KEENHash: Hashing Programs into Function-Aware Embeddings for Large-Scale Binary Code Similarity Analysis

ZHIJIE LIU, ShanghaiTech University, China
QIYI TANG, Tencent Security Keen Lab, China
SEN NIE, Tencent Security Keen Lab, China
SHI WU, Tencent Security Keen Lab, China
LIANG FENG ZHANG, ShanghaiTech University, China
YUTIAN TANG[*], University of Glasgow, United Kingdom

Binary code similarity analysis (BCSA) is a crucial research area in many fields such as cybersecurity. Specifically, function-level diffing tools are the most widely used in BCSA: they perform function matching one by one for evaluating the similarity between binary programs. However, such methods need a high time complexity, making them unscalable in large-scale scenarios (e.g., $1/n$-to-n search). Towards effective and efficient program-level BCSA, we propose KEENHash, a novel hashing approach that hashes binaries into program-level representations through large language model (LLM)-generated function embeddings. KEENHash condenses a binary into one compact and fixed-length program embedding using K-Means and Feature Hashing, allowing us to do effective and efficient large-scale program-level BCSA, surpassing the previous state-of-the-art methods. The experimental results show that KEENHash is at least 215 times faster than the state-of-the-art function matching tools while maintaining effectiveness. Furthermore, in a large-scale scenario with 5.3 billion similarity evaluations, KEENHash takes only 395.83 seconds while these tools will cost at least 56 days. We also evaluate KEENHash on the program clone search of large-scale BCSA across extensive datasets in 202,305 binaries. Compared with 4 state-of-the-art methods, KEENHash outperforms all of them by at least 23.16%, and displays remarkable superiority over them in the large-scale BCSA security scenario of malware detection.

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

Additional Key Words and Phrases: BCSA, LLM, Program, Clone Search

ACM Reference Format:
Zhijie Liu, Qiyi Tang, Sen Nie, Shi Wu, Liang Feng Zhang, and Yutian Tang. 2025. KEENHash: Hashing Programs into Function-Aware Embeddings for Large-Scale Binary Code Similarity Analysis. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA036 (July 2025), 30 pages. https://doi.org/10.1145/3728911

1 Introduction

Binary code similarity analysis (BCSA) is a crucial research area in the fields of cybersecurity, software engineering, and reverse engineering [16, 26, 28, 44, 49, 55, 72, 95]. It involves the comparison of binary code (e.g., program and function) to identify similarities and differences among them, which is applied to a wide range of applications, including code clone search [16, 26, 87],

[*] Yutian Tang (yutian.tang@glasgow.ac.uk) is the corresponding author.

Authors' Contact Information: Zhijie Liu, ShanghaiTech University, Shanghai, China, liuzhj2022@shanghaitech.edu.cn; Qiyi Tang, Tencent Security Keen Lab, Shanghai, China, work_t71@163.com; Sen Nie, Tencent Security Keen Lab, Shanghai, China, snie@tencent.com; Shi Wu, Tencent Security Keen Lab, Shanghai, China, shiwu@tencent.com; Liang Feng Zhang, ShanghaiTech University, Shanghai, China, zhanglf@shanghaitech.edu.cn; Yutian Tang, University of Glasgow, Glasgow, United Kingdom, yutian.tang@glasgow.ac.uk.

ACM 2994-970X/2025/7-ARTISSTA036
https://doi.org/10.1145/3728911

malware analysis [16, 25, 28, 44, 46, 83], vulnerability detection [32, 86, 93], software composition analysis [49, 50, 97], and so forth. Among these analyses, program-level BCSA [42] stands out as a powerful technique that can analyze and compare similarities between binaries (i.e., binary programs) [16, 44, 83] which are larger and more complex objects than functions (i.e., function-level BCSA, evaluates the similarity between binary functions). This type of analysis includes 1-to-1 and 1/n-to-n similarity comparisons of binaries. Particularly, in the **vital large-scale scenarios** (i.e., 1/n-to-n), such as program clone search [16, 44], malware detection [25, 83], and threat intelligence [20, 58, 60, 69, 83], it must be both accurate and efficient for evaluating huge amounts of similarities among binaries.

Unfortunately, in spite of the significance on program-level BCSA, none of the previous works could achieve satisfying results on the large-scale 1/n-to-n comparisons. Specifically, most previous works on BCSA [16, 26, 28, 84, 87] focus on *function-level* similarity: they generate one embedding for each function in a binary, and iteratively compare these embeddings with embeddings from another binary, to determine the (similar) function-matched proportion (i.e., similarity) between two binaries. ***Limitation.1:*** Such a matching approach has an unscalable $O(nm^3)$ time complexity (n is the number of comparisons among binaries and m is the number of functions to one binary) [57], making it impossible to be applied to large-scale BCSA, even with faster heuristic strategy [19, 28, 58] (Sec. 4.3). ***Limitation.2:*** $\mathrm{PSS_O}$ [16] is the only recent work focusing on large-scale program-level BCSA. However, it only uses simple features (e.g., call graph and edge counts of control flow graphs) for generating program-level embedding without considering the rich semantics in binary functions, leading to poor performance in large-scale experiments (Sec. 4.4 to 4.7).

To fill the aforementioned gaps, we propose KEENHash, a novel hashing approach to hash binaries into fixed-length representations for *effective and efficient* BCSA. To achieve this goal, KEENHash condenses a binary into one compact and fixed-length program embedding (e.g., 8KB). It leverages K-Means and Feature Hashing technique [89] to classify and represent decompiled pseudo functions (with function embeddings) into a bit-vector for approximating function matching, thereby allowing us to do large-scale (1-to-n and n-to-n) program-level BCSA that previous function-level methods fail to do. Specifically, KEENHash involves three stages to hash a binary: Function Embedding Generation (Sec. 3.3), Program Embedding Generation (Sec. 3.4), and Similarity Evaluation (Sec. 3.5). In the first stage, we train a large language model (LLM) to encode the decompiled pseudocode of each function (i.e., pseudo function) into corresponding embedding, in which the rich semantics of pseudo functions are extracted and maintained. Taking the idea of classifying similar functions into the same class and thus representing function matching, in the second stage, we cluster extensive source functions using K-Means (an effective clustering algorithm to generate labels), and then feed the label information (derived from K-Means through classification) of the pseudo functions into the Feature Hashing module to produce the compact vector (KEENHash-stru, Sec. 3.4.1) that can represent the whole binary. By applying such K-Means and Feature Hashing techniques, an original binary with the size of MBs of pseudo function embeddings can be condensed into only 8KB, significantly accelerating the (function matching) process of large-scale program-level BCSA. Furthermore, massive code reuse (e.g., > 70%), a widespread practice in software development [50], can misleadingly increase the similarity between non-same-class binaries (Sec. 4.4) and weaken indirectly the significance of binaries' unique feature parts. We introduce KEENHash-sem (Sec. 3.4.2) from the perspective of program semantics with the weighted average of function embeddings. Comparing these two approaches, KEENHash-stru is better in code obfuscation scenarios (Sec. 4.5), while KEENHash-sem is more effective in massive code reuse ones (Sec. 4.4). Finally, in the third stage, we use the compact program embedding (KEENHash-stru or sem) representing the information of the whole binary to perform the large-scale BCSA.

Our experiments illustrate that KEENHash can maintain an effective performance on function matching while faster than SigmaDiff [34] and BinDiffMatch [19, 58], the two state-of-the-art binary diffing tools for function matching with heuristic strategy, by 1,254 and 215 times on the average of each matching between two binaries (Sec. 4.3). Furthermore, in a large-scale scenario with 5.3 billion similarity evaluations among binaries (Sec. 4.6), *a typical workload within one day* [25, 81, 83], KEENHash takes at most only 395.83 seconds, while SigmaDiff and BinDiffMatch will cost 323 and 56 days, which is unscalable. Additionally, on program clone search (a representative task for large-scale BCSA), KEENHash shows effective performance on a total of 202,305 Linux/Windows binaries with real-world cases across various compile environments including optimization levels, compilers, architectures, and obfuscations, significantly outperforming all other state-of-the-art BCSA methods including PSS$_{\text{O}}$ (Sec. 4.4 to 4.6) by at least 23.16% (Sec. 4.6). Moreover, KEENHash displays remarkable superiority over other methods in the large-scale BCSA security scenario of malware detection (Sec. 4.7).

To summarize, our paper makes three contributions:

- We propose KEENHash, based on LLM-generated function embeddings, taking two perspectives of function matching and program semantics to hash any binary to a compact and fixed-length representation for effective and efficient large-scale program-level BCSA;
- We evaluate KEENHash on function matching, against the state-of-the-art tools SigmaDiff and BinDiffMatch. KEENHash is able to maintain an effective performance and demonstrates a significant speed advantage, being 1,254 and 215 times faster (Sec. 4.3). In a large-scale scenario with 5.3 billion similarity evaluations, it takes at most 395.83 seconds while SigmaDiff and BinDiffMatch will cost 323 and 56 days (Sec. 4.3 and 4.6), respectively;
- We evaluate KEENHash on program clone search with 5 large-scale datasets in a total of 202,305 Linux/Windows binaries. Experimental results show that KEENHash outperforms all other state-of-the-art BCSA methods including PSS$_{\text{O}}$ by at least 23.16% (Sec. 4.4 - 4.6), and displays remarkable superiority over them in the large-scale security scenario of malware detection (Sec. 4.7).

Online SDK Availability for K(EEN)Hash: https://www.binaryai.cn.

2 Preliminary

2.1 Problem Definition

Program-level BCSA is a task to evaluate the similarity between two binary programs. Given two binaries q and r, the similarity evaluation process is presented as follows:

Definition 1: (Similarity Evaluation). For q and r, the similarity evaluation process measures the similarity score between them, which is formulated as follows:

$$Similarity\ Score = F(Enc(q), Enc(r)) \tag{1}$$

Where function Enc encodes a program q or r based on their information to a representation; and, function F further measures their similarity score. The larger the similarity score, the more similar the two programs q and r are. This definition also holds between functions.

In this study, we focus on large-scale program-level BCSA. To evaluate the performance of methods, we utilize the program clone search [16] as the evaluation task. *Notably, program clone search is also one of the critical large-scale scenarios in threat intelligence and BCSA (e.g., finding similar binaries to unknown malware for better understanding)* [83].

Definition 2: (Same Class Program). Programs in the same class are clones of each other. A clone c of a program p is defined as that c is compiled from the same or different code version source code to p with various compilation environments. For example, c compiled from source code s

using GCC v13.2 with O0 is a clone of p compiled from s using GCC v10.5 with O3; and, c compiled from s is a clone of p compiled from s' where s is another version to s' (e.g., malware variants).

Definition 3: (Program Clone Search). Given an unknown query binary $q \in Q$, a query program dataset Q, and a program repository dataset R containing a large amount of unknown or known binaries r, the task of program clone search is to input q and retrieve the most Top-k similar binaries $\{r_1, r_2, ..., r_k | r_i \in R\}$ from R, ranked by their similarity scores. The more binaries of the same class to q are returned and the higher they rank among the Top-k retrieved binaries, the better the performance of program-level BCSA methods.

The clone search procedure is presented as follows:

❶ **Repository Preprocessing.** Before retrieving Top-k similar programs, the repository R needs to be built first. Additionally, for each program $r \in R$, the program-level BCSA method $Enc(r)$ to get its representation, such as embedding, for subsequent similarity comparisons in retrieving;

❷ **Query Preprocessing.** Given a query program q, like the first step ❶, the program-level BCSA method $Enc(q)$ to get its representation;

❸ **Retrieving.** We send q to the similarity search system (e.g., FAISS [27] and Milvus [41, 88]) built based on R to retrieve the most Top-k similar programs from R by leveraging program representations with function F. The search procedure has many indexes [88] such as `FLAT`, `HNSW`, `IVF_FLAT`, and so forth. In this study, to accurately evaluate the performance of program-level BCSA methods, we use the `FLAT` index by default through brute force search.

2.2 Motivation

In this section, we introduce the motivation behind the design of KEENHash for **large-scale program-level BCSA**. Binary diffing [28] is a widely used method to identify differences between two binaries, enabling various analyses. Where, function-level binary diffing (i.e., similar function matching) [7, 52, 58] is popular since functions represent sufficiently detailed information about the decomposition of the program functionality. Therefore, using the results of (similar) function matching proportion as a measure can effectively capture the structural similarity between binaries. However, the approach introduced in Sec. 1 costs a $O(nm^3)$ time complexity for n similarity evaluations. Even with heuristic strategies, scalability remains unsatisfiable on large-scale BCSA scenarios (e.g., BinDiffMatch takes 56 days to evaluate 5.3 billion similarities. See Sec. 4.3). Nevertheless, considering that function matching is essentially a form of classification, grouping similar functions into the same class which is dynamically expanding. Thus, as long as a precise and comprehensive classifier is used to classify functions, the matching is equivalent to and transformed into classification. In addition, by encoding classified results as positions (e.g., Feature Hashing [89]) in a vector, n similarity comparisons only require $O(n) \ll O(nm^3)$ time complexity where the dimension (length) of the vector is fixed. With this property, we can hash any binary, with a varying number of functions, to a compact and fixed-length vector, with hardware acceleration, to support fast comparison in large-scale scenarios. We propose KEENHash-stru based on this insight (Sec. 3.4.1). Moreover, considering that massive code reuse (a widespread practice) can misleadingly increase the similarity between two non-same-class binaries (Sec. 1 and 3.4.2), we introduce KEENHash-sem from the perspective of program semantics in Sec. 3.4.2 to mitigate this issue in large-scale BCSA.

2.3 Assumption of Processed Binary

In this section, we outline the assumption of the processed binary for KEENHash. Binary packing [25] is the technique for compressing original binaries to reduce their size and obfuscate their contents. Packed binaries are decompressed in memory during runtime before the original content in binaries is executed, and are difficult to accurately decompress statically [67]. Directly using decompilers, like Ghidra [7] or IDA Pro [74], may not identify and analyze all functions within

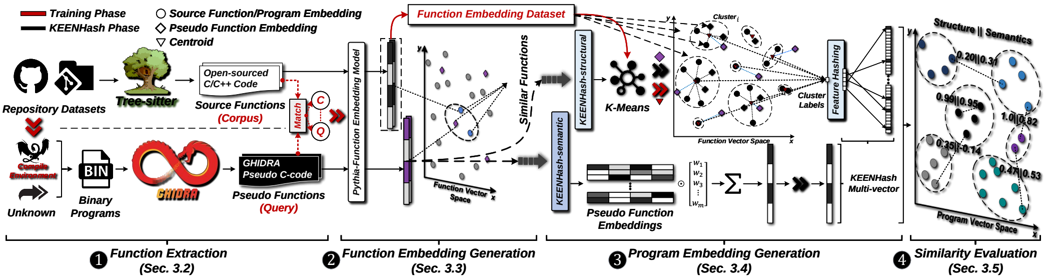

Fig. 1. Workflow of KEENHash.

packed binaries. KEENHash hashes programs based on decompiled results through Ghidra (see Sec. 3.2) which may be affected by packing techniques. Additionally, how to unpack any packed binary goes beyond the scope of our paper. Thus, to avoid biases, in this study, we divide the range of binaries that KEENHash can process into programs without packing, or packed programs can be unpacked easily (i.e., UPX [80]).

3 Methodology

In this section, we introduce the workflow of KEENHash for large-scale program-level BCSA.

3.1 Overview

As shown in Fig. 1, the workflow of KEENHash includes four phases: ❶ function extraction (Sec. 3.2), ❷ function embedding generation (Sec. 3.3), ❸ program embedding generation (Sec. 3.4), and ❹ similarity evaluation (Sec. 3.5). Specifically, to hash a given binary to a multi-vector for similarity analysis, ❶ KEENHash extracts the C-like pseudocode functions in the binary through Ghidra [7]. Then, ❷ KEENHash leverages a large language model (LLM) to generate the embeddings of these pseudo functions. Next, after obtaining the pseudo function embeddings, ❸ KEENHash leverages the function matching-based structural feature (KEENHash-stru) as well as the function-intrinsic semantic feature (KEENHash-sem) to transform pseudo function embeddings into compact and fixed-length structural and semantic program embeddings, respectively, forming a multi-vector. Where the second is proposed by considering huge cose reuse cases (see Sec. 4.4). Eventually, ❹ KEENHash leverages the corresponding and respective similarity evaluation metrics for the generated structural and semantic embedding to compare the similarities of the given binary with other program embeddings, to support various large-scale program-level BCSA tasks.

3.2 Function Extraction

The initial step of KEENHash involves extracting C-like pseudocode functions (pseudo functions) from binaries where these pseudo functions are translated from corresponding binary functions in the binaries through Ghidra [7]. The C-like pseudocode, rather than others (e.g., assembly code, byte code, and so forth) [26, 84, 87], shields the details of assembly instructions from various architectures (i.e., unify to the same code format) and is close to the code in high-level languages such as C/C++ (i.e., one model can easily encode both source and pseudo functions in the same tokenization). Moreover, to train the subsequent LLM (function embedding model) for generating function embeddings, source functions from open-sourced C/C++ projects are extracted, with their pseudo functions compiled through various compile environments. Massive C/C++ source functions are also extracted for KEENHash-stru (see Sec. 3.4.1) when generating program embeddings. Contrary to the intuitive imagination of KEENHash only hashing binaries, we do not adopt a binary-to-binary training method to LLM like previous works [26, 87] (i.e., with only pseudo/binary functions), but instead use a ***source-to-binary*** approach (i.e., with both source and pseudo functions). The reason is three-fold: (1) the language of C-like pseudocode is close to the languages in source code from C/C++ projects; (2) a source function acts as an anchor to many corresponding binary functions compiled from different compile environments; and (3) it is essential to support generating massive source function embeddings instead of pseudo ones (decompilation for binaries is time and resource-consuming) for KEENHash-stru (see Sec. 3.4.1). Additionally, we only choose C/C++ open-sourced projects for fast construction of the automatic compilation pipeline inspired by jTrans [87], due to compilation compatibility reasons. The specific function extraction process is listed as follows:

3.2.1 Training Phase. KEENHash performs function extraction and ground-truth matching.
- **Source Function Extraction.** The process extracts C/C++ source functions from open-sourced C/C++ projects, such as GitHub repositories [35]. For each project, we collect all the C/C++ source code files across all versions through `git tags`. All source code files are deduplicated with the sha256 hash values. We also leverage tree-sitter [79] to parse these files and extract all unique source functions (through the sha256 hash values of their content without comments and whitespaces) with line numbers in files from them. The mapping relationships among projects, project versions, source code files, line numbers, and source functions are preserved during the extraction.
- **Pseudo Function Extraction.** For a given binary, we leverage Ghidra [7] to decompile it and extract all its binary functions that are translated to the language of C-like pseudocode (i.e., pseudo functions). Moreover, the relative virtual addresses (*rva*) to the binary functions are also extracted.
- **Source and Pseudo Function Matching.** Given a C/C++ project and corresponding compiled (unstripped) binaries through one specific compile environment (e.g., <GCC v13.2, O3, x86, 64-bit>), the matching process matches the source functions with their pseudo ones. A matched pair of the source and pseudo functions is an invertible mapping of the source one and its compiled binary one. Specifically, we compile the project and generate the debugging information (DWARF [45]). Then, we perform pseudo function extraction to extract the mapping between pseudo functions and their *rva*. Meanwhile, we parse the debugging information to extract the mapping between *rva* and corresponding source files with line numbers. After further performing the source function extraction to the project and getting the third mapping, we can merge these mappings to get the 1-to-n matching from the source functions to the pseudo ones. Moreover, we use sha256 hashed values to deduplicate pseudo functions for each source. By performing matching and deduplication on extensive C/C++ projects and binaries, we get a *Corpus* dataset C and a *Query* dataset Q where they contain matched and unique source functions, and matched pseudo functions, respectively. The matching between C and Q is a 1-to-n mapping across binaries in one project, in different projects, and through various compile environments. C and Q are further used for training the subsequent function embedding model.

3.2.2 Hashing Phase. Only binaries are processed to extract pseudo functions. Specifically, given a binary, we leverage Ghidra to decompile it and extract its pseudo functions, preparing to generate their function embeddings.

3.3 Function Embedding Generation

The foundation of KEENHash is the function embedding model for generating function embeddings to both source and pseudo functions (Sec. 3.2) for subsequent function-aware program embedding generation. The objective of the model is to generate function representations (embeddings) such that similar source and pseudo functions are gathered naturally in the vector (embedding) space. Conversely, dissimilar functions remain distanced from each other. To place the representations of both kinds of functions in the same space, we train our model based on the pairs of matched source and pseudo ones extracted from Sec. 3.2.1. While the grammar of source and pseudo code is similar [7], significant differences can still exist in the code due to various language features (e.g., function inlining [48]), compile environments (e.g., optimization level [26]), and decompilation (e.g., accessing data members and functions [7]), resulting in different formats. Recent work [33, 70] shows that existing LLMs, trained on code in various languages, can provide the capacity to understand and discriminate the intricate details and similarities of code syntax and semantics across different formats [9, 59, 98]. Therefore, we leverage LLM to overcome this issue and generate function embeddings. Instead of training a model from scratch, we use a pre-trained one for the transfer of knowledge [40, 70] and further fine-tune it on the pairs of matched functions through contrastive learning [73, 99]. Through this approach, we enable the LLM to draw similar functions closer together while pushing dissimilar ones farther apart. Specifically, we leverage `Pythia-410M` (contain 410M parameters) [17, 31], a transformer-based language model widely adopted by the research community, as the initialized base model for further fine-tuning. Furthermore, we highlight that our function embedding model differs from existing state-of-the-art ones, such as jTrans [87] and CLAP [84], in its ability to map both source and pseudo (binary) functions into the same space, while theirs only support binary ones. Our model is also more effective than theirs, focusing on binary function embeddings and correspondingly generated program embeddings (see [54]).

3.3.1 Training Phase. In the training phase, we further fine-tune `Pythia-410M` in a supervised manner on the pairs of matched functions through contrastive learning. Contrastive learning [24] is a technique, engaging in-batch negative samples, for a model to learn an embedding space where similar sample pairs stay close to each other while dissimilar ones are far apart, leading to better performance on discriminating functions [77]. In particular, we leverage Contrastive Language-Image Pre-training (CLIP) [73, 99] for fine-tuning our model due to the different code formats of source and pseudo functions. Specifically, ❶ we first perform tokenization on all source and pseudo functions in the training dataset that converts them into sequences of tokens. ❷ In the training epochs, batches are generated randomly and dynamically for more effective learning [99]. Thus, to generate one batch, we randomly sample N pairs of matched similar source and pseudo functions from C and Q with the near-deduplication procedure used in StarCoder [59] for more diverse training data, where functions between pairs are considered dissimilar. We pass the tokenized sequences of the $2N$ functions to `Pythia-410M` and obtain their embeddings by extracting the n-dimensional output of the last hidden layer of the model where $n = 1024$ (each dimension in `float32`, i.e., 4 bytes) [31]. Here, we denote $\mathbf{e}_i^s$ and $\mathbf{e}_j^p \in \mathbb{R}^n$ as the embeddings of the i_{th} and j_{th} source and pseudo functions, respectively. Furthermore, $\mathbf{e}_i^s$ and $\mathbf{e}_j^p$ are considered a match (i.e., positive pair) if i is equal to j; otherwise, they are deemed unmatched (i.e., negative pair). ❸ In each batch, CLIP evaluates the $N \times N$ cosine similarity matrix [99] between all the pairs of functions based on their embeddings. ❹ The training objective is to generate function embeddings in such a way that the similarity values of N positive pairs are maximized, while the similarity values of the $N \times (N-1)$ negative pairs are minimized. Therefore, we apply the softmax loss for language image pre-training [99], across source and pseudo functions, to the previously generated cosine similarity matrix. The specific loss function is defined as follows:

$$\mathcal{L} = -\frac{1}{2N}\sum_{i=0}^{N-1}\Big(\overbrace{\log \frac{e^{t\mathbf{x}_i\cdot\mathbf{y}_i}}{\sum_{j=0}^{N-1} e^{t\mathbf{x}_i\cdot\mathbf{y}_j}}}^{\text{source}\rightarrow\text{pseudo softmax}} + \overbrace{\log \frac{e^{t\mathbf{x}_i\cdot\mathbf{y}_i}}{\sum_{j=0}^{N-1} e^{t\mathbf{x}_j\cdot\mathbf{y}_i}}}^{\text{pseudo}\rightarrow\text{source softmax}} \Big) \tag{2}$$

Where $\mathbf{x}_i = \mathbf{e}_i^s / \|\mathbf{e}_i^s\|_2$ and $\mathbf{y}_j = \mathbf{e}_j^p / \|\mathbf{e}_j^p\|_2$; t is a freely learnable parameter for scaling logits [73].

3.3.2 Hashing Phase. Given a bunch of pseudo functions extracted from a binary, KEENHash leverages the function embedding model to generate corresponding function embeddings, for subsequent program embedding generation.

3.4 Program Embedding Generation

Functions are self-contained code modules designed to perform specific tasks, and their combination constitutes the functionality and representation of a program. The core of KEENHash lies in integrating the functions in a binary to generate a compact and fixed-length embedding that represents the binary, for large-scale BCSA. Therefore, we approach it from two perspectives, respectively: ❶ program structure (i.e., the function matching between binaries) and ❷ program semantics (i.e., amplification of unique feature semantics between binaries when comparison). As mentioned in Sec. 2.2, for ❶, the extent to which functions match between two binaries can serve as a metric for assessing their similarity. However, the time complexity of direct matching is prohibitively high (Sec. 2.2). Thus, we transform the function matching problem into a classification one, to achieve matching at a much lower time complexity. As for ❷, due to the widespread practice of massive code reuse [49, 50, 64, 90], it often leads to the structure (e.g., function matching, call graph, and so forth) of two binaries appearing very similar. This results in difficulty in distinguishing binaries (in large-scale scenarios) in the same or different classes but with similar structures (see example in Sec. 4.4) since the significance of the unique feature parts is indirectly weakened. Therefore, we explore the integration of function embeddings by capturing the semantic differences and maximizing the unique features among functions to effectively reflect the overall program semantics. We denote the first method as KEENHash-stru (Sec. 3.4.1) and the second as KEENHash-sem (Sec. 3.4.2). Together, their respective program embeddings combine into multi-vectors and can be selectively utilized based on specific conditions. Moreover, we highlight that both methods should be in an unsupervised manner due to the prohibitive costs of crafting large-scale and diverse labeled training datasets of binaries for real-world scenarios and generalization [16, 28].

3.4.1 Structure-based Embedding Generation. The insight of KEENHash-stru is performing function classification for function matching based on the pseudo functions to the given binary. Therefore, it is crucial to find a classifier that can efficiently classify a function such that similar functions are in the same class and dissimilar ones are separated. However, it is challenging to craft a high-quality training dataset for training a multi-classifier in a supervised manner due to the two aspects of labeling functions and determining the number of labels. To overcome this issue, we regard the classification as the 1-NN search [27] by our function embedding model. Our model unifies source and pseudo functions within the same vector space, enabling us to perform the 1-NN search on a vast training dataset crafted based on source functions to classify pseudo ones. The extraction of source functions is achieved from open-sourced C/C++ projects, eliminating the need for reverse engineering of binaries. This approach allows for the easy expansion of the 1-NN training dataset, encompassing a wide variety of function semantics. Additionally, the model is capable of grouping similar functions together, while distinctly separating dissimilar ones, resulting in the massive collection of source functions forming clustered distributions directly. Therefore, by employing the suitable clustering algorithm, we can automatically extract function classes and label functions for the 1-NN dataset. Moreover, to avoid the high time cost of 1-NN search, centroid-based clustering is preferred where each cluster is represented by a centroid instead of all inner-cluster data points.
Clustering. Specifically, we first perform source function extraction (see 3.2.1) on massive projects to extract a large collection of distinct C/C++ source functions. Next, our trained function embedding model is applied to these functions to generate function embeddings. These embeddings are used as the training dataset for the subsequent centroid-based clustering. Here, we utilize K-Means [15], an

effective unsupervised algorithm scalable on large-scale datasets, to perform clustering. Formally, the K-Means clustering algorithm partitions the training dataset of the source functions into n clusters $S = \{S_0, S_1, ..., S_{n-1}\}$ by maximizing the cosine similarity between functions in the same cluster. $c_i \in C$ is the centroid and representation for the cluster of S_i, labeled with i. C is the set of all centroids, in the size of $n \ll$ the number of source functions that serves as the training dataset of 1-NN. While training the K-Means model is expensive, it is a one-time and offline process in a period that does not impact the efficiency of KEENHash-stru. In addition, the cluster size of n to K-Means is a critical hyperparameter that can affect the performance of the subsequent classification task. Recall that our objective is to transform the matching problem into the classification one, and therefore the performance of the matching task is the evaluation metric for finding the most effective n to train the K-Means model. To this end, we set n to 2^k where $k \in \mathbb{N}^+$ to appropriately reduce its search range. We systematically study a suitable n in the function matching task between binaries in the same class in Sec. 4.3.

Generation. After obtaining the n centroids, we take them as the training dataset of 1-NN and perform the 1-NN search to all pseudo functions of the binary for generating the program embedding. Formally, we denote the collection of pseudo functions with q numbers as $P = \{P_0, P_1, ..., P_{q-1}\}$ where P_i is the i_{th} pseudo function. Then after the search, we get the corresponding collection of retrieved Top-1 most similar centroids $R = \{c_i^0, c_j^1, ..., c_k^{q-1}\}$ with cosine similarity where c_j^i represents that P_i retrieve the $c_j \in C$ centroid. Therefore, taking the labels $L = \{i, j, ..., k\}$ of R, the binary is transformed into a collection of labels, and for any two binaries, their respective pseudo functions with the same labels are considered matches. Additionally, to further enhance the efficiency of similarity evaluation with bitwise operations, we attempt to transform L of the binary into a fixed-length bit-vector $\mathbf{v}$ in dimension n: $\mathbf{v} = [v_0, v_1, ..., v_{n-1}]$ where $v_i = 1$ if $i \in L$; otherwise, $v_i = 0$. This transformation captures the presence of each centroid in the result of the 1-NN search, and multiple identical labels are consolidated as a single one since the pseudo functions are classified into the same class, potentially presenting similar semantics. Moreover, each element in $\mathbf{v}$ is represented as a single bit to minimize space requirements.

However, the dimension n of $\mathbf{v}$ can be very high (e.g., $n = 2^{22}$ in 512KB), and the number of pseudo functions in the binary can be extremely smaller. Directly using the dimension n can consume a substantial amount of space and is computationally prohibitive [44]. Furthermore, the size of n for K-Means may not be reducible due to the consideration regarding the performance of function matching (Sec. 4.3). To address this issue, we employ Feature Hashing [89], which hashes the high-dimensional input vector $\mathbf{v} \in \{0, 1\}^n$ into a lower one $\{0, 1\}^m$ with the mapping function $\phi : \mathcal{V} \longrightarrow \{0, 1\}^m$ where $\mathcal{V}$ is the domain of all possible $\mathbf{v}$. Since $m \ll n$, Feature Hashing reduces $\mathbf{v}$ to a more compact representation, allowing for significant savings in space and computational resources. Moreover, previous research demonstrates that Feature Hashing approximately preserves the original similarity (i.e., function matching results) between hashed vectors with a high probability [46], and the penalty incurred from using it only grows logarithmically with the number of samples compared [89].

In particular, according to $\mathbf{v}$, we leverage a uniform hash function $H : \mathcal{L} \longrightarrow [0, m)$ to hash a label $i \in \mathcal{L} \wedge v_i = 1$ ($\mathcal{L}$ is the domain of all possible labels, i.e., all clustered centroids) to the new position $j \in [0, m)$ of hashed bit-vector $\mathbf{v}' = [v'_0, v'_1, ..., v'_{m-1}]$ in dimension m. Furthermore, a sign hash function $\zeta : \mathcal{L} \longrightarrow \{-1, +1\}$ is applied to the label i to get its signed value for leading to an unbiased estimate [89]. In case of collision where multiple labels map to the same position j, the sum of their signed values, followed with an indicator function $\mathbf{1}_{x \neq 0}(x)$ for preserving bitwise operations and space requirements, is taken as the value for $v'_j \in \{0, 1\}$. Formally, for a given $\mathbf{v}$, the ϕ to get $\mathbf{v}'$ is defined as follows:

$$v'_j = \phi_j(\mathbf{v}) = \mathbf{1}_{x\neq 0}(0 + \sum_{k\in\{i|H(i)=j\wedge v_i=1\}} \zeta(k)) \tag{3}$$

where $\mathbf{1}_{x\neq 0}(x) = 1$ if $x \neq 0$; otherwise, $\mathbf{1}_{x\neq 0}(x) = 0$.

In addition, the selection of the hashed length m is critical since it balances the efficiency and effectiveness, as well as space requirements. Based on our experience, we aim to select a larger m as much as possible. Specifically, we set $m = 2^{16}$ (e.g., 512KB to 8KB), which is a reasonable upper bound of space size and applies to the vector database Milvus [88], supporting at most 2^{18} in bits. We also perform a discussion on the selection of m in [54].

3.4.2 Semantics-based Embedding Generation. KEENHash-stru compares binaries through the structural features of function matching. However, the semantic differences between functions are normalized to only two states: matched or unmatched, i.e., any function is considered equally significant. Thus, structurally similar (i.e., massive code reuse [50]) binaries in the same or different classes (e.g., across compile environments. See the previously mentioned example in Sec. 4.4) may affect KEENHash-stru to some extent of distinguishing them in large-scale BCSA (Sec. 4.4), due to lacking the capability of simultaneously capturing the semantic differences and maximizing the unique features. This problem is also faced by other structure-based methods with even more negative impact (Sec. 4.4). To mitigate this issue, we introduce KEENHash-sem which integrates function semantics based on significance to derive the program semantics.

Generation. A potential way for generating the semantic embedding to a binary involves averaging directly the pseudo function embeddings (Mean Pooling) [14]. However, this strategy ignores the quantity of information for each function such as the size of a function [94]. Inspired by the success of TF-IDF [91] and SIF [14] techniques which model sentence semantics based on the weighted average of word embeddings to express the most significant words, we propose a similar strategy to derive program semantics through function ones. Notably, the feature functions are exclusively included in the same-class binaries and not present across any non-same-class ones, i.e., the unique features. Thus, we utilize the intrinsic information of the function as weights to assess its significance and thereby, maximize the semantics of feature functions for amplifying the unique similarities (the feature functions between same-class binaries) or differences (the respective feature functions between non-same-class ones) while offsetting the ones of the reused for stronger distinguishing ability. Specifically, we determine the weights for a pseudo function by modeling based on two effective factors of its lines of code (LoC) and the number of strings (NoS). Guided by heuristics, we posit that a pseudo function's importance in a binary increases with its LoC and NoS since a function with a higher LoC likely handles more complex logic, making it crucial within the overall program and a larger NoS may indicate extensive functionality in processing user inputs, displaying data, or executing other tasks that heavily involve string operations. Therefore, for a pseudo function P_i with its LoC_i and NoS_i, its weight w_i is specified as follows:

$$w_i = f_1(\text{LoC}_i, \alpha) + f_2(\text{NoS}_i, \beta) \tag{4}$$

where f_1 and f_2 are designed to compute the partial weights based on LoC_i and NoS_i, respectively. α and β are hyperparameters that adjust and scale the influence of LoC_i and NoS_i on the overall weight of P_i. Typically, f_1 and f_2 are determined empirically and experimentally. We evaluate the performance of program clone search on the IoT (malicious) and BinaryCorp (benign) repository datasets (Sec. 4.1), without their query parts, to adjust and find an optimal configuration for them:

$$f_1(\text{LoC}_i, \alpha) = \frac{(\text{LoC}_i)^{\alpha_1}}{\alpha_2}, \quad f_2(\text{NoS}_i, \beta) = \frac{(\text{NoS}_i)^{\beta_1}}{\beta_2} + 1 \tag{5}$$

where $\alpha_1, \alpha_2, \beta_1, \beta_2 = 0.4, 5, 0.45, 1$ (obtained through grid search [62]). For a binary with its pseudo functions $P = \{P_0, P_1, ..., P_{q-1}\}$ and corresponding function embeddings $E = \{\mathbf{e}_0^p, \mathbf{e}_1^p, ..., \mathbf{e}_{q-1}^p\}$, its program embedding is formulated as:

$$\mathbf{v} = \frac{1}{q} \sum_{i=0}^{q-1} w_i \frac{\mathbf{e}_i^p}{\|\mathbf{e}_i^p\|_2} \tag{6}$$

With this approach, KEENHash-sem can maximize the feature function semantics within the same-class binaries through the weights to effectively distinguish them from other non-same-class ones but with massive code reuse (see Sec. 4.4). Moreover, we have experimented with other features to set weights, such as ① the vertex centrality in call graph and ② system API call usages (e.g., the count of API calls in a function). However, the experimental results show that using them for setting weights is good but less effective than LoC and/or NoS (e.g., ②), or is even less than Mean Pooling (e.g., ①). In this study, we do not delve into these features.

3.5 Similarity Evaluation

Here, we present the similarity evaluation metric used by KEENHash for comparing two binaries.
KEENHash-stru. The program embedding produced by KEENHash-stru is a bit-vector, with each element indicating the classification-based function matching. We leverage Jaccard similarity [46] to evaluate similarities due to its proportional measure property (see [54]).
KEENHash-sem. For the float vector of KEENHash-sem, we leverage cosine similarity, which is consistent with the comparison between function embeddings [14].

4 Evaluation

In this section, we attempt to investigate KEENHash on its performance of large-scale program-level BCSA (see Sec. 2.1) by answering the following research questions:

- **RQ1:** How *effective* including *scalable* is KEENHash in the task of function matching for large-scale BCSA scenarios?
- **RQ2:** How does KEENHash perform on program clone search with various respective large-scale repositories?
- **RQ3:** Is KEENHash effective against code obfuscation on large-scale program clone search?
- **RQ4:** Is KEENHash effective on program clone search with *larger-scale* repository?
- **RQ5:** How does KEENHash perform on malware detection from the large-scale BCSA and clone search perspective?

4.1 Dataset

In this section, we briefly introduce our datasets and the details of them can be found in [54].
Training Dataset. Two training datasets are included for ❶ the function embedding model and ❷ the K-Means model, respectively. For ❶, we collect (and build) open-sourced C/C++ projects (along with corresponding collected binaries across various architectures if possible) through ArchLinux official repositories (AOR) [12], Arch User Repository (AUR) [13], and Linux Community [39], ultimately amassing around 910K projects. The source functions (with matched pseudo ones) related to the evaluation of RQ1 and the effectiveness of our function embedding model (see [54]) are excluded, preventing data leakage. Eventually, we obtain 40.51M matched function pairs with an average of 556 tokens per function. As for ❷, we follow previous studies [78, 92] to collect a large number of diverse open-source C/C++ projects by crawling from Github [35] and GNU/Linux community [39]. In total, 11,013 projects, including malicious ones (e.g., gh0st RAT malware [23]),

are obtained, containing 56M unique C/C++ source functions. Such a substantial source function dataset is essential for the generalization of KEENHash-stru.

Test Dataset. The test dataset is used to evaluate the performance of KEENHash on function matching (Sec. 4.3). Specifically, we use the binary diffing dataset in DeepBinDiff [28]. In total, there are 2,098 binaries across various versions and optimization levels. The function matching ground truth is obtained through the Function Extraction (Sec. 3.2).

Repository and Query Dataset. To evaluate KEENHash on BCSA (Sec. 2.1), we collect 5 datasets:

▶ **IoT.** We collect recent 37,657 nonpacked C/C++ (detected with DIE [43]) IoT malware samples from MalwareBazaar [66] across 21 malware families. We randomly divide the dataset into repository and query datasets in a 9:1 ratio and each family has at least two samples in the query.

▶ **BinaryCorp (BC).** The dataset [87] is crafted based on AOR and AUR across 5 optimization levels. There are 9,819 source code and 45,593 distinct C/C++ binaries with 9,498 sample families. We randomly divide the dataset into repository and query in a 7.5:2.5 ratio and each family has at least one in the query. All binaries are stripped. Notably, it contains massive code reuse (Sec. 4.4).

▶ **BinKit (BK).** BinKit [55] dataset is crafted from 51 GNU software packages with 235 unique C/C++ source code (i.e., sample families). It is diverse along different optimization levels, compilers, architectures, and obfuscations. Like BinaryCorp, it also contains massive code reuse (Sec. 4.4).

• **Normal (BinKit$_N$/BK$_N$):** The normal one is compiled with 288 different compile environments for 67,680 binaries to 51 packages. It covers 8 architectures (arm, x86, mips, and mipseb, each available in 32 and 64 bits), 9 compilers (5 versions of GCC and 4 versions of Clang), and 4 optimization levels (O0, O1, O2, and O3);

• **Obfuscation (BinKit$_O$/BK$_O$):** The obfuscation one is compiled with 4 obfuscation options, instruction substitution (SUB), bogus control flow (BCF), control flow flattening (FLA), and all combined (ALL), through Obfuscator-LLVM [53]. The same architectures and 5 optimization levels (extra Os) are also covered, and 37,600 binaries are generated.

The BK$_N$ dataset is divided randomly into repository and query in a 9:1 ratio. 10% of the samples are randomly selected from the BK$_O$ dataset as the query to maintain experimental consistency in Sec. 4.5 and 4.6. All binaries are stripped.

▶ **MLWMC (MC).** MLWMC [25] is a recent real-world PE 32 malware dataset. It contains 67,000 malware samples across 670 malware families. We consider the 49,820 nonpacked C/C++ samples, belonging to a total of 615 malware families where each family contains at least 20 samples. Moreover, we divide the dataset in the same way as IoT.

4.2 Experiment Setup

Function Embedding Model. The maximum length of the embedding model is 2048, the training epoch is 196, the batch size (i.e., N) is 512, and the learning ratio is 0.001.

K-Means Model. The cluster size is set to 2^k where $k \in [16, 22]$ with the iteration of 30. Furthermore, we use FAISS-GPU [51] to train K-Means models. Thus, the maximum cluster size is limited to 2^{22} due to the VRAM constraint. The clustering takes at most 20 hours for the size of 2^{22}.

KEENHash. The parameter settings are shown in Sec. 3.4.

Baseline. For RQ1, we use popular open-sourced Diaphora [52], the most recent state-of-the-art academic SigmaDiff [34], and commercial BinDiffMatch [58] tools for *function matching* between binaries. Diaphora uses function hash values and calling relationships. SigmaDiff and BinDiffMatch employs function embeddings, generated through deep/machine learning (DL/ML), and a call graph-based heuristic strategy to find the most similar functions. There are other relevant tools such as DeepBinDiff [28] for direct binary diffing, and Asm2Vec [26] and PalmTree [61] for function embedding generation. In this study, we do not include them since SigmaDiff is an upgrade of DeepBinDiff [28, 34], and both SigmaDiff and BinDiffMatch have already covered function embedding generation for function matching, better than Asm2Vec and PalmTree.

For RQ2 to RQ5, we include 4 state-of-the-art structure-based methods as our baselines: SSDEEP [56], TLSH [71], Vhash [83], and PSS_O [16]. SSDEEP and TLSH are two fuzzy hash algorithms (on whole files) that are widely used in binary similarity evaluation. Vhash, a widely-recognized security BCSA method, is an in-house similarity clustering algorithm, based on a simple structural feature hash. It empowers VirusTotal to find similar files and perform threat intelligence. However, VirusTotal does not provide the information of similarity space. Thus, we define the Vhash in the Hamming space due to the best performance in experiments. PSS_O is a spectral-based method. It represents a binary by calculating the spectrum of its call graph and the edge counts in the control flow graphs (CFGs) of functions. In this study, we include SSDEEP and TLSH (belonging to fuzzy hashing families) due to their popularity in the industry for large-scale program-level BCSA [4, 66, 83], though previous works [42, 85] claim that they may have poor performance. For comprehensive comparison, therefore, we incorporate the most recent methods from both industry and academia (i.e., Vhash and PSS_O). We also leverage 4 methods to compare with KEENHash for the ablation study purpose in RQ2 and RQ3: Mean Pooling (Sec. 3.4.2), $\text{KEENHash}^{16}_{\text{w/oFH}}$, LoC, and NoS. $\text{KEENHash}^{16}_{\text{w/oFH}}$ is the variant of KEENHash-stru but with a cluster size of 2^{16} for K-Means and thereby without Feature Hashing, for assessing the necessity of the Feature Hashing module. LoC and NoS represent using either one of the features in KEENHash-sem, for demonstrating their respective contribution. Here, we do not include function matching (and DL/ML)-based BCSA methods for comparison, such as SigmaDiff [34], BinDiffMatch [58], and other function embedding methods [26, 61, 84], since they do not support direct use in large-scale scenarios (Sec. 4.3). The effectiveness of our function embedding model and correspondingly generated program embeddings, compared with others, are discussed in [54], as mentioned before. We also take no string-based method in as it lacks robustness and is vulnerable to simple string-based attacks (Sec. 4.4).

Experiment Environment. All the experiments are run on a Linux server running Ubuntu 20.04 with AMD EPYC 7K62 48-Core Processor, 1TB RAM, and 8 Nvidia A100 GPUs. The program clone search is implemented atop Milvus [88].

4.3 RQ1: Effectiveness and Scalability of Function Matching

Motivation. For KEENHash-stru, it is essential to choose an effective K-Means model to reflect the results of function matching accurately. Furthermore, we show that KEENHash-stru is scalable on *large-scale* BCSA scenarios but existing state-of-the-art function matching methods are not.

Approach. We train multiple K-Means models by selecting various cluster sizes of 2^k. These K-Means models are used for binary diffing to match same-class pseudo functions (compiled from the same source functions) between two same-class binaries. Same-class pseudo functions should be classified into the same clusters, whereas those from different classes should be assigned to distinct clusters. The methods used for comparison are Diaphora, SigmaDiff, and BinDiffMatch. The compared pairs of binaries can be found here [19], aligning with BinDiffMatch. In total, there are 1,926 pairs of binaries with 101,289 pairs of matched functions. The evaluation metrics include Precision (the ratio of corrected matches to all derived results), Recall (the ratio of corrected matches to all the ground-truth data), and F1-Score (the harmonic mean of Precision and Recall). We also measure the time cost for function matching to compare their scalability in large-scale scenarios.

Result. Table 1 shows that the performance of function matching with our K-Means models is enhanced as the cluster size increases. In particular, the K-Means (n=2^{22}) achieves the highest 0.7573, 0.7730, and 0.7651 in Precision, Recall, and F1-Score, significantly outperforming Diaphora. Compared to SigmaDiff and BinDiffMatch, the performance of the K-Means is relatively poorer, but

Table 1. The performance of function matching for K-Means of KEENHash-stru in different cluster size settings.

Method	Precision	Recall	F1-Score
Diaphora [52]	0.7121	0.5944	0.6480
SigmaDiff [34]	0.8640	0.7827	0.8213
BinDiffMatch [58]	0.9652	0.8870	0.9244
K-Means (n=2^{22})	0.7573	0.7730	0.7651
K-Means (n=2^{21})	0.7282	0.7714	0.7492
K-Means (n=2^{20})	0.7023	0.7656	0.7326
K-Means (n=2^{19})	0.6403	0.7542	0.6926
K-Means (n=2^{18})	0.6113	0.7182	0.6605
K-Means (n=2^{17})	0.5675	0.7016	0.6274
K-Means (n=2^{16})	0.5295	0.7138	0.6080

Table 2. Statistics of function matching time cost for K-Means (n=2^{22}), SigmaDiff, and BinDiffMatch.

Method	Mean Cost (1 core)	RQ1 Scenario (1 core)	RQ4 Scenario (48 cores)
SigmaDiff	0.25074 seconds	483 seconds	323 days
BinDiffMatch	0.04314 seconds	83 seconds	56 days
K-Means (n=2^{22})	0.00020 seconds	0.38574 seconds	395.83 seconds

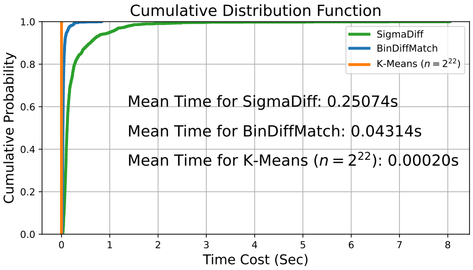

Fig. 2. The time cost of performing function matching for K-Means (n=2^{22}), SigmaDiff, and BinDiffMatch with 1 core. K-Means (n=2^{22}) takes an average of 0.00020 seconds to complete one matching between two binaries, while SigmaDiff and BinDiffMatch require 0.25074 and 0.04314 seconds.

still able to maintain an effective capacity of 93% to SigmaDiff and 83% to BinDiffMatch in F1-Score (*and it is the only one that supports large-scale scenarios.* See later). This is deemed reasonable as the K-Means models are unable to distinguish the similarities among classified functions and do not incorporate call graphs from binaries to provide essential diffing information for matching functions, resulting in a lower Recall. The classification nature can also result in multiple matched functions being classified into the same clusters, raising the rate of false positives (cartesian product) and lowering Precision.

• **Scalability.** Additionally, we plot the cumulative distribution of the time cost for only function matching procedure of pairs of binaries in Fig. 2 among SigmaDiff, BinDiffMatch, and K-Means (n=2^{22}) with 1 core (Diaphora is out of the bound of 1 second). Table 2 shows the statistics of time cost for them on function matching across mean time for one matching, RQ1 scenario total time cost, and RQ4 scenario (Sec. 4.6) total time cost. The function embeddings, call graphs, and function classifications (K-Means) are generated offline, which is reasonable in large-scale scenarios. The results show that K-Means is on average 1,254 times and 215 times faster than SigmaDiff and BinDiffMatch, respectively. Particularly, in the large-scale RQ4 scenario of Sec. 4.6 (see Table 2), SigmaDiff and BinDiffMatch will cost 323 and 56 days to perform 5.3 billion similarity evaluations (*a typical workload within one day*) between binaries with 48 processes (48 cores) running in parallel, which is unscalable and unavailable. Whereas KEENHash$^{16}_{\text{stru}}$ (Sec. 4.4) based on K-Means (n=2^{22}) takes only 395.83 seconds. Therefore, we consider the K-Means (n=2^{22}) model to be effective, as it provides an effective function matching capability and is well-suited for scalability for program-level BCSA on large-scale scenarios. As for Diaphora, SigmaDiff, and BinDiffMatch, they are unable to support such large-scale scenarios and are thereby excluded from the following RQs.

Answer 1: KEENHash-stru can effectively (0.7651 in F1-Score) and scalably perform function matching in large-scale scenarios. While the state-of-the-art matching methods are not scalable, being at least 215 times slower. In a scenario with 5.3 billion evaluations, KEENHash-stru takes 395.83 seconds while they will cost at least 56 days.

Table 3. Program clone search against string obfuscation. The string-based method Minhash$_s$ is vulnerable to the simple attack.

Method	mAP@100		mP@100	
	Mirai$_N$	Mirai$_O$	Mirai$_N$	Mirai$_O$
Minhash$_s$	**1.0**	**0.1909**	**1.0**	**0.1040**
PSS$_O$	0.9977	0.9887	0.9960	0.9894
Vhash	0.6442	0.6442	0.5350	0.5350
TLSH	0.8583	0.8472	0.8393	0.8398
SSDEEP	0.9901	0.9900	0.9900	0.9900
KEENHash$^{16}_{stru}$	0.9997	0.9999	0.9996	0.9997
KEENHash$^{f}_{sem}$	0.9937	0.9939	0.9883	0.9873

Table 4. The performance of program clone search of KEENHash on respective datasets of IoT, BinaryCorp, BinKit$_N$, and MLWMC.

Method	mAP@100					mP@100				
	IoT	BC	BK$_N$	MC	Avg.	IoT	BC	BK$_N$	MC	Avg.
PSS$_O$	0.9383	0.3212	0.6803	0.7164	0.6641	0.8991	0.0141	0.3033	0.3666	0.3958
Vhash	0.6789	0.5646	0.3379	0.7377	0.5798	0.6455	0.0207	0.0616	0.3581	0.2715
TLSH	0.9526	0.3768	0.4883	0.6625	0.6201	0.9304	0.0168	0.0809	0.2758	0.3260
SSDEEP	0.9393	0.2371	0.1598	0.6412	0.4944	0.8050	0.0055	0.0369	0.2436	0.2728
KEENHash$^{16}_{w/oFH}$	0.9580	0.6613	0.6410	0.7430	0.7508	0.9343	0.0261	0.4226	0.3972	0.4451
Mean Pooling (Baseline)	0.9599	0.7289	0.7369	0.7482	0.7935	0.9335	0.0333	0.5263	0.4081	0.4753
LoC	0.9595	0.7438	0.7406	0.7577	0.8004	0.9336	0.0339	0.5548	0.4123	0.4837
NoS	0.9604	0.7610	**0.7972**	0.7486	0.8168	0.9330	0.0341	**0.6604**	0.4022	0.5074
KEENHash$^{16}_{stru}$	0.9602	0.7247	0.7270	0.7519	0.7910	**0.9363**	0.0322	0.5467	**0.4144**	0.4824
KEENHash$^{f}_{sem}$	**0.9608**	**0.7628**	0.7911	**0.7581**	**0.8182**	0.9361	**0.0346**	0.6531	0.4126	**0.5091**

4.4 RQ2: KEENHash on Program Clone Search

Motivation. In this RQ, we attempt to evaluate KEENHash performance on program clone search, reflecting its capacity in program-level BCSA on respective and different large-scale scenarios.
Approach. We perform program clone search on 4 repository and query datasets (except BinKit$_O$), respectively. Here, we denote KEENHash-stru and sem as KEENHash$^{16}_{stru}$ (16 represents Feature Hashing size of 2^{16}) and KEENHash$^{f}_{sem}$ (f represents float vectors), respectively. The evaluation metrics include mAP@k (Mean Average Precision at k) and mP@k (Mean Precision at k) [22] where k represents retrieving the most Top-k similar results. mAP@k is a metric that evaluates the mean (across all retrieving results) of the average of the Precision@k_i ($k_i \in \{i|1 \leq i \leq k \wedge r(i) = 1\}$ where $r(i) = 1$ represents that the i_{th} retrieved sample is in the same class to the query sample; otherwise, $r(i) = 0$) to one retrieving. The higher the ranking of same-class results returned, the greater the mAP@k. However, mAP@k cannot assess the proportion of same-class results returned. Therefore, we augment it with mP@k, which calculates the mean of the Precision@k across all retrieving. Where Precision@k_i presents the ratio of retrieved same-class samples to the retrieved Top-k_i ones. Here, we fix $k = 100$ to evaluate the KEENHash BCSA capacity on multiple same-class binaries.
• **String-based Method.** We introduce 1 string-based method to demonstrate its vulnerability to simple string-based attacks (the reason for excluding it from the following RQs). The string-based method (Minhash$_s$) extracts strings from a binary through command `strings` and uses MinHash [21] to generate 128 hash values as its representation, in the Jaccard similarity space. Real-world malware usually avoids meaningful string literals, making challenges to the string-based analysis systems. Mirai [11] is a famous botnet family, targeting various kinds of IoT devices for DDoS attacks. The initial version has been open-sourced since 2016 [47]. By analyzing the source code of Mirai, we discover that Mirai encrypts almost all of its strings. Therefore, we substitute the secret key originally used by Mirai and encrypt all strings in the source code using the same encryption strategy, where the program behavior remains unchanged. Furthermore, we compile both original (Mirai$_N$) and newly obfuscated (Mirai$_O$) versions across 9 architectures, 5 optimization levels, and 2 Mirai options, obtaining 90 binaries per version. These binaries are taken as the query dataset to search against the IoT repository which contains Mirai variants.

According to the results shown in Table 3, apart from Minhash$_s$, all other methods show consistent performance in mAP@100 and mP@100 against both Mirai$_N$ and Mirai$_O$ since they do not mainly rely on string features. However, turn to Minhash$_s$, its mAP@100 and mP@100 decrease by 80.91% and 89.6% from Mirai$_N$ to Mirai$_O$. The reason for Minhash$_s$, against Mirai$_O$, still achieving a 0.1040 mP@100 is due to the unobfuscated strings added during the compilation process. Such a result alerts that Minhash$_s$ is not a reliable system since adversaries can easily breach it using simple string encryption strategies for hiding the original information of string literals. Therefore, we exclude string-based methods from the comparison in RQ2 to RQ5.

Result. Table 4 presents the program clone search results on the respective datasets, including their average results. KEENHash (i.e., KEENHash$^{16}_{\text{stru}}$ and KEENHash$^{f}_{\text{sem}}$) methods outperform all other structure-based methods, by averages of at least 12.69% and 8.66% in mAP@100 and mP@100 (i.e., PSS$_{\text{O}}$), respectively. Furthermore, KEENHash demonstrates more distinct advantages on the two benign datasets with massive code reuse (see below). For instance, PSS$_{\text{O}}$ on BinaryCorp scores a rather low mAP@100 at only 0.3212 (e.g., lower than KEENHash$^{16}_{\text{stru}}$'s 0.7247). Even within malware datasets, KEENHash continues to outperform these methods in both metrics. Since these structure-based methods, depending only on simple features (e.g., PSS$_{\text{O}}$'s), extract no semantic information, the effectiveness of similarity evaluation between binaries can be significantly impacted across compilation environments (e.g., O0 vs. O3 [26]) and (malware) variants. Instead, KEENHash leverages semantics in pseudo functions, mitigating the impact of these factors.

- **Ablation Study.** Comparing KEENHash$^{16}_{\text{stru}}$ with KEENHash$^{16}_{\text{w/oFH}}$, Table 4 shows that the former outperforms the latter across all datasets. For instance, KEENHash$^{16}_{\text{stru}}$ achieves 0.7270 mAP@100 on BinKit$_N$, significantly surpassing KEENHash$^{16}_{\text{w/oFH}}$'s by 8.6%. As indicated by Sec. 4.3, smaller cluster sizes n of K-Means affect the function matching results, which in turn impacts the quality of the generated program embeddings. These two experimental results demonstrate the necessity of a well-configured K-Means (i.e., n=2^{22}) in conjunction with Feature Hashing.

Regarding LoC and NoS, in general, both LoC and NoS perform better than Mean Pooling (without any feature), indicating that either of them has a positive contribution. For example, LoC and NoS get 0.7438 and 0.7610 mAP@100, higher than Mean Pooling's 0.7289. In addition, KEENHash$^{f}_{\text{sem}}$ with both features is generally better than LoC and NoS. For instance, on MLWMC, KEENHash$^{f}_{\text{sem}}$ achieves 0.7581 mAP@100, outperforming both LoC and NoS. Therefore, we can conclude that the introduction of both LoC and NoS is useful for enhancing KEENHash$^{f}_{\text{sem}}$ (which obtains the best results on average, compared with either LoC or NoS).

As for Mean Pooling, we analyze and discuss it later as it relates to RQ2 and RQ3.

- **Massive Code Reuse.** The two KEENHash and the pooling methods exhibit similar performance on IoT and MLWMC while KEENHash$^{f}_{\text{sem}}$ performs significantly better than others on the benign datasets (e.g., 5.42% and 12.68% improvements in mAP@100 and mP@100 to Mean Pooling for BinKit$_N$). A common feature of these *two* datasets is that multiple non-same-class binaries can be compiled (same compile environment) from a single project, and they share (i.e., massive code reuse) a large number of the common functions (e.g., around 72%/63%, in Jaccard similarity, between `cp` with O3 and `ln` with O3/`cp` with O0 in `Coreutils-8.29`. KEENHash$^{f}_{\text{sem}}$ is better in distinguishing these kinds of cases) [12, 13, 39], leading to difficulty for BCSA methods to distinguish them under the two datasets (*across compile environments*). Thus, only maximizing the unique features can more effectively distinguish them (Sec. 3.4.2). Therefore, such a result illustrates the advantages of integrating function semantics with their intrinsic information to distinguish binaries with massive code reuse by maximizing such feature function semantics. Furthermore, though KEENHash$^{16}_{\text{stru}}$ has a similar performance to Mean Pooling on the 4 datasets, it shows significantly greater robustness across compile environments and under code obfuscation (Table 5 and 6).

- **Robustness across Compile Environments.** We evaluate the robustness of KEENHash across compile environments (see Table 5) on BinKit$_N$. Here we denote <x, y> (e.g., <O3, O0>) as that the new query and repository contain only the binaries with options x and y from the original ones, respectively. Thus, the primary differences between the new query and the repository are focused on x and y. Here, we only list the representative results in Table 5, where the second/third-row option represents query/repository. The results show that KEENHash methods outperform all other

Table 5. The performance of KEENHash on program clone search across compile environments on BinKit$_N$ query and repository. The second/third row represents query/repository (samples) with specific compile environments.

Method	Metric	Optimization				Compiler	Architecture		
		O0	O1	O2	O3	GCC	ARM	MIPS	x86
		O1	O2	O3	O0	Clang	MIPS	x86	ARM
PSS$_\text{O}$	mAP@k	0.4833	0.3606	0.4486	0.1475	0.3344	0.3179	0.2152	0.4491
	mP@k	0.1712	0.1282	0.1334	0.0727	0.1572	0.0916	0.1034	0.1644
Vhash	mAP@k	0.2969	0.2778	0.3083	0.2655	0.0095	0.0152	0.0078	0.0086
	mP@k	0.0232	0.0229	0.0245	0.0222	0.0045	0.0048	0.0034	0.0040
TLSH	mAP@k	0.0682	0.2955	0.4046	0.0735	0.1210	0.0547	0.0515	0.0814
	mP@k	0.0226	0.0479	0.0489	0.0268	0.0446	0.0237	0.0255	0.0375
SSDEEP	mAP@k	0.0540	0.0864	0.1050	0.0481	0.0265	0.0584	0.0320	0.0510
	mP@k	0.0144	0.0184	0.0172	0.0165	0.0163	0.0187	0.0139	0.0181
Mean Pooling (Baseline)	mAP@k	0.7387	0.6956	0.7077	0.7155	0.7558	0.7095	0.7263	0.6824
	mP@k	0.3859	0.3701	0.3627	0.3791	0.5171	0.3865	0.3648	0.3421
KEENHash$^{16}_{\text{stru}}$	mAP@k	0.7841	0.7569	0.7293	0.8072	0.8125	0.7870	0.7843	0.7487
	mP@k	0.4571	0.4350	0.4051	0.4654	0.6106	0.4385	0.4144	**0.4122**
KEENHash$^{f}_{\text{sem}}$	mAP@k	**0.8195**	**0.7817**	**0.7684**	**0.8080**	**0.8245**	**0.7958**	**0.8024**	**0.7640**
	mP@k	**0.4688**	**0.4613**	**0.4364**	**0.4689**	**0.6330**	**0.4620**	**0.4373**	0.4100

Table 6. The performance of KEENHash on program clone search against code obfuscation on BinKit$_N$ repository and BinKit$_O$ query (samples) with specific compile environments.

Method	Metric	SUB N	BCF N	FLA N	ALL N	O N
PSS$_\text{O}$	mAP@k	0.6675	0.3727	0.2085	0.1231	0.3483
	mP@k	0.3043	0.2170	0.1315	0.0867	0.1872
Vhash	mAP@k	0.0034	0.0068	0.0048	0.0070	0.2176
	mP@k	0.0023	0.0041	0.0033	0.0036	0.0536
TLSH	mAP@k	0.3776	0.0899	0.0580	0.0228	0.1395
	mP@k	0.0778	0.0361	0.0232	0.0096	0.0373
SSDEEP	mAP@k	0.1062	0.0627	0.0338	0.0145	0.0552
	mP@k	0.0393	0.0325	0.0188	0.0119	0.0260
Mean Pooling (Baseline)	mAP@k	0.7673	0.6359	0.4377	0.2608	0.5312
	mP@k	0.5227	0.5178	0.3825	0.2281	0.4166
KEENHash$^{16}_{\text{stru}}$	mAP@k	0.7757	**0.8207**	**0.7735**	**0.7070**	**0.7704**
	mP@k	0.5679	**0.6789**	**0.5915**	**0.5593**	**0.6006**
KEENHash$^{f}_{\text{sem}}$	mAP@k	**0.8026**	0.7353	0.5032	0.2726	0.5849
	mP@k	**0.6428**	0.6350	0.4427	0.2483	0.4974

structure-based methods across all options, especially on <O3, O0>. As mentioned earlier, simple features can be significantly affected by compilation environments, weakening the effectiveness of them. Furthermore, both KEENHash methods significantly outperform the pooling method (e.g., 9% and 8% improvements in mAP@100 and mP@100 on <O3, O0>), illustrating that direct integrating on function semantics exhibits poorer robustness.

> **Answer 2:** Both KEENHash methods significantly outperform state-of-the-art methods across 4 datasets in terms of performance by an average of at least 12.69% in mAP@100 on program clone search. In addition, KEENHash$^{f}_{\text{sem}}$ is more effective than KEENHash$^{16}_{\text{stru}}$ and Mean Pooling in the massive code reuse scenario. Both KEENHash methods also show greater robustness across compilation environments than others including Mean Pooling.

4.5 RQ3: KEENHash against Code Obfuscation

Motivation. Code obfuscation is the process of modifying binaries to make them no longer useful to hackers while maintaining them fully functional. On the contrary, it can interfere with existing BCSA methods. In this RQ, we assess the robustness of KEENHash against code obfuscation.
Approach. We use BinKit$_O$'s query to retrieve binaries from BinKit$_N$'s repository in an obfuscation vs. normal scenario. In BinKit$_O$, SUB transforms fragments of assembly code to their equivalent form through predefined rules; BCF modifies the CFGs of functions by adding extensive irrelevant basic blocks; FLA changes the original CFG using a complex hierarchy of new conditions as switches; and, ALL combines all obfuscations above. The evaluation metrics include mAP@100 and mP@100.
Result. Table 6 presents the retrieving results against code obfuscation where ’O’/‘N’ represents the complete query or repository of BinKit$_O$/BinKit$_N$. According to the results, KEENHash methods consistently outperform other structure-based methods in all scenarios (e.g., KEENHash$^{16}_{\text{stru}}$ outperforms PSS$_\text{O}$ by 58% and 47% in mAP@100 and mP@100 on <ALL, N>). In obfuscation scenarios, semantics in pseudo functions still distinguish binaries better than simple features.

Furthermore, in general, KEENHash$^{16}_{\text{stru}}$ has better robustness with 0.7704 mAP@100 and 0.6006 mP@100 on <O, N> than KEENHash$^{f}_{\text{sem}}$'s 0.5849 and 0.4974. Although code obfuscation can impact the effectiveness of function embeddings, especially for <ALL, N>, our K-Means model mitigates this by transforming function similarity into a matching problem (i.e., 0 or 1) and still capturing

Table 7. Program clone search against all datasets of IoT, BinaryCorp, BinKit$_N$, MLWMC, BinKit$_O$, Mirai$_N$, and Mirai$_O$. The query datasets are shown in the table and the repository is the merge of repositories of IoT, BinKit$_N$, BinaryCorp, and MLWMC.

Method	**mAP@100**							**mP@100**						
	IoT	BinaryCorp	BinKit$_N$	MLWMC	BinKit$_O$	Mirai$_N$	Mirai$_O$	IoT	BinaryCorp	BinKit$_N$	MLWMC	BinKit$_O$	Mirai$_N$	Mirai$_O$
PSS$_\text{O}$	0.9144	0.3136	0.6798	0.7153	0.3256	0.8781	0.8430	0.8436	0.0123	0.2779	0.3623	0.1615	0.8470	0.8183
Vhash	0.6812	0.5651	0.3414	0.7375	0.2179	0.3931	0.3931	0.6156	0.0206	0.0604	0.3585	0.0524	0.3917	0.3917
TLSH	0.9438	0.3769	0.4888	0.6801	0.1362	0.6692	0.6732	0.9148	0.0168	0.0797	0.3206	0.0346	0.6555	0.6661
SSDEEP	0.7045	0.2375	0.1606	0.6413	0.0516	0.0778	0.0	0.2258	0.0054	0.0375	0.2430	0.0222	0.0066	0.0
KEENHash$^{16}_{\text{stru}}$	0.9599	0.7243	0.7270	0.7515	**0.7704**	**0.9997**	**0.9999**	**0.9354**	0.0322	0.5466	**0.4141**	**0.6004**	**0.9996**	**0.9997**
KEENHash$^{f}_{\text{sem}}$	**0.9609**	**0.7627**	**0.7911**	**0.7553**	0.5666	0.9937	0.9939	0.9352	**0.0346**	**0.6530**	0.4116	0.4910	0.9883	0.9873

enough relationships between normal and obfuscated functions. Thereby, it can reduce the impact of similarities among normal and obfuscated functions. On <SUB, N>, KEENHash$^{f}_{\text{sem}}$ is slightly better since the impact of SUB on function embeddings is relatively weak. Additionally, the two KEENHash methods perform significantly better than Mean Pooling in both metrics (e.g., 23.92% and 18.4% improvements in mAP@100 and mP@100 on <O, N> for KEENHash$^{16}_{\text{stru}}$), demonstrating again the advantages of structure and semantics-based program embedding generation of KEENHash.

> **Answer 3:** KEENHash is more robust against code obfuscation than state-of-the-art methods by at least 23.66% - 42.21% in mAP@100 on <O, N>. In addition, the performance of KEENHash$^{16}_{\text{stru}}$ is generally better than KEENHash$^{f}_{\text{sem}}$. Both KEENHash methods are also more robust than Mean Pooling by 5.37% - 23.92% in mAP@100 on <O, N>.

4.6 RQ4: KEENHash on Larger-scale Repository

Motivation. The comparisons should not be affected greatly by the scale and distributions of binaries. In this RQ, we evaluate the robustness of KEENHash in distinguishing binaries on a larger-scale dataset, simulating the diversity of data distributions in real-world scenarios (e.g., across Linux/Windows and benign/malicious binaries).

Approach. We merge the repositories of IoT, BinKit$_N$, BinaryCorp, and MLWMC in sequence to form a larger repository, where the order of samples within each repository is maintained. Furthermore, we use 7 query datasets from all previous experiments. In total, there are 171,075 and 31,230 binaries in the repository and query, respectively, indicating 5.3 billion similarity evaluations (*a typical evaluation workload within one day* based on VirusTotal-related reports [25, 81, 83]). The metrics include mAP@100 and mP@100.

Result. The experimental results are shown in Table 7. By Cliff's Delta effect size [65] between the pair of results (i.e., two metrics) of respective repositories and the larger repository to each method, the measured effect sizes show that only PSS$_\text{O}$ and SSDEEP have small and large differences, while others have negligible differences, in general. Where SSDEEP shows a significant drop in both metrics on the IoT query (PSS$_\text{O}$ has a drop in mP@100 by 5.6%). Furthermore, by investigating Mirai$_N$ and Mirai$_O$, the 4 structure-based methods, with simple features, have an obvious performance decrease compared to the results in Table 3. In contrast, KEENHash methods maintain consistent performance. As a result of the dataset merge, the capability of structure-based methods to differentiate Mirai from binaries in BinaryCorp, BinKit$_N$, and MLWMC is negatively affected, demonstrating that they can be potentially and significantly affected by the scale and distributions of binaries. Overall, across the 7 queries, two KEENHash methods can maintain consistent performance (only nearly consistent for KEENHash$^{f}_{\text{sem}}$ to BinKit$_O$ due to code obfuscation

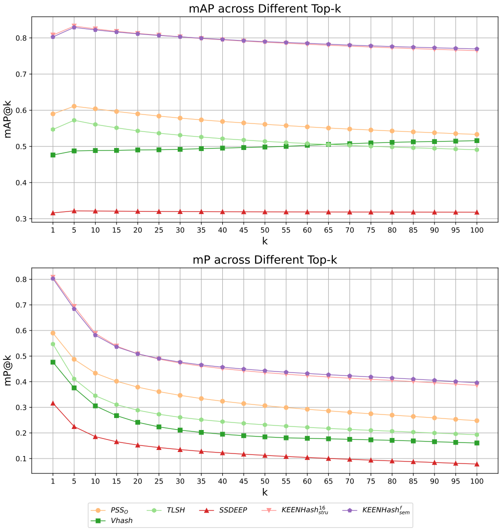

Fig. 3. Program clone search on all binaries.

Table 8. Malware detection against $\text{Mirai}_{N/O}$ (malicious), IoT (malicious), and BinKit_N (benign) dataset. Numbers represent how many binaries are *misclassified* by the BCSA methods in different K-NN settings.

Method	$\text{Mirai}_{N/O}$			IoT			BinKit_N		
	False Negative Number						False Positive Number		
	K=1	K=3	K=5	K=1	K=3	K=5	K=1	K=3	K=5
PSS_O ✗	30	18	12	29	45	51	5	16	31
Vhash ✗	40	40	60	34	46	59	2	4	5
TLSH ✗	50	52	52	14	18	17	1	1	1
SSDEEP ✗	**0**	174	174	**0**	1143	1514	3856	1	1
$\text{KEENHash}^{16}_{\text{stru}}$ ✓	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
$\text{KEENHash}^{f}_{\text{sem}}$ ✓	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

to the effectiveness of function embeddings), while the 4 structure-based methods always have obvious and slight (e.g., IoT) degradation to some of these queries.

We further plot the performance of program clone search across multiple $k \in [1, 100]$ in Fig. 3, where the query is the merged one of all the 7 queries. The results show that in general, KEENHash can achieve at least 0.7647 mAP@100 and 0.3858 mP@100 and outperforms other structure-based methods (by at least 23.16% and 13.79% in $k = 100$, i.e., PSS_O) across different k, indicating its enhanced capacity on large-scale BCSA. Moreover, we highlight that $\text{KEENHash}^{16}_{\text{stru}}$ and $\text{KEENHash}^{f}_{\text{sem}}$ costs only 395.83 and 90.31 seconds with CPU (48 cores), respectively, for 5.3 billion similarity evaluations (i.e., scalable in large-scale scenarios).

Answer 4: In general, KEENHash outperforms other structure-based methods by at least 23.16% in mAP@100, and shows greater robustness, against a larger-scale dataset. Additionally, it is scalable for large-scale BCSA scenarios. For instance, in the scenario with 5.3 billion similarity evaluations, KEENHash takes at most 395.83 seconds.

4.7 RQ5: KEENHash on Malware Detection

Motivation. In previous RQs, we have demonstrated the superiority of KEENHash from aspects including effectiveness and robustness. In this RQ, we show how KEENHash can be helpful in a more specific and critical large-scale BCSA security application: malware detection.

Approach. We leverage the merged repository used in RQ4 as the malware and benign binary database. Additionally, we perform malware detection (i.e., benign or malicious) by leveraging IoT, $\text{Mirai}_{N/O}$, and BinKit_N queries against the merged repository from the malware security aspect. We use K-NN as the binary classifier and the numbers of false negatives (i.e., misclassified as benign samples) and false positives (i.e., misclassified as malicious samples) as the metrics.

Result. As shown in Table 8, KEENHash (both structural and semantic) outputs *zero* misclassification in all three datasets across malicious and benign, in all K-NN settings. In comparison, all other BCSA methods can make mistakes, even for the widely-recognized security method Vhash from

VirusTotal. We identified two primary causes of misclassified cases: (1) the query binary belongs to a less common architecture (e.g., m68k), resulting in significant code differences compared to the repository ones; and (2) the query binary is a variant of repository ones, but with substantial changes in code functions (e.g., much more functions and invocations), which simple features (e.g., spectrum of call graph) fail to capture them. Thanks to the capability to capture function semantics and represent binaries, KEENHash can accurately classify these challenging malware samples, while other methods with simple features struggle with such variations, resulting in false negatives. For false positives, the shortcomings of these methods across compilation environments (see Sec. 4.4) lead to prioritizing the retrieval of malicious binaries in certain cases.

> **Answer 5:** Thanks to the powerful LLM-based function embedding and effective program representation, KEENHash demonstrates superior performance in the large-scale BCSA scenario of malware detection, and significantly outperforms previous methods including the widely-recognized method Vhash from VirusTotal.

5 Limitations

KEENHash suffers from the limitations to the property of static analysis including decompilation. For example, packed binaries and binaries with payloads are difficult to decompile accurately to obtain precise function results, thus affecting its performance. Furthermore, substantial variations in or among functions warrant caution, like significant source code revisions (e.g., code refactoring), aggressive inter-procedural compiler optimizations (e.g., function inlining and link-time optimizations), and great inter or inner-procedural code obfuscations (e.g., function merging). These variations may disrupt the similarity between program structures or affect the performance of our function embedding model. In the future, we plan to systematically evaluate the impact of such variations and propose corresponding mitigation strategies. In addition, the maximum length of input tokens to our function embedding model is limited due to space consideration [82]. This limitation can be overcome with new LLM architectures introduced in the future.

6 Related Work

Function-level Similarity Analysis. Recently, there has been a tremendous increase in the popularity of (binary) function-level similarity analysis [26, 42, 55, 68, 84, 86, 87, 95, 96]. Considering performance and scalability, these function representation techniques mainly focus on static analysis and deep learning. Gemini [93] extracts crafted features of each basic block from Genius [32] and employs GNN to learn the representations of CFG to functions. jTrans [87] leverages a transformer-based method with a jump-aware representation of the analyzed binary functions and a newly-designed pre-training task to generate embeddings, encoded with CFG information. VulHawk [64] proposes an intermediate representation function model with the language model and GCN, followed by an entropy-based adapter to transfer function embedding space from different file environments into the same one to alleviate the differences caused by various file environments. CLAP [84] boosts superior transfer learning capabilities by effectively aligning binary code with their semantics explanations. However, it is difficult to directly adapt them in program-level BCSA due to the scalability problem.

Program-level Similarity Analysis. There are few recent studies related to program-level BCSA [16, 28, 42, 44]. SSDEEP [56] is a fuzzy hashing technique based on Context Triggered Piecewise Hashing (CTPH) to hash files into hash strings. TLSH [71] is a fuzzy hashing method based on k-skip N-gram features, followed by LSH to hash feature counts into a vector with 128 buckets. Vhash [83] is an in-house similarity clustering algorithm, based on a simple structural feature hash. PSS$_O$ [16] is a spectral-based method that represents programs through their spectrums of call graphs and the edge counts of CFGs. These methods are limited to their performance on large-scale BCSA. There are also some studies evaluating similarity based on dynamic analysis and symbolic execution [10, 46, 63, 68]. However, such methods are limited to the explorable execution space and the runtime overhead of the representation generation in large-scale scenarios. Furthermore, some work [28, 34, 52, 58] focuses on binary diffing. While these methods also fail to scale to large-scale scenarios due to the scalability problem, which is shown in our experiment.

7 Conclusion

In this paper, we propose a novel large-scale program-level BCSA hashing approach KEENHash, for evaluating similarities among binaries. KEENHash captures binaries from the dual perspectives of function matching based on K-Means and Feature Hashing, and program semantics by integrating function embeddings to generate respective compact and fixed-length program embeddings. Our experimental results demonstrate that KEENHash is at least 215 times faster than the state-of-the-art function matching tools while maintaining effectiveness. Furthermore, in a large-scale scenario with 5.3 billion similarity evaluations, KEENHash takes only 395.83 seconds while the previous tools will cost at least 56 days. We also evaluate the two KEENHash methods on the program clone search of large-scale BCSA across extensive datasets in a total of 202,305 binaries. Compared with 4 state-of-the-art methods, KEENHash outperforms all of them by at least 23.16% and displays remarkable superiority over them in the BCSA security scenario of malware detection. Such results demonstrate the outstanding effectiveness of KEENHash on large-scale program-level BCSA.

Acknowledgments

We thank Dr. Pengfei Jing for the helpful discussion when preparing the manuscript. This work is partially supported by the National Natural Science Foundation of China (Grant No. 62202306 and Grant No. 62372299). Zhijie Liu would like to dedicate this paper to the love of his fiancée.

References

[1] 2025. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[2] 2025. Hybrid Search. https://milvus.io/docs/multi-vector-search.md.

[3] 2025. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. https://github.com/llvm/llvm-project.

[4] 2025. TLSH is a fuzzy matching program and library. https://tlsh.org/.

[5] 2025. tshd malware. https://www.virustotal.com/gui/file/289616b59a145e2033baddb8a8a9b5a8fb01bdbba1b8cf9acadcdd92e6cc0562.

[6] 2025. Zygug malware. https://www.virustotal.com/gui/file/fa541d1274b450c2bbdc0c29531b847fb06baf30da46367c100c917ef5e8cbe8.

[7] National Security Agency. 2024. Ghidra Software Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra.

[8] Jina AI. 2023. jina-embeddings-v2-base-en. https://huggingface.co/jinaai/jina-embeddings-v2-base-en.

[9] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[10] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-based malware detection using dynamic analysis. *Journal in computer Virology* 7 (2011), 247–258.

[11] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*. 1093–1110.

[12] Archlinux. 2021. Arch linux. https://archlinux.org/packages/.

[13] Archlinux. 2021. Arch User Repository. https://aur.archlinux.org/.

[14] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A simple but tough-to-beat baseline for sentence embeddings. In *International conference on learning representations*.

[15] David Arthur, Sergei Vassilvitskii, et al. 2007. k-means++: The advantages of careful seeding. In *Soda*, Vol. 7. 1027–1035.

[16] Tristan Benoit, Jean-Yves Marion, and Sébastien Bardin. 2023. Scalable program clone search through spectral analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 808–820.

[17] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*. PMLR, 2397–2430.

[18] BigCode. 2023. starcoderbase-1b. https://huggingface.co/bigcode/starcoderbase-1b.

[19] BinaryAI. 2024. BinaryAI BindiffMatch algorithm. https://github.com/binaryai/bindiffmatch.

[20] Xander Bouwman, Harm Griffioen, Jelle Egbers, Christian Doerr, Bram Klievink, and Michel Van Eeten. 2020. A different cup of {TI}? the added value of commercial threat intelligence. In *29th USENIX security symposium (USENIX security 20)*. 433–450.

[21] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 21–29.

[22] Ben Carterette and Ellen M Voorhees. 2011. Overview of information retrieval evaluation. In *Current challenges in patent information retrieval*. Springer, 69–85.

[23] ChangC. 2020. The gh0st RAT malware. https://github.com/Cc28256/CcRemote.

[24] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[25] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. 2023. Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 60–74.

[26] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[27] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[28] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*.

[29] EleutherAI. 2023. Pythia-160M. https://huggingface.co/EleutherAI/pythia-160m.

[30] EleutherAI. 2023. Pythia-1B. https://huggingface.co/EleutherAI/pythia-1b.

[31] EleutherAI. 2023. Pythia-410M. https://huggingface.co/EleutherAI/pythia-410m.

[32] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 480–491.

[33] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[34] Lian Gao, Yu Qu, Sheng Yu, Yue Duan, and Heng Yin. 2024. SigmaDiff: Semantics-aware deep graph matching for pseudocode diffing. In *Network and distributed system security symposium*.

[35] GitHub. 2024. GitHub: Let's build from here. https://github.com/.

[36] GNU. 2019. GNU Coretuils. https://www.gnu.org/software/coreutils/.

[37] GNU. 2019. GNU Difftuils. https://www.gnu.org/software/diffutils/.

[38] GNU. 2019. GNU Findutils. https://www.gnu.org/software/findutils/.

[39] GNU. 2024. The GNU Operating System and the Free Software Movement. https://www.gnu.org/home.en.html.

[40] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[41] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, et al. 2022. Manu: a cloud native vector database management system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3548–3561.

[42] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–38.

[43] horsicq. 2024. Detect It Easy. https://github.com/horsicq/Detect-It-Easy.

[44] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. 2013. {MutantX-S}: Scalable Malware Clustering Based on Static Features. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 187–198.

[45] UNIX International. 2024. DWARF Debugging Information Format. https://dwarfstd.org/.

[46] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*. 309–320.

[47] jgamblin. 2016. Mirai BotNet. https://github.com/jgamblin/Mirai-Source-Code.

[48] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 2023. 1-to-1 or 1-to-n? Investigating the effect of function inlining on binary similarity analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26.

[49] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[50] Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2023. Third-Party Library Dependency for Large-Scale SCA in the C/C++ Ecosystem: How Far Are We?. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1383–1395. doi:10.1145/3597926.3598143

[51] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[52] joxeankoret. 2024. Diaphora. https://github.com/joxeankoret/diaphora.

[53] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. doi:10.1109/SPRO.2015.10

[54] KEENHash. 2025. KEENHash online artifact. https://github.com/Rroscha/KEENHash.

[55] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1661–1682.

[56] Jesse Kornblum. 2006. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation* 3 (2006), 91–97.

[57] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.

[58] Tencent Keen Security Lab. 2024. BinaryAI: Binary File Security Analysis Platform. https://www.binaryai.cn/home.

[59] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[60] Vector Guo Li, Matthew Dunn, Paul Pearce, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. 2019. Reading the tea leaves: A comparative analysis of threat intelligence. In *28th USENIX security symposium (USENIX Security 19)*. 851–867.

[61] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3236–3251.

[62] Petro Liashchynskyi and Pavlo Liashchynskyi. 2019. Grid search, random search, genetic algorithm: a big comparison for NAS. *arXiv preprint arXiv:1912.06059* (2019).

[63] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177.

[64] 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 *NDSS*.

[65] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[66] MalwareBazaar. 2024. MalwareBazaar | Malware sample exchange. https://bazaar.abuse.ch/.

[67] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. 2020. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In *NDSS 2020, Network and Distributed System Security Symposium, 23-26 February 2020, San Diego, CA, USA*. Internet Society.

[68] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. {BinSim}: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *26th USENIX Security Symposium (USENIX Security 17)*. 253–270.

[69] Jaron Mink, Hadjer Benkraouda, Limin Yang, Arridhana Ciptadi, Ali Ahmadzadeh, Daniel Votipka, and Gang Wang. 2023. Everybody's Got ML, Tell Me What Else You Have: Practitioners' Perception of ML-Based Security Tools and Explanations. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2068–2085.

[70] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[71] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 7–13.

[72] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. 2024. len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 152–152.

[73] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PMLR, 8748–8763.

[74] Hex Rays. 2024. IDA Pro: The best-of-breed binary code analysis tool, an indispensable item in the toolbox of world-class software analysts, reverse engineers, malware analyst and cybersecurity professionals. https://hex-rays.com/ida-pro/.

[75] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*. Springer, 230–253.

[76] Silvia Sebastián and Juan Caballero. 2020. Avclass2: Massive malware tag extraction from av labels. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 42–53.

[77] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2198–2210.

[78] Wei Tang, Zhengzi Xu, Chengwei Liu, Jiahui Wu, Shouguo Yang, Yi Li, Ping Luo, and Yang Liu. 2022. Towards understanding third-party library dependency in c/c++ ecosystem. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[79] tree sitter. 2024. Tree-sitter is a parser generator tool and an incremental parsing library. https://tree-sitter.github.io/tree-sitter/.

[80] upx. 2024. The Ultimate Packer for eXecutables. https://github.com/upx/upx.

[81] Kevin van Liebergen, Juan Caballero, Platon Kotzias, and Chris Gates. 2023. A Deep Dive into the VirusTotal File Feed. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 155–176.

[82] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[83] VirusTotal. 2024. VirusTotal - free online virus, malware and URL scanner. https://www.virustotal.com.

[84] 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. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (2024).

[85] Huaijin Wang, Zhibo Liu, Shuai Wang, Ying Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2024. Are We There Yet? Filling the Gap Between Binary Similarity Analysis and Binary Software Composition Analysis. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 506–523.

[86] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. sem2vec: Semantics-aware Assembly Tracelet Embedding. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–34.

[87] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13.

[88] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data*. 2614–2627.

[89] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*. 1113–1120.

[90] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 860–872.

[91] Ho Chung Wu, Robert Wing Pong Luk, Kam Fai Wong, and Kui Lam Kwok. 2008. Interpreting TF-IDF term weights as making relevance decisions. *ACM Transactions on Information Systems (TOIS)* 26, 3 (2008), 1–37.

[92] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. 2023. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 270–282.

[93] 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*. 363–376.

[94] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. 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*. 1393–1405.

[95] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*. 1145–1152.

[96] 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. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[97] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2SFinder: Detecting open-source software reuse in COTS software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049.

[98] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 39–51.

[99] Xiaohua Zhai, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. 2023. Sigmoid loss for language image pre-training. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 11975–11986.

Table 9. The performance of program clone search with different hashed lengths for KEENHash-stru.

Method	mAP@100					mP@100				
	IoT	BinaryCorp	BinKit$_N$	MLWMC	Avg.	IoT	BinaryCorp	BinKit$_N$	MLWMC	Avg.
KEENHash$^{10}_{\text{stru}}$	0.9596	0.6915	0.7262	0.7424	0.7799	0.9360	0.0296	0.5468	0.3961	0.4771
KEENHash$^{12}_{\text{stru}}$	0.9605	0.7191	0.7267	0.7502	0.7891	0.9360	0.0312	0.5458	0.4105	0.4809
KEENHash$^{14}_{\text{stru}}$	0.9603	0.7237	0.7270	0.7518	0.7907	0.9365	0.0318	0.5465	0.4135	0.4821
KEENHash$^{16}_{\text{stru}}$	0.9602	0.7247	0.7270	0.7519	0.7910	0.9363	0.0322	0.5467	0.4144	0.4824
KEENHash$^{18}_{\text{stru}}$	0.9604	0.7253	0.7270	0.7518	0.7911	0.9363	0.0325	0.5467	0.4150	0.4826

A Hashed Length of Feature Hashing

In this section, we introduce KEENHash-stru with different parameter settings of hashed length for comparison. Here, we denote KEENHash with different settings as KEENHash$^{y}_{\text{stru}}$ where $y \in \{10, 12, 14, 16, 18\}$ represents the Feature Hashing size (length) of 2^y. We evaluate the performance of KEENHash$^{y}_{\text{stru}}$ through the task in Sec. 4.4 (the results of other tasks show a similar trend here). The experimental results are shown in Table. 9. On average, the performance of KEENHash$^{y}_{\text{stru}}$ improves in tandem with increases in the hashed length 2^y. In addition, Cliff's Delta effect size [65] measures that the effect size between any pair of results of KEENHash$^{y}_{\text{stru}}$, across the 4 datasets, remains below 0.1, indicating a negligible difference. Although the performance increases on these datasets are minimal, we consider that the hashed length 2^y should be sufficiently large within an appropriate range to minimize the impact of the number of functions in binaries on KEENHash-stru for even larger-scale scenarios (e.g., 10 million binaries). Therefore, as mentioned in Sec. 3.4.1, we set the hashed length to 2^{16} (i.e., $y = 16$).

B Similarity Evaluation for KEENHash-stru

The program embedding produced by KEENHash-stru is a bit-vector, with each element indicating the classification of functions for representing function matching. Generally, two popular similarity evaluation metrics are appropriate for KEENHash-stru: Hamming distance [27] and Jaccard similarity [46]. In this study, we opt for the latter since the Hamming distance, quantifying absolute differences, is more likely to lead to false positives. For instance, consider programs A, B, and C with 10K, 4K, and 4 numbers of 1s in their bit-vectors. A and B are of the same class, with the only difference being that A is compiled with O0 while B is compiled with O3. The hamming distance between A and B is at least 6K, whereas the distance between B and C is no more than 4K + 4, leading to B and C appearing more similar in the similarity comparison. While, Jaccard similarity incorporates proportional measures (e.g., 4K and 10K are proportionally closer than 4 and 4K), mitigating the negative effects of the absolute differences.

C Dataset in Detail

This section extends the original description of the dataset used in our experiment.

Training Dataset. Two training datasets are included in this study for ❶ the function embedding model and ❷ the K-Means model, respectively. For ❶, to obtain a large number of matched source and pseudo functions, we build the automatic compilation pipeline based on ArchLinux official repositories (AOR) [12] and Arch User Repository (AUR) [13], following the same setting in jTrans (i.e., BinaryCorp) [87]. We compile all the projects through the command `makepkg`. In addition, over a period of three years, we collect open-sourced C/C++ projects from the Linux Community, along with their corresponding compiled binaries across various architectures (e.g., x86, arm, and so forth), ultimately amassing around 900K projects. Furthermore, the source functions (with matched pseudo ones) causing data leakage are excluded through sha256 (Sec. 3.2.1) from the training dataset for the evaluation of the effectiveness of our K-Means model (Sec. 4.3) and our function embedding model (Appendix E). The source and pseudo functions and pairs are extracted through the Function Extraction (see Sec. 3.2.1). Eventually, we obtain 4.51M matched function pairs with an average of 556 tokens per function as the training dataset for the function embedding model. As for ❷, to obtain massive and diverse C/C++ source functions, we follow previous studies [78, 92] to collect a large number of open-source C/C++ projects by crawling from Github [35] and GNU/Linux community [39]. In total, 11,013 projects, including malicious ones (e.g., gh0st RAT malware [23]), are obtained, containing 56M unique C/C++ source functions. A substantial source function dataset is essential for the generalization of KEENHash-structural.

Test Dataset. The test dataset is used to evaluate the performance of our K-Means model (Sec. 4.3). Specifically, we use the binary diffing dataset in DeepBinDiff [28]. The dataset, compiled with GCC 5.4, utilizes three popular binary sets of Coreutils [36], Diffutils [37], and Findutils [38], across various versions (5 versions for Coreutils, 4 versions for Diffutils, and 3 versions of Findutils) and optimization levels (O0, O1, O2, and O3). In total, there are 2,098 binaries. Moreover, the function matching ground truth is obtained through the Function Extraction in a total of 101,289 pairs of matched functions across 1,926 compared pairs of same-class binaries.

Repository and Query Dataset. To evaluate KEENHash on program clone search, we collect five datasets:

► **IoT.** We collect 37,657 nonpacked C/C++ (detected with DIE [43]) IoT malware samples from MalwareBazaar [66] across 21 malware families, spanning from January 2020 to July 2023. The malware families are obtained from VirusTotal [83] reports through avclass [75, 76], containing many notorious ones such as Mirai, Gafgyt, Tsunami, and so forth. Furthermore, each family contains at least 20 samples. We randomly divide the dataset into repository and query datasets in a 9:1 ratio and each family has at least two samples in the query.

► **BinaryCorp.** BinaryCorp [87] dataset is crafted based on AOR and AUR. Where, AOR contains tens of thousands of diverse packages, ranging from editor, HTTP server, compiler, graphics library, cryptographic library, and so forth. AUR contains over 77,000 packages uploaded and maintained by users. Furthermore, ArchLinux provides a useful tool `makepkg` for users to build their packages from source code. Wang et al. [87] choose the C/C++ project in the pipeline to build the datasets across five optimization levels of O0, O1, O2, O3, and Os. In total, 9,819 source code are collected and 45,593 distinct C/C++ binaries in x86 are generated, with 9,498 sample families. The sample family number is lower than the project number since binaries compiled (in the same compile environments) from different projects may have the same SHA256 hash values. We randomly divide the dataset into repository and query in a 7.5:2.5 ratio and each family has at least one sample in the query. Furthermore, all binaries are stripped, which is practical in real-world scenarios.

► **BinKit.** BinKit [55] dataset is crafted from 51 GNU software packages with 235 unique C source code (i.e., sample families). The 51 GNU packages are chosen due to their popularity and accessibility as they are real-world applications that are widely used on Linux, and their source code is publicly available. The compiled binaries are also diverse along different optimization levels, compilers, architectures, and obfuscations.

- **Normal (BinKit$_N$):** The normal one is compiled with 288 different compile environments for a total of 67,680 binaries to the 51 packages. It covers 8 architectures (arm, x86, mips, and mipseb, each available in 32 and 64 bits), 9 compilers (5 versions of GCC v{4.9.4, 5.5.0, 6.4.0, 7.3.0, 8.2.0} and 4 versions of Clang v{4.0, 5.0, 6.0, 7.0}), and 4 optimization levels (O0, O1, O2, and O3);
- **Obfuscation (BinKit$_O$):** The obfuscation one is compiled with 4 obfuscation options including instruction substitution (SUB), bogus control flow (BCF), control flow flattening

(FLA), and all combined (ALL), through Obfuscator-LLVM [53] as the compiler. Where SUB transforms fragments of assembly code to their equivalent form through predefined rules; BCF modifies the control flow graph (CFG) of functions by adding a large number of irrelevant basic blocks and branches; FLA changes the original CFG using a complex hierarchy of new conditions as switches; and, ALL combines all obfuscations above. The same architectures and 5 optimization levels (extra Os) are also covered. Therefore, a total of 37,600 binaries are generated.

The BinKit$_N$ dataset is divided randomly into repository and query in a 9:1 ratio. 10% of the samples are randomly selected from the BinKit$_O$ dataset as the query to maintain experimental consistency in Sec. 4.5 and 4.6. All binaries are stripped.

▶ **MLWMC.** MLWMC [25] is an open PE 32 real-world malware dataset collected through VirusTotal from August 2021 to March 2022. It contains 67,000 malware samples across 670 malware families obtained from VirusTotal reports through avclass where each family contains 100 malware samples. These families belong to 13 threat categories: 36% (282) of the families are classified as grayware, 15% (120) as downloaders, 11% (87) as worms, 10% (78) as backdoors, 5% (41) as viruses, and the remaining 23% (62) includes ransomware, rogueware, spyware, miners, hacking tools, clickers, and dialers. In this study, we consider the 49,820 nonpacked C/C++ samples, belonging to a total of 615 malware families where each family contains at least 20 samples. Moreover, we divide the dataset in the same way as IoT.

D Runtime Overhead

Here, we count the average time spent, across our collected 5 datasets, on each step of KEENHash for hashing a binary into corresponding program embeddings. In Function Extraction, the main time cost is in the decompilation process, which takes around 1 minute to decompile (1 core) one binary on average (at most 43 minutes for a binary in the size of 220MB). In Function Embedding Generation, for one binary (200 functions on average), it takes about 0.06 seconds on average (at most 12 seconds for binaries containing around 40K functions). In Program Embedding Generation, for one binary, it takes 0.3 and 0.1 seconds on average for KEENHash-stru (at most 35 seconds) and KEENHash-sem (at most 9 seconds), respectively. It is notable that these steps can be completed offline in large-scale scenarios for BCSA such as building a large-scale repository (Sec. 2).

E Function Embedding Discussion

As our function embedding model is the foundation of KEENHash, its effectiveness and ability to generalize in generating program embeddings are essential. Here, we discuss it with two state-of-the-art publically available models jTrans [87] and CLAP [84] on binary functions. We only focus on binary functions since the two models are only available on assembly code (they do not support source functions) and the performance between source and pseudo ones, for function matching, are already evaluated in Sec. 4.3. Furthermore, we employ the test dataset of BinaryCorp [87] for the experiment which aligns with the one evaluated in jTrans and CLAP. The test dataset contains 2,911,846 functions across 9,351 binaries from 1,974 source code with 5 optimization levels. On average, 1 source function (i.e., class) corresponds to 5.6 binary functions.

Function Embedding Comparing. We perform the function clone search for evaluation where the subject changes from programs to pseudo (binary) functions (Sec. 2.1). The test dataset is randomly divided into repository and query datasets in a 7.5:2.5 ratio where each class (if the class size > 1) has functions in both repository and query. The metrics include mAP@k and mP@k. The results are shown in Table 10. Our function embedding model (i.e., KEENHash) surpasses both jTrans and CLAP in terms of both mAP@k and mP@k metrics across $k \in \{1, 10, 50, 100\}$ on such a large and

Table 10. The performance of function clone search of KEENHash.

Method	Metric	Top-1	Top-10	Top-50	Top-100
jTrans	mAP@k	0.2841	0.3065	0.2846	0.2714
	mP@k	0.2841	0.1464	0.0640	0.0457
CLAP	mAP@k	0.6423	0.6567	0.6288	0.6186
	mP@k	0.6423	0.3129	0.1015	0.0648
KEENHash	mAP@k	0.7244	0.7422	0.7202	0.7116
	mP@k	0.7244	0.3261	0.1162	0.0764

Table 11. Program clone search across function embedding models.

Method	mAP@100	mP@100
jTrans-Mean Pooling	0.6321	0.0293
CLAP-Mean Pooling	0.8340	0.0333
KEENHash-Mean Pooling	0.8445	0.0349

diverse dataset. The results demonstrate that our model can retrieve more same-class functions that are also ranked higher compared to jTrans and CLAP, underscoring the effective discriminative capacity of our function embedding model in generating pseudo function embeddings. For instance, our model achieves 0.7116 and 0.0764 in mAP@100 and mP@100, outperforming jTrans by 44.02% and 3.07%, and CLAP by 9.30% and 1.16%.

Program Embedding Ablation Study. We leverage the function embeddings produced through KEENHash, jTrans, and CLAP to generate respective program embeddings for further demonstrating the effectiveness of different function embeddings to program embeddings. Specifically, to avoid biases, we employ Mean Pooling (see Sec. 4.2) as the program embedding generation approaches for conducting the ablation study. KEENHash-stru and KEENHash-sem-based approaches are excluded since jTrans and CLAP do not support aligning both source and binary functions within the same space, and they model binary functions based on assembly code, which differs from that of KEENHash. We perform the program clone search where the test dataset of BinaryCorp is randomly divided into repository and query datasets in an 8:2 ratio. Each query binary has at least one same-class sample in the repository. The metrics include mAP@100 and mP@100. Table 11 reveals that the Mean Pooling based on KEENHash outperforms those based on jTrans and CLAP. For example, KEENHash-Mean Pooling achieves 0.8445 in mAP@100, outperforming jTrans by 21.24%, and CLAP by 1.05%. This outcome indicates that a more effective function embedding model can potentially enhance the effectiveness of generated program embeddings. Furthermore, our function embedding model encodes source and pseudo functions into the same space, supporting both KEENHash-stru and sem methods where the former is better in code obfuscation scenarios (Sec. 4.5), while the latter is more effective in huge code reuse ones (Sec. 4.4).

F KEENHash Discussion

Comparison with Different Sizes of LLMs. As mentioned in Sec. 3.3, we fine-tune the `Pythia-410M` to have our function embedding model. However, LLMs with larger model sizes are generally expected to improve their comprehension. Here, we demonstrate that `Pythia-410M` achieves the best balance between performance and resource consumption. Therefore, we select it for our paper. The training, validation, and test datasets are split from the one introduced in Sec 4.1, with a ratio of 8:1:1. The various pre-trained base LLMs for fine-tuning, across different sizes, are selected with their

Table 12. Performance of fine-tuned function embedding models across different base LLMs with different sizes.

Fine-tuned Base LLM	MRR	Recall@1	Recall@5
StarCoder-1B [18]	0.8588	0.7932	0.9291
Pythia-1B [30]	0.8572	0.7975	0.9286
Pythia-410M [31]	0.8523	0.7915	0.9267
Pythia-160M [29]	0.7958	0.7257	0.8793
Jina-137M [8]	0.7723	0.6947	0.8651

popularity and can be found in Table 12. We use pseudo functions to retrieve source functions to evaluate the quality of both modalities. The evaluation metrics include MRR, Recall@1, and Recall@5, following the ones used in previous works [84, 87]. As shown in Table 12, `Pythia-410M` has a similar performance compared with `Pythia-1B` [30] and `StarCoder-1B` [18]. The larger-scale LLMs have almost no performance improvement, while they will cost more resources including: computing consumption, time, and video memory. Compared with `Pythia-160M` [29] and `Jina-137M` [8], `Pythia-410M` significantly outperforms both of them. Therefore, we consider `Pythia-410M` to be the optimal among these LLMs, due to its performance and resource consumption.

Combination of KEENHash-stru and sem. KEENHash-stru and sem represent a binary program from two different perspectives (Sec. 3.4), each with its own advantages (Sec. 4.4 and 4.5). Combining their respective advantages may further enhance the performance and robustness of KEENHash. A potentially direct strategy is to use hybrid search [2]. It conducts respective KEENHash-stru and sem similarity evaluations simultaneously, and merges and reranks/reweights the two sets of (paired) results based on normalized similarity scores. We leverage hybrid search on KEENHash for program clone search against the IoT dataset, with the same settings in Sec. 4.4. The importance (i.e., weight) of KEENHash-stru and sem is set to 0.5 each. Our experimental results show that the hybrid search achieves 0.9384 mAP@100 and 0.9300 mP@100. The direct combining strategy does not exhibit significant improvement compared to KEENHash-stru and sem (see Table 4). We plan to explore how to effectively combine these two types of program embeddings in the future.

Fragility of NoS. KEENHash-sem leverages NoS feature (see Sec. 3.4.2) as one of the factors for weighting function embeddings and performs well shown in our experiments. However, NoS is expected to be fragile against the string `libcall` expansion with known small lengths [1, 3]. Thus, it may vary with compilation options (e.g., O0 vs. O3), affecting weighting results. In addition, real-world malware also usually obfuscates, hides, and avoids string literals in functions (e.g., tshd [5] and Zygug [6] in MLWMC [25]), which can render NoS less effective. To mitigate this issue, we also introduce the LoC feature as another factor to enhance the robustness of KEENHash-sem (see Table 4).

Received 2024-10-30; accepted 2025-03-31