 2/3 for pre/post address adjacency edges, and 4 for data-co-use edges. Furthermore, the algorithm constructs string nodes and the edges linking these nodes to their corresponding function nodes. Based on the string content su utilized by the

---

Algorithm 2: EESG Construction Algorithm

Input: Feature Sets s_e, Function Sets s_u, Target Function f_t and Max Depth md
Output: the EESG of f_t

1 $eesg, ns, es \leftarrow NULL$;
2 **Function** *EESG_Construct(s_e, s_u, f_t, md)*:
3      $nss \leftarrow f_t$;
4      $eesg.addNode(f_t, type = function)$;
5      $ns.add(f_t)$;
6   **for** $d = 0$ *to* md **do**
7          $nes \leftarrow NULL$;
8   **for** $n \in nss$ **do**
9              $cd, dcu, aa, su \leftarrow \text{GetExternal}(s_u, s_e, n)$;
10             $nes.add(BuildEdges(n, cd, 0, 1, function))$;
11             $nes.add(BuildEdges(n, aa, 2, 3, function))$;
12             $nes.add(BuildEdges(n, dcu, 4, 4, function))$;
13             $BuildBiEdges(n, su, 5, 5, string)$;
14         $nss \leftarrow nes$
15   **return** $eesg$;
16 **end**

17 **Function** *BuildEdges(src, des_f, r_1, r_2, $node\_type$)*:
18     $n \leftarrow NULL$;
19   **for** $des \in des_f$ **do**
20         $e \leftarrow (src, des, type = r_1)$;
21   **if** $des \notin ns$ **then**
22             $essg.addNode(des, type = node\_type)$;
23             $n.add(des)$;
24             $ns.add(des)$;
25   **if** $e \notin es$ **then**
26   **if** $node\_type == function$ **then**
27                 $eesg.addEdge(src, des, type = r_1)$;
28                 $es.add(src, des, type = r_1)$;
29             $eesg.addEdge(des, src, type = r_2)$;
30             $es.add(des, src, type = r_2)$;
31   **return** n;
32 **end**

---

function node, the edges are categorized as type 5. The process culminates with the algorithm returning the $eesg$ of the F_t. The function $GetExternal$ at line 9 retrieves all function nodes and string nodes connected with the function node n through the four types of edges.

3) Semantic Enhancement Model: After the construction of the EESG for the target function F_t, we observe that the inherent ability of graph neural networks (GNNs) [45] to aggregate neighboring node information for updating target node embedding aligns well with the requirement to learn the semantic knowledge of the external environment of F_t node. Therefore, we designed the SEM model based on GNN to enhance the representation of function nodes, capable of learning the structure of the EESG and incorporating both external and internal semantics.

Figure 4 depicts the structure of SEM. After encoding the EESG, we select a GNN to integrate the embedding of neighboring nodes relevant to the F_t. In particular, we select Relational Graph Convolutional Neural Network (RGCN) [45]. Notably, while utilizing a more advanced GNN could enhence
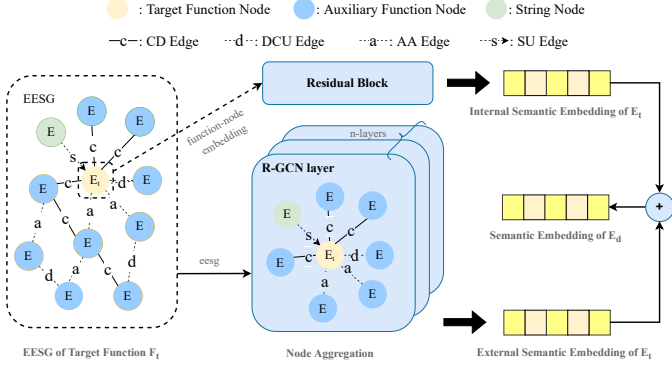
Fig. 4: Structure of SEM

SEM's performance, it lies outside of this study's primary focus. Compared with the GCN [38], the RGCN contemplates node interrelations and adeptly addresses different edge relationship impacts within the graph structure, aligning seamlessly with EESG's graph structure.

$$E_i^{(l+1)} = LeakyRelu(\sum_{r \in R} \sum_{j \in N_i^r} \frac{1}{c_{i,r}} W_r^{(l)} h_j^{(l)}) \quad (1)$$

$$E_t = RGCN(F_t) + Resisual\_block(HermesSim(F_t)) \quad (2)$$

For the edges of EESG, we set different trainable parameters for different edge types by the Pytorch [19] framework based on the RGCN model. It guides function nodes to learn unique semantic information based on edge types. Specifically, for each function node F_i, we update its function embedding following Equation 1 (from the RGCN model), where N_i^r denotes the set of neighbor nodes related to F_i through relation r, while $C_{i,r}$ represents a normalization factor set to the cardinality of N_i^r. $W_r^{(l)}$ denotes the linear transformation function, applied uniformly across neighbor nodes connected by identical edge types, with the count of $W_r^{(l)}$ corresponding to the number of EESG's edge types. The layer count, l, indicates the layer number of the RGCN network. SEM's structure facilitates the embedding amalgamation of all neighboring nodes for a given function node at each layer through adjustable parameters, continuously refining its embeddings. As the number of neural network layers increases, each target function node incrementally learns the semantic information from increasingly distant auxiliary function nodes. Since we adopt four levels of EESG in depth (in Section III-C2), consistently, a four-layer neural network structure for SEM is applied to learn the semantics of all the auxiliary function nodes of the target node on EESG. After these neural network layers are updated, each function node's embedding is encoded with the semantics of its comprehensive external environment. Thus, F_t learns its external semantic embedding. To better integrate the internal semantics of the function with its external semantics and prevent it from forgetting its internal semantics, SEM introduces a residual block with trainable parameters. This residual block transforms the initial embedding of F_t into an internal semantic embedding with a dimension consistent with the dimension of the external semantic embedding by the linear layer of pytorch. Finally, the SEM integrates two embeddings, internal semantic embedding, and external semantic embedding, to obtain the final semantic embedding representation of the target function F_t, as shown

in Equation 2.

D. Similarity Combination

In this section, we describe how to combine data feature similarity with the cosine similarity of embeddings and outline the training and inference process for the entire framework.

1) Jaccard similarity: We combine the similarities of data features between homologous functions with the cosine similarity of semantic embeddings by a feed-forward neural network (FFN) to obtain the final similarity. While prior studies [61] have utilized these data features to identify binary functions, their efficacy in recognizing homologous functions is limited. Firstly, not all functions contain such data features, as evidenced by the study [27]. Secondly, as discussed in Section I, due to the existence of optimization strategies such as function inlining, which cannot be resolved by data features. Therefore, These features serve as an auxiliary tool to fine-tune the calibration of semantic embedding similarity, rather than being the sole basis of function recall.

We employ the Jaccard similarity to calculate the similarity of data between two functions. In particular, this similarity is computed as $Sim = \frac{|F_m \cap F_n|}{|F_m \cup F_n|}$, where $\cap / \cup$ represents the intersection/union of the features of the functions F_m and F_n. Next, we compute the cosine similarity (Sim_{cos}) between the semantic embedding representations (E_d and $E_{d'}$) of the two functions, as derived from the SEM. As shown in Equation 3, this cosine similarity is then concatenated with the data similarities to construct a similarity vector. Finally, an FFN is utilized to predict the ultimate similarity between the two functions by the similarity vector. It's noteworthy that even in the absence of data features for similarity combination, our SEM (without data features) demonstrates commendable performance in semantic embedding, as evidenced by the results in the ablation study (in Section IV-E).

$$Sim = Tanh(FFN(Concat(Sim_{cos}, Sim_{data}))) \quad (3)$$

2) Framework Training and Inference: As shown in Equation 3, the final similarity score generated by the framework is derived from the $Tanh$ activation function, yielding values within the range of $[-1, 1]$. Therefore, we employ the Mean Squared Error (MSE) as our loss function, as delineated in Equation 4, where m denotes the batch size during the training process, while Sim_p and Sim_n represent the similarity scores of homologous and non-homologous functions to the target function, respectively. The target similarity for homologous functions is set to 1, and conversely, -1 for non-homologous functions. Furthermore, we utilize the Adam optimizer to train the parameters, aiming to minimize the loss. Meanwhile, in the BINENHANCE framework, only the parameters of the SEM and FFN in the Similarity Combination participate in the end-to-end training of the framework. Other modules, such as HermesSim and MPNET, are only utilized for node initial embedding generation.

$$Loss = \frac{1}{m} \sum_m \frac{1}{2}(1 - Sim_p)^2 + \frac{1}{2}(1 + Sim_n)^2 \quad (4)$$

When calculating the similarity between functions, we follow the common practice in most binary code search methods

by first extracting features and constructing the function's EESG. We then execute existing internal code semantics models and MPNET, followed by a whitening transformation to complete the node embedding assignment for the EESG. After updating the nodes with SEM, we combine the data feature similarities with embedding cosine similarity to obtain the final function similarity.

IV. EVALUATION

In this section, we apply BINENHANCE to the existing state-of-the-art binary code search methods for the performance evaluation of multiple related tasks. In the experiments, we aim to answer the following research questions: **RQ1**: How much could be improved when BINENHANCE is applied to baselines (including HermesSim [30], Asteria [59], Asm2vec [26], TREX [44], and Gemini [56])? **RQ2**: Is BINENHANCE robust against different compiler optimization options and architectures? **RQ3**: Does BINENHANCE effectively alleviate the impact of function inlining in binary code search? **RQ4**: What is the contribution of each part in BINENHANCE? **RQ5**: Does BINENHANCE improve the efficiency of baselines? **RQ6**: What is the performance of BINENHANCE in detecting 1-day vulnerabilities in real-world firmware?

A. Implementation and Experiment Setup

We use IDA Pro 7.5 [18] for disassembling binary files and Python 3.8 for programming BINENHANCE with libraries, such as DGL [16], and Pytorch [19]. Our evaluation is conducted on an Ubuntu 22.04 with an Intel Xeon 128 core 3.0GHz CPU, 1TB memory, and 2 Nvidia V100 32GB GPUs.

1) Datasets: In the experiments, as shown in Table I, we select three public datasets [37, 59] and collect a firmware dataset. We divide the public datasets into a training set, validation set, and testing set in an 8:1:1 ratio based on the granularity of the project. The test data differs from the training and validation data to make the evaluation result reliable.

- **Dataset** $D1$. The dataset is from $Asteria$ [59], including three architectures: ARM, $X86$, and $X64$ with four levels of optimizations: $O0$, $O1$, $O2$, and $O3$. In total, this dataset contains $2,751,667$ functions. It includes widely utilized tool libraries like $binutils$ and spans various categories, such as communication (*e.g.*, $openssl$, acl), compression (*e.g.*, $p7zip$, $szip$), file operation (*e.g.*, $file$, $json$) and others, covering 260 distinct projects.
- **Dataset** $D2\_norm$. The dataset is from $BinKit$ [37], $D2\_norm$ spans three architectures (ARM, $MIPS$, $X86$) and four optimization levels ($O0$, $O1$, $O2$, $O3$), featuring a total of $1,654,805$ functions from 51 projects like $coreutils$ and gnu. $D2\_norm$ uses standard compilation.
- **Dataset** $D2\_noinline$. $D2\_noinline$ and $D2\_norm$ contain identical software projects. The key distinction lies in their compilation process: $D2\_noinline$ prohibited function inlining during compilation. Therefore, $D2\_noinline$ contains $1,991,864$ functions, exceeding the size of $D2\_norm$.
- **Dataset** $D3\_firmware$. It consists of 37 firmware images from 8 vendors (*e.g.*, $DLINK$ and $ASUS$) that span a variety of device categories such as cameras and routers. We use $Binwalk$ [1] to parse the images and extract the $6,817$ binaries from them. We collect them to assess the ability of BINENHANCE to detect 1-day real-world vulnerabilities.

2) Evaluation Metrics: In practice, the binary code search tasks are often accompanied by a big function pool where most are non-homologous functions. Therefore, we select Mean Average Precision (MAP) as the evaluation metric, as shown in Equation 5, where AP denotes the average precision of one prediction result, and k is the ranking position of the prediction functions. $P(k)$ is the proportion of homologous functions among the top-k prediction functions. $sim(k)$ indicates whether the k_{th} prediction is a homologous function or not, specifically 1 for yes, and 0 for no. N indicates the total number of homologous functions in this test data. MAP represents the mean value of the average precision of the multiple prediction results. q is the serial number of the prediction and Q is the total number of predictions. $AP(q)$ is the average precision of the $q_t h$ predicted result.

$$AP = \frac{\sum_{k=1}^{N}(P(k) \times sim(k))}{N}, \ MAP = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \tag{5}$$

3) Baselines: To demonstrate the effectiveness of BINENHANCE to existing methods, we select the following state-of-the-art (SOTA) methods of binary search from various perspectives as baselines based on their open-source availability and the requirement for generating function embeddings. For all the baselines, we utilize the official code and default parameters. Moreover, we retrain all the baselines for each dataset (except $D3\_firmware$) using the official training code to prevent performance discrepancies from varying dataset distributions.

- **Gemini [56].** This baseline utilizes Structure2vec [25] to learn ACFG and encode function embeddings.
- **Asm2vec [26].** This baseline uses the PV-DM model to learn instruction sequence embedding representations.
- **Asteria [59].** This baseline utilizes the Tree-LSTM to learn ASTs and obtain embedding representations.
- **TREX [44].** This baseline inputs the sequence generated by the micro-trace into the transformer model for training.
- **HermesSim [30].** This baseline utilizes Ghidra [17] to lift binary code to the toy IR and proposes a Semantic-Oriented Graph (SOG) to generate function embeddings.

For external semantic learning methods, we also evaluate BMM [29] that use function call relationships (in Section IV-E). We additionally evaluate the influence of varying dimensions on function embedding, aiming to identify the suitable dimension. Ultimately, considering both the MAP score and computational time, we employ the whitening transformation module (in Section III-C1) to reduce the dimensions of the embedding derived from the baseline methods. Details of the dimension setting are shown in Table II.

B. RQ1: MAP Improvement Evaluation

In real-world binary code search tasks, the size of the function pool is often significantly larger than the pool size of the experiments. To evaluate the impact of increasing pool size, we set the function pool sizes to (2, 16, 32, 128, 512, 1024, 2048, 4096, 8192, 10000). Table III illustrates the MAP scores of five baseline methods and their enhancement via BINENHANCE across two distinct public datasets ($D1$, $D2\_norm$). As shown in Table III, the enhancement versions consistently outperform the baselines in all the function pool

TABLE I: The details of three public datasets and one firmware dataset

Dataset	architectures	options	Functions	Applications	Projects	Source
D1	ARM, X86, X64	O0-O3	2,751,667	RQ1, RQ2	260	Asteria
D2_norm	ARM, MIPS, X86	O0-O3	1,654,805	RQ1, RQ2, RQ3, RQ4, RQ5	51	BinKit
D2_noinline	ARM, MIPS, X86	O0-O3	1,991,864	RQ3	51	BinKit
D3_firmware	ARM, MIPS, X86	UNKNOWN	6,817(binaries)	RQ6	37	Real-world

TABLE II: Function Embedding dimensions

Model	Original model	+BINENHANCE
Gemini	64	64
Asm2vec	200	64
Asteria	150	128
TREX	768	128
HermesSim	384	128

Fig. 5: MAP scores in different pool sizes ($D1$)

Fig. 6: MAP scores in different pool sizes ($D2\_norm$)

sizes, with maximal MAP improvements of 27.4% and 28.3% on the respective datasets, and an average MAP improvement of 16.1% (from 53.6% to 69.7%). Notably, some methods after using BINENHANCE even surpassed originally more advanced methods. For instance, in $D1$ with the function pool size of 10000, Asm2vec's MAP score increases from 9.1% to 36.5%, overtaking TREX's 21.4%. It indicates that while the existing methods inherently possess the capability to capture some complex semantic information, they lack an efficient mechanism for integrating and utilizing this information. BI-NENHANCE addresses this gap by systematically curating and amalgamating that semantic information through EESG.

As illustrated in Figure 5 and 6, there is a notable decline in the MAP scores across all baselines as the size of the function pool expands. This trend highlights the increasing challenge of discerning and recalling homologous functions amidst a growing array of options. However, it's particularly noteworthy that the disparity in MAP scores between the BINENHANCE model (represented by a dashed line) and the original model (represented by a solid line) progressively widens. For instance, within a function pool of just two functions, the original Asteria model scores 94.5% in $D1$, which increases to 95.8% after BINENHANCE enhancement (an increase of 1.3%). This gap further escalates to 15% when the function pool increases to 10,000. This widening margin underscores BINENHANCE's superior stability and accuracy in recalling homologous functions from a large function pool, making it more practical. Thanks to the implementation of function embedding enhancement and similarity combination, BINENHANCE effectively captures and leverages the subtle semantic similarities between homologous functions, thereby achieving markedly improved performance.

> **Answer to RQ.1.** BINENHANCE demonstrates significant improvement across all the baselines, evidenced by the average increment in MAP scores on the two public datasets (16.1%, from 53.6% to 69.7%). Furthermore, its improvement for each baseline is positively correlated with the size of the function pool.
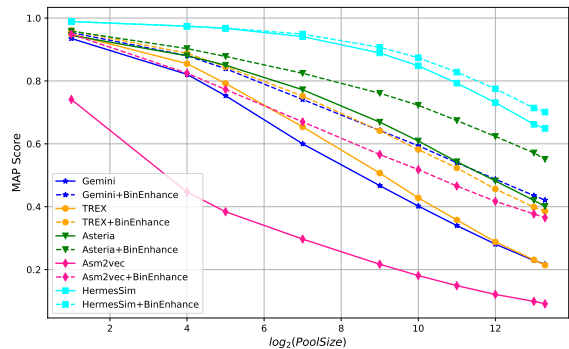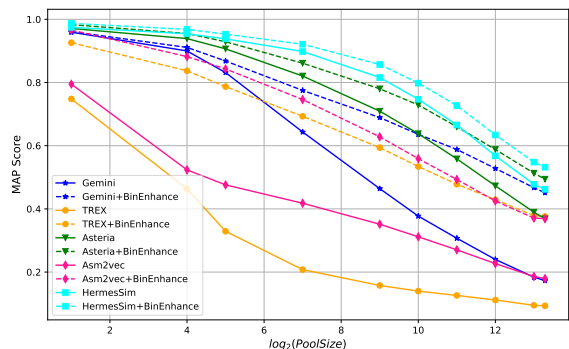
C. RQ2: Robustness Evaluation

In this section, we conduct extensive experiments to asses BINENHANCE's ability to improve baseline models under various architectures and compilation optimization options, aiming to verify its adaptability in real-world scenarios. Cross-architecture experiments are conducted on the $O2$ level compilation optimization option, and cross-optimization option experiments are conducted on $X86$ architecture. The results, detailed in Tables IV and V, show BINENHANCE's significant enhancements in cross-architecture and cross-optimization option experiments. Specifically, BINENHANCE, when applied in HermesSim, has the best results in cross-architecture and cross-optimization option tasks in all the public datasets. However, using the Transformer model without a specific cross-architecture design, TREX lagged behind Gemini in these tasks but mostly showed superior performance in cross-compilation optimizations, especially with BINENHANCE.

Figure 7 and 8 show that the improvement of the baseline models caused by BinEnhence is mostly stable across

TABLE III: MAP scores of different methods of binary code search tasks in different function pool sizes.

Models	D1						D2_norm					
	P=2	P=128	P=1024	P=4096	P=10000	Avg.	P=2	P=128	P=1024	P=4096	P=10000	Avg.
Gemini	93.5	60.0	40.2	28.1	21.7	48.7	95.9	64.3	37.7	24.0	17.3	47.8
Gemini+BINENHANCE	**95.4**	**74.1**	**59.4**	**48.7**	**42.1**	**63.9**	**96.1**	**77.5**	**63.6**	**52.8**	**45.1**	**67.0**
TREX	94.9	65.4	42.8	28.8	21.4	50.7	74.8	20.8	14.0	11.2	9.3	26.0
TREX+BINENHANCE	**96.0**	**75.2**	**58.2**	**45.6**	**38.6**	**62.7**	**92.6**	**69.3**	**53.4**	**42.9**	**37.6**	**59.2**
Asm2vec*	74.1	29.7	18.1	12.1	9.1	28.6	79.5	41.8	31.2	22.7	17.9	38.6
Asm2vec*+BINENHANCE	**94.9**	**67.0**	**51.8**	**41.7**	**36.5**	**58.4**	**96.6**	**74.6**	**55.9**	**42.5**	**36.9**	**61.3**
Asteria	94.5	77.2	60.9	48.2	40.1	64.2	97.1	82.1	63.8	47.3	36.9	65.4
Asteria+BINENHANCE	**95.8**	**82.4**	**72.3**	**62.3**	**55.1**	**73.6**	**98.3**	**86.1**	**72.9**	**58.8**	**49.4**	**73.1**
HermesSim	98.9	94.1	84.8	73.1	64.9	83.2	97.5	89.8	74.7	56.8	46.2	73.0
HermesSim+BINENHANCE	**98.9**	**94.9**	**87.4**	**77.5**	**70.1**	**85.8**	**98.7**	**92.1**	**79.8**	**63.4**	**53.2**	**77.4**

∗ Asm2vec only support X86 architecture.

TABLE IV: MAP scores of different methods in different optimization options and architectures (D1).

Models	Cross-architecture			Cross-optimization option					
	ARM,X86	ARM,X64	X64,X86	O0,O1	O0,O2	O0,O3	O1,O2	O1,O3	O2,O3
Gemini	33.6	32.9	52.2	35.5	30.6	25.2	53.3	42.0	54.0
Gemini+BINENHANCE	**58.2**	**63.0**	**76.0**	**58.3**	**51.4**	**42.1**	**68.6**	**58.5**	**68.0**
TREX	29.4	32.4	56.5	42.1	37.0	29.2	71.1	58.1	69.0
TREX+BINENHANCE	**66.3**	**68.3**	**83.9**	**67.1**	**61.0**	**55.2**	**84.2**	**74.3**	**74.8**
Asm2vec*	-	-	-	1.5	1.6	1.7	29.4	25.7	43.4
Asm2vec*+BINENHANCE	-	-	-	**43.4**	**42.3**	**35.1**	**68.4**	**52.3**	**64.6**
Asteria	68.2	60.2	69.3	58.9	49.4	42.0	57.4	45.9	57.7
Asteria+BINENHANCE	**79.4**	**79.4**	**81.2**	**73.2**	**61.2**	**56.5**	**77.9**	**63.4**	**71.6**
HermesSim	90.7	89.9	89.9	83.1	84.7	79.5	81.9	82.4	79.2
HermesSim+BINENHANCE	**96.3**	**96.2**	**97.4**	**91.5**	**91.6**	**87.1**	**96.2**	**91.9**	**91.6**

∗ Asm2vec do not support cross-architecture.

architectures and compilation optimization options (the interval between dashed and solid lines mostly does not vary significantly). In some tasks with MAP improvement relatively larger changes, there is no optimization strategy for the function in the O0 case, while there are some optimization strategies that change the structure such as CFG under the other optimization options, resulting in a very low score for the Asm2vec original model in the tasks (functions under O0 case recall functions under other options). BINENHANCE mitigates the effects of the optimization strategy, and the boosts are larger in the O0 case than other optimization options, so there are some fluctuations of Asm2vec. These results underscore BINENHANCE's robustness and consistency in diverse compilation environments. Its stability stems from the EESG's ability to create a consistent external environment for homologous functions, which combined with the SEM model, ensures a reliable performance enhancement across all test tasks.

> **Answer to RQ.2.** BINENHANCE stably enhances baselines across cross-architecture and cross-optimization option tasks, without succumbing to significant performance dips under various compilation settings.

D. RQ3: Impact of function inlining

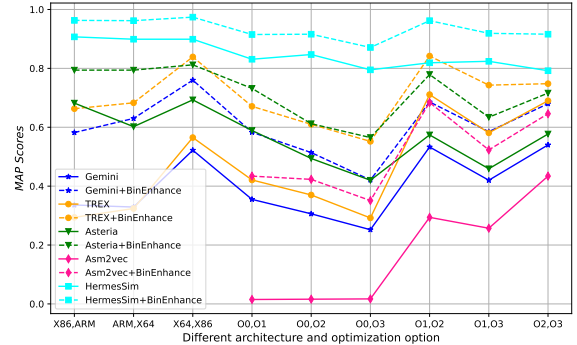In this section, we explore the effects of BINENHANCE on function inlining, employing datasets D2_norm and

Fig. 7: MAP scores in different compilation settings (D1)
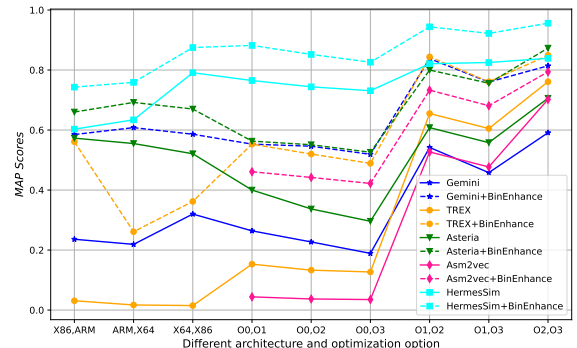

Fig. 8: MAP in different compilation settings (D2_norm)

TABLE V: MAP scores of different methods in different optimization options and architectures ($D2\_norm$).

Models	Cross-architecture			Cross-optimization option					
	ARM,X86	ARM,MIPS	MIPS,X86	O0,O1	O0,O2	O0,O3	O1,O2	O1,O3	O2,O3
Gemini	23.6	21.9	32.0	26.4	22.7	18.9	54.2	45.8	59.1
Gemini+BINENHANCE	**58.5**	**60.8**	**58.6**	**55.3**	**54.6**	**51.9**	**83.8**	**76.0**	**81.4**
TREX	3.1	1.7	1.5	15.3	13.3	12.7	65.5	60.5	76.1
TREX+BINENHANCE	**56.1**	**26.1**	**36.2**	**55.3**	**52.0**	**48.9**	**84.4**	**76.1**	**84.9**
Asm2vec*	-	-	-	4.4	3.7	3.5	52.7	47.7	70.2
Asm2vec*+BINENHANCE	-	-	-	**46.1**	**44.2**	**42.2**	**73.3**	**68.1**	**79.4**
Asteria	57.3	55.5	52.1	40.0	33.7	29.6	60.8	55.8	70.6
Asteria+BINENHANCE	**66.0**	**69.2**	**67.0**	**56.3**	**55.1**	**52.6**	**80.0**	**75.6**	**87.3**
HermesSim	60.3	63.4	79.1	76.5	74.4	73.1	82.1	82.5	83.9
HermesSim+BINENHANCE	**74.3**	**75.9**	**87.5**	**88.2**	**85.2**	**82.6**	**94.4**	**92.2**	**95.6**

∗ Asm2vec do not support cross-architecture.

TABLE VI: MAP scores of different methods of binary code search tasks in inlining and non-inlining compilation settings.

Models	Normal-to-Normal						Normal-to-Noinline					
	P=2	P=128	P=1024	P=4096	P=10000	Avg.	P=2	P=128	P=1024	P=4096	P=10000	Avg.
Gemini	90.6	57.5	38.0	27.6	21.7	47.1	89.6	50.3	28.4	18.9	14.0	40.2
Gemini+BINENHANCE	**93.6**	**73.6**	**62.7**	**56.1**	**51.3**	**67.5**	**92.3**	**69.6**	**57.1**	**49.0**	**44.9**	**62.6**
TREX	73.0	24.3	16.3	13.4	11.9	27.8	71.9	22.1	12.7	9.6	8.1	24.9
TREX+BINENHANCE	**92.8**	**70.4**	**60.4**	**53.0**	**48.7**	**65.1**	**91.0**	**66.7**	**52.8**	**44.0**	**41.4**	**59.2**
Asm2vec*	86.7	40.7	30.2	23.8	19.3	40.1	11.2	6.0	3.0	2.3	1.7	4.8
Asm2vec*+BINENHANCE	**95.0**	**72.9**	**60.3**	**50.5**	**46.6**	**65.1**	**92.9**	**62.6**	**47.7**	**39.1**	**37.0**	**55.9**
Asteria	94.9	71.9	56.8	47.2	39.2	62.0	54.7	29.2	21.3	17.5	14.1	27.4
Asteria+BINENHANCE	**97.1**	**77.7**	**67.8**	**59.2**	**53.4**	**71.0**	**59.1**	**49.2**	**38.2**	**31.3**	**27.1**	**41.0**
HermesSim	93.9	85.3	74.5	62.7	53.3	73.9	92.7	84.5	72.0	57.6	48.6	71.1
HermesSim+BINENHANCE	**97.9**	**87.9**	**79.1**	**69.6**	**61.7**	**79.2**	**97.6**	**87.1**	**75.9**	**64.5**	**56.4**	**76.3**

∗ Asm2vec only support X86 architecture.
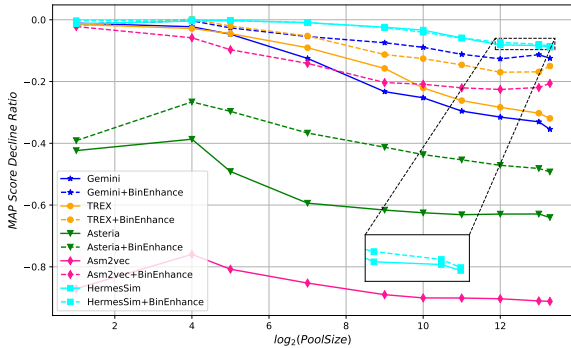

Fig. 9: MAP Score Decline Ratio in function inlining

Fig. 10: MAP scores of different situations

$D2\_noinline$ for this purpose. To evaluate the impact of function inlining, we first extract homologous functions with and without inlining from the two datasets. This extraction is accomplished by analyzing the function call graph and the corresponding symbol information provided by the datasets. Based on these lists of homologous functions, we reconstruct two types of homologous function pairs: **Normal-to-Normal** (NI, both functions are from the normally compiled datasets $D2\_norm$) and **Normal-to-Noinline** (NO, one function from the $D2\_norm$ and the other from the without function inlining datasets $D2\_noinline$). The functions in both homologous function pairs are identical. We evaluate these two types of homologous function pairs separately, as shown in Table VI,
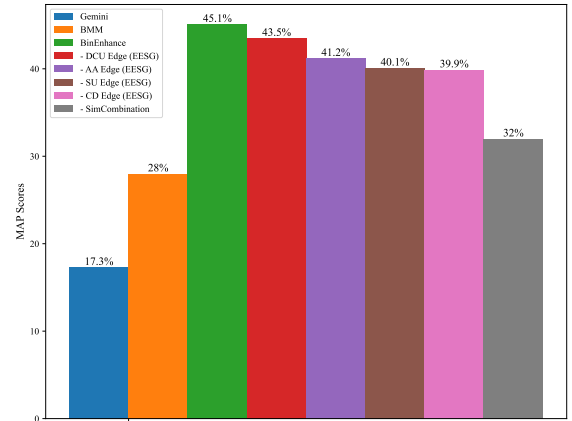
which presents the results for two different tasks.

Firstly, whether in NI or NO tasks, BINENHANCE significantly improves MAP scores compared to the baseline method. Since the only difference between the two homologous function pairs is the presence or absence of inlining, the impact of function inlining can be evaluated by the MAP difference between NI and NO. We use the MAP score decline ratio = $(MAP_{NI} - MAP_{NO})/MAP_{NI}$ as the measurement standard. For clarity, Figure 9 illustrates the effect of BINENHANCE on function inlining, with the y-axis representing the MAP score decline ratio, which indicates the MAP score loss introduced

by function inlining. We observed that, after applying BINEN-HANCE, most baseline methods showed a reduced performance loss (the dashed line is located above the solid line). Both the original model of HermesSim and the model enhanced by BINENHANCE exhibited a relatively small decrease in MAP when dealing with function inlining. HermesSim differs from most other methods that focus on binary code design. It uses a compiler-independent toy IR [17] to convert it into SOG and then employs multi-head softmax to selectively learn the semantics of different nodes on SOG, which is constrained by the analysis tool [17]. Conversely, Asm2vec is significantly affected by function inlining, but BINENHANCE greatly mitigates its MAP score loss. These results are attributed to our proposed stable EESG for cross-compilation settings.

> **Answer to RQ.3.** Function inlining leads to a substantial performance loss in the binary code search task, but BINENHANCE mitigates the decline and improves the baseline model's ability to cope with optimization strategies such as function inlining.

E. RQ4: Ablation Study

To evaluate the impact of different types of edge in EESG and similarity combination on the performance of the BINEN-HANCE framework, we design different ablation experiments on the test dataset separately. For clarity of comparison and time cost, we select Gemini as an example, Figure 10 shows the results of all experiments including:

- **Gemini**: the original model has a MAP score of 17.3%.
- **BINENHANCE**: indicates the complete framework of BINENHANCE with a MAP score of 45.1%.
- **-CD Edge**: EESG does not exist CD edges, with a reduced MAP score of 5.2%, which supplements the semantics through function calls, mitigating the performance loss.
- **-DCU Edge**: EESG does not exist DCU edges, with a reduced MAP score of 1.5%. Global data does not exist in all functions, so its contribution is relatively smaller.
- **-AA Edge**: EESG does not exist AA edges, with a reduced MAP score of 3.9%, which establishes edges between multiplexed functions through location information, supplementing the semantics with location semantic information.
- **-SU Edge**: EESG does not exist SU edges, with a reduced MAP score of 5.0%, which demonstrates that our SEM can merge string semantics with function semantics.
- **-SimCombination**: BINENHANCE does not use the similarity combination module, with a reduced MAP score of 13.1%, it decreases the most, which illustrates that it is effective for the task when data features exist.

In addition, we select BMM [29] for evaluating the performance of function call relationship methods, and the final score is 28.0%, which is 17.1% lower than our method. This also proves that our EESG is superior to function call graphs.

> **Answer to RQ.4.** Each component of BINENHANCE plays a crucial role in the final result, and the absence of any one of them can lead to a degradation in the performance of the binary search task.

TABLE VII: Efficiency evaluation

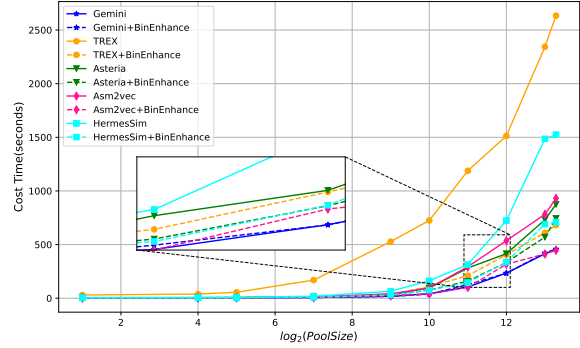| Model	Feature	Train	Embed	Search	All
Asteria	2.14	86.40	1.21	149.20	238.95
BINENHANCE	**1.22**	**2.25**	**0.84**	**118.32**	**122.63**
Asteria+BINENHANCE	2.14	88.65	2.05	**118.32**	211.16

Fig. 11: The impact of embedding dimension on time cost.

F. RQ5: Efficiency Evaluation

Table VII shows the time cost required for each step of processing a binary file for Asteria as well as BINENHANCE. The additional time cost incurred by BINENHANCE is mainly in the training (2.25s) and the generating function embedding (0.84s), which is because these two operations cannot be performed in parallel with the baseline method (feature extraction can be operated in parallel), but in reality, the extra time cost is not high compared to the baseline method. In addition, BINENHANCE ends up producing smaller function embedding dimensions compared to the baseline method. As a result, the time cost spent on binary code search by our method is drastically reduced, as shown in Figure 11. It shows that as the function pool size increases, the time spent on binary code search increases immediately. It is worth mentioning that, as shown in Table II, for all the baseline methods, we have reduced the function embedding dimension for almost all of them. More specifically, the function embeddings of BINENHANCE, when applied to the binary code search task, take less time (even less than 1/4) compared to the original methods, and the saved time increases as the function pool size increases.

> **Answer to RQ.5.** The additional time cost of BINENHANCE is in training and generating function embeddings, but it significantly reduces the time for binary code search tasks, resulting in an overall lower time cost compared to the original model.

G. RQ6: 1-day Vulnerability Search

In this section, we utilized the $D3\_firmware$ alongside CVE vulnerability functions dataset to determine the effectiveness of the BINENHANCE in enhancing real-world 1-day

TABLE VIII: Results of vulnerability search (*fail to recall*).

ID	CVE	Tot.	Gemini	Asteria	TREX	HermesSim	Ours
1	2014-4877	20	3	19	14	0	0
2	2016-8622	9	9	8	7	8	2
3	2016-6301	10	10	0	10	0	0
4	2016-8618	12	11	11	6	6	3
5	2018-19519	6	6	6	6	0	0
6	2018-1000301	1	0	0	0	0	0
7	2018-16230	7	4	4	6	3	3
8	2018-16452	20	1	19	19	2	1
9	2018-16451	3	3	2	3	2	2
10	2020-8306	20	1	18	18	4	2
11	2021-22924	3	2	3	1	2	2
12	2022-0778	5	5	5	5	0	0
	MAP	-	14.2	26.8	15.2	60.2	**67.9**
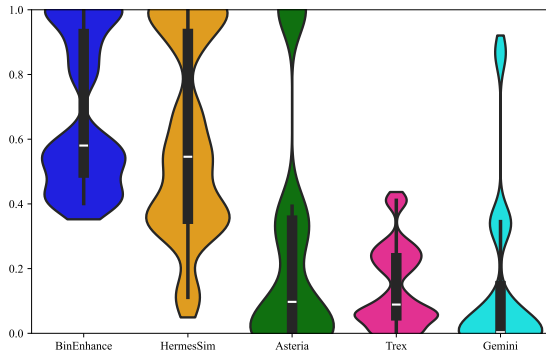
Fig. 12: The violin plot for each method on vulnerability search.

vulnerability search capabilities. To ensure comprehensive coverage of both sources [43, 59], we selected 12 high-impact and commonly reported CVE vulnerabilities [3–13, 15], focusing on widely used open-source components (e.g., $OpenSSL$, $Libcurl$). The evaluation involves homologous function recall using HermesSim and BINENHANCE for each CVE vulnerability function and those undetected in the decompressed firmware binaries. Like the HermesSim experiment, we identify the ground truth by manually examining the top-20 results of all methods and then we obtain 116 vulnerability functions from $D3\_firmware$. As shown in Table VIII, BINENHANCE recalls 101 vulnerability functions with a MAP score of 67.9%, which is 12 more than the recall number of HermesSim with an increased MAP score of 7.7%. BINENHANCE Through incorporating external semantic information that contrasts with the similar internal code structure of these non-homologous functions, BINENHANCE successfully reduces false positives in HermesSim.

Figure 12 shows the distribution of AP scores for each method in 12 CVE vulnerabilities. The internal code semantic methods, such as Gemini and Asteria, fail to recall many vulnerabilities and low AP scores. It means that even if they recall vulnerabilities, the ranking of these vulnerabilities is not high. In our final false analysis of BINENHANCE, two primary factors were identified. First, BINENHANCE is partly contingent on the performance of HermesSim. For example, the $curl\_mvaprintf$ function is recalled as the $curl\_maprintf$ function of $CVE-2016-8618$. The cosine similarity score calculated by HermesSim is more than its homologous func-

tion. Second, certain non-homologous functions share similar external environments, yet these instances are relatively rare. In a word, BINENHANCE can enhance the semantics contained in the embeddings generated by the original model in most cases by introducing external environment semantics (EESG in Section III-C2).

> **Answer to RQ.6.** BINENHANCE identified 101 1-day vulnerabilities in 37 firmware images with 67.9% MAP scores, which is 12 more vulnerabilities detected than HermesSim, and a MAP score of 7.7% higher.

V. RELATED WORK

In the field of Binary Code Search, various techniques have been proposed to tackle the challenge of identifying similarities between the binary code. We categorize the related work into two main groups: methods only utilizing internal code features, and those combining internal features and call graphs (CG).

Internal Code Semantic Methods (ICSM). Several prior studies have concentrated on analyzing internal code features of binary code snippets to detect similarity. These methods largely utilize features like text semantics, control flow graphs (CFGs), or abstract syntax trees (ASTs). For instance, text semantic-based approaches such as asm2vec [26], InnerEYE [64], Trex [44], jTrans [53], and Inter-BIN [48], interpret programming languages through natural language processing techniques to compute similarities. Similarly, Gemini [56] and ISRD [55] use CFGs to analyze structural features in binary code for similarity assessment. Asteria [59] and others [22, 54] involve the use of ASTs for function matching, focusing on syntactic structure analysis. HermesSim [30] and VulHawk[43] convert binary code into IR and construct semantic graphs (e.g., SOG) to encode functions. Despite their effectiveness in some scenarios, these methods can not fully capture the complete semantics of the binary code, limiting their effectiveness in certain complex cases (inlining, etc.).

The combination of CG and ICSM Methods. To overcome the limitations inherent in solely internal code semantic methods, recent studies have explored incorporating external semantic features, especially function call graphs. These integrated approaches aim to improve binary code similarity detection. αdiff [41] represents an innovative example, using a two-dimensional vector for each node's in/out-degree on the function call graph to represent function features. Furthermore, BMM [29] is another noteworthy work that extends the idea of integrating call graphs with internal semantic information. This approach constructs a representation of binary code by combining a control flow graph, data flow graph, and function call graph. By capturing both the structural and behavioral aspects of the code, BMM [29] demonstrates superior performance in identifying code similarities across diverse binary.

In this paper, we propose a novel method that goes beyond the existing approaches by integrating both internal semantic information and external semantic information for binary code search. By combining the comprehensive understanding of the code's internal structure with the contextual knowledge derived from external semantic information, our method aims to capture a more holistic representation of binary code snippets. This integration allows us to leverage both the local

semantics within a code snippet and the broader inter-function relationships, leading to improved accuracy and robustness in detecting similarities between binary codes.

VI. DISCUSSION

In this section, we discuss the limitations and future research of the BINENHANCE framework. BINENHANCE significantly enhances existing binary code search methods by incorporating external semantics. It demonstrates stable performance improvements across various scenarios, including cross-architecture, cross-optimization options, function inlining, and function pool expansion. The framework's compatibility with multiple baseline methods indicates its potential to enhance semantic learning within any function's internal features.

Our experiments reveal that integrating Address-Adjacency (AA) semantics leads to notable improvements. Nonetheless, the efficacy of AA edge is constrained to reused code searches, lacking the capability to enrich non-reused code semantically. It may also be vulnerable to layout obfuscation techniques and could inadvertently integrate irrelevant code snippet semantics. Furthermore, our 1-day detection experiments indicate a rarity of non-homologous functions with similar external environments, posing a risk of false positives. To address these challenges, future research will focus on developing adaptable algorithms for AA edge creation, aiming to infuse meaningful location semantics more accurately.

Currently, many deep learning-based binary code search methods, particularly those employing natural language processing, lack mechanisms to manage complex scenarios such as cross-architecture. These methods typically either input assembly languages from various instruction sets into large self-learning models (e.g., TREX and Ordermaters [60]) or focus on learning representations under a single instruction set (e.g., jTrans [53] and Palmtree [39]). We posit the urgency of adopting Transformer-based models to comprehend cross-architecture semantics in assembly code, akin to multilingual models [24]. Our future work will explore this approach and external semantic models with better learning capabilities.

VII. CONCLUSION

In this paper, we propose a binary code search enhancement framework BINENHANCE, which enhances internal code semantic models with valuable external environment semantic information, thereby reducing the false positive and false negative. We design an EESG to model a function's external environment among homologous functions, linking the target function with the auxiliary functions and the target function with strings via diverse semantic dependencies. Moreover, We introduce whitening transformation and SEM to merge external semantics with internal ones, forming a comprehensive function representation. We utilize data feature similarity to assist in calibrating BINENHANCE. Our approach demonstrates substantial improvements over existing methods in terms of performance and search speed in six tasks, as evidenced by experiments. Additionally, we have evaluated BINENHANCE against HermesSim in detecting 1-day vulnerabilities within a firmware dataset and show its advantages.

REFERENCES

[1] "Craigheffner," https://networkx.org/, 2024.
[2] "Cve-cve-2014-0106," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0106, 2024.
[3] "Cve-cve-2014-4877," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-4877, 2024.
[4] "Cve-cve-2016-6301," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6301, 2024.
[5] "Cve-cve-2016-8618," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8618, 2024.
[6] "Cve-cve-2016-8622," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8622, 2024.
[7] "Cve-cve-2018-1000301," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000301, 2024.
[8] "Cve-cve-2018-16230," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16230, 2024.
[9] "Cve-cve-2018-16451," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16451, 2024.
[10] "Cve-cve-2018-16452," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16452, 2024.
[11] "Cve-cve-2018-19519," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19519, 2024.
[12] "Cve-cve-2020-8036," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8036, 2024.
[13] "Cve-cve-2021-22924," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22924, 2024.
[14] "Cve-cve-2021-44228," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228, 2024.
[15] "Cve-cve-2022-0778," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0778, 2024.
[16] "dgl," https://www.dgl.ai/, 2024.
[17] "ghidra," https://github.com/NationalSecurityAgency/ghidra/, 2024.
[18] "Ida pro," https://hex-rays.com/IDA-pro/, 2024.
[19] "pytorch," https://pytorch.org/, 2024.
[20] "Synopsys 2023 open source security and risk analysis report." https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html, 2024.
[21] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," *ACM Sigsoft Software engineering notes*, vol. 30, no. 5, pp. 156–165, 2005.
[22] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 95–104.
[23] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 678–689.

[24] Z. Chi, L. Dong, F. Wei, N. Yang, S. Singhal, W. Wang, X. Song, X.-L. Mao, H. Huang, and M. Zhou, "In-foxlm: An information-theoretic framework for cross-lingual language model pre-training," *arXiv preprint arXiv:2007.07834*, 2020.

[25] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*. PMLR, 2016, pp. 2702–2711.

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

[27] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 780–791.

[28] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao, and H. Jin, "Cross-language binary-source code matching with intermediate representations," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 601–612.

[29] Y. Guo, P. Li, Y. Luo, X. Wang, and Z. Wang, "Exploring gnn based program embedding technologies for binary related tasks," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 366–377.

[30] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue, "Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection," in *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*, 2024.

[31] H. Huang, A. M. Youssef, and M. Debbabi, "Binse-quence: Fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017, pp. 155–166.

[32] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis," *ACM Transactions on Software Engineering and Methodology*, 2022.

[33] G. Jian, Y. Xin, F. Ying, J. Yu, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *the 33rd ACM/IEEE International Conference*, 2018.

[34] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, "Binaryai: Binary software composition analysis via intelligent binary source code matching," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[35] V. Karande, S. Chandra, Z. Lin, J. Caballero, L. Khan, and K. Hamlen, "Bcd: Decomposing binary code into components using graph-based clustering," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 393–398.

[36] 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)*. IEEE, 2017, pp. 342–352.

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

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

[39] 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*, 2021, pp. 3236–3251.

[40] J. Lin, D. Wang, R. Chang, L. Wu, Y. Zhou, and K. Ren, "Enbindiff: Identifying data-only patches for binaries," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[41] 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*, 2018, pp. 667–678.

[42] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 389–400.

[43] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "Vulhawk: Cross-architecture vulnerability detection with entropy-based binary code search."

[44] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2776–2790, 2022.

[45] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*. Springer, 2018, pp. 593–607.

[46] P. Shirani, L. Collard, B. L. Agba, B. Lebel, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," *Springer, Cham*, 2018.

[47] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "Mpnet: Masked and permuted pre-training for language understanding," *Advances in Neural Information Processing Systems*, vol. 33, pp. 16 857–16 867, 2020.

[48] Q. Song, Y. Zhang, B. Wang, and Y. Chen, "Inter-bin: Interaction-based cross-architecture iot binary similarity comparison," *IEEE Internet of Things Journal*, vol. 9, no. 20, pp. 20 018–20 033, 2022.

[49] J. Su, J. Cao, W. Liu, and Y. Ou, "Whitening sentence representations for better semantics and faster retrieval," *arXiv preprint arXiv:2103.15316*, 2021.

[50] W. Tang, P. Luo, J. Fu, and D. Zhang, "Libdx: A cross-platform and accurate system to detect third-party libraries in binary code," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 104–115.

[51] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "Libdb: an effective and efficient framework for detecting third-party libraries in binaries," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 423–434.

[52] 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.

[53] 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*, 2022, pp. 1–13.

[54] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[55] X. Xu, Q. Zheng, Z. Yan, M. Fan, A. Jia, and T. Liu, "Interpretation-enabled software reuse detection based on a multi-level birthmark model," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 873–884.

[56] 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*, 2017, pp. 363–376.

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

[58] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "Modx: binary level partially imported third-party library detection via program modularization and semantic matching," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1393–1405.

[59] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 224–236.

[60] 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 *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1145–1152.

[61] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: Detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.

[62] D. Zhang, P. Luo, W. Tang, and M. Zhou, "Osldetector: Identifying open-source libraries through binary analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1312–1315.

[63] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 442–454.

[64] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *arXiv preprint arXiv:1808.04706*, 2018.