BinQuery: A Novel Framework for Natural Language-Based Binary Code Retrieval

BOLUN ZHANG[*], Institute of Information Engineering at Chinese Academy of Sciences, China
ZEYU GAO[†], Tsinghua University, China
HAO WANG[†], Tsinghua University, China
YUXIN CUI[†], Tsinghua University, China
SILIANG QIN[*], Institute of Information Engineering at Chinese Academy of Sciences, China
CHAO ZHANG[†‡], Tsinghua University, China
KAI CHEN[*‡], Institute of Information Engineering at Chinese Academy of Sciences, China
BEIBEI ZHAO[*], Institute of Information Engineering at Chinese Academy of Sciences, China

Binary Function Retrieval (BFR) is crucial in reverse engineering for identifying specific functions in binary code, especially those associated with malicious behavior or vulnerabilities. Traditional BFR methods rely on heuristics, often lacking the efficiency and adaptability needed for large-scale or diverse binary analysis tasks. To address these challenges, we present `BinQuery`, a Natural Language-based BFR (NL-based BFR) framework that uses natural language queries to retrieve relevant binary functions with improved flexibility and precision. `BinQuery` introduces innovative techniques to bridge information gaps between binary code and natural language, achieves fine-grained alignment for enhanced retrieval accuracy, and leverages Large Language Models (LLMs) to refine queries and generate diverse descriptions. Our extensive experiments indicate that `BinQuery` surpasses current state-of-the-art methods, achieving a **42.55%** increase in recall@1 and a **4× improvement** in performance on comparable benchmarks.

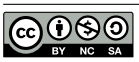
CCS Concepts: • **Security and privacy → Software reverse engineering**.

Additional Key Words and Phrases: Binary Analysis, Deep Learning, Representation Learning

ACM Reference Format:
Bolun Zhang, Zeyu Gao, Hao Wang, Yuxin Cui, Siliang Qin, Chao Zhang, Kai Chen, and Beibei Zhao. 2025. BinQuery: A Novel Framework for Natural Language-Based Binary Code Retrieval. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA052 (July 2025), 23 pages. https://doi.org/10.1145/3728927

[*]Also affiliated with School of Cyber Security, University of Chinese Academy of Sciences, China
[†]Also affiliated with JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.
[‡]Corresponding authors.

Authors' Contact Information: Bolun Zhang, State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering at Chinese Academy of Sciences, Beijing, China, zhangbolun@iie.ac.cn; Zeyu Gao, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, gaozy22@mails.tsinghua.edu.cn; Hao Wang, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, hao-wang20@mails.tsinghua.edu.cn; Yuxin Cui, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, yx-cui24@mails.tsinghua.edu.cn; Siliang Qin, State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering at Chinese Academy of Sciences, Beijing, China, qinsiliang@iie.ac.cn; Chao Zhang, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, chaoz@tsinghua.edu.cn; Kai Chen, State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering at Chinese Academy of Sciences, Beijing, China; Beibei Zhao, Institute of Information Engineering at Chinese Academy of Sciences, Beijing, China, zhaobeibei@iie.ac.cn.



ACM 2994-970X/2025/7-ARTISSTA052
https://doi.org/10.1145/3728927

1 Introduction

Reverse engineering [3, 31] is essential for analyzing binary programs without access to their source code. An essential task in this field involves locating specific functions that may exhibit malicious behaviors [30, 42, 56, 64] or contain vulnerabilities [16, 33, 38, 41, 58, 63]. However, manually locating these functions can be time-consuming and complex, especially given the opaque nature of binary code. To address this challenge, significant efforts have been made to automate the process of locating relevant functions in binaries, which we refer to as Binary Function Retrieval (BFR).

Traditionally, BFR has been achieved through the use of binary analysis tools [24, 39, 44, 47, 50] to extract features like strings [10], constants [43], and program structures [25, 27, 28, 59] These features are then leveraged in conjunction with manually crafted rules to retrieve relevant functions. However, this approach is limited by its reliance on expert knowledge and heuristics, often proving inefficient and time-consuming.

Notably, previous work [17] has demonstrated that specific reverse engineering tasks can be effectively articulated using natural language queries. Building on this insight, we refer to this approach as Natural Language-based Binary Function Retrieval (NL-based BFR), where the goal is to locate specific functions in binaries with greater flexibility and efficiency by leveraging natural language as a query mechanism.

While significant progress has been made in using natural language queries for *source code* retrieval [15, 37], especially with advances in cross-modal contrastive learning [45] (also known as alignment training), the direct application of these techniques to binary function retrieval faces unique challenges. It arises both from the modality gap between binary code and natural language, as well as the specific nature of the retrieval scenario. The three main challenges are outlined as follows:

C1: Both binary and text modalities suffer from the loss of semantic information in the program. Binary code, stripped of identifier names [26] and comments during compilation, lacks high-level semantic information such as function purpose and logic. Conversely, natural language focuses on high-level abstractions, missing low-level implementation details like specific operations and data manipulations. This information loss hinders the model's ability to learn semantic mappings between the modalities, leading to poor performance.

C2: Existing methods, based on function-level alignment, struggle to meet the fine-grained query demands. In real-world reverse engineering scenarios, natural language queries may target specific parts of a function's logic rather than the overall functionality. However, in existing work, both the construction of cross-modal datasets and model training are based on the function as a unit, causing the model to struggle with learning fine-grained semantics which leads to poor performance.

C3: The descriptive perspectives of training texts and real-world queries differ significantly. Real-world queries typically exhibit diverse requirements, including functionality, implementation, or structural characteristics, and often include domain-specific terminology. However, existing methods for constructing training data tend to yield single-perspective and static text data. This discrepancy between real-world usage scenarios and training data can result in a lack of generalization capability in practical situations, leading to suboptimal performance.

To address these challenges, we introduce `BinQuery`, a novel framework that enhances NL-based BFR tasks through data preparation, model training, and retrieval inference phases.

For C1: We leverage source code as a supervisory signal during training. Source code preserves the semantic information lost in binary code and the details absent in natural language descriptions, serving as an effective bridge between the two. All training objectives are aligned across the three modalities—binary, source, and text—allowing comprehensive utilization of the information

contained within the source code. Additionally, we perform extensive ablation experiments to validate the effectiveness of this approach and elucidate its underlying mechanisms.

For C2: We design snippet-level alignment training, providing the model with more fine-grained training data than functions, enabling it to learn semantic knowledge that existing methods cannot acquire. *In the data preparation phase*, we design a novel algorithm that utilizes large language models to extract source code snippets and map them to all modalities. *In the model training phase*, we integrate function-level and snippet-level training objectives, enhancing the model's ability to understand fine-grained details.

For C3: We use an LLM as a bridge between the training text and real-world queries. *In the data preparation phase*, we create a role-based description generation scheme that guides LLMs to produce diverse natural language descriptions from various perspectives, addressing the limited diversity in training descriptions. *In the retrieval inference phase*, we instruct the LLM to augment the original query, standardize its expression, introduce domain knowledge, and even perform guesswork and inference on certain problems to enhance retrieval effectiveness.

We implement `BinQuery` and evaluate it on two representative NL-based BFR datasets. The results demonstrate that `BinQuery` significantly outperforms existing state-of-the-art (SOTA) solutions. On the ViC [17] dataset, which targets general reverse engineering scenarios, `BinQuery` achieves an impressive **42.55% improvement** in the recall@1 metric compared to the best SOTA method. In the Magma [21] dataset, focused on challenging 1-day vulnerability retrieval scenarios, `BinQuery` achieves a **4x increase** in recall@1, substantially enhancing the practicality of NL-based BFR in vulnerability discovery. In summary, our contributions are as follows:

- We introduce `BinQuery`, a novel NL-based BFR framework that employs cross-modal alignment techniques and addresses its unique challenges, including information loss across different modalities, granularity inconsistencies between training and inference stages, and descriptive mismatches between training data and real-world queries.
- We design a new model training strategy to improve text-binary retrieval performance. Specifically, we leverage source code as a supervisory signal to overcome the information loss across binary and text modalities. Additionally, we propose a snippet-level alignment training method to provide the model with fine-grained training data, enabling it to capture detailed semantic information that function-level methods cannot.
- We use LLMs to bridge the gap between training text and real-world queries. First, we create a role-based description generation scheme to produce diverse natural language descriptions, improving the model's generalization to real-world queries. Second, we introduce LLM-driven query augmentation techniques that standardize query expressions, integrate domain knowledge, and perform inferential guesswork to boost retrieval effectiveness.
- Comprehensive experiments show that `BinQuery` consistently outperforms existing state-of-the-art methods. Notably, it achieves a 42.55% improvement in recall@1 for general reverse engineering scenarios and a 4x increase in recall@1 for the challenging 1-day vulnerability search scenario.
- We release our `BinQuery` framework to the research community to facilitate future advancements in NL-based BFR.

2 Background and Related Works

In this section, we first briefly introduce the specific approach of alignment training. Then, we enumerate some related BFR works, including those that perform retrieval using other modalities. Finally, we discuss previous approaches to constructing training data using LLM, both in general scenarios and under binary analysis.

2.1 Cross-Modal Alignment Training

Cross-modal contrastive learning has become a powerful method for unified representation across modalities like images and text. A key work is CLIP [45], which uses image-text pairs to align visual and textual embeddings in a shared space. The training objective of CLIP is to ensure that the correct image-text pairs are more similar to each other than to any non-matching pairs. To do this, CLIP uses the following loss function:

$$\mathcal{L}^{clip} = -\frac{1}{n}\sum_{i=1}^{n}\left[\log\frac{\exp(E^{img}(x_i)\cdot E^{txt}(y_i)/\tau)}{\sum_{j=1}^{n}\exp(E^{img}(x_i)\cdot E^{txt}(y_j)/\tau)} + \log\frac{\exp(E^{img}(x_i)\cdot E^{txt}(y_i)/\tau)}{\sum_{j=1}^{n}\exp(E^{img}(x_j)\cdot E^{txt}(y_i)/\tau)}\right]$$

This process is also referred to as alignment, which aims to coordinate the embeddings of different modalities so that semantically similar data points from each modality are positioned closely within the shared latent space.

2.2 Existing BFR Approaches

Rule-based approaches [9, 23, 43] often use static analysis to extract static and dynamic features from binary code. These approaches involve the manual creation and adjustment of heuristic rules to automatically select target binary functions. However, these methods heavily rely on expert knowledge, are inefficient, have limited generalization capabilities, and are not well-suited for handling large-scale BFR demands.

A number of studies [17, 29, 51] have emerged with the rise of deep learning methods, particularly contrastive learning [20, 46]. These studies share a common characteristic: they map the query and candidate binary code functions into a high-dimensional vector space. Through training, they ensure that the vector corresponding to the query code is closest in space to that of the target binary program, thus efficiently pinpointing the target binary function.

Binary-based BFR, where the query and target consist of binary code, is also known as Binary Code Similarity Detection (BCSD) [7, 22, 34, 57]. These methods are typically used when a reverse engineer has acquired the binary code of a target function but needs to locate functionally equivalent code within a large set of binary functions. The main challenge stems from variations in binary code, which arise due to different compilation options, diverse architectures, and varying platforms.

Source-based BFR, first introduced by CodeCMR [61], uses a network architecture that combines CNN [6] with LSTM [19] to map query source code into vectors and a graph neural network combined with LSTM to map target binary code into vectors. This approach aims to align the two vector spaces through contrastive learning, thereby enabling the specific retrieval of binary code functions using source code.

In real-world scenarios, data from specific modalities is sometimes difficult to obtain. For example, in 1-day vulnerability retrieval scenarios, a reverse engineer may only have the CVE description, creating an urgent need to use natural language as a query for retrieval.

2.3 LLM Synthetic Data

Constructing ground truth is crucial for deep learning-based technologies. The recognized capabilities of Large Language Models have led to the emergence of LLM-based dataset construction methods, known as LLMs synthetic data [35, 48]. In general domains, researchers like Furu Wei [54] trained a highly effective natural language embedding model by requesting the LLM to generate similar but semantically different negative samples for a given sentence, constructing a high-quality contrastive learning dataset.

In binary analysis, CLAP [51] requires the LLM to generate natural language descriptions for source code and uses these descriptions for cross-modal supervision, resulting in a binary embedding

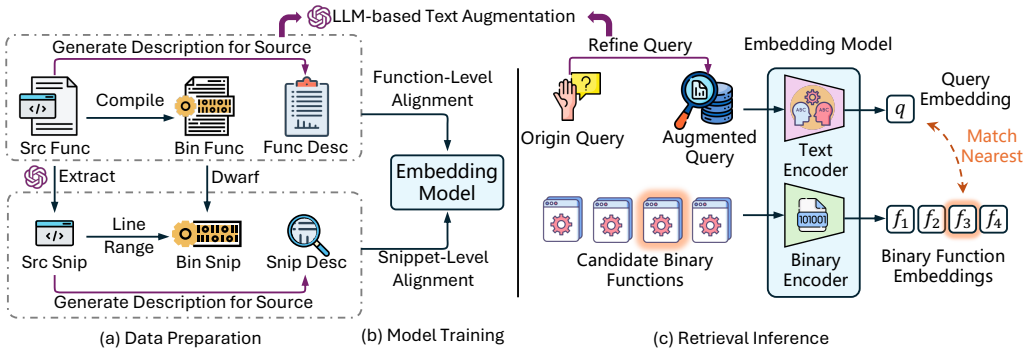


Fig. 1. Overview of BinQuery. During training, the embedding model consists of three encoders for all modalities, including a source encoder for supervision. Only binary and text encoders are used for inference.

model with strong generalization capabilities. COMBO [62] simultaneously uses source code and natural language to form a fused embedding and employs this fused embedding to supervise the training of the binary model. However, it is precisely this fusion strategy that causes the model obtained by this method to be unable to perform retrieval using either the source code or text as a single modality.

3 Methodology

The workflow of our research is illustrated in Figure 1, divided into three stages: (1) *Data Preparation.* Our method begins by collecting data at *function* and *snippet* levels across *source*, *binary* and *text* modalities. (2) *Model Training.* Our model targets two objectives: function-level (Section 3.1.2) and snippet-level (Section 3.3.2), both of which are supervised by source code modality. (3) *Retrieval Inference.* We augment the query provided by the user with LLM to produce a better query for retrieval (Section 3.4.2). Subsequently, this augmented query, along with all candidate functions, is encoded into embeddings by the models. The target function is identified as the one whose embedding is closest to the query embedding within the vector space.

We structure the following sections according to our approach to various challenges. First, we explain the incorporation of source code supervision in alignment training (Section 3.1). Next, we describe the snippet-level data collection process (Section 3.2) and the design of snippet-level alignment training objectives (Section 3.3). Finally, we discuss the use of LLM for text augmentation on training data and queries to enhance retrieval performance (Section 3.4).

This organization by challenge highlights the motivation for each technique. However, since the implementation of these proposals spans different operational phases of our framework, we provide Table 1 to clarify the logical structure and relationship between our proposals and the three stages of BinQuery: Data Preparation, Model Training, and Retrieval Inference.

The table explicitly maps each core proposal (*Source Supervision*, *Snippet Alignment*, *Text Augmentation*) to the primary challenge it addresses (*C1*, *C2*, *C3*) and indicates, using section references, during which stage(s) each proposal is actioned. Specifically, *Source Supervision* (for *C1*) involves activities during both *Data Preparation* (as detailed in Section 3.1) and *Model Training* (covered in Sections 3.1 and 3.3). Similarly, *Snippet Alignment* (for *C2*) requires *Data Preparation* (Section 3.2) and *Model Training* (Section 3.3). In contrast, *Text Augmentation* (for *C3*) encompasses techniques applied during *Data Preparation* and later during *Retrieval Inference* (both discussed in Section 3.4),

Table 1. Outline of Section 3, relationships between our proposal and the three stages.

Proposal	For Challenge	Data Preparation	Model Training	Retrieval Inference
Source Supervision	C1	3.1	3.1, 3.3	-
Snippet Alignment	C2	3.2	3.3	-
Text Augmentation	C3	3.4	-	3.4

but not during the core *Model Training* phase itself (indicated by '-'). This table serves as a concise reference to understand how our technical contributions are distributed across the `BinQuery` workflow to tackle the identified challenges.

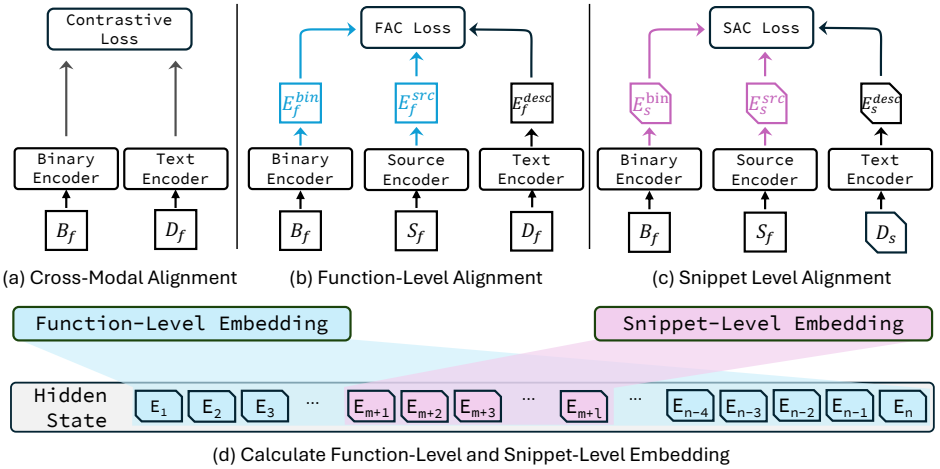


Fig. 2. Key points of the training process in `BinQuery`. `B` stands for binary, `S` for source, `D` for description, and `E` for embedding. Subscript `f` for function and `s` for snippet.

3.1 Source Code Supervision

In this section, we describe how we introduce source code as an additional supervision and train the binary, source, and text encoders together to obtain our embedding model. This involves obtaining data for different modalities and the training algorithm.

3.1.1 Obtain Cross-Modal Data with Compilation and LLM. We use the `apt` [11] build system on Ubuntu [4] for automated compilation. We modify the compilation environment by intercepting the default *gcc* compiler and setting optimization levels from `O0` to `O3` and `Os`. To get the corresponding text data, we use an LLM to generate explanations for the source code used in model training. The details of generating descriptions will be discussed in Section 3.4.1 and are not covered here.

3.1.2 Function-Level Training Objective with Source Supervision. We use three different encoders for alignment training, in which the source encoder is only used during training as a supervision. Only binary and text encoders are used for inference, as shown in Figure 1(c).

Here, we introduce how source code supervision is incorporated at the function level during training, while the snippet-level approach will be discussed in Section 3.3.2. For any two modalities, we use contrastive loss as the training objective. As shown in Figure 2(a), we use the binary and text modalities as an example. Assume we have a set of functions $f = \{f_1, f_2, \ldots, f_n\}$. Their binary code's

embedding calculated by binary encoder can be represented as $E^{bin}(f) = \{E^{bin}_{f1}, E^{bin}_{f2}, \ldots, E^{bin}_{fn}\}$, their corresponding descriptions' embedding calculated by text encoder can be represented as $E^{desc}(f) = \{E^{desc}_{f1}, E^{desc}_{f2}, \ldots, E^{desc}_{fn}\}$. Let τ denote temperature, and we can define the following loss.

$$\mathcal{L}^{bin-desc}(f) = -\frac{1}{n}\sum_{i=1}^{n} log\frac{exp(E^{bin}_{fi} \cdot E^{desc}_{fi}/\tau)}{\sum_{j=1}^{n} exp(E^{bin}_{fi} \cdot E^{desc}_{fj}/\tau)}$$

This operation will occur symmetrically between every pair of modalities, as shown in Figure 2(b), and we formally define the loss as follows:

$$\mathcal{L}_{FAC}(f) = \sum_{\substack{m_1, m_2 \in \{\text{bin,src,desc}\} \\ m_1 \neq m_2}} \mathcal{L}^{m1-m2}(f)$$

Since the training objective above is at the function level, we refer to this loss function as the Function Alignment Contrastive (FAC) loss.

3.2 Snippet-Level Data Preparation

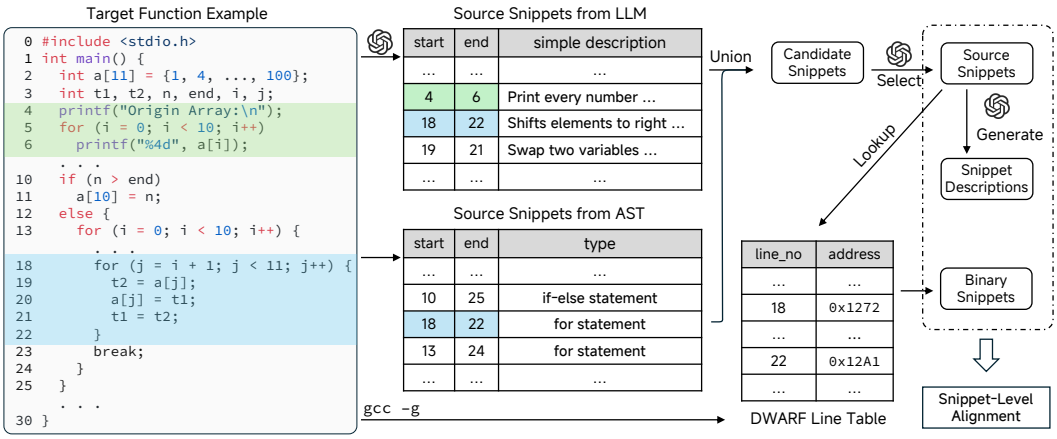


Fig. 3. A Flowchart of Snippet-Level Data Generation with the Help of LLM

It is difficult to formally define and extract semantically complete snippets using fixed rules. Since LLMs have been trained on large amounts of code data, and their capabilities on source code are now on par with human experts [12, 60], we prompt the LLM to perform snippet extraction.

3.2.1 Recursive Splitting Snippet Extraction Strategy. Directly instructing the LLM to generate descriptions can lead to several issues, such as selecting overly fine-grained or coarse-grained snippets and potentially missing valuable information. To mitigate these concerns, we implement a two-stage snippet extraction strategy, as shown in Figure 3: (1) Use the LLM and rules to extract as many snippet candidates as possible. (2) Use the LLM to select the most valuable snippets. The following are the implementation details.

We instruct the LLM to recursively split the target function to cover the snippets of interest comprehensively, as we observe that function snippets have nested relationships, with complex snippets built from simpler ones. Below is a brief description of the prompt we used.

I will provide you with a numbered code snippet, and you need to help me partition it by functionality. You should reply to me with a **JSON** like:

Snippet := {
    range: [start: *int*, end: *int*],
    simple_description: *string*,
    sub_snippets: [**Snippet**, ...]
}

In this prompt, we request the LLM to output snippets in a recursive JSON structure. While the simple description generated in this step is not used for training, requiring the model to provide the rationale for its results and generating additional information helps LLMs produce better output [5, 55]. Additionally, before the LLM provides its answer, we present an example in advance to improve performance using a few-shot prompt [49]. The LLM's output forms the first set of snippets in Figure 3.

We extract another set of snippets based on rules to complement the extraction from the LLM, aiming to address the issue of missing code snippets by the LLM. We parse source code functions into AST using `tree-sitter` [2] and pick {`where`, `if-else`, `compound`, `switch`} statements as supplements to candidates. This way, we obtain the second set of snippets in Figure 3, and then we combine both sets of snippets into a union, which serves as the candidates for LLM to select from.

We input all candidate snippets into the large language model (LLM) and instruct it to select up to three snippets that are representative and logically complete. To help the LLM better understand our intent, we introduce a hypothetical scenario at the beginning, assuming that *these snippets will be used to teach programming beginners*, and we instruct the LLM to prioritize snippets that offer greater educational value.

3.2.2 Map Snippets to Other Modalities. We add the `-gdwarf-4` option to the original compile commands, ensuring that all resulting binaries include DWARF debug information [8]. By extracting the `line_table` from the `debug_line` section, we establish a mapping between source code line numbers and binary function addresses. Using this mapping table, we map each source snippet to the corresponding address range in the binary program, ultimately obtaining binary snippets.

Unlike the binary or source modality, in the text modality, a snippet is not a subsequence of a function, so we generate descriptions for each source snippet individually. For each snippet, we provide the containing function and specify the snippet's line range to help the LLM reference its context and provide a better explanation. The details of generating these descriptions will be discussed further in Section 3.4.1.

3.3 Snippet-Level Alignment Training

In this section, we first describe how our model is modified to produce function-level and snippet-level embeddings. We then present the loss function for snippet-level alignment training.

3.3.1 Weighted pooling for function and snippet-level embeddings. As shown in Figure 2(d), we consider different tokens when calculating function-level and snippet-level embeddings. Let $T = \{t_1, t_2, \ldots, t_n\}$ denote the function token sequence. The encoder generates a sequence of token embeddings $E = \{e_1, e_2, \ldots, e_n\}$ where e_i is the embedding for token t_i. *For function-level embedding*, we use the common approach of averaging all tokens in the function through mean pooling as $E_f = \frac{1}{n}\sum_{i=1}^{n} e_i$. *For snippet-level embedding*, our approach is based on two insights: (1) tokens related to the snippet represent its semantics, and (2) tokens outside the snippet but within the

same function represent its context. We design the following weighted pooling scheme:

$$E_s = \frac{\sum_{i=1}^{n} w_i \cdot e_i}{n} \quad \text{where} \quad w_i = \begin{cases} c & \text{if } t_i \in S \\ 1 & \text{otherwise} \end{cases}$$

This structure is only employed for the binary and source modalities, excluding the text modality. Since the text modality lacks the inherent structure where snippet sequences are components of function sequences, we generate distinct descriptions for functions and snippets, computing their embeddings separately. This explains why in Figure 2(c), the binary and source modalities are computed using function-level input, whereas the text modality employs snippet-level input.

3.3.2 Snippet-Level Training Objective with Source Supervision. Unlike the function-level alignment in Section 3.1.2, we use a different strategy for negative sample sampling here. During function-level alignment, any two different functions are considered negative samples, but in snippet-level alignment, different snippets from the same function are excluded from the loss calculation. This is because they share the same context, and treating them as negative samples degrades performance.

Assume we have snippets from n functions $s = \{s_1, s_2, \ldots, s_m\}$, one function may have multiple fragments. Binary code snippet embeddings from s are represented as $E^{bin}(s) = \{E^{bin}_{s1}, E^{bin}_{s2}, \ldots, E^{bin}_{sm}\}$. Similarly, snippet description embeddings from s are $E^{desc}(s) = \{E^{desc}_{s1}, E^{desc}_{s2}, \ldots, E^{desc}_{sm}\}$.

We introduce an indicator function to show if two snippets are from the same function:

$$\mathbb{I}(s_i, s_j) = \begin{cases} 0 & \text{if from same function and } i \neq j \\ 1 & \text{otherwise} \end{cases}$$

The indicator function helps define a loss function that pulls the embeddings of matching binary and source code snippets closer while pushing apart those from different functions. Let τ denote temperature, and we can define the following loss.

$$\mathcal{L}_s^{bin-desc}(f) = -\frac{1}{m} \sum_{i=1}^{m} \log \frac{\exp(E^{bin}_{si} \cdot E^{desc}_{si} / \tau)}{\sum_{j=1}^{m} \mathbb{I}(s_i, s_j) \cdot \exp(E^{bin}_{si} \cdot E^{desc}_{sj} / \tau)}$$

We apply this pattern to each modality pair to define the Snippet Alignment Contrastive (SAC) loss:

$$\mathcal{L}_{SAC}(f) = \sum_{\substack{m_1, m_2 \in \{\text{bin,src,desc}\} \\ m_1 \neq m_2}} \mathcal{L}_s^{m_1 - m_2}(f)$$

Combining the FAC and SAC loss, training loss is defined as $\mathcal{L}(f) = \mathcal{L}_{CMC}(f) + \mathcal{L}_{SAC}(f)$.

3.4 Text Augmentation with LLM

We perform text augmentation in both the data preparation and retrieval inference stages. During data preparation, the LLM generates a rich variety of data that enhances the embedding model's generalization capability. During retrieval inference, the LLM helps filter noise from real-world queries and supplements missing information in queries through its reasoning capabilities.

3.4.1 Role-based Source Description Generation. We employ DeepSeek Coder [13, 14] to generate natural language descriptions for source code, instead of directly targeting binary code, for two reasons: Symbolic information is lost in binary code, hindering the LLM's understanding. Furthermore, LLMs are better at understanding source code, as they are typically trained on larger source code datasets compared to binary code.

We propose role-based prompts to guide the LLM in explaining the target source code functions or snippets from multiple perspectives. Below is a brief description; we list the roles for which the LLM is required to act and the perspective from which the LLM is expected to provide explanations.

(1) Architect: The LLM is required to describe the target source code from a functional perspective, providing data on *what this code does*.
(2) Developer: The LLM is required to describe the target source code from an implementation perspective, providing data on *how this code does it*.
(3) Reverse Engineer: The LLM is required to describe the target source code mainly from its structure. This means focusing on *parts that stay the same after compilation*. It should avoid mentioning variable names, strings, or other specific details from the source.

The design above is based on the fact that analysis goals and target programs vary in real-world scenarios, resulting in different description focuses. Real-world queries may focus on *what* and *how*, leading to the introduction of *architect* and *programmer* roles. As for *reverse engineer*, our main idea is that natural language descriptions from source code may depend on specific information like programmer-chosen names, but such information is often lost in binary code. Thus, we want the LLM to focus on *structural features*, as they remain consistent through compilation.

3.4.2 Query Augmentation. When the embedding model receives a query from a reverse engineer, we employ an LLM to augment the original query before proceeding with the retrieval of binary functions. The main components of our prompt consist of the following three parts:

(1) Require LLM to generate a direct description and, if possible, make reasonable inferences.
(2) Provide examples of role descriptions from the existing dataset and request the generation of descriptions that match the language style and sentence structure.
(3) Provide the output in JSON format and require it to be returned according to it.

For example, if a CVE description mentions *a buffer overflow in serialization*, the original query may focus on the overflow's impact, which misaligns with the model's training goals. An LLM may guess that the target function involves serializing data into a buffer, thus enhancing retrieval effectiveness. Real-world reverse engineering processes inspire this design, as reverse engineers often make informed guesses about possible implementations based on incomplete information, thereby identifying potential vulnerabilities.

4 Experimental Setup

In this section, we describe the detailed settings of our experiments. We will present the details of the datasets in Section 4.1, and then outline the details of our model configuration in Section 4.2, followed by an introduction to the state-of-the-art (SOTA) baselines we compared against in Section 4.3, and finally, describe the metrics used for comparison in Section 4.4.

4.1 Dataset

In our experiments, we utilize three datasets: the BLA dataset we created for training, the ViC dataset, and the Magma dataset from prior works for comparison with previous studies.

Binary Language Alignment (BLA) Dataset for Training. The dataset is generated as described in Sections 3.1.1, 3.2, and 3.4.1. We use pyelftools [1] to parse dwarf information [8], IDA Pro [24] v7.6 to extract binary code, tree-sitter v0.22.4 to get supplement snippets, and DeepSeek Coder v2 [13, 14] to generate descriptions.

ViC Dataset for General NL-based BFR Evaluation. The dataset [17] comprises 257 reverse engineering query requests, handcrafted by binary reverse engineers, targeting closed-source software across platforms such as MacOS, Windows, and Linux. Each query is paired with its corresponding ground-truth target binary function. During the evaluation, we use each query in natural language to retrieve the target binary function from all functions in the program, and other functions serve as distractors. We simulate a reverse engineer needing to locate specific binary functions of interest in a target program in this dataset.

Magma Dataset for 1-day Vulnerability Search. Magma [21] is a dataset for evaluating fuzzer performance, containing 138 real-world vulnerabilities from 9 open-source projects. We manually compile the source code using default options and select 130 vulnerabilities recorded in the CVE database for our tests. During the evaluation, we use vulnerability descriptions from the CVE database as queries to retrieve vulnerable binary functions from the specific project. We simulate a scenario of a 1-day vulnerability search where a security researcher obtains vulnerability information from the CVE and locates the target function in the binary program.

4.2 Model Configuration

For the encoders of the binary code, source code, and language modalities, we employ hyper-parameter settings equivalent to BERT-base, with L=12, H=768, and A=12, totaling 110 million parameters. We use AdamW [36] as an optimizer, with the learning rate set to $1.0e-4$, the dropout rate set to 0.1, the weight decay set to 0.01, and the temperature $\tau = 0.07$. We use a cosine learning rate scheduler, with the warm-up steps set to 0.05 of the maximum training length. All training is conducted on a machine equipped with 8 NVIDIA RTX 4090 GPUs, Intel Xeon 6236 CPUs, and 378GB of memory.

We use two setups to evaluate our proposed method. The full setup involves training the model with the best effort to achieve optimal performance, enabling direct comparison with existing work. However, due to the high training cost, we also adopt a limited setup, where the model is trained on a fixed amount of data (inferior performance compared to the full setup). This allows us to conduct multiple fair comparison experiments within our resource constraints, facilitating a comprehensive analysis of the contributions of various components in our proposed method.

Full Setup. We initialize the corresponding encoders of our model using the binary encoder and language encoder from CLAP and initialize the source encoder with *gte-base* [32], until convergence on the validation set, and then obtain the final results on the test set.

Limited Setup. We initialize the model parameters using Xavier initialization [18] with default settings and perform Masked Language Modeling (MLM) pre-training for each modality. For the subsequent ablation experiments, regardless of the specific strategies employed, we start training from this pre-trained model and limit the training to 10 million iterations before evaluating the results. This approach ensures a fair comparison across different ablation strategies by maintaining a consistent starting point.

4.3 Baselines

To evaluate `BinQuery`, we compare its performance against two categories of baseline models:

General Models. We include widely-used text embedding models: text-embedding-ada-002, text-embedding-3-small, and text-embedding-3-large from OpenAI [40], as well as gte-base-en-v1.5 and gte-large-en-v1.5 from Alibaba-NLP [32]. These models serve as standard benchmarks for general text embedding tasks. Comparing against them helps establish a performance baseline and determine the necessity of domain-specific adaptations for NL-based BFR.

Specialized Models for Binary Analysis. We compare against CLAP [51] and ViC [17]. These models represent relevant state-of-the-art approaches in the binary analysis domain related to our task.

- **CLAP** [51]: utilizes contrastive language-assembly pre-training, relevant for understanding binary semantics from text.
- **ViC** [17]: directly targets NL-based BFR using LLMs.

Comparing against these models helps demonstrate the effectiveness of `BinQuery` relative to current specialized approaches.

Other Related Works (Indirect Comparison). Other related works are considered conceptually through ablation studies rather than direct end-to-end comparison, for the following reasons:

- **CodeCMR** [61]: Focuses on source-to-binary retrieval, performing direct contrastive alignment solely between source and binary embeddings. As it does not handle NL queries, it is unsuitable for direct NL-BFR comparison. To conceptually evaluate its strategy of direct two-modality alignment, our `BinQuery-BT` ablation setting mirrors this structure by aligning only binary and text embeddings. This allows assessment of such a direct alignment approach for the NL-BFR task.
- **COMBO** [62]: Primarily improves Binary Code Similarity Detection (BCSD), using fused source and text embeddings to supervise the binary encoder. Since it cannot perform retrieval using only an NL query, direct NL-BFR comparison is unsuitable. We conceptually evaluate its approach of incorporating source code during training via our `BinQuery-BST` (binary-source-text) ablation setting. This allows assessing the benefit of using this additional source modality compared to the direct binary-text alignment performed in the `BinQuery-BT` setting.

4.4 Evaluation Metric

In this section, we define the Mean Average Precision (MAP) and recall@k metrics, which are used to evaluate the performance of a retrieval system.

4.4.1 MAP. It averages the Average Precision (AP) over all queries. For each query q_i, $|r_i|$ is the number of retrieved binary functions. Precision(k) is the proportion of relevant items in the top k retrieved functions. The indicator function rel(k) is 1 if the item at rank k is relevant, else 0. The total relevant binary functions for query q_i is $|\{p_i\}|$, and n is the total number of queries. The MAP is computed using this formula:

$$\text{MAP} = \frac{1}{n}\sum_{i=1}^{n} \text{AP}(q_i) \quad \text{where} \quad \text{AP}(q_i) = \frac{\sum_{k=1}^{|r_i|} \text{Precision}(k) \cdot \text{rel}(k)}{|\{p_i\}|}$$

MAP offers a single value indicating the system's effectiveness, considering precision and ranking across multiple queries.

4.4.2 Recall@k. It measures how many relevant items are retrieved within the top k results for a given query. For each query q_i, k is the rank cutoff for evaluating recall. $\{r_i^{(k)}\}$ is the set of top k retrieved functions for q_i. $|\{p_i\}|$ is the total number of relevant functions for q_i. $|\{p_i\} \cap \{r_i^{(k)}\}|$ is the number of relevant functions retrieved in the top k. Recall@k for query q_i is given by:

$$\text{Recall@k}(q_i) = \frac{|\{p_i\} \cap \{r_i^{(k)}\}|}{|\{p_i\}|}$$

Recall@k(q_i) measures the fraction of relevant functions retrieved in the top k results for q_i. Averaging Recall@k over all queries gives a measure of the system's ability to retrieve relevant items within the top k.

5 Evaluation

To prove `BinQuery`'s effectiveness in addressing previous challenges, we propose these research questions (RQs):

- **RQ1:** How does `BinQuery` perform compared to SOTA NL-based BFR solutions?
- **RQ2:** How does `BinQuery` perform in challenging vulnerability search task?
- **RQ3:** How does source code supervision affect the final results?

- **RQ4:** How does snippet-level alignment affect the final results?
- **RQ5:** Is it reasonable to use different encoders to process different modalities?
- **RQ6:** How does role-based description affect model performance?
- **RQ7:** What impact does Query Augmentation have?

5.1 Performance (Full Setup) (RQ1)

In this experiment, we evaluate `BinQuery`'s performance on the ViC dataset, as introduced in Section 4.1. We use language descriptions manually crafted through reverse engineering as queries to retrieve corresponding functions from entire closed-source software projects. We report the experimental results as shown in Table 2, which includes the results of BinQuery with and without the query augmentation mechanism.

Our proposed method outperforms all baselines. `BinQuery` achieves the best results on all metrics, notably improving recall@1 from 27.99 to 39.9, a 42.55% increase over the second best, CLAP. *Improvement in the Recall@1 metric is crucial for the future automation of binary program analysis*, as it allows the model to accurately identify the target binary function at rank 1, enabling further analysis without human intervention.

Our proposed method shows a greater lead when the rank cutoff is smaller. At recall@50, `BinQuery` improved from 77.83 to 84.12 compared to the second-best SOTA, ViC, showing an increase of 8.08%. In comparison, the increase in performance at recall@1 is more pronounced. From observing the results, we find that the primary reason is that our method enhances the ranking of samples that were previously retrieved by other methods, effectively clustering target functions at the top. This leads to a more significant advantage when dealing with smaller cutoff values. For samples that all methods fail to retrieve, it is likely due to other parts of the closed-source software containing more relevant, unannotated distractors, rendering these samples exceedingly difficult to retrieve.

We do not discuss the impact of query augmentation here, and it will be discussed in Section 5.7.

Answering RQ1

`BinQuery` significantly outperforms all SOTA methods in NL-based BFR tasks and **leads across all evaluation metrics**. Notably, it achieves an impressive **42.55%** improvement in the recall@1 metric, bringing it closer to the fully automated binary program analysis in the future.

5.2 Vulnerability Search (Full Setup) (RQ2)

In Table 3, we present the results of our proposed method compared to existing NL-based BFR approaches on the vulnerability search task.

5.2.1 `BinQuery` *significantly outperforms previous approaches.* The data on the left side of each cell shows the results obtained using the original CVE descriptions. `BinQuery` improves the MAP metric from 5.82 to 19.24, representing a **130.58%** increase over ViC, and achieves **399.68%** of the recall@1 score of the second-best method, indicating that our work significantly outperforms existing approaches. We believe the more valuable improvement lies in `BinQuery`'s increase in the recall@50 metric from 25.82 to 43.87, achieving a **69.91%** improvement. This improvement enables security researchers to pre-select 50 candidate functions using NL-based BFR, achieving a recall rate above 40%, which assists in practical security analysis. Our method significantly enhances usability for real-world tasks when compared to previous approaches.

5.2.2 Potential Information Leak. CVE descriptions may leak information like function names or strings within functions. This leakage enables the model to take shortcuts, compromising the

Table 2. Results of different NL-based BFR approaches on ViC Dataset

Models		MAP	recall@1	recall@5	recall@20	recall@50
text-embedding-ada-002		9.78	3.04	17.11	32.7	43.00
text-embedding-3-small		18.65	12.17	24.27	39.29	50.44
text-embedding-3-large		21.91	14.12	30.41	45.93	58.02
gte-base-en-v1.5		2.83	0.75	4.51	9.40	18.42
gte-large-en-v1.5		8.53	3.88	12.40	23.97	33.40
CLAP		39.98	27.99	52.39	67.87	74.33
ViC		35.35	24.24	47.68	68.26	77.83
BinQuery	w/o qa	**50.69**	**39.90**	62.20	**78.54**	**84.12**
	w/ qa	48.05	35.43	**62.44**	78.27	83.33

[1] The first five entries in the table are general models, and the next are specialized models for binary analysis. We show the performance of BinQuery with and without query augmentation.
[2] The best results are indicated in bold, and the second-best is underlined (excluding another setup of BinQuery).

Table 3. Results of different NL-based BFR approaches on Magma Dataset

Models		MAP	recall@1	recall@5	recall@20	recall@50
text-emb-3-large		3.24/3.17	1.54/1.69	3.85/4.77	8.23/8.00	17.62/14.92
ViC		5.82/3.34	3.08/1.54	7.46/3.85	10.46/8.46	25.82/13.61
CLAP		3.27/1.56	1.54/0.00	1.92/1.54	7.85/4.62	16.82/6.67
BinQuery	w/o qa	13.93/4.8	8.08/1.54	17.31/6.15	31.54/11.54	36.82/16.92
	w/ qa	**19.24/8.66**	**12.31/5.38**	**25.00/9.23**	**37.44/18.85**	**43.87/30.00**

[1] The two sets of data, separated by a slash, show results from the original CVE description (left) and masked CVE description (right).
[2] The best results are indicated in bold, and the second-best results are underlined (excluding another setup of BinQuery).
[3] We choose the best-performing general-purpose model as a representative.

objective evaluation of NL-based BFR. We manually process CVE descriptions to mask information leaks, replacing function names with *the function* and summarizing strings to prevent exact matches. The right side of each cell shows results after masking leaks in descriptions.

BinQuery, like other methods, experiences a performance drop when information leaks are masked. However, we observe that BinQuery still maintains 30% accuracy on recall@50, retaining fairly strong usability, while other methods achieve at most 14.92%, making them almost unusable.

CVE-2017-0663: A remote code execution vulnerability in libxml2 could enable an attacker using a specially crafted file to execute arbitrary code within the context of an unprivileged process. This issue is rated as High due to the possibility of remote code execution in an application that uses this library. Product: Android. Versions: 4.4.4, 5.0.2, 5.1.1, 6.0, 6.0.1, 7.0, 7.1.1, 7.1.2. Android ID: A-37104170.
CVE-2016-10267: LibTIFF 4.0.7 allows remote attackers to cause a denial of service (divide-by-zero error and application crash) via a crafted TIFF image (related to libtiff/tif_ojpeg.c:816:8).

5.2.3 Vulnerability Search is Challenging. The results for the vulnerability search task are significantly lower than those for the ViC dataset, as shown in Section 5.1. Compared to the manually

crafted queries from reverse engineering in the ViC dataset, the queries from CVE descriptions in the Magma dataset are notably harder, and we identify the following two main issues: (1) *CVE descriptions often lack crucial details.* CVE-2017-0663 exemplifies this problem. Its description only includes the version number and impact (remote code execution), lacking retrieval utility. (2) *Domain knowledge is essential.* CVE-2016-10267 exemplifies this. The crucial cue for retrieval is *TIFF image*, which requires an understanding of TIFF images and how they are parsed to enable effective retrieval.

These issues mirror real-world scenarios where information sources are often unreliable and evaluate the practical effectiveness of different approaches.

Answering RQ2

Although vulnerability search is a challenging task, `BinQuery` achieves impressive improvement, reaching **nearly 4 times** the performance of SOTA on recall@1. In recall@50, `BinQuery` achieves **43.87%** accuracy, effectively aiding reverse engineers in real-world tasks compared to previous methods. Furthermore, our experiments show that even with information leaks masked, our method maintains strong performance, indicating its robustness in practical scenarios.

Table 4. Ablation Study of Training Strategies

Models	Setup			Result	
	Role-Based	FAC	SAC	MAP	recall@20
`BinQuery-BT`	✓			10.16(-47.00%)	28.33(-34.01%)
`BinQuery-BST`	✓	✓		13.21(-31.09%)	30.96(-27.88%)
`BinQuery-NoRole`		✓	✓	15.00(-21.75%)	37.55(-12.53%)
`BinQuery-Limit`	✓	✓	✓	**19.17**	**42.93**

[1] All results are obtained on the ViC dataset without query augmentation.
[2] All models are trained in a limited setup, introduced in Section 4.2.
[3] `BinQuery-BT` trained on binary and text modalities, function-level.
[4] `BinQuery-BST` trained on binary, source, and text modalities, function-level.
[5] `BinQuery-NoRole` trained without training text augmentation.
[6] Numbers in brackets show performance decline compared to `BinQuery-Limit`.

5.3 Source Code Supervision (Limited Setup) (RQ3)

We evaluate the effect of source code supervision on the ViC dataset and report the results in Table 4. By comparing the results of `BinQuery-BT` and `BinQuery-BST`, we find that incorporating source code supervision quantitatively increases MAP from 10.16 to 13.21, resulting in a roughly 30% improvement in overall performance, while recall@20 increased from 28.33 to 30.96. Overall, source code supervision can effectively improve performance on the NL-based BFR task.

Source code supervision boosts generalization ability. To explore the improvement mechanism, we record performance trends on the test set during training, plot the curve in Figure 4(b) and compare it with the results in Table 4 to gain insights. On the test set, the performance curves for `BinQuery-BST` and `BinQuery-BT` nearly overlap towards the end of training (Figure 4(b)), suggesting little difference within that specific data distribution. However, in the real-world scenarios represented by the ViC dataset (Table 4), source code supervision (`BinQuery-BST`) significantly improves results compared to direct binary-text alignment (`BinQuery-BT`). Without source code supervision, the model does not effectively learn the semantics of each modality due to information loss in both, reducing generalization in real-world scenarios.

Answering RQ3

Introducing source code as a supervisory signal significantly enhances the model's performance on the NL-based BFR task. Further analysis reveals that this improvement is due to source code supervision mitigating information loss across modalities, thereby improving the model's generalization capability.

5.4 Snippet-Level Alignment Training (Limited Setup) (RQ4)

We validate the effect of snippet-level training on the ViC dataset and report the results in Table 4. When we compare the `BinQuery-BST` and `BinQuery` rows in the table, we observe that using snippet alignment training improves the MAP score from 13.21 to 19.17 and increases the recall@20 from 30.96 to 42.93. This indicates that introducing snippet-level training can effectively improve the model's final performance in retrieval tasks. Comparing the impacts of different mechanisms on the final model performance, snippet-level alignment is the most influential during training.

The improvement comes from snippet-level semantics fitting practical retrieval scenarios better. For instance, a query like "String copy with quotes handling" is hard to learn at the function level because snippets often serve larger functionalities like encoding or printing, which aren't finely described in function-level training. Our approach splits functions into various granularities, enhancing performance in retrieval scenarios.

Answering RQ4

Snippet-level alignment provides finer semantic supervision, effectively enhancing NL-based BFR performance. It is the most crucial design in the training process compared to others.

5.5 Sharing Parameters Between Modalities (RQ5)

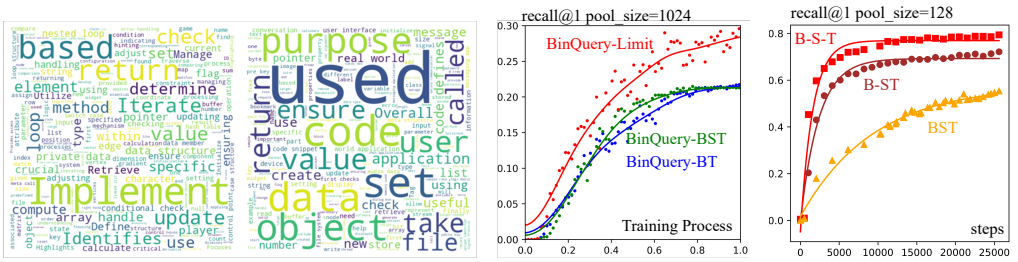
In Section 3.1.1, it raised a concern about whether to use different encoders for different modalities, and we discuss this issue here. We conduct the experiment in Figure 4(c) to study the effectiveness of sharing parameters across different modalities. The figure includes three settings. `B-S-L` means each modality uses separate models, as in `BinQuery`. `B-SL` shares one model between source and language modalities, while `BSL` shares one model across all modalities. Results show that `BinQuery`'s approach is best for the NL-based BFR task, with the fastest improvement and best final results. Here's the reasons:

(1) The binary modality has a unique distribution. For example, binary code includes jumps to addresses as immediate values, requiring special preprocessing and model designs [51, 53].

(2) Using the same model for different modalities requires learning their joint probability distribution, enhancing applicability but increasing training complexity, necessitating large data for fused models. With limited data and computational power, separate models for each modality achieve better results faster.

Answering RQ5

Employing individual encoders for different modalities is justified. In our experiment, `BinQuery` with separate encoders (`B-S-T`) achieves the fastest improvements and the best performance in the NL-based BFR task, outperforming models that share encoders across modalities.



(a) word cloud with and without role-based generation (b) model setup curve (c) model structure curve

Fig. 4. Role-based Description Generation and Test Curves during Training

5.6 Role-based Description Generation (Limited Setup) (RQ6)

We evaluate the contribution of our three-role description generation strategy to the system and briefly discuss its underlying mechanism. For comparison, we use a naive approach to generate source descriptions by asking the LLM to explain the source code directly, serving as the control, labeled as `BinQuery-NoRole`.

From table 4, comparing `BinQuery-NoRole` and `BinQuery-Limit`, we find that without the role-based description generation mechanism, MAP drops by 21.75% and recall@20 by 12.53%. This indicates that our role-based description generation plays a critical role.

To better understand this, we create word cloud diagrams for both the augmented and non-augmented data, shown in Fig 4(a). We use the Python `wordcloud` library with the hyperparameter `relative_scaling` set to 1.0. Before generating the word cloud, we remove the words *function* and *program* due to their high frequency. From the figure, we derive the following observation:

(1) Our approach generates more specific programming concepts such as *pointer*, *position*, *map*, and *flag*. In the control group, without specific instructions, the LLM tends to explain the functional intent of the source code, leading to less frequent appearance of such concepts.

(2) Our approach generates a dataset with a more balanced distribution, while the control group shows a higher frequency for certain words. A balanced distribution prevents the model from overfitting to specific expressions or formats.

Answering RQ6

Role-based description generation effectively improves NL-based BFR performance, primarily by incorporating additional programming concepts and achieving a more balanced data distribution.

5.7 Query Augmentation (Full Setup) (RQ7)

In Sections 5.1 and 5.2, we omitted discussions about query augmentation when presenting the results. In this section, we discuss the exact impact that the query augmentation mechanism brings. When tested on the ViC dataset, the query augmentation mechanism resulted in a slight performance loss (recall@50 from 84.12 to 83.33). However, when tested on the Magma dataset, the QA mechanism led to a significant improvement (MAP from 36.82 to 43.87).

5.7.1 Case Study. To explain this phenomenon, we manually analyzed several cases and attempted to identify the reasons for them.

Good Case from LLM Reasoning. In the description of CVE-2018-18557, it is stated that "the target function handles JBIG data of unrestricted size within a predefined memory area, leading to an out-of-bounds issue". The LLM, simulating an experienced reverse engineer, hypothesized that this is due to the content expanding after decoding, so it suggested "permitting it to decode and store the data in the predefined memory area," which aligns with the actual scenario. As a result, the target function's rank improved from **357th** to **1st**, accurately identifying the target.

Bad Case from Hallucination. We choose a query from the ViC dataset for demonstration. The target function is `import_real` from the `gbonds` project, and the original query is "PRIVATE. Import a file". It is so vague that it ranked only 33rd during retrieval. After enhancement by the LLM, the following details were added to the description, such as "processing its contents" and "storing the minimum values". Unfortunately, these details do not actually exist in the target function but were fabricated by the LLM. After the enhancement, the priority of the target function was reduced to the 170th position because these details do not exist within the function.

5.7.2 Domain Specific Expert Knowledge. By comparing the performance of general models in Tables 2 and 3, we find that they have narrowed the gap with specialized models in vulnerability search tasks and even surpass them in certain metrics, such as recall@50. As discussed in Section 5.6, real-world tasks often involve queries that contain specific domain concepts—for example, the TIFF format in our earlier example. General models have a natural advantage in domain knowledge due to their training on vast and diverse datasets. In contrast, specialized models, being smaller and trained on limited data, may lag behind. Since large language models (LLMs) are powerful general models, our proposed query augmentation mechanism offers an excellent opportunity to introduce domain-specific knowledge into the search process, effectively combining the strengths of both specialized and general models.

Answering RQ7

Query augmentation presents both pros and cons for NL-based BFR tasks. In simpler scenarios, it may slightly reduce performance due to LLM hallucinations. In real-world scenarios, LLM-enriched queries with domain-specific knowledge greatly boost the retrieval of relevant functions, especially in complex tasks. Overall, the advantages of query augmentation outweigh its disadvantages.

6 Discussion

In this section, we will conduct some additional discussions. First, we will discuss some limitations related to our method. Then, we will present the relationship between our work and previous works to address potential concerns regarding compatibility and innovativeness. Finally, we will offer an outlook on possible future work that could build upon our contributions.

6.1 Limitations

6.1.1 Snippet Extraction Effectivness. when constructing our dataset, we extract snippets from functions using a combination of heuristic rules and LLM inference. However, the lack of objective evaluation metrics for snippet quality leaves room for improvement. Our proposed method has already demonstrated the effectiveness of snippet alignment training. In the future, we can further enhance the results by refining the snippet extraction algorithm.

6.1.2 Illusion in Query Augmentation. In the NL-based BFR scenario, we leverage the reasoning capabilities of LLMs, significantly enhancing performance. However, due to the hallucinations of

LLMs, there are slight performance drops in specific scenarios. Future work could mitigate this by refining prompt quality or fine-tuning LLMs, potentially leading to further performance increases.

6.2 Relationship with Related Works

Our contributions are orthogonal to existing research, allowing them to integrate seamlessly with prior advancements and enhance overall performance. For the base model of binary code [52, 53], our proposed method can leverage their improvements by enhancing the quality of binary embeddings to achieve better retrieval performance. For data augmentation efforts, such as ViC [17], focus on building more robust datasets to enhance model performance. Our work also benefits from these augmented datasets, ultimately leading to improved results. For example, our method can complement BCSD's binary code encoder architectures and ViC's dataset construction enhancements. This synergy creates a cumulative effect, leveraging the strengths of each approach to achieve superior results.

6.3 Future Works

6.3.1 Binary Code RAG. Retrieval-augmented generation integrates external knowledge by combining information retrieval and text generation. Binary code is viewed as a new knowledge resource in security research. Our method enables LLMs to retrieve relevant binary code from large repositories, generating high-quality responses. This capability is particularly useful in reverse engineering and security analysis.

6.3.2 NL-based Software Supply Chain Analysis. NL-based BFR enables us to go beyond the limitations of traditional software supply chain analysis. By incorporating natural language processing, our approach analyzes code and also correlates information from various modalities, such as documentation, licenses, and community feedback. This cross-modal analysis helps us identify and understand complex relationships and risks within the supply chain, providing a more accurate and comprehensive security assessment.

7 Conclusion

In conclusion, we present `BinQuery`, a novel framework that tackles the key challenges of NL-based BFR. `BinQuery` leverages source code as a supervisory signal and introduces snippet-level granularity during training to improve retrieval performance. Additionally, role-based description generation and LLM-based query augmentation enhance the system's adaptability to real-world queries. Our evaluations on two representative datasets demonstrate `BinQuery`'s superior performance across all metrics compared to existing state-of-the-art methods. These contributions provide a robust foundation for further research in binary analysis, with potential applications in areas such as vulnerability detection, reverse engineering, and large-scale software supply chain analysis.

Data Availability

We provide the code for the data preparation, model architecture and model training in repository https://github.com/BinQueryGroup/OpenBinQuery. Additionally, we offer a demo of the model in action and provide sample datasets to examine the details of `BinQuery`.

Acknowledgments

We would like to sincerely thank all the reviewers for their insightful feedback that greatly helped us to improve this paper. Additionally, special thanks are extended to Zihan Sha, Yi Yang, Ying Cao and Dandan Xu for their invaluable comments. The Institute of Information Engineering authors are

supported in part by the National Natural Science Foundation of China (U24A20236, 92270204), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118). The authors from Tsinghua University are supported in part by the National Natural Science Foundation of China (U24A20337) and the Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd. Support specifically for Bolun Zhang is provided by the Key Laboratory of Network Assessment Technology of Chinese Academy of Sciences under Grant CXJJ-22S022.

References

[1] Eli Bendersky. 2024. pyelftools: A pure-Python library for parsing ELF and DWARF. https://github.com/eliben/pyelftools. Accessed: 2024-10-31.

[2] Max Brunsfeld. 2018. Tree-sitter: An Incremental Parsing System for Programming Tools. https://tree-sitter.github.io/tree-sitter/. Accessed: 2024-10-26.

[3] Francesco Buonamici, Monica Carfagni, Rocco Furferi, Lapo Governi, Alessandro Lapini, and Yary Volpe. 2018. Reverse engineering modeling methods and tools: a survey. *Computer-Aided Design and Applications* 15, 3 (2018), 443–464.

[4] Canonical Ltd. 2024. *Ubuntu.* Canonical Ltd. Available at https://ubuntu.com.

[5] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. 2023. A Survey of Chain of Thought Reasoning: Advances, Frontiers and Future. *ArXiv* abs/2309.15402 (2023). doi:10.48550/arXiv.2309.15402

[6] Leon O Chua. 1997. CNN: A vision of complexity. *International Journal of Bifurcation and Chaos* 7, 10 (1997), 2219–2425.

[7] Victor Cochard, Damian Pfammatter, Chi Thang Duong, and Mathias Humbert. 2022. Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. IEEE, 60–73.

[8] DWARF Debugging Information Format Committee. 2024. DWARF Debugging Standard. https://dwarfstd.org/ Accessed: 2024-10-26.

[9] Roland Croft, Dominic Newlands, Ziyu Chen, and Muhammad Ali Babar. 2021. An Empirical Study of Rule-Based and Learning-Based Approaches for Static Application Security Testing. In *ESEM '21: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 11-15, 2021*, Filippo Lanubile, Marcos Kalinowski, and Maria Teresa Baldassarre (Eds.). ACM, 8:1–8:12.

[10] darx0r. 2024. Stingray: Static Analysis for Shellcode. Available at: https://github.com/darx0r/Stingray.

[11] Debian Project. 2024. *Advanced Package Tool (APT).* Debian Project. Available at https://wiki.debian.org/Apt.

[12] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation. *ArXiv* abs/2308.01861 (2023). doi:10.48550/arXiv.2308.01861

[13] Guo et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv* (2024). https://arxiv.org/abs/2406.11931 Accessed: 2024-10-26.

[14] Guo et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *arXiv* (2024). https://arxiv.org/abs/2401.14196 Accessed: 2024-10-26.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics. doi:10.18653/v1/2020.findings-emnlp.139

[16] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 896–899. doi:10.1145/3238147.3240480

[17] Zeyu Gao, Hao Wang, Yuanda Wang, and Chao Zhang. 2024. Virtual Compiler Is All You Need For Assembly Code Search. In *Proceedings of the 62st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

[18] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.

[19] Alex Graves and Alex Graves. 2012. Long short-term memory. *Supervised sequence labelling with recurrent neural networks* (2012), 37–45.

[20] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, Vol. 2. IEEE, 1735–1742.

[21] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. doi:10.1145/3428334

[22] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. 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*.

[23] Hex-rays. 2023. Hex-Rays - State. https://hex-rays.com/products/ida/support/idadoc/1379.shtml

[24] Hex-Rays. 2024. *IDA Pro: Interactive Disassembler*. Version 9.0, Available at: https://hex-rays.com/ida-pro.

[25] Dongxing Huang, Yong Tang, Yi Wang, and Shuning Wei. 2018. Toward efficient and accurate function-call graph matching of binary codes. *Concurrency and Computation: Practice and Experience* 31 (2018). doi:10.1002/cpe.4871

[26] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1631–1645. doi:10.1145/3548606.3560612

[27] Clemens Jonischkeit and Julian Kirsch. 2017. Enhancing Control Flow Graph Based Binary Function Identification. (2017), 8. doi:10.1145/3150376.3150384

[28] Chariton Karamitas and A. Kehagias. 2018. Efficient features for function matching between binary executables. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), 335–345. doi:10.1109/SANER.2018.8330221

[29] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. *Advances in neural information processing systems* 33 (2020), 18661–18673.

[30] Jihun Kim and Jonghee M. Youn. 2017. Malware behavior analysis using binary code tracking. *2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT)* (2017), 1–4. doi:10.1109/CAIPT.2017.8320724

[31] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), 628–639. https://api.semanticscholar.org/CorpusID:202676778

[32] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281* (2023).

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

[34] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. \(\alpha\)Diff: Cross-Version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 667–678. doi:10.1145/3238147.3238199

[35] Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang. 2024. On llms-driven synthetic data generation, curation, and evaluation: A survey. *arXiv preprint arXiv:2406.15126* (2024).

[36] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bkg6RiCqY7

[37] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]

[38] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society.

[39] National Security Agency. 2024. *Ghidra: Software Reverse Engineering Framework*. Available at: https://ghidra-sre.org/.

[40] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[41] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 1dVul: Discovering 1-Day Vulnerabilities through Binary Patches. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), 605–616. doi:10.1109/DSN.2019.00066

[42] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral clustering of HTTP-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 26.

[43] polymorf. 2020. IDA Pro plugin to find crypto constants. https://github.com/polymorf/findcryptyara

[44] radare project. 2024. *radare2: Open-source Reverse Engineering Framework*. Available at: https://rada.re/n/.

[45] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8748–8763.

[46] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[47] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[48] Ruixiang Tang, Xiaotian Han, Xiaoqian Jiang, and Xia Hu. 2023. Does synthetic data generation of llms help clinical text mining? *arXiv preprint arXiv:2303.04360* (2023).

[49] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[50] Vector 35 Inc. 2024. *Binary Ninja: Binary Analysis Platform*. Available at: https://binary.ninja/.

[51] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision. *arXiv preprint arXiv:2402.16928* (2024).

[52] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. *arXiv preprint arXiv:2402.18818* (2024).

[53] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-Aware Transformer for Binary Code Similarity Detection. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 1–13. doi:10.1145/3533767.3534367

[54] Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Improving text embeddings with large language models. *arXiv preprint arXiv:2401.00368* (2023).

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, E. Chi, F. Xia, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv* abs/2201.11903 (2022).

[56] Yao Xin-lei. 2012. Program Malicious Behavior Recognizing Method Based on Model Checking. *Computer Engineering* (2012).

[57] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 363–376.

[58] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020). doi:10.1145/3395363.3397361

[59] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2019. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Transactions on Software Engineering* 45 (2019), 1125–1149. doi:10.1109/TSE.2018.2827379

[60] Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, and Hari Sundaram. 2023. CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation. *ArXiv* abs/2311.08588 (2023). doi:10.48550/arXiv.2311.08588

[61] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.).

[62] Yifan Zhang, Chen Huang, Yueke Zhang, Kevin Cao, Scott Thomas Andersen, Huajie Shao, Kevin Leach, and Yu Huang. 2022. COMBO: Pre-Training Representations of Binary Code Using Contrastive Learning. *CoRR* abs/2210.05102 (2022). arXiv:2210.05102

[63] Dongdong Zhao, Hong Lin, Linjun Ran, Mushuai Han, Jing Tian, Liping Lu, Shengwu Xiong, and Jianwen Xiang. 2019. CVSkSA: cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph. *Software Quality Journal* 27 (2019), 1045–1068.

[64] Yujie Zhao, Zhanyong Tang, Guixin Ye, Dongxu Peng, Dingyi Fang, Xiaojiang Chen, and Z. Wang. 2020. Semantics-aware obfuscation scheme prediction for binary. *Comput. Secur.* 99 (2020), 102072. doi:10.1016/J.COSE.2020.102072

Received 2024-10-31; accepted 2025-03-31