Enhancing DNN-Based Binary Code Function Search With Low-Cost Equivalence Checking

Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, *Member, IEEE*, Qiyi Tang, Sen Nie, and Shi Wu

Abstract—Binary code function search has been used as the core basis of various security and software engineering applications, including malware clustering, code clone detection, and vulnerability audits. Recognizing logically similar assembly functions, however, remains a challenge. Most binary code search tools rely on program structure-level information, such as control flow and data flow graphs, that is extracted using program analysis techniques or deep neural networks (DNNs). However, DNN-based techniques capture lexical-, control structure-, or data flow-level information of binary code for representation learning, which is often too coarse-grained and does not accurately denote program functionality. Additionally, it may exhibit low robustness to a variety of challenging settings, such as compiler optimizations and obfuscations. This paper proposes a general solution for enhancing the top-k ranked candidates in DNN-based binary code function search. The key idea is to design a low-cost and comprehensive equivalence check that quickly exposes functionality deviations between the target function and its top-k matched functions. Functions that fail this equivalence check can be shaved from the top-k list, and functions that pass the check can be revisited to move ahead on the top-k ranked candidates, in a deliberate way. We design a practical and efficient equivalence check, named `BinUSE`, using *under-constrained* symbolic execution (USE). USE, a variant of symbolic execution, improves scalability by initiating symbolic execution directly from function entry points and relaxing constraints on function parameters. It eliminates the overhead incurred by path explosion and costly constraints. `BinUSE` is specifically designed to deliver an assembly function-level equivalence check, enhancing DNN-based binary code search by reducing its false alarms with low cost. Our evaluation shows that `BinUSE` can enable a general and effective enhancement of four state-of-the-art DNN-based binary code search tools when confronted with challenges posed by different compilers, optimizations, obfuscations, and architectures.

Index Terms—Reverse engineering, symbolic execution, software similarity, deep learning

1 Introduction

Binary code search determines the degree of similarity between two pieces of assembly code, and it is critical in many binary code security applications. For instance, malware analysis identifies malware samples that behave similarly in order to uncover malware families and avoid the need to re-analyze known malware samples [1], [2]. Patch presence testing extracts security patch signatures and conducts binary code search to decide whether critical patches have been properly deployed in an executable [3]. Binary code search is also the basis of many binary code comprehension tasks, such as code cloning and plagiarism detection [4], [5].

Given the prosperous development of machine learning techniques and their widespread application in downstream tasks like software embedding [6], [7], the majority of contemporary binary code search tools aim to train a machine learning model to capture binary code similarity [8], [9], [10], [11]. In particular, recent advances in deep neural networks (DNN) and representation learning have enabled the promising approach of training DNN models to learn optimal code representations capable of discriminating between similar assembly functions [12], [13], [14], [15], [16], [17].

To learn code representations, DNN models are trained with (lightweight) lexical-, control structure-, or data flow-level features. Such representations, despite being easy to extract, may not preserve program semantics to a great extent. Additionally, lightweight features are typically not robust to challenges such as compiler optimizations or obfuscations, which make semantically similar assembly code appear to be dramatically different. Hence, DNN models may exhibit low discriminability and low robustness, resulting in a high number of false alarms in their retrieved top-k candidates.

This paper aims to enhance binary code function search in a principled and efficient approach. Given a target function f_t and a function repository RP, our key idea is to employ a low-cost equivalence check to quickly identify functions in RP that deviate semantically from f_t, and should thus be shaved from the retrieved top-k ranked candidates. As a result, functions that pass the check can be reconsidered for inclusion in the retrieved top-k candidates. Our main results of boosting DNN-based binary function search tools are shown in Table 1. In short, this work

- *Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, and Shuai Wang are with the Department of Computer Science and Engineering, HKUST, Clear Water Bay, Kowloon, Hong Kong. E-mail: {hwangdz, pmaab, yyuanaq, zliudc, shuaiw}@cse.ust.hk.*
- *Qiyi Tang, Sen Nie, and Shi Wu are with Tencent Security Keen Lab, Shenzhen, Guangdong 518000, China. E-mail: {dodgetang, snie, shiwu}@tencent.com.*

Manuscript received 14 June 2021; revised 19 January 2022; accepted 20 January 2022. Date of publication 8 February 2022; date of current version 9 January 2023.
This work was supported in part by a Bridge Gap Fund of TTC/HKUST under Grant BGF.003.2021 and CCF-Tencent Open Research Fund.
(Corresponding author: Shuai Wang.)
Recommended for acceptance by J. Sun.
Digital Object Identifier no. 10.1109/TSE.2022.3149240

TABLE 1
Enhance Top-k and MRR Accuracy of DNN-Based Tools With BinUSE

	BinaryAI [15]	asm2vec [14]	ncc [13]	PalmTree [18]
top-1	+13.2%	+23.8%	+34.5%	+40.9%
top-3	+21.2%	+30.7%	+34.0%	+41.0%
top-5	+22.2%	+34.0%	+30.1%	+36.8%
MRR	+17.2%	+27.7%	+32.1%	+37.9%

See Definition of Top-k and MRR in Section 2.

delivers a consistent and highly-encouraging enhancement of four state-of-the-art DNN-based tools, despite the fact that these tools are on the basis of different neural models and learning techniques. According to our observation, BinaryAI achieves the highest accuracy compared with the rest models. Nevertheless, our work still identifies a considerable space for improvement. In particular, the top-1 accuracy of BinaryAI is notably enhanced by 13.3%. Similarly, the mean reciprocal rank (MRR) score of BinaryAI is also largely improved for 17.2%. We present the detailed evaluation and discussion in Section 8.

To design a low-cost and practical equivalence check, we build and check the input-output relations of assembly functions using constraint solving and under-constrained symbolic execution (USE) techniques [19]. In comparison with standard symbolic execution, USE is meant to perform flexible and speedy symbolic reasoning directly from function entry points, skipping the costly path prefix from main to target functions. We *optimize* the standard USE scheme as a practical tool, namely BinUSE, particularly for equivalence checking of assembly functions. BinUSE launches USE traversal from the function entry point, and traverses each path until reaching the first *external function callsite*, denoting an informative and critical node on CFG. Then, BinUSE uses symbolic formulas of external callsite inputs to form symbolic constraints of each path, and to match two functions, BinUSE explores matching symbolic constraints collected from every path in each function.

BinUSE is unsound due to an over-estimation of legitimate function input space, an unsound memory model of the symbolic execution used, and several engineering challenges like cross-architecture matching (see Section 4). However, in contrast to previous binary code equivalence checking that analyzes only basic blocks or a single execution trace [3], [4], [20], [21], BinUSE is sufficiently scalable to cross-check *all* assembly functions in a pair of coreutils executables within 56.6 CPU minutes (only 25.0s to check two assembly functions). Similarly, BinUSE takes 4,137.6 CPU minutes (about 2.1 wallclock hours on our server with 32 CPU cores) to cross-check functions in a pair of binutils executables. On average, BinUSE takes about 10.9s to check two assembly functions; note that binutils programs have significantly more functions than that of coreutils programs. A variety of difficult comparison settings are considered, including different compilers, optimizations, architectures, and obfuscations. BinUSE achieves average false positive rates of 25.0% and false negative rates of 4.2%. Our evaluation shows that BinUSE can successfully boost four cutting-edge DNN-based binary code search tools under different settings, confirming the approach's efficacy and generalizability. We further demonstrate optimizations that enable BinUSE to handle binutils executables which are much larger than coreutils executables and how BinUSE helps to improve the accuracy of DNN-based vulnerability search. In summary, we make the following contributions:

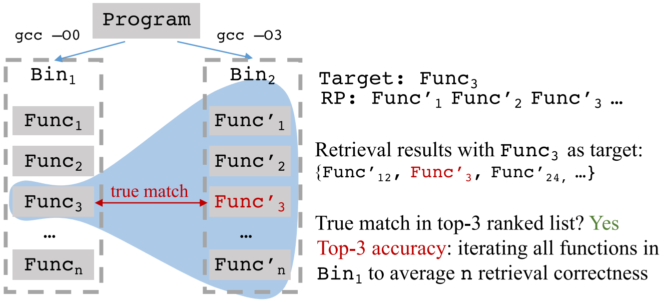

Program
gcc -O0
gcc -O3
Bin$_1$
Bin$_2$
Func$_1$
Func$_2$
Func$_3$
…
Func$_n$
Func'$_1$
Func'$_2$
Func'$_3$
…
Func'$_n$
true match
Target: Func$_3$
RP: Func'$_1$ Func'$_2$ Func'$_3$ …
Retrieval results with Func$_3$ as target:
{Func'$_{12}$, Func'$_3$, Func'$_{24}$, …}
True match in top-3 ranked list? Yes
Top-3 accuracy: iterating all functions in Bin$_1$ to average n retrieval correctness

Fig. 1. Assessment setup for binary code function search. The top-k accuracy is computed by averaging results of taking each function in Bin$_1$ as the "target function".

- At the conceptual level, we advocate a new focus to enhance DNN-based binary code function search which currently exhibits low accuracy. Instead of designing new DNNs (which are in principle difficult to precisely capture semantics), we design a low-cost equivalence check to flag and shave assembly functions that deviate from the target function semantically.
- At the technical level, we present an equivalence check by optimizing the standard USE scheme to further reduce its cost. This equivalence check is particularly designed for assembly functions, taking into account a variety of technical challenges and optimization opportunities, e.g., collecting symbolic constraints over external callsites reachable from function entry points to reduce complexity.
- At the empirical level, our evaluation shows that the designed equivalence check is general and effective to enhance DNN-based binary function search tools with low cost. The equivalence check shows excellent performance in a variety of challenging settings, including general equivalent function matching and CVE search.

We have released the source code of BinUSE and evaluation data publicly available for reproducibility at [22]. We will maintain BinUSE to benefit future research.

2 Preliminaries

2.1 Formulation and Metrics

Most recent works perform semantics-aware assembly function search [6], [8], [12], [13], [14], [15], [18], [23]. That is, they aim to determine the semantical similarity of two assembly functions in binary code, although these two functions may appear syntactically distinct, for example, as a result of compiler optimization. The function search task is similar to that of information retrieval, in that given a target assembly function f_t and a repository of assembly functions RP, binary code search engines retrieve the top-k functions $f_t \in RP$ ranked by their semantic similarity with f_t.

Fig. 1 illustrates a frequently used and challenging assessment setup in this sector [6], [8], [12], [13], [14], [15],

[23], where we prepare two executable $\texttt{Bin}_1$ and $\texttt{Bin}_2$ compiled from the same program using different compilation settings. Comparing each pair of functions yields a similarity score, and the top-k accuracy is formulated in the following way

$$\frac{1}{N} \times \sum_{i=1}^{n} p_k(f_i) \tag{1}$$

where N is the total number of functions in the program. To understand this standard formulation of top-k accuracy: we iteratively take each function f_i in $\texttt{Bin}_1$ as the target function to query the RP that is formed by all functions in $\texttt{Bin}_2$. To compute the top-k accuracy, let the correct match (i.e., ground truth) of f_i be f_i'. In Formula 1, $p_k(f_i)$ yields one if f_i' is within the retrieved top-k ranked candidates, whereas $p_k(f_i) = 0$ if f_i' is not. Note that the correct match f_i' (ground truth) denotes functions sharing *identical functionality* with the target function f_i. For assessment in Fig. 1, the standard and common setting is to deem a function f_i' sharing the same function name with f_i to be the ground truth. Another commonly-used metric, known as mean reciprocal rank (MRR) score, can be computed in the following way

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\mathrm{rank}_i}$$

where $|Q|$ denotes the total number of queries launched toward RP, and rank_i represents the rank of the correct match for ith query returned by the function search tool. A larger MRR score implies a more accurate function search. For instance, given three queries toward RP where the correct matches for each query are placed at the 4-th, 2-th, and 6th ranked candidates, respectively. Then, the MRR is computed as $\frac{1}{3}(\frac{1}{4} + \frac{1}{2} + \frac{1}{6})$ which is 0.31.

Clarification on Forming Dataset of "Equivalent Code". Careful readers may wonder that conceptually, DNN-based binary diffing tools are designed to find *similar* code. We clarify that the function search setting over *equivalent functions* is indeed a common setup adopted by most, if not all, (DNN-based) works in this field [6], [8], [12], [13], [14], [15], [23]. This design choice can be explained from three aspects: 1) in practice, it is quite hard to create a large set of "similar binary code," whereas equivalent binary code can be well-prepared using different compilation settings, 2) benchmarking equivalent code can reflect the performance of these tools on the same baseline, and therefore, this setting is widely used in this field, and 3) in addition to malware analysis, some other relevant and important applications (e.g., patch analysis [3]; CVE search) would require to precisely match equivalent code fragments. For instance, in searching the infamous Heartbleed vulnerability, OpenSSL version 1.0.1h, though manifests similar functionality as to version 1.0.1f (which is vulnerable), has been patched and is free from Heartbleed.

2.2 Binary Code Equivalence Checking

In addition to popular DNN-based representation learning, another line of research is to perform code equivalence checking, using program input-output relations obtained through symbolic execution (SE) [4], [20], [24], [25], [26], [27]. Given symbolic formulas representing binary code input-output relations, constraint solver is then used to check the equivalence of symbolic formulas. Equivalence checking results in strong resiliency to challenging settings such as compiler optimization and obfuscation, since these settings should not change program input-output relations.

Working Example. We now give a simple working example to demonstrate how program equivalence is checked using SE. Consider the code fragment below:

```
a = 4*a + 15;
b = a - 15;
p = b + 2;
```

```
m = 4*m + 10;
n = m - 10;
s = n + 2;
```

where a and m are the inputs of two code fragments, respectively. SE represents inputs as free symbols and interprets each statement within the symbolic domain. In our case, the output formulas are shown below:

```
p = f1(a) = 4*a + 2
```

```
s = f1(m) = 4*m + 2
```

And the equivalence of the above code fragments can be checked by forming the following constraint

$$a = m \wedge p \neq s$$

where the constraint solver searches the input space of a and m to check whether there are two inputs a and m that make the output formulas inequivalent. In case the constraint solver finds *no* satisfiable solutions (i.e., the solver yields `unsat`), these two code snippets are rigorously checked as equivalent.[1]

Limitation. The proposed technique gives rigorous proof of program equivalence. Nevertheless, SE and constraint solving suffer from *low scalability*, due to path explosion, reasoning complex constraints, and domain-specific challenges for binary analysis [28]. To date, equivalence check-based methods are mostly used for basic blocks or execution traces comparison [3], [4], [20].

Binary Classification Task. We note that the equivalence checking enabled by SE denotes a *binary classification* task. That is, instead of computing a floating number representing the similarity score (as how DNN-based binary code search tools do; Section 2.1), equivalence checking primarily answers a "yes/no" question. Accordingly, the standard classification errors can be defined as follows:

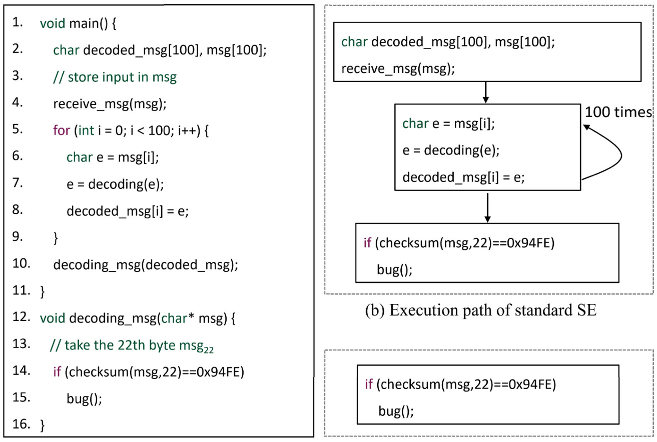
- *false positive (FP)*, which implies that assembly functions of different functionality are treated as equivalent.
- *false negative (FN)*, denoting that assembly functions of identical functionality are treated as inequivalent.

While equivalence checking cannot be used to directly compute a top-k accuracy, we use equivalence checking enabled by USE to eliminate false alarms made by DNN models, as will be introduced in Section 4.

2.3 Under-Constrained Symbolic Execution

To address the scalability issue of SE, USE [19] is proposed to directly check arbitrary code components (e.g., a function)

[1] Side effects are usually not considered when checking equivalence.



```
1.  void main() {
2.    char decoded_msg[100], msg[100];
3.    // store input in msg
4.    receive_msg(msg);
5.    for (int i = 0; i < 100; i++) {
6.      char e = msg[i];
7.      e = decoding(e);
8.      decoded_msg[i] = e;
9.    }
10.   decoding_msg(decoded_msg);
11. }
12. void decoding_msg(char* msg) {
13.   // take the 22th byte msg22
14.   if (checksum(msg,22)==0x94FE)
15.     bug();
16. }
```

(a) Sample program for message decoding

```
char decoded_msg[100], msg[100];
receive_msg(msg);

char e = msg[i];
e = decoding(e);
decoded_msg[i] = e;

if (checksum(msg,22)==0x94FE)
  bug();
```

(b) Execution path of standard SE

```
if (checksum(msg,22)==0x94FE)
  bug();
```

(c) Execution path of USE

Fig. 2. Comparing the high-level technical concepts (in terms of path coverage) between SE and USE using "bug finding" as an example.

rather than the entire program, thus reducing the complexity of SE in a principled and systematic way. Additionally, because certain complex constraints on inputs are not modeled, constraint solving is likely to become easier. Existing research has illustrated that USE can allow to directly check code fragments deeply hidden in the call chain [19].

To illustrate the high-level technical difference (in terms of path coverage) between SE and USE, Fig. 2a presents a case to analyze a message decoding program and identifies a bug in `decoding_msg`. The `main` function receives the encoded message with `receive_msg`, and performs decoding process within a loop statement. The decoded message is then passed to function `decoding_msg`, where a bug (marked as `bug` at line 15 in Fig. 2a) is hidden within the `if` branch. SE can be impeded in analyzing this simple case due to high computing resource usage and verbose constraints. As shown in Fig. 2b, SE starts from the `main` entry point to interpret program statements with symbolic variables. When analyzing the loop statement, symbolic variables in memory are increasingly created, and each symbolic value processed by `decoding` may likewise continue to grow. Therefore, the memory usage could become unrealistic, and the produced symbolic constraints may be too complex to solve.

USE reduces the complexity in a principled manner. To reach `bug`, USE directly analyzes `decoding_msg`. The resulting path, as shown in Fig. 2c, imposes no complex constraint on the decoded message $\texttt{msg}_{22}$ and likely induces a much easier constraint to solve. More expensive whole program analysis can be delayed until needed.

Limitation. By directly analyzing arbitrary code fragments, USE relaxes constraints over inputs of code fragments and could consequently induce much simpler constraints. However, when using USE for equivalence checking, it can induce *false negatives* by treating two equivalent code fragments as unequal. Section 2.2 has introduced equivalence checking by constructing constraints to check the presence of inputs that lead to output deviations. Two code fragments can pass the equivalence checking in case *no* inputs induce output deviations. However, USE, by relaxing constraints over inputs, could find satisfiable solutions which are indeed invalid when taking the path prefix from `main` to the target code fragments into account. Again, successfully finding a satisfiable solution means that two code fragments fail the equivalence checking. Overall, USE, in principle, should deliver a *complete*, efficient, yet *unsound* equivalence check, potentially inducing *false negatives* which are usually undesirable. We will continue analyzing the completeness of our customized USE implementation, namely `BinUSE`, in Section 4.

2.4 Application Scope

The search granularity of our proposed technique, `BinUSE`, is at the level of functions. As clarified in Section 2.1, This design decision is aligned with most works in this field [6], [8], [12], [13], [14], [15], [18], [23]. Nevertheless, the core technique of `BinUSE`, i.e., USE, is generally applicable to code components of different granularities [19].

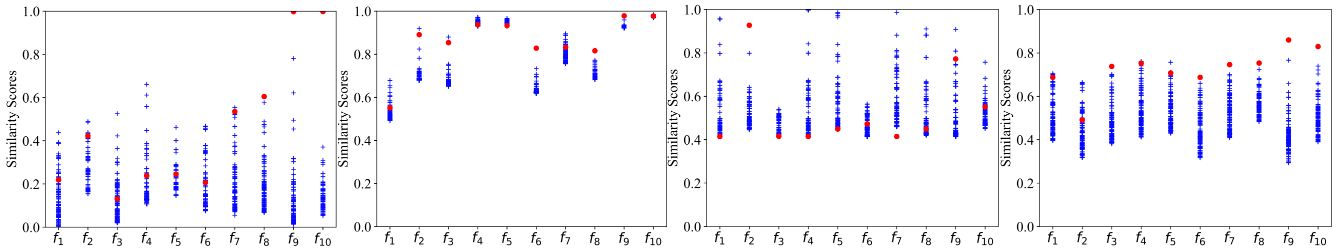
As reported in Table 1, USE-based equivalence checking is shown to deliver a general enhancement of cutting-edge DNN-based binary code search. Furthermore, `BinUSE` should be applicable to augment other code search tools, as long as they do not use rigorous semantics information (e.g., input-output relations) for matching. We focus on DNN-based binary code search, as DNN-based binary code matching has been shown to outperform most conventional methods [14], [15], [16], [18]. In Section 8.5, we show the generalizability of `BinUSE` by boosting binary code search tools following conventional structure-level comparison.

3 Research Motivation

Popular DNN-based binary code search learns code representations (i.e., "code embedding") that can distinguish similar binary code components with the rest [6], [8], [9], [10], [12], [14], [17], [29]. DNN-based approaches learn code representations from lexical, control structure, or data flow facts, e.g., two instructions that access the same memory location [13]. A well-trained DNN model will convert input binary sample (or machine instructions) into numerical vectors, where two similar programs should have a closer cosine distance. By primarily learning from "fuzzy" and lightweight data and control features, DNN-based approaches exhibit high flexibility and scalability, promoting analysis of large-scale binary samples. However, the learned lexical, control, or data features do *not* necessarily and precisely denote functionality. In short, we deem the learned embedding representations mainly suffer from the following two drawbacks:

- *Low Discriminability.* This drawback denotes that DNN models can treat logically different functions as similar. Accordingly, low discriminability leads to reporting many FP matching results.
- *Low Robustness.* Overall, robustness refers to the resistance across a variety of imperfect conditions when running the software or algorithm. Accordingly, low robustness implies that DNN models can suffer from matching functions sharing equal logic but appear to be syntactically different. In general, low robustness shall lead to reporting many FN matching results.

Motivating Example. We deem that the low discriminability and robustness as a common observation of existing DNN-based models, which, to a certain extend, has also been pointed out by recent DNN-based binary search tools such as `asm2-vec` [14]. Nevertheless, existing DNN-based tools, including



(a) BinaryAI (b) asm2vec (c) ncc (d) palmtree

Fig. 3. Matching ten randomly-selected functions from the `coreutils` dataset using four popular DNN-based function search tools. For each case, we report the top-50 functions matched by DNN models (the top-k matching has been formalized in Section 2.1). The red dot in each figure denotes the true positive match (which should manifest the highest similarity for each case, if these DNN-based tools are making accurate predictions), while the blue dots (we have 49 blue dots for each case) are irrelevant functions in the top-50 matched functions.

recent "semantics-based" approaches like `asm2vec`, though good at learning a scalable view of code representations, generally fail to understand semantics precisely. The rest of this section presents a motivating example to illustrate the relatively low discriminability and robustness of popular DNN-based tools. The Linux `coreutils` program suite has 105 programs, where each program has on average 103.7 functions. Without loss of generality, we randomly select ten programs and use one function from each program to launch a function matching task. The detailed setup has been given in Section 2.1, and we measure top-50 accuracy. We benchmark four popular DNN-based tools, `BinaryAI` [15], `asm2vec` [14], `PalmTree` [18], and `ncc` [13]; we discuss details of these models in Section 8.1. For this task, we use `gcc -O0` to compile the target function f_t and `gcc -O3` to compile the RP.

Fig. 3 presents the comparison results, where red dots denote the similarity score of comparing assembly functions compiled from the same source functions, while blue dots denote similarity scores of comparing other functions. Ideally, a high similarity score for red dots indicate DNN models' robustness toward compiler optimization: assembly functions compiled from the same source function, though exhibiting distinct syntactical appearance, have identical semantics. Accordingly, discriminability is reflected when blue dots are well separated, meaning that assembly functions compiled from different source functions show dramatic differences in the view of DNN models.

We report that out of 4×10 test cases, 20 cases have the red dots within the top-5 highest similarity, and 13 cases have the red dots in the top-1 highest similarity (e.g., f_9 and f_{10} in the `BinaryAI` evaluation). Cases like f_7 in Fig. 3c and f_5 in Fig. 3b are even at the 50th (lowest among top-50 cases), indicating a very low similarity score. In short, for the evaluated cross optimization setting (`gcc -O0` versus `-O3`), we find that DNN models struggle to match the target function to its true positive match in RP, indicating low robustness toward compiler optimization. On the other hand, it is also obvious that dots are not well separated from each other in many cases: for instance, dots in f_7 in Fig. 3b and f_6 in Fig. 3c are very close with each other, indicating that DNN models struggle to distinguish assembly functions with distinct semantics.

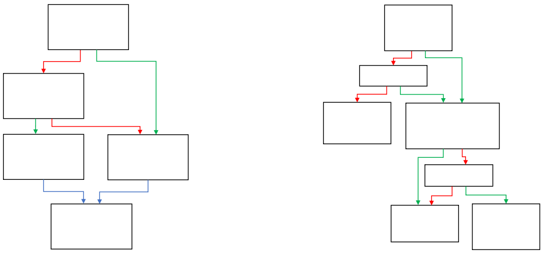
Fig. 4 presents a false prediction: two assembly functions compiled from different source code are incorrectly deemed "similar." This is likely due to the similarity of the "contextual flow graph" extracted by `ncc` (see Section 8.1 for its model details), which, cannot reflect the functionality deviations. Our study also shows that function `rpl_fflush` and its optimized version (presented in Fig. 11 when discussing evaluation results in Section 8.5), are assigned with a very low similarity score. The optimized `rpl_fflush` has a largely changed CFG, appearing structurally different but indeed retaining the same logic.

Clarification. We clarify that the compilation setting used in our motivation example denotes an *easy task*. Soon in our evaluation (Sections 8.5, 8.6, and 8.7), we show that DNN-based tools, including `BinaryAI` and `PalmTree` which show relatively better accuracy in this motivating example, generally struggle in matching assembly functions from Linux `coreutils` and `binutils` test suites, and real-world complex software like OpenSSL and Wireshark, particularly under cross-architecture, cross-compiler, and obfuscation settings. This indicates that low robustness and discriminability are general concerns for today's DNN-based function search tools.

4 Research Overview

Fig. 3 has illustrated that modern DNN-based binary code search learns code representation from coarse-grained features and exhibits low discriminability and low robustness. Hence, typically top-k matched functions have close similarity scores, while ground truth matches might not have high enough similarity due to challenging settings such as compiler optimizations or obfuscations.

Our preliminary study manually inspected the top-k matching results of these DNN models, and we suspect that a simple equivalence check could effectively reduce FPs. For instance, we can feed the target function f_t and a function in the top-k match with the same input, and compare their outputs to see if they differ. These input-output relations can act as an oracle, quickly exposing and shaving



(a) `rpl_fflush` compiled by `gcc -O0` (b) `xrealloc` compiled by `gcc -O3`

Fig. 4. Two assembly functions in `coreutils` program `shuf` is *incorrectly* deemed as "highly similar" by DNN-based binary matching tools.

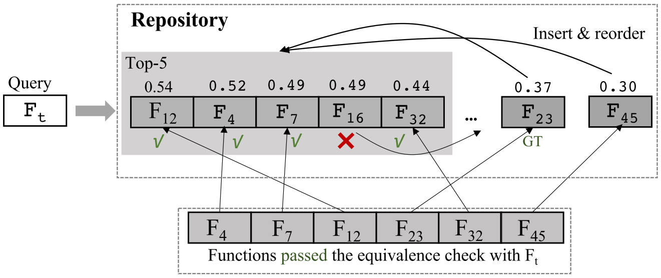


Fig. 5. Overview: regulating top-k ($k = 5$) retrieval of DNN-based tools with low-cost equivalence checking enabled by `BinUSE`. GT denotes the ground truth for this query.

logically different functions, whereas the true positive matches gradually bubble out in a top-k retrieval.

Workflow Overview. Fig. 5 depicts the overview of our workflow to enhance the top-k retrieval of DNN-based tools. In short, this research aims to provide a low-cost equivalence check that makes binary (true/false) decisions on whether two assembly functions are identical. This way, we can smoothly regulate the search results of DNN-based tools, by reducing the FPs in the top-k retrieval. For instance, many "blue dots" in our motivating example (Fig. 3) can be decided as dissimilar with the target function by simply executing both functions with an input and comparing the equivalence of their outputs. With such a low-cost equivalence check, we can easily shave these FPs off the top-k retrieval, thus optimizing cases like f_7 in Fig. 3b and f_9 in Fig. 3c.

Checking the equivalence of software is overall expensive. Therefore, as a practical tradeoff, we accept an equivalence check with relatively lower precision, in the sense that two functions that pass the check may nonetheless be different. This way, by augmenting DNN-based binary code search with low-cost equivalence checking, a practical synergistic effect can be achieved, resulting in speedy service with higher accuracy. In the following section, we discuss the pros and cons of each equivalence check proposal in accordance with Fig. 6, and illustrate our implemented equivalence check (referred to as `BinUSE`) in Fig. 6e.

Options for Function Equivalence Check. Fig. 6 analyzes potential options to design assembly function equivalence check. We now discuss each option in details.

Fig. 6a: Concrete Execution-based Equivalence Check. One option is to use randomly-sampled values as inputs and compare the concrete execution outputs [21], [29], [30], [31], [32]. However, as illustrated in Fig. 6a, this method may cover only small input spaces, likely inducing high FPs by treating different functions as equivalent. We also clarify that directly executing assembly functions without setting up the proper execution context can be very challenging [31].

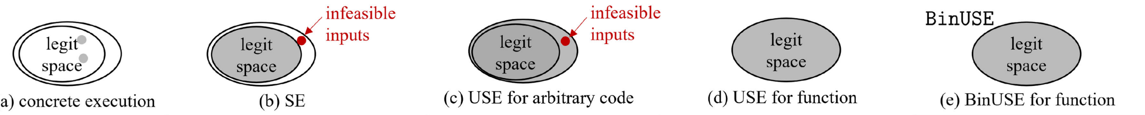
Fig. 6b: SE-based Equivalence Check. On the other end of the spectrum, SE precisely models the input constraints and constructs equivalence checks within the legitimate input spaces. Therefore, it should be accurate. Nevertheless, SE suffers from low scalability, whose execution can hardly reach functions deeply hidden in the call chain. In Section 8.3, we show the low speed of performing whole-program SE using recent versions of KLEE and MoKLEE [33], [34].

Fig. 6c: USE-based Equivalence Check. USE can launch symbolic reasoning at arbitrary program points, and it promotes standard SE by skipping expensive path prefixes. However, ignoring path prefix, as mentioned in Section 2.3, indicates that USE cannot model constraints on inputs of the target code fragments. Therefore, USE overly explores the full input space. Section 2.3 has clarified that USE enables complete albeit unsound equivalence checking, as it may find counterexamples outside the legitimate input space, thus violating the equivalence checking constraint given in Section 2.2.

Fig. 6d: USE-based Function Equivalence Check. Despite the general difficulty of addressing unsoundness, our observation on real-world software induces a key assumption in this research:

> In general, functions in real-world programs adhere to the *defensive programming* principle [35], [36], [37], which states that no particular function should make assumptions about its inputs (e.g., a pointer passed by the caller could be invalid). That is, the inputs to functions may be any value in the input space.

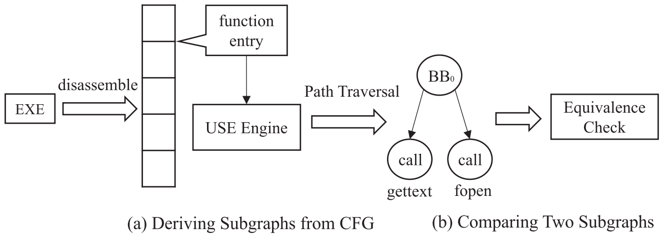
As illustrated in Fig. 6d, this assumption offers a unique opportunity to provide a "soundy" equivalence check, particularly for functions, because when analyzing functions in real-world software, the legitimate input space should be aligned with the complete input space that USE can explore. The likelihood of performing unsound equivalence checking in practice should be slim. On the other hand, we clarify



	(a) concrete execution	(b) SE	(c) USE for arbitrary code	(d) USE for function	(e) BinUSE for function
soundness	√	√ sound	X unsound	soundy (?? FN rate)	X (very low FN rate)
completeness	X (low coverage ➔ high FP)	√? assume no side effect; cover full CFG; pairwise path match	√? assume no side effect; cover full CFG; pairwise path match	√? assume no side effect; cover full CFG; pairwise path match	X (low FP rate)
speed	fastest	too slow for non-trivial code	still very slow	still very slow	56.6 mins to check two `coreutils` executable

Outer ellipse: entire input space (e.g. 2^{32} -1 for `uint32`). **Inner ellipse**: legit input space (specified by path prefix). **Shaded region**: input space explored by different methods.

Fig. 6. Different options for equivalence checking. Our approach, as an optimized USE implementation particularly over assembly functions, is referred to as `BinUSE` in this paper. Comparing to standard USE, `BinUSE` further trades completeness for speed.



(a) Deriving Subgraphs from CFG (b) Comparing Two Subgraphs

Fig. 7. High-level workflow of `BinUSE`.

that in case users aim to leverage USE to compare arbitrary code fragments, e.g., a recently patched code block with those un-patched versions [38], the aforementioned key assumption is *not* applied. Therefore, users might need to resort to using USE (while suffering from unsoundness) or standard SE (with low scalability).

Indeed, our evaluation (Table 7) shows that for our customized USE (dubbed `BinUSE`; see below), the average FN rates are very low (4.2%), even when very challenging cross-optimization, cross-compiler, cross-architecture, and obfuscation settings are assessed. More importantly, we manually analyzed `BinUSE`'s errors; we report that they are *not* due to the violation of the defensive programming assumption mentioned above. We list results that can induce FPs and FNs to `BinUSE` in Section 9. In summary, this principle is seen as obeyed by programmers of popular program suites like `coreutils`, `binutils`, and complex real-world software like OpenSSL and Wireshark. It's also interesting to launch an empirical study, as part of this paper's future work, to determine whether common software adheres to defensive programming concepts. Given that said, our preliminary study shows that the standard USE design continues to suffer from slow analysis speed (see paragraph *Baseline* in Section 5). Therefore, we demand further optimization on speed, as discussed below.

Fig. 6e: `BinUSE`-based Function Equivalence Check. We depict the USE approach launched in this work, namely `BinUSE`, in Fig. 6e, where we further *optimize*the standard USE approach by trading completeness for speed, particularly for equivalence checking of assembly functions. `BinUSE` launches USE traversal from the entry point of each assembly function to explore each path. When traversing a path, it stops whenever reaching the first external function callsite. In this research, we assume that external function callsites (e.g., a function call to `libc` functions) representing informative and critical nodes on CFG. `BinUSE` computes the symbolic formulas of external callsite's inputs to form the symbolic constraints of each path. To match two functions, `BinUSE` explores matching symbolic constraints collected from every path in each function. `BinUSE` is not sound. However, the average FN rate is very low according to our empirical results. Moreover, `BinUSE` enables low-cost checks under various challenging settings (e.g., cross-architecture). `BinUSE` can finish checking two `coreutils` executables within 56.6 CPU minutes (on average 25s per pair of functions), including all symbolic execution and constraint solving tasks. This indicates its practical usage in production.

5 Design of `BinUSE`

This section introduces the design of `BinUSE`, a USE-based equivalence checking tool. Procedures to regulate top-k retrieval of a DNN model with `BinUSE` will be given in Section 6. Fig. 7 depicts the high-level workflow of `BinUSE`. Overall, given an input executable file, `BinUSE` first performs reverse engineering to recover the assembly function information (see Section 2.4 for reverse engineering assumptions). Then, it starts from the entry point of each assembly function to launch USE path by path (Fig. 7a), where each path traversal stops when reaching the first external callsite. As a result, a subgraph will be generated, where each leaf node corresponds to one external callsite. To compare two assembly functions, we compare their derived subgraphs (Fig. 7b), by launching constraint solving to cross-check the semantics equivalence of external callsite inputs and path constraints (see details in Sections 5.3 and 5.4). Before elaborating on the design of `BinUSE`, we first clarify our assumption regarding reverse engineering below and baseline in Section 5.1.

Assumption on Reverse Engineering. Our analysis is at the function-level, and it does not assume the presence of program symbols or debugging information. Stripped binary could be processed with no additional difficulty, as long as functions have been identified for use. Our analysis is also platform-neutral; we evaluated three cross-architecture settings for x86 64-bit, x86 32-bit, and ARM architectures. We also evaluate different compilers (`gcc` and `clang`), optimization levels, and commonly-used obfuscation methods. `BinUSE` manifests generally encouraging results regarding all of these settings. Nevertheless, we clarify that assembly function search task by nature can be impeded by function inlining; if a query function is inlined into its callers, the function-level search can be undermined. We clarify that this is a common limitation in this line of works, not only `BinUSE`.

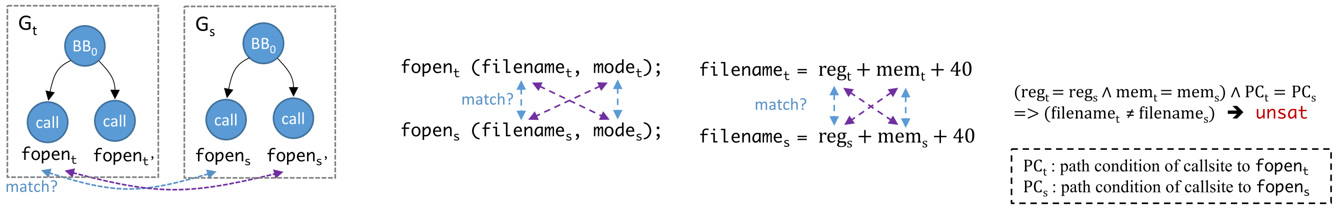
5.1 Baseline

To conduct function equivalence checking, the baseline approach is to perform standard intra-procedural analysis starting from the entry point of a function and iterating every execution path. Free symbols are created whenever loading from unknown data, including function parameters, global data, and other memory regions. We then collect output symbolic formulas of CPU registers and memory at the exit point of a path to construct an input-output relation. This denotes a standard setting in this line of work [20], and conceptually corresponds to the standard USE scheme illustrated in Fig. 6d.

We tentatively explored this scheme; while this design enables a full-fledged modeling of input-output relations, our preliminary study reveals its low scalability. The induced intra-procedural CFG is often very complex. Additionally, we implemented the standard CoP algorithm [4] to reduce complexity by extracting the longest common subsequence (LCS) of equivalent basic blocks on a path. However, certain implementation details (e.g., block splitting and merging) remain unknown. While the USE method primarily reduces the complexity of whole program SE, our tentative study in analyzing real-world executables illustrates a demanding need for further calibration and optimization.

5.2 Generating Subgraphs From CFG

Considering the difficulty of fully exploring the CFG of a function, we first extract a subgraph. The extracted subgraph


G_t
BB_0
call
call
fopen_t
fopen_t'
G_s
BB_0
call
call
fopen_s
fopen_s'
match?
fopen_t (filename_t, mode_t);
match?
fopen_s (filename_s, mode_s);
filename_t = reg_t + mem_t + 40
match?
filename_s = reg_s + mem_s + 40
(reg_t = reg_s ∧ mem_t = mem_s) ∧ PC_t = PC_s
=> (filename_t ≠ filename_s) ➔ unsat
PC_t : path condition of callsite to fopen_t
PC_s : path condition of callsite to fopen_s

(a) external callsite permutation (permutations on $\texttt{fopen}_{t'}$ are omitted for readability)

(b) call parameter permutation

(c) symbol permutation

(d) equivalence check

Fig. 8. Comparing two subgraphs and callsites.

should retain representative features of the corresponding CFG, and reasonably reduce the analysis complexity. In this research, we deem *external function callsites* as representative and critical nodes on CFG. From a holistic perspective, most compiler optimizations and obfuscations are designed to perform intra-procedural transformations [39]; external function calls, mainly formed by calling dynamically linked libraries, should not be changed. Soon in Section 5.5, we will introduce an optimization to take return instructions (`ret`) as another representative nodes into this subgraph.

`BinUSE` is designed to traverse every execution path from the entry point of each assembly function. When encountering a loop, as a common workaround, we unroll the loop (the unroll factor is currently set as 5). When analyzing an execution path, `BinUSE` recursively inlines every callsite on the path that has its implementation in the binary code. `BinUSE` stops whenever encountering the first *external callsite*. Consistent with standard SE, we create free symbols to represent values stored in registers or memory cells when they are accessed for the first time and the stored symbolic value is unknown. When we meet an external callsite, we collect symbolic formulas for each function call's input (see below for information on recovering function prototype) to form the "outputs" of this path. We also record path constraints as the *pre-requisite* of reaching this external callsite.

5.3 Comparing Two Subgraphs

Fig. 8 illustrates the processes required to compare two subgraphs G_t and G_s derived from the assembly function f_t and f_s, respectively. As shown in Fig. 8a, the CFG traversal and stop criteria proposed in Section 5.2 forms a subgraph with a reasonable degree of complexity from each CFG, with each leaf denoting an external function callsite. As a result, comparing two CFGs becomes a matter of comparing these two subgraphs: we compare each callsite iteratively until we find that a permutation that makes external callsites in G_t pairwise equivalent with external callsites in G_s. Note that we allow only a subset of callsites in G_t to match another subset of callsites in G_s. The reason is that compiler optimizations can sometimes eliminate C library function calls, therefore allowing a subset of library calls to match another subset of library calls would not neglect to match G_t with highly optimized G_s. In contrast, two functions are deemed inequivalent if no permutations are found.

While comparing callsites pairwise in G_t and G_s can introduce a lot of permutations, we clarify that we will proceed with heavyweight equivalence checking (see Fig. 8d) only when two callsites refer to the same external function. External function names can be obtained from even stripped ELF binary code. For instance, we will compare two callsites in G_t and G_s in case they both refer to `fopen`.

C Library Calls Replacement. When certain input parameters of a C library call are constant, a compiler might replace this C library call with others [40]. In addition, compilers might occasionally replace a common C library call with a safer version. For instance, by replacing `printf` with `__printf_chk`, stack overflow is detected prior to computing the results [41].

Library callsites matching deems a critical step in equivalence check, and to this end, we manually collected the following list where each entry is also a list L. `BinUSE` is constructed to treat library calls within each L as identical. For instance, in addition to comparing two `printf` callsites (Section 5.3), we consider a `printf` callsite and `__printf_chk` to be identical and proceed to construct constraints for the equivalence check. To our best knowledge, this list encompasses *all* possible C library replacements discovered in our test cases, which include Linux `coreutils` and `binutils` test suites, as well as a CVE database including vulnerable functions from complex software like OpenSSL, Wireshark, bash, and ffmpeg. As such, we assume that the following list is comprehensive and can benefit future research in this field.

```
SIMILAR_LIB_CALLS = [
    ['gettext', 'dgettext', 'dcgettext'],
    ['printf', '__printf_chk'],
    ['fprintf', '__fprintf_chk'],
    ['sprintf', '__sprintf_chk'],
    ['snprintf', '__snprintf_chk'],
    ['strcpy', '__strcpy_chk'],
    ['asprintf', '__asprintf_chk'],
    ['realpath', '__realpath_chk'],
    ['siglongjmp', '__longjmp_chk'],
    ['toupper', '__ctype_toupper_loc'],
    ['tolower', '__ctype_tolower_loc'],
    ['fseeko', 'fseeko64'],
    ['lseek', 'lseek64'],
    ['open', 'open64'],
    ['__xstat', '__xstat64']
]
```

Calculating Quantitative Matching Scores. Assuming there are n paths in G_t (each path terminates in an external callsite), and we identify p paths whose external callsites have semantically equivalent callsites in G_s (see Section 5.4), a score s expressing the confidence of treating assembly function f_t and f_s as equivalent is calculated as $^p/_n$. This confidence score will be saved and used when calibrating results of DNN models. Given that said, we clarify that when analyzing `coreutils` executables, 87.9% cases have a confidence score of 1.0, indicating that for the majority of the time, all paths on G_f are matched to G_s.

TABLE 2
Distribution of Number of Parameters in Each External Callsite

≤ 2	3–4	5	6	7	≥ 8
73.0%	23.0%	3.9%	$<$ 0.1% (1 case)	$<$ 0.1% (1 case)	0

5.4 Comparison of Two Callsites

To compare two callsites in machine code, we first recover function prototypes to determine the number of parameters to extract. To do so, we leverage a commercial software decompiler, IDA-Pro [42], to recover the function prototype information. While prior research has demonstrated IDA-Pro's inadequate support for function information recovery [43], we find that the standard C library function information is well-maintained in the FLIRT database [44] of IDA-Pro. Hence, using IDA-Pro (version 7.3) ensures the credibility of analyzing external callsites to a great extent. Nonetheless, we agree that if the executables contain some user-defined (or some third-party defined) library calls, FLIRT is unable to handle them. Recovering such information requires inference about the number of function parameters; recent advances in recovering function prototype information can be referred [43], [45], [46]. Given a callsite of N parameters, we extract N symbolic formulas following the calling convention on corresponding architectures. For instance, we load N memory cells from the top of the stack for x86 32-bit architectures.

Call Parameter Permutation. As shown in Fig. 8b, to check two callsites, we search for a permutation of function parameter pairwise match. We note that instead of simply comparing the ith parameter Arg_t^i and Arg_s^i between two callsites, we allow a more conservative design by searching for the existence of a permutation. For instance, two callsites of `fopen` in Fig. 8b will be deemed equivalent, in case `filename`$_\mathtt{t}$ matches `mode`$_\mathtt{s}$ and `filename`$_\mathtt{s}$ matches `mode`$_\mathtt{t}$. Note that this design makes our equivalence check more conservative, being robust to compiler optimizations, potential obfuscations, but could introduce FPs. As aforementioned, compiler optimizations might replace certain library function calls into others, where function parameters might be relocated or removed. Considering the following two C library functions:

```
char* gettext(char *msgid)
char* dgettext(char *domainname, char *msgid)
```

where compiler optimizations may replace a callsite of `dgettext` with `gettext` when `domainname` is constant. To match a callsite of `dgettext` with a callsite of `gettext` in highly optimized code, we need to match the second parameter in `dgettext` with the first parameter in `gettext`.

The above case also reveals the observation that two "identical" function callsites could have different number of arguments. To this end, our current implementation would deem two callsites as equivalent, by setting a threshold γ such that γ arguments in the first callsite should be pairwisely matched with the other callsite. This way, we practically alleviate obstacles raised by inconsistent number of function parameters. Overall, we consistently and conservatively design `BinUSE` to be robust against various real-world obstacles, principally eliminating potential FNs. Note that given the number of parameters in two callsites, we always use the smaller number as the dividend to compute a ratio $r \leq 1$ and compare with γ. γ is empirically decided as 0.5 in our current implementation.

The call parameter permutation offers a more conservative design and is robust to various compiler optimizations that may change callsites (an example is given above). Nevertheless, if the two function callsites involve many parameters, the number of permutations to check will grow exponentially. Our empirical observation demonstrates that this permutation method does not impose excessive overhead, as nearly all commonly used software has a limited number of function parameters. To further justify this design decision, we give a distribution of the number of parameters for all external callsites encountered in our test cases in Table 2. Note that distribution data in Table 2 is obtained from analyzing all of our evaluated programs, including Linux program suites `coreutils` and `binutils`, and a CVE database containing vulnerable functions in real-world complex software like OpenSSL, ffmpeg, Wireshark (see details in Section 8). It is observed from these empirical results that most external callsites has less or equal to 2 parameters, and nearly all external callsites have less or equal to 5 parameters. As a result, we deem that our design decision of permutating parameters does not add significant cost, but can help make `BinUSE`'s overall design more conservative and robust.

Equivalence Checking of Two Parameters. We then construct constraints for equivalence check of a pair of parameters (see Fig. 8d). Let PC_t and PC_s be the path constraints collected from the entry point of the analyzed assembly functions reaching to these two callsites. Let Arg_t^i and Arg_s^j be the ith and jth parameters of these two callsites. Formally, we check

$$X = \pi(Y) \wedge (PC_t(X) \wedge PC_s(Y)) \Rightarrow (Arg_t^i(X) \neq Arg_s^j(Y))$$

where $X = [x_0, \ldots, x_m]$ denotes the list of symbols used by either Arg_t^i or PC_t. Similarly, $Y = [y_0, \ldots, y_n]$ denotes the list of symbols used by either Arg_s^j or PC_s. As shown in Fig. 8c, We check to see if there existed a permutation $\pi(Y)$ that would pairwisely match X under the conjunction of path constraints $PC_t(X) \wedge PC_s(Y)$ to make the above constraint *`unsat`*. In case such a permutation can be found, Arg_t^i and Arg_s^j are checked as equivalent. We note that permutation can make `BinUSE` more conservative (less FNs) and robust against changes introduced by cross-architecture, cross-compiler, and obfuscations. Nevertheless, this design decision may potentially introduce extra FPs. We view this as a design tradeoff to calibrate FP/FN rates.

In addition to permutation, at this step we also normalize a constant in symbolic formulas if it denotes a memory address. We conduct a straightforward approach of determining the constant that represents a memory address. First, when performing USE path by path, we flag a constant as a memory address whenever we observe that it is used to construct the base address of code pointers. Second, we make an assumption shared by many advanced static disassemblers [47], [48], [49] such that a constant will be treated as a memory address if it points to the data or text sections of an ELF format executable.

Symbols (X and Y) are collected by traversing the constructed symbolic formulas and path constraints. Each time when checking the above constraint, we set the timeout for the employed solver, Z3 [50], to N seconds. In case the SMT solver yields an `unsat`, or it cannot find a `sat` solution within N seconds, two function call arguments are deemed equivalent. N is set as 15 seconds for our current implementation, and our observation shows that most tasks can be finished within this threshold. Setting a timeout may cause inequivalent arguments to be treated as equivalent (i.e., FPs), since no counterexamples were found within N seconds. Setting a timeout, however, does not introduce FN, but can accelerate the analysis of large-scale binary samples.

5.5 Optimization: Collecting and Comparing Path Constraints on Paths Without External Callsites

While the traversal strategy introduced in this section can subsume most real-world cases (see discussions in Section 8.2), some corner cases may exist to impede our analysis. In particular, there may be *no* external callsites on an execution path. In such cases, instead of simply skipping this path analysis, we collect the path constraints PC from the function entry point until reaching the return instruction (`ret`) at the end of the execution path. However, if we cannot construct any path condition when traversing this path, we skip comparing this path. From a holistic view, each return instruction `ret` will be treated as a special "external callsite" with no parameters, and to decide if two paths with no external callsites can be matched, we use the following constraint to check their associated path constraints PC_t and PC_s

$$X = \pi(Y) \Rightarrow (PC_t(X) \neq PC_s(Y))$$

where $X = [x_0, \ldots, x_m]$ denotes the list of symbols used by PC_t and $Y = [y_0, \ldots, y_n]$ denotes the list of symbols used by PC_s. That is, we check if there was a permutation $\pi(Y)$ to pairwisely match X to make the above constraint *unsat*. If not, two path contraints are checked as equivalent.

This optimization will extend the subgraph generated during our symbolic traversal (Section 5.2) with extra nodes, denoting the path constraints collected from execution paths with no external callsites. Then, to compare two formed subgraphs G_t and G_s derived from assembly function f_t and f_s (Section 5.3), we still cross compare nodes on G_t and G_s denoting external callsites, and also cross compare nodes on G_t and G_s denoting paths with no external callsites using the above constraint.

According to our empirical results, this optimization can facilitate comparing a number of paths with no external callsites, and further improve the accuracy of DNN-based tools. We compare the enhanced top-1 accuracy of four evaluated DNN models (model details are clarified in Section 8.1) in Table 3. We use the Linux `coreutils` test suite for the comparison, and the accuracy is computed by averaging all 12 comparison settings evaluated in this research (see Table 7 for details of these settings). In short, after using the optimization proposed in this section, `BinUSE` can further enhance the accuracy of DNN models for about 2.45% on average. We also discuss some other corner cases we have encountered and our workaround in Section 9.1.

TABLE 3
Demostrating Enhancement Over DNN Models Due to the Optimization Proposed in Section 5.5

	`BinaryAI`	`asm2vec`	`ncc`	`PalmTree`
w/o opt.	59.7%	45.1%	67.3%	62.6%
with opt.	62.1%	47.5%	67.5%	64.9%

Other Optimization Opportunities. It is worth noting that we also tentatively explored other optimization methods. For instance, currently, the customized USE scheme used by `BinUSE` does not consider the impact of the return values from external callsites. Suppose the input/output constraints of an external callsite (e.g., a callsite to the standard libc function `strcpy`) has been modeled by `BinUSE`, it is feasible to continue the symbolic execution with the modeled return values. From a holistic view, this may likely take additional constraints on the return values of external functions into consideration during equivalence checking, which may help to improve the true positive rates. Despite the promising potential of this optimization, we clarify that it is generally difficult to model input/output relations of external callsites, which may require considerable manual efforts to hardcode such information for `BinUSE`. We also notice that popular symbolic execution engines like KLEE have encoded such input/output constraints for some standard libc functions, which might be leveraged to enhance `BinUSE`. On the other hand, it is generally hard, if at all possible, to encode such information for some user-defined (or some third-party defined) library calls.

6 Boosting DNN-Based Tools

We now discuss strategies to refine ranked candidates of DNN-based binary code function search. Recall the problem formulation in Section 2.1 such that given a target function f_t, we compare f_t with repository functions $f_s \in RP$ and decide the logic similarity between each function pair. Suppose DNN models compare f_t with each $f_s \in RP$, and each comparison yields a similar score $Pred_{dnn}(f_t, f_s) \in [0, 1]$. Let $f_s \in P_{dnn}$ denote the top-k ranked candidates ($|P_{dnn}| = k$) and $N_{dnn} = RP \setminus P_{dnn}$. Similarly, after checking f_t with each $f_s \in RP$, `BinUSE` returns a usually small set of functions P_{use} passing the equivalence check, and each pair associated with a confidence score $Pred_{use}(f_t, f_s) \in (0, 1]$.[2] N_{use} denotes the negative predictions such that $N_{use} = RP \setminus P_{use}$.

Algorithm 1 presents the algorithm to boost DNN-based binary code function search with `BinUSE`, in accordance with the aforementioned notations. To ease the presentation, let P_{dnn} be a map data structure, whose keys are the top-k functions and values are the corresponding similarity scores. Similarly, P_{use} maps each function in the repository to its equivalence checking results (the confidence scores) with f_t. For a function $f_s \in RP$, Algorithm 1 handles three situations to boost top-k retrieval as follows:

$\underline{f_s \in P_{dnn} \wedge f_s \in P_{use}(\text{lines}4-6)}$. Overall, `BinUSE` accepts $\underline{Pred_{dnn}(f_t, f_s)}$ in case $f_s \in P_{use} \wedge f_s \in P_{dnn}$. That is, `BinUSE`

[2] Section 5.3 clarifies how to compute the confidence score; for our evaluation, 87.9% positive cases are associated with a confidence score of 1.0.

and DNN models reach to an agreement, and f_s will be kept in top-k P_{dnn}. In Algorithm 1, f_s is kept in a temporary map P together with its confidence score s' and similarity score s in P_{dnn} and P_{use}, respectively (line 6).

Algorithm 1. Enhancing DNN-Based Function Search and Retrieve Top-k Ranked Candidates

```
1:  function Enhancement($P_{dnn}$: dict, $P_{use}$: dict)
2:      $P \leftarrow$ dict()                          ▷ $P$ stores the final results.
3:      for $(f, s) \in$ items($P_{dnn}$) do
4:          if $P_{use}[f] > 0$ then                  ▷ $f \in P_{dnn} \wedge f \in P_{use}$
5:              $s' \leftarrow P_{use}[f]$
6:              $P[f] \leftarrow (s', s)$
7:          else                                     ▷ $f \in P_{dnn} \wedge f \in N_{use}$
8:              $P[f] \leftarrow (0.0, s)$
9:      for $(f, s) \in$ items($P_{use}$) do
10:         if $f \notin$ keys($D$) && $s \geq \alpha$ then   ▷ $f \in N_{dnn} \wedge f \in P_{use}$
11:             $P[f] \leftarrow (s, 0.0)$
12: $P \leftarrow$ sort($P$)                    ▷ Sort $P$ in descending order
13: return retrieve-top-$k(P)$              ▷ Retrieve top-$k$ from the sorted $P$
```

$f_s \in P_{dnn} \wedge f_s \in N_{use}(\text{lines}7-8)$. For such cases, we shave f_s from the top-k P_{dnn}. The reason is that BinUSE makes low FNs, and in case $f_s \in N_{use}$, we have a high confidence of taking f_s out of P_{dnn}. Note that f_s could still be retrieved in top-k, unless we find cases satisfying the condition on lines 9-11 and move them from N_{dnn} into P_{dnn}. Therefore, we still put f_s in P, though the "confidence score" for BinUSE to match f_s and the target function is 0.0, as illustrated in line 8.

$f_s \in N_{dnn} \wedge f_s \in P_{use}(\text{lines}9-11)$. Given that P_{use} may induce FPs due to its over-approximation, we resort to rely on a pre-trained threshold, α, to selectively accepts P_{use}. f_s will be added into P, only if its associated confidence score $Pred_{use}(f_t, f_s) \geq \alpha$ (line 10). We empirically decide α as 0.41 in the current implementation. See relevant discussions in Section 7.

P maps functions f_i to their associated confidence score s'_i (decide by BinUSE) and similarity score s_i (decide by the DNN models). On line 12, we reorder functions in P in descending order. That is, function f_i will be put ahead of f_j in P if the confidence score $s'_i > s'_j$, and if $s'_i = s'_j$, we further compare the similarity score to see if $s_i > s_j$. We finally return the top-k elements in P on line 13.

7 Implementation

We implement BinUSE with approximately 5,500 lines of Python code on the basis of a binary analysis framework angr [51]. By linking with the popular angr ecosystem which lifts assembly code into the platform-neutral VEX intermediate language, BinUSE can process executables from different architectures. More importantly, a rich set of analysis facilities (e.g., symbolic execution) are already available in angr, therefore saving the effort of building it from scratch. When analyzing each assembly function, we set a timeout threshold as 10 minutes. We only have negligible timeout cases (see Section 8.3).

Deciding the Threshold α. As discussed in Section 6, we define α as a threshold to regulate the calibration of DNN predictions. In this section, we illustrate an empirical procedure to decide the α. In general, setting a large α indicates overconfidence on BinUSE's analysis results (leaving more FNs in the calibrated final predictions). A small α, however, may overlook opportunities to match assembly functions and induce more FPs. Hence, we empirically decide α to maximize the portion of true positives among all positive predications made by BinUSE.

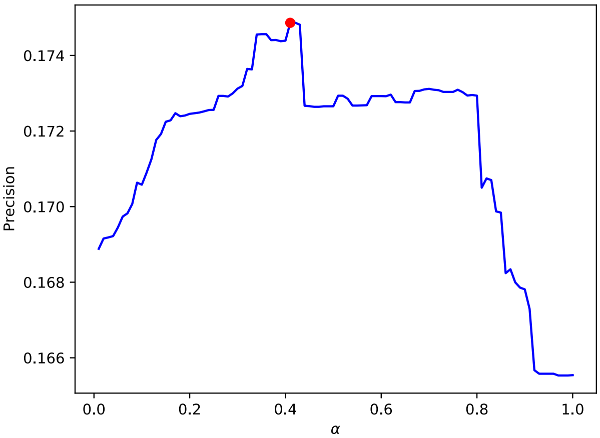


Fig. 9. Precision changes w.r.t. different α using the gcc -O0 versus gcc -O3 comparison setting. The optimal result is achieved when $\alpha = 0.41$.

To this end, we generating two sets of binary code by compiling coreutils using gcc with -O0 and -O3 optimizations. Let each coreutils program have on average n functions, and the entire coreutils test suite has m programs, we use BinUSE to cross compare in total $n \times m$ function pairs, following the standard setting introduced in Fig. 1. For instance, get_global_opt in ls compiled with gcc -O0 should be matched to the same function in ls compiled with gcc -O3. Given α, we keep positive predications made by BinUSE whose confidence score $s \geq \alpha$. We then compute precision $\frac{|TP|}{|T|}$, where $|T|$ denotes total number of positive cases, and $|TP|$ denotes the number of true positives in BinUSE's matching results. We iterate α every 0.02 step from 0.01 until 1.0 and plot Fig. 9. According to Fig. 9, $\alpha = 0.41$ induces the highest precision.

Generalization of α. Our employed test suite, coreutils, contains over 100 programs of diverse functionality, which are all commonly used on Linux. More importantly, while $\alpha = 0.41$ is decided over coreutils test cases compiled using gcc with -O0 and -O3 settings, this configuration is shown as effective over other challenging settings introduced by different compilers, optimizations, architectures, and obfuscations. Table 4 evaluates how different α changes the enhancement under different comparison settings. It is seen that $\alpha = 0.41$ constantly shows encouraging results *across all compilation settings*. This evaluation illustrates the generalization of our selected α in most usage scenarios.

Nevertheless, we admit that the software dataset suffers from non-negligible domain shift which makes it possible that this empirical selection of hyper-parameter is not generalizable *across different datasets*. Users can follow the same procedure to decide their optimal choice of α over other datasets. In fact, when benchmarking binutils test cases, we find that the optimal α is 0.68, rather than 0.41, though the latter configuration also achieves close precision.

TABLE 4
Average Top-1 Accuracy Enhancement Over the coreutils Dataset Using $\alpha = 0.41$ Over 11 Comparison Settings

Comparison Setting	Obfuscation	$\alpha = 0.01$	$\alpha = 0.21$	$\alpha = 0.41$	$\alpha = 0.61$	$\alpha = 0.81$
gcc -O0 versus clang -O3	-sub	+12.1%	+12.1%	+12.2%	+12.1%	+11.4%
gcc -O0 versus clang -O3	-bcf	+14.2%	+14.2%	+14.2%	+14.2%	+14.2%
gcc -O0 versus clang -O3	-fla	+20.2%	+20.1%	+20.0%	+19.5%	+18.4%
-m32 -O0 versus gcc -O3	NA	+9.7%	+9.9%	+9.9%	+9.7%	+9.7%
-m32 -O0 versus clang -O3	-sub	+8.4%	+7.9%	+8.1%	+8.1%	+8.1%
-m32 -O0 versus clang -O3	-bcf	+9.1%	+9.2%	+9.4%	+9.4%	+9.5%
-m32 -O0 versus clang -O3	-fla	+17.3%	+17.1%	+16.3%	+15.0%	+14.2%
arm -O0 versus gcc -O3	NA	+12.4%	+12.2%	+12.3%	+12.0%	+12.1%
arm -O0 versus clang -O3	-sub	+11.8%	+11.6%	+11.7%	+10.9%	+11.0%
arm -O0 versus clang -O3	-bcf	+15.3%	+15.0%	+15.0%	+15.0%	+15.0%
arm -O0 versus clang -O3	-fla	+18.9%	+18.6%	+18.2%	+17.2%	+16.2%

Note that $\alpha = 0.41$ is decided using the gcc -O0 versus gcc -O3 comparison setting (with no obfuscation), which is not *in these 11 comparison settings. We find that $\alpha = 0.41$ manifests at least top-3 best accuracy enhancement over its four competitors across all settings, illustrating decent generalization of $\alpha = 0.41$.*

8 Evaluation

8.1 Evaluation Setup

Dataset and Compilation Setting. We evaluate BinUSE using Linux coreutils dataset (ver. 8.28). coreutils dataset contains 106 programs. We compile programs with seven different settings (see Table 5). We use gcc 7.5.0 and clang 4.0.1 to compile programs. We compile programs with no optimization (-O0) and the highest optimization (-O3). To facilitate a cross-architecture comparison, we compile the binary code on three different architectures, 32-bit x86, 64-bit x86 and ARM. We report that each coreutils executable compiled with -O3 option has on average 103.7 functions. In other words, given a pair of coreutils executables, BinUSE needs to cross compare 103.7×103.7 assembly functions. Furthermore, we benchmark BinUSE using the Linux binutils dataset (ver. 2.36) in Section 8.7. binutils dataset contains 112 programs. Each binutils executable has on average 1,765.0 functions. Hence, given a pair of binutils executables, BinUSE needs to launch a much larger number of cross comparisons; see Section 8.7 for our optimization to reduce the number of comparisons. We use the same seven settings to compile binutils programs on x86 32-bit, x86 64-bit, and ARM architectures. We also assess a popular downstream application, searching for vulnerable assembly functions. We use the CVE dataset released by [26], which is also adopted by asm2vec. These assembly functions are from complex real-world software, including OpenSSL, Wireshark, ffmpeg, and ntpd; see details in Section 8.6.

For all test programs, we generate obfuscated binary code with Obfuscator-LLVM [52], which contains three obfuscation schemes. Instruction substitution (-sub) implements several mapping rules to convert certain instructions into semantics-equivalent instructions. Bogus control flow (-bcf) inserts opaque predicates into randomly selected code blocks. It changes the control flow structures by adding extra nodes and edges [39]. Control-flow flattening (-fla) changes the CFG of a function into a “flatten” structure. The original execution flow is preserved by a deliberately constructed C switch statement [39].

TABLE 5
Seven Compilation Settings Over Our Test Datasets

Compiler	Opt.	Platform	Obfuscation
gcc	-O0	x86 64-bit	NA
gcc	-O3	x86 64-bit	NA
clang	-O3	x86 64-bit	-sub
clang	-O3	x86 64-bit	-bcf
clang	-O3	x86 64-bit	-fla
gcc	-O0	x86 32-bit	NA
gcc	-O0	ARM (aarch64)	NA

These seven settings constitute 12 comparison settings, as shown in Table 7.

Overall, different compilation settings reported in Table 5 result into a total of 12 cross-comparison settings (see Table 7). We report the statistics about the complexity of evaluated assembly functions in Table 6. The results are computed by averaging all assembly functions in coreutils, binutils, and the database containing real-world vulnerable software under all different compilation, optimization, and obfuscation settings. Particularly, for each assembly function, we measure the number of covered paths, all instructions analyzed by BinUSE, as well as encountered external callsites. It is seen that programs in binutils and the real-world vulnerable software manifest comparable complexity, whereas coreutils programs have relatively few and shorter paths. We also report the cyclomatic number [53], a commonly-used metrics to assess complexity of a CFG: cyclomatic number is defined as $e - n + 2$, where e and n are the numbers of edges and basic blocks in the CFG, respectively. In short, we interpret that all assembly functions evaluated in this research have reasonable complexity; according to our observation, these assembly programs often contain nested loops and many function calls (which are inlined by BinUSE during analysis), inducing complex symbolic constraints. Nevertheless, BinUSE can rapidly finish the analysis of each assembly function in several minutes; see Section 8.3.

DNN Models. We use BinUSE to enhance four cutting-edge DNN-based binary code function search tools: BinaryAI [15], asm2vec [14], PalmTree [18], and ncc [13]. BinaryAI, published at AAAI’20, conducts assembly function embedding by computing basic block embeddings with BERT [54] and then conducting graph embedding with gated graph neural network (GGNN) [55]. BinaryAI provides APIs to access its pre-trained model and performs binary function match (see https://github.com/binaryai). We clarify that the training data of BinaryAI is not

TABLE 6
Statistics of Assembly Functions in Different Datasets

	coreutils	binutils	real-world vulnerable software
#instructions	1385.7	7461.0	9556.4
#paths	6.745	37.18	21.98
#external callsites	6.02	17.14	17.58
cyclomatic	19.11	36.91	32.83

The "cyclomatic" denotes the cyclomatic number [53], a common metric (higher is better) to assess the complexity of control flow graphs.

TABLE 7
BinUSE Performance

Setting	Obf.	FN Rate	FP Rate	Failed Rate
gcc -O0 versus gcc -O3	NA	3.3%	24.4%	13.2%
gcc -O0 versus clang -O3	-sub	2.2%	24.6%	11.1%
gcc -O0 versus clang -O3	-bcf	2.6%	24.2%	11.8%
gcc -O0 versus clang -O3	-fla	1.9%	26.3%	12.7%
-m32 -O0 versus gcc -O3	NA	6.0%	23.2%	14.6%
-m32 -O0 versus clang -O3	-sub	5.1%	24.2%	14.4%
-m32 -O0 versus clang -O3	-bcf	5.6%	24.0%	15.4%
-m32 -O0 versus clang -O3	-fla	4.7%	25.5%	15.2%
arm -O0 versus gcc -O3	NA	5.4%	24.9%	13.7%
arm -O0 versus clang -O3	-sub	5.6%	25.9%	13.7%
arm -O0 versus clang -O3	-bcf	4.5%	26.0%	15.3%
arm -O0 versus clang -O3	-fla	4.1%	26.9%	14.0%
Total	NA	4.2%	25.0%	13.8%

disclosed. From its paper, it is stated that BinaryAI is pretrained over millions of binary samples [15] compiled with different compilers, optimizations, and on architectures. We find that BinaryAI manifests sufficiently high accuracy which is comparable to its paper [15]. To our best knowledge, BinaryAI denotes the state-of-the-art binary embedding tool, which has been evaluated [15] to outperform other popular embedding models that can be smoothly extended for binary code, including Structure2-vec [12], Word2vec [56], BERT [54], MPNN [57] and CNN models.

asm2vec, published at IEEE S&P'20, generates assembly function embedding using primarily an extended PV-DM language embedding model [58] and graph neutral networks. We setup its official client which requires IDA-Pro. Unfortunately, asm2vec does not provide a pre-trained model to reproduce its reported results. We therefore follow its paper by uploading executables in our datasets compiled on x86 32-bit, 64-bit and ARM architectures with no optimization to its server to form the function repository and to train a model. During experiments, we iteratively upload executable and for each function inside the upload executable, asm2vec returns the top-15 semantically similar functions in the function repository.

ncc, published at NeurIPS'18, generates code embeddings from LLVM IR code by constructing a contextual flow graph, which subsumes data flow and control flow features. It then uses GNN-based embedding models to extract a numerical representation. We leverage a popular static binary lifter, RetDec [59], to convert binary code into LLVM IR. To prepare inputs for ncc, we tried other binary lifters, including mcsema [60] and mctoll [61]. They manifest much worse LLVM IR lifting results compared with RetDec. ncc is trained using coreutils.

PalmTree [18], published at CCS'21, provides a novel language model for x86 machine instruction embedding. PalmTree features a flexible self-supervised training procedure over unlabeled assembly code, whose generated representation is shown as effective over popular downstream tasks like code similarity analysis, function prototype inference and static analysis. We clarify that PalmTree focuses on computing a novel embedding of only machine instructions. According to its paper, it leverages mean pooling for basic block-level embedding, and Gemini [12] for control graph (function)-level embedding. We follow its paper to equip PalmTree with mean pooling for basic block embedding. As for graph-level embedding, we inquired the authors of Gemini for the graph embedding model and setup details but do not receive response by the time of writing. Therefore, we set the graph embedding model employed by BinaryAI, GGNN, for the graph embedding of PalmTree. We refer this implementation as PalmTree (mean/GGNN) in the evaluation. Furthermore, since PalmTree manifests relatively lower accuracy (see Table 8), we also replace mean pooling with HBMP [62], a popular recurrent neural network (RNN) model, for basic block embedding. Table 8 shows that HBMP can reasonably enhance the accuracy of PalmTree. This improvement of PalmTree is referred to as PalmTree (HBMP/GGNN) in the evaluation.

We use the instruction embedding model of PalmTree pre-trained and released by the authors [63]. It is disclosed by the authors that this model is trained with x64 binaries from coreutils and binutils. This pre-trained model has the embedding vector size as 128. Therefore, we benchmark PalmTree over 64-bit and 32-bit x86 executables and skip analyzing executables on ARM. The function embedding model for PalmTree is trained with binutils, compiled by gcc and clang, with -O0 and -O3. For PalmTree (mean/GGNN), we configure the mean pooling dimension and graph embedding dimension as 128. The message passing step, a key hyper-parameter for GGNN, is 5. For PalmTree (HBMP/GGNN), we set the dimension of HBMP as 128; other settings are all the same with PalmTree (mean/GGNN).

Clarification on Training Dataset Selection. We use normal binary code to train those DNN models, whereas the trained DNN models are assessed on its robustness in cross compilers, cross optimizations, cross architectures, and obfuscations settings. It is clear that DNN models are not exposed to those challenging binary code samples, i.e., not having those samples in the training datasets. On one hand, we clarify that this setup is standard and shared by most existing works in this field [14], [15], [18], [29], [64]. The reason is that we do not assume what obfuscation methods are applied, given that obfuscation methods are diverse and generally unpredictable in real-world settings [39]. On the other hand, benchmarking binary code samples not in the training dataset might potentially raise the concern such that the training and evaluation datasets do not share a "similar distribution." Recent research has illustrated the high potential of using diversely-optimized code samples to augment deep learning models and enhance its robustness of learned embedding representation [65]. We leave it as

TABLE 8
MRR and Top-1 Accuracy Comparison With the State-of-the-Art (DNN-Based) Binary Code Diffing and Search Tools

Comparison Setting	Obfuscation	`BinaryAI`		`ncc`		`asm2vec`		`PalmTree(mean/GGNN)`		`PalmTree(HBMP/GGNN)`	
		MRR	Top-1	MRR	Top-1	MRR	Top-1	MRR	Top-1	MRR	Top-1
`gcc -O0` vs. `gcc -O3`	NA	0.587	0.548	0.275	0.185	0.443	0.383	0.516	0.399	0.618	0.507
`gcc -O0` vs. `clang -O3`	`-sub`	0.633	0.570	0.621	0.516	0.345	0.277	0.569	0.457	0.665	0.548
`gcc -O0` vs. `clang -O3`	`-bcf`	0.588	0.528	0.531	0.394	0.330	0.260	0.386	0.258	0.498	0.364
`gcc -O0` vs. `clang -O3`	`-fla`	0.402	0.345	0.467	0.343	0.256	0.197	0.195	0.108	0.190	0.084
`-m32 -O0` vs. `gcc -O3`	NA	0.579	0.538	0.228	0.135	0.369	0.319	0.271	0.147	0.257	0.154
`-m32 -O0` vs. `clang -O3`	`-sub`	0.622	0.561	0.516	0.379	0.354	0.292	0.310	0.193	0.281	0.163
`-m32 -O0` vs. `clang -O3`	`-bcf`	0.584	0.528	0.442	0.287	0.353	0.289	0.251	0.141	0.247	0.141
`-m32 -O0` vs. `clang -O3`	`-fla`	0.395	0.338	0.403	0.242	0.319	0.255	0.122	0.046	0.130	0.049
`arm -O0` vs. `gcc -O3`	NA	0.570	0.527	0.248	0.158	0.109	0.074	NA	NA	NA	NA
`arm -O0` vs. `clang -O3`	`-sub`	0.622	0.561	0.559	0.420	0.166	0.129	NA	NA	NA	NA
`arm -O0` vs. `clang -O3`	`-bcf`	0.569	0.499	0.526	0.388	0.145	0.107	NA	NA	NA	NA
`arm -O0` vs. `clang -O3`	`-fla`	0.399	0.344	0.476	0.345	0.157	0.127	NA	NA	NA	NA

one future work to exploring augmenting DNN-based models with binary code samples transformed via standard optimization and obfuscation techniques.

8.2 `BinUSE` Performance

Table 7 reports the performance of `BinUSE` regarding in total of 12 comparison settings over the `coreutils` dataset. Most comparisons entail challenging *cross-compiler*, *cross-optimization*, and *cross-architecture* settings. For instance, the last comparison in Table 7 denotes a highly difficult setting which is cross-architecture (ARM versus x86 64-bit), cross-compiler (`gcc` versus `clang`), cross-optimization (`-O0` versus `-O3`), and also applied the control flow flattening obfuscation (-fla) which extensively changes the control flow structures.

Overall, `BinUSE` is designed to be conservative. For instance, we allow pairwise comparison and permutation for function parameters and symbols (Section 5.3), although such permutations incur more constraints to solve and potential FPs. We interpret the overall FN rate (4.2%) is practical and reasonable. Analyzing real-world binary code reveals many engineering issues and corner cases, some of which are difficult, if at all possible, to be addressed without manual effort. Section 9 gives further discussions. `BinUSE` can fail to analyze a number of functions with no external callsites identified. Given each `coreutils` program contains on average 103.7 functions, about 18.7 functions do not have external callsites, which primarily contributes to the FPs. In addition, recall to speed up the analysis, `BinUSE` only analyzes a subgraph of each function, which can mistakenly treat different functions as equivalent and also induce FPs.

In addition, the underlying reverse engineering and symbolic execution tools can throw exceptions and terminate the analysis for 13.8% of test cases (see the last column of Table 7). Overall, the symbolic execution engine, `angr`, can throw errors when inferring control transfer destinations of code pointers. In particular, Table 7 shows that `BinUSE` failed much more functions compiled with control flow flattening obfuscation -fla. As previously mentioned, this obfuscation converts CFG into a C switch statement and stitches basic blocks with a dispatch node. Code pointers are frequently used in the dispatcher node to guide control transfers, inducing higher chances of failures when concretizing symbolic code pointers. Interested readers can refer to Section 9.2 for further information on these reverse engineering tool chain issues. Similarly, IDA-Pro can have incorrect external function recognition results.

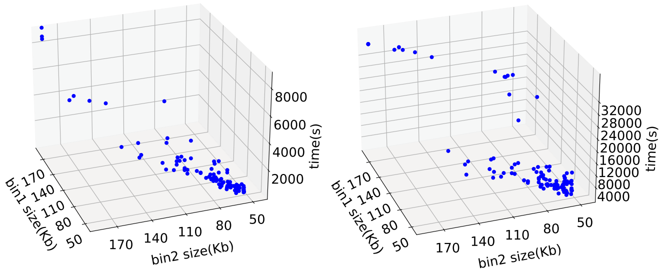
Results Interpretation & Usage Discussion. Overall, to apply the USE scheme in checking the equivalence of assembly functions, we addressed a number of design challenges and corner cases. To take both comprehension and scalability into account, `BinUSE` is designed to only analyze a subgraph of the entire function. We emphasize that `BinUSE` is *not* designed for stand-alone usage, but to compensate mainstream DNN-based code search. Indeed, we report that among all functions that can pass the equivalence check, 87.9% have a confidence score of 1.0. With most passed candidates tied at 1.0, it is unrealistic to compute top-k accuracy. On the other hand, we report how DNN-based binary function search tools are enhanced by `BinUSE` in Section 8.5, Section 8.6, and Section 8.7, respectively.

The preceding evaluation, as well as the evaluations that follow, all assume that RP is formed using highly optimized (or obfuscated) binary code. We clarify that when the reverse configuration is used, i.e., when highly optimized binary code is searched in the RP formed by unoptimized binary code (`-O0`), the performance should be different. Nonetheless, we emphasize that employing highly optimized and obfuscated binary code to generate RP is a common setting that is shared by most prior works and all tools benchmarked in this research. Note that this is a practical configuration. For example, we create the RP using binary samples that were collected in the wild and may be highly optimized and obfuscated. We then query RP and look for vulnerabilities in those binary samples using a known vulnerability pattern produced in a standard and non-optimized form (`-O0`).

8.3 Processing Time

Our `BinUSE` experiments are conducted on a Ubuntu 18.04 machine with Intel Xeon CPU E5-2678 and 256GB RAM. `BinUSE` takes on average 56.6 CPU minutes to process two `coreutils` executables (on average 25.0s to check two functions), including all symbolic execution and constraint solving tasks. Recall we set 10 minutes as the timeout threshold of `BinUSE` when analyzing a function: there are only three timeout cases out of all analyzed functions. We list those three functions in [22].

This illustrates the strength of the USE scheme and our practical USE design for equivalence checking. This also indicates the engineering quality of `angr`. We report that about 23.5% of processing time is spent on symbolic execution. Constraint solving takes the triple amount of time (76.5%). `BinUSE` exhausts every path starting from the



(a) Symbolic Execution (b) Constraint Solving

Fig. 10. Processing time breakdown for compilation setting `gcc -O0` versus `gcc -O3`.

function entry point, inlining every encountered user function calls until reaching the first external callsite. This entails the major consumption of CPU resources. To compare with `BinUSE`, we tested KLEE [33] (ver. 2.1) and an optimized version named MoKLEE [34] with 10 `coreutils` executables and a timeout of 10 hours. `KLEE` takes several hours to process one executable (3 timeouts) while MoKLEE has 8 timeouts and 2 exceptions. See our results at [66], [67].

`BinUSE` also has reasonable memory usage: our `BinUSE` experiments were conducted by launching 30 `angr` processes simultaneously on the server. The total peak memory usage of 30 processes was seen as below 40GB. Overall, `BinUSE` analyzes a subgraph of each individual function and all inlined callees on the subgraph. Therefore, in contrast to standard SE (e.g., [34]), memory usage is not a primary concern for `BinUSE`.

Fig. 10 presents the processing time breakdown for comparison `gcc -O0` versus `gcc -O3`. We report the processing time of performing symbolic execution and solving constraints in Figs. 10a and 10b, respectively. Overall, we find that the processing time scales approximately linear w.r.t. the size of the executable files. This is intuitive: large executable files have more functions, thus prolonging the symbolic execution time of `BinUSE`. Similarly, large executable files may likely contain more complex symbolic constraints, thus prolonging the time taken to solve symbolic constraints. Nevertheless, it is seen that the majority of the binary code samples can be analyzed within 2,000 CPU seconds for symbolic execution, and 4000 CPU seconds for constraint solving. We thus interpret the cost as reasonable.

8.4 DNN Model Comparison

We first run four DNN-based binary function search tools, `BinaryAI`, `asm2vec`, `ncc`, and `PalmTree`, on 12 comparison settings (refer to the first column of Table 8). `PalmTree` cannot process executables on ARM platforms; we thus skip the corresponding evaluation. Table 8 summarizes the performance results. `BinaryAI` is seen to outperform all models across all different settings. While cross-architecture settings impose major challenges, `BinaryAI` seems more robust to cross-architecture changes, given it learns from the platform-neutral microcode lifted by IDA-Pro [15]. Obfuscation, in particular control flow flattening (-fla), primarily and consistently undermines the top-1 accuracy. When lifting binary code into LLVM IR as the inputs of `ncc`, we encountered plenty of reverse engineering errors. The binary lifter, `RetDec`, throws exceptions when processing certain binary code. For such cases, we only measure the top-1 for the successfully processed binary code (about 40% remaining cases for binary code compiled with `clang -O3`). The remaining functions are relatively simpler, which explains the surprisingly higher accuracy for a few comparison settings of `ncc`.

`BinaryAI` is maintained by an industrial giant (Tencent), indicating more resource devoted to training the model and better engineering quality. We cannot recover the high accuracy reported in the `asm2vec` paper: we emphasize that both software engineering and security communities have pointed out similar issues [16], [68], [69]. Our evaluation shows that `asm2vec` has 38.3% top-1 accuracy, which, although lower than the accuracy reported in its paper, is *highly consistent* with findings of recent research [16], [68], [69]. Nevertheless, `asm2vec` still entails one cutting-edge DNN framework in this field (outperforming traditional CFG-based matching [14]). `PalmTree` (mean/GGNN) manifests reasonable accuracy. It notably outperforms `asm2vec` for the `-sub` obfuscation setting over 64-bit x86 binary code. Nevertheless, `PalmTree` (mean/GGNN) becomes less accurate in terms of 32-bit x86 executables. The reason could be that the pre-trained model provided by `PalmTree` primarily uses 64-bit x86 executables as the training data. We clarify that `PalmTree` does not provide full details of model re-training, and therefore, it is unclear for us to somehow re-train its released model using 32-bit x86 executables. `PalmTree` (HBMP/GGNN) shows promising improvement over `PalmTree` (mean/GGNN) on 64-bit executables; its accuracy is close to `BinaryAI`. Nevertheless, `BinUSE` still offers a high enhancement for `PalmTree` (mean/GGNN), as will be reported in Section 8.5.

8.5 DNN Model Enhancement

To measure the enhancement of DNN-based approaches using `BinUSE`, we try to answer two questions: 1) *RQ1*: can `BinUSE` enhance different DNN-based binary code function search tools? and 2) *RQ2*: can `BinUSE` enhance `BinaryAI` of different settings? For *RQ2*, we take `BinaryAI` as the target since it notably outperforms the other three models. Also, in addition to the enhancement of DNN-based methods, we also explore *RQ3*: is `BinUSE` general enough to enhance conventional binary diffing tools based on program structure-level information? For *RQ3*, we benchmark a popular binary diffing tool, FuncSimSearch [70], developed and maintained by Google Project Zero.

RQ1. Table 9 presents the evaluation results in terms of different settings. Consistently, we measure top-1, top-3, and top-5 enhancement. Table 9 shows that all DNN-based approaches can be noticeably improved using `BinUSE`. This is intuitive and consistent with our motivation: DNN models are generally learning from coarse-grained code features which are not resilient toward various challenging settings, thus making very high false alarms. `BinUSE` is designed to address their key limitation in a consistent manner. `BinUSE` is seen to identify more opportunities to improve other DNN-based tools rather than `BinaryAI`. As previously mentioned, `BinaryAI` has higher accuracy than others: `BinaryAI` should have fewer false predictions to be regulated. It is also interesting to compare `BinaryAI` and `PalmTree` (HBMP/GGNN): while these two models show close accuracy (the latter model is slightly worse) in Table 8,

TABLE 9
Boosting DNN-Based Tools Over coreutils Programs With BinUSE

Comparison	Obf.	BinaryAI(%)		ncc (%)		asm2vec (%)		PalmTree (mean/GGNN) (%)		PalmTree (HBMP/GGNN) (%)	
		MRR	Top-1/Top-3/Top-5	MRR	Top-1/Top-3/Top-5	MRR	Top-1/Top-3/Top-5	MRR	Top-1/Top-3/Top-5	MRR	Top-1/Top-3/Top-5
gcc -O0 versus gcc -O3	NA	+16.9	+11.0/+22.3/+24.2	+34.4	+30.1/+42.3/+42.2	+20.7	+16.9/+23.0/+27.5	+27.4	+30.7/+30.4/+24.1	+21.6	+24.8/+22.5/+17.9
gcc -O0 versus clang -O3	-sub	+15.6	+12.1/+18.4/+19.1	+24.9	+30.5/+22.9/+17.2	+26.5	+21.5/+29.4/+34.7	+25.1	+30.3/+24.1/+20.0	+18.2	+23.3/+15.9/+12.5
gcc -O0 versus clang -O3	-bcf	+17.3	+14.2/+21.9/+20.5	+32.0	+40.7/+27.2/+21.3	+27.9	+24.2/+30.6/+33.9	+35.5	+40.6/+36.7/+30.9	+28.0	+34.3/+26.6/+21.6
gcc -O0 versus clang -O3	-fla	+25.0	+20.0/+29.6/+31.8	+30.1	+35.9/+30.3/+23.0	+34.0	+29.4/+37.8/+41.8	+47.2	+48.4/+52.6/+49.8	+49.6	+53.1/+54.0/+49.4
-m32 -O0 versus gcc -O3	NA	+14.4	+9.7/+18.8/+19.3	+34.6	+30.9/+42.6/+42.4	+20.9	+17.5/+22.8/+25.8	+40.8	+45.1/+43.2/+37.7	+43.4	+47.3/+45.7/+41.6
-m32 -O0 versus clang -O3	-sub	+11.4	+8.1/+13.8/+16.2	+28.9	+37.0/+25.7/+16.3	+22.8	+19.7/+26.0/+26.6	+38.8	+42.9/+40.5/+35.9	+42.2	+45.9/+44.9/+40.4
-m32 -O0 versus clang -O3	-bcf	+12.5	+9.4/+16.8/+15.6	+32.3	+40.9/+28.8/+20.5	+22.5	+20.6/+24.5/+25.8	+41.0	+42.9/+45.3/+41.8	+41.3	+42.9/+46.0/+42.5
-m32 -O0 versus clang -O3	-fla	+20.9	+16.3/+25.6/+27.7	+33.1	+42.3/+28.8/+21.0	+24.6	+22.1/+26.5/+29.0	+45.8	+44.9/+53.2/+51.3	+47.4	+47.1/+53.8/+52.3
arm -O0 versus gcc -O3	NA	+17.3	+12.3/+21.9/+23.7	+35.8	+32.4/+42.6/+42.6	+34.1	+29.0/+38.2/+42.1	NA	NA/NA/NA	NA	NA/NA/NA
arm -O0 versus clang -O3	-sub	+15.0	+11.7/+17.6/+18.8	+26.6	+34.9/+22.1/+14.8	+31.4	+27.2/+34.8/+38.3	NA	NA/NA/NA	NA	NA/NA/NA
arm -O0 versus clang -O3	-bcf	+17.4	+15.0/+20.9/+19.9	+28.3	+36.6/+23.6/+17.3	+32.8	+28.5/+36.4/+39.6	NA	NA/NA/NA	NA	NA/NA/NA
arm -O0 versus clang -O3	-fla	+22.6	+18.2/+26.4/+29.6	+30.2	+36.9/+28.8/+22.5	+33.7	+28.3/+38.1/+42.3	NA	NA/NA/NA	NA	NA/NA/NA

BinUSE delivers a much higher enhancement toward PalmTree (HBMP/GGNN). The main reason is that PalmTree (HBMP/GGNN) is seen to behave strangely over small assembly functions. Therefore, when assessing PalmTree (HBMP/GGNN), we only use functions with over three basic blocks, resulting in fewer analyzed functions. As a result, the enhancement ratio thus becomes higher, as the divider (i.e., the total number of analyzed functions; the "N" in Formula 1) of this ratio is small.

This evaluation subsumes four recent DNN-based binary code search tools which have been shown to suffer from similar issues and can be consistently enhanced. We envision the opportunities to primarily enhance the performance of other research sharing the similar methodology and potential limits. We leave it as one future work to boost other DNN models with BinUSE.

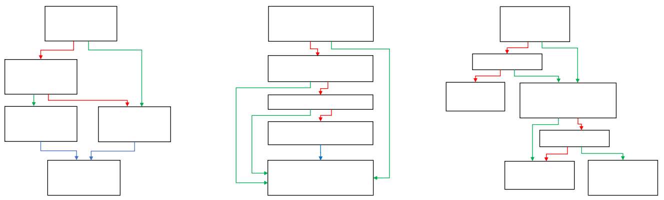
Case Study. Fig. 11 presents a case study by comparing coreutils program shuf compiled with gcc -O0 versus compiled with clang -O3. Recall ncc, as reported in Section 3, makes similar errors. We consistently pick this case given its smaller size to ease our presentation. Fig. 11a presents the CFG of function rpl_fflush when compiled with gcc -O0. Fig. 11b presents the CFG of rpl_fflush when clang -O3 is used.

When full optimization is enabled, the relatively "flattened" CFG of rpl_fflush in Fig. 11a is converted into a visually linear representation. BinaryAI instead treats another function, xrealloc (Fig. 11c), as the top-1 match with rpl_fflush in Fig. 11a. We interepret the results as reasonable; xrealloc is seen to share more structure-level similarity with rpl_fflush, and thus misleading BinaryAI. In contrast, BinUSE constructs symbolic formulas representing the inputs of two external callsites in rpl_fflush, and use these formulas to determine the semantics similarity between rpl_fflush in Fig. 11a and rpl_fflush in Fig. 11b.

RQ2. To answer *RQ2*, we studied three key hyper-parameters of representation learning relevant to the embedding vector dimensions. In general, different dimensions primarily influence model accuracy: longer embedding could convey subtle information on input data, while smaller ones might not represent the semantics well enough. Nevertheless, longer vectors indicate more challenges for model training, and might potentially undermine the model robustness. BinaryAI subsumes three embedding dimension-related hyper-parameters, which are token embedding dimension, constant embedding dimension, and graph embedding dimension. Recall BinaryAI first performs basic block-level embedding and then graph-level embedding: token and constant embedding dimensions are hyper-parameters used in the first step, whereas the dimension of graph embedding primarily calibrates the quality of graph embedding. Table 10 reports that for all hyper-parameters, despite different embedding dimensions, BinUSE consistently enhances the accuracy. Overall, this evaluation reveals an intuitive observation: equivalence check consistently resolves high false alarms despite changes in the model settings, indicating the generalizability of BinUSE from another important aspect.

RQ3. This research primarily focuses on DNN-based binary code function search, given that DNN-based approaches manifest highly promising accuracy and have largely outperformed conventional program structure-based algorithms like graph isomorphism [71]. Nevertheless, it is easy to see that BinUSE is not limited to enhancing only DNN-based approaches. In principle, we argue that binary diffing based on program structures generally suffers from making low discriminable and low robust predications. *RQ3*empirically validates our argument, by using BinUSE to boost one state-of-the-art binary diffing tool, FuncSimSearch, which computes the Simhash score over control flow graphs to efficiently decide the distance of assembly functions.

Table 11 reports the evaluation results. We find that FuncSimSearch shows much worse results compared with contemporary DNN-based methods. Therefore, BinUSE can largely enhance its accuracy for all the assessed settings. The relatively low accuracy of FuncSimSearch is also pointed out in the asm2vec paper. This is reasonable, given modern DNN models have shown a highly encouraging capability of com-



(a) rpl_fflush compiled by gcc -O0

(b) rpl_fflush compiled by clang -O3

(c) xrealloc compiled by clang -O3

Fig. 11. Case study using coreutils program shuf.

TABLE 10
Boosting BinaryAI With Different Dimensions

graph embedding		16	32	64	128	256
	Top-1	+15.5%	+16.0%	+13.2%	+7.5%	+10.7%
	Top-3	+13.1%	+16.1%	+8.3%	+9.5%	+9.4%
	Top-5	+15.9%	+13.9%	+13.0%	+14.5%	+12.9%
	MRR	+14.0%	+14.5%	+11.4%	+8.5%	+10.0%
token embedding		16	32	64	128	256
	Top-1	+16.5%	+10.6%	+14.3%	+7.5%	+11.1%
	Top-3	+19.1%	+9.8%	+16.8%	+9.5%	+10.0%
	Top-5	+15.3%	+13.5%	+13.6%	+14.5%	+12.2%
	MRR	+15.6%	+10.8%	+13.6%	+8.5%	+10.2%
constant embedding		16	32	64	128	256
	Top-1	+14.4%	+7.5%	+14.8%	+18.5%	+15.2%
	Top-3	+14.2%	+9.5%	+17.2%	+18.4%	+14.0%
	Top-5	+12.8%	+14.5%	+14.4%	+15.1%	+13.6%
	MRR	+12.9%	+8.5%	+14.3%	+16.6%	+13.7%

prehending complex (fuzzy) structures. In addition, DNN models can learn from comprehensive sets of historical data. On the other hand, BinUSE exhibits highly promising results in boosting FuncSimSearch. In sum, our exploration toward *RQ3* shows that code structure-based binary diffing tools share common limits of making low discriminable and low robust predictions, particularly against changes introduced by optimizations, obfuscations and platforms. We show that equivalence check enabled by BinUSE can reduce high false alarms of structure-based approaches in a general manner.

Overall, evaluation in this section consistently demonstrates that high false alarms made by popular structure-based (DNN) tools can be enhanced by BinUSE in a general, efficient, and unique manner. We thus advocate combining binary code search with BinUSE and achieve a synergistic effect in production usage.

8.6 Vulnerable Function Searching

We launch a case study by applying BinUSE to augment a vulnerability search task toward a public vulnerability dataset. This application mimics a common security usage scenario: given an assembly function f from a suspicious piece of executable, we search against a database D of functions with known vulnerabilities and decide if f can be matched with any function in D.

As with asm2vec, we use the dataset D released by [26]. This database contains program samples of eight CVE vulnerabilities. We evaluate seven CVEs, because the other CVE, venom, requires rebuilding qemu (ver. 2.4.0), which cannot be processed by Obfuscator-LLVM. D contains 12 functions of seven CVEs, including the infamous Heartbleed exploiting OpenSSL crypto library, and Shellshock allowing remote attackers to execute arbitrary commands on the victim machine. To enhance the difficulty, D also contains 1,225 “negative samples”, denoting functions with no vulnerability. A vulnerability search engine must match vulnerable inputs with correct vulnerability samples in D at top-1, without interference from the remainder (benign) functions.

TABLE 11
Enhancing FuncSimSearch

Config.	gcc -O0 versus *	gcc -O0 -m32 versus *	arm -O0 versus *
Top-1	+49.8%	+48.2%	+49.0%
Top-3	+49.8%	+50.7%	+52.6%
Top-5	+47.5%	+48.6%	+50.7%
MRR	+48.5%	+47.4%	+49.2%

At this step, we compile samples in D into a database D_{asm} of assembly functions using four obfuscation settings, -sub, -bcf, -fla, and -hybrid, respectively. Note that the hybrid setting, referred to as -hybrid, combines all three obfuscation methods together during compilation. We also enable full optimization -O3 when compiling each sample program and the target function. In short, given a heavily-optimized (-O3) assembly function with known vulnerabilities, we retrieve its matched functions from D_{asm} and check if its correct match, functions with the same vulnerabilities, exist in top-1 ranked candidates.

At this step, we measure asm2vec, BinaryAI, and two versions of PalmTree. We omit to evaluate ncc as we find its employed binary lifter failed too many cases when processing these real-world complex software. We report the evaluation results for each setting in Tables 12, 13, 14, and 15. asm2vec seems to struggle with OpenSSL and Wireshark, given that both programs are highly complex. For three versions of OpenSSL and Wireshark, asm2vec rank the true matches much lower. For instance, asm2vec ranks the true match of Heartbleed vulnerability in OpenSSL (ver. 1.0.1f) at top-17, implying that users may need to manually compare at least 17 copies of programs in D_{asm} to confirm that a Heartbleed vulnerability exists in the suspicious input. In contrast, BinUSE can successfully match the suspicious input with the Heartbleed vulnerability in D_{asm} at top-1. When analyzing another infamous CVE, ws-snmp, asm2vec also achieves a much lower accuracy. We find that this vulnerability contains a large CFG, which presumably hinders asm2vec's graph-level embedding computation that is based on random walk. Given that said, with the help of BinUSE, asm2vec can place the true vulnerable function at top-1.

Similar observations are obtained from evaluations of BinaryAI and PalmTree. BinaryAI generally shows promising accuracy compared with asm2vec. Our manual

TABLE 12
Augmenting Vulnerability Search of `asm2vec` Using `BinUSE`

Vulnerability	CVE	Software/Version	-sub	-bcf	-fla	-hybrid
Shellshock #1	2014-6271	Bash 4.3	1 → 1	1 → 1	1 → 1	1 → 1
		Bash 4.3.30	1 → 1	1 → 1	1 → 1	1 → 1
Shellshock #2	2014-6271	Bash 4.3	1 → 1	1 → 1	1 → 1	1 → 1
		Bash 4.3.30	1 → 1	1 → 1	1 → 1	1 → 1
ffmpeg	2015-6826	ffmpeg 2.6.4	1 → 1	1 → 1	1 → 1	1 → 1
Clobberin' Time	2014-9295	ntpd 4.2.7	1 → 1	1 → 1	1 → 1	1 → 1
		ntpd 4.2.8	1 → 1	1 → 1	1 → 1	1 → 1
Heartbleed	2014-0160	OpenSSL 1.0.1e	1 → 1	1 → 1	1 → 1	21 → 1
		OpenSSL 1.0.1f	1 → 1	1 → 1	2 → 1	17 → 1
		OpenSSL 1.0.1g	1 → 1	1 → 1	1 → 1	27 → 1
wget	2014-4877	wget 1.8	1 → 1	1 → 1	1 → 1	1 → 1
ws-snmp	2011-0444	Wireshark 1.12.8	1 → 1	8 → 1	> 50 → 1	> 50 → 1

The numbers before and after the arrow represent the top-k ranking. For instance, "1" means an input software with a vulnerability can be matched with correct vulnerability samples in the databse [26] at top-1. Improvement enabled by `BinUSE` *is shown as "→".*

investigation shows that `BinaryAI` extracts representative constant strings from these binary executables; relying on these constant strings enables accurate matching between vulnerable functions. While `BinaryAI` can still make a number of mistakes, `BinUSE` can effectively improve its accuracy by correctly matching all vulnerable functions at top-1. Compared with `BinaryAI` and `asm2vec`, `PalmTree` shows worse accuracy in this evaluation. We evaluate two configurations of `PalmTree` in Table 14 and Table 15, respectively. While our enhancement, by replacing its default mean pooling with HBMP, reasonably increases the accuracy in this task, `PalmTree` still makes a considerable number of inaccurate matching, particularly for obfuscated binary code. For instance, when enabling control flow flattening, denoted as `-fla` and `-hybrid` in Table 14 and Table 15, the true matching is lower than top-1000. Users can hardly identify the CVE vulnerability from the suspicious input, given that their true matches in D_{asm} is ranked in such low positions. We also wish to clarify that there are a few cases in Table 15 which are not in top-1 after enhancement of `BinUSE`(e.g., the wget case under the `-hybrid` compilation setting). We clarify that all of these cases that are not in the top-1 are because multiple cases, including the true positive case itself, are in a tie at top-1. While these increase the difficulty for users to confirm the vulnerability, `BinUSE` still largely reduces the effort. For instance, according to Table 15, while users may need to check 944 cases to confirm that the suspicious input really contains CVE 2014-4877 (if it is at all possible), users only need to check 14 cases after using `BinUSE`.

Discussion of `BinUSE` *'s Enhancement.* Despite the errors made by existing DNN-based tools for this task, `BinUSE` can successfully identify the true matches with the highest confidence scores respectively. In addition to precisely capturing the semantics-level constraints, our manual investigation shows that `BinUSE`, by matching the names of external callsites, indeed exposes a "shortcut" of searching for vulnerable functions. In particular, we find that in these real-world programs (e.g., OpenSSL), each assembly function, including functions with CVE vulnerabilities, usually has a distinct callsite pattern. In other words, employing external callsite names has already helped to match quite a number of vulnerable

TABLE 13
Augmenting Vulnerability Search of `BinaryAI` Using `BinUSE`

Vulnerability	CVE	Software/Version	-sub	-bcf	-fla	-hybrid
Shellshock #1	2014-6271	Bash 4.3	1 → 1	1 → 1	1 → 1	1 → 1
		Bash 4.3.30	1 → 1	1 → 1	1 → 1	1 → 1
Shellshock #2	2014-6271	Bash 4.3	1 → 1	1 → 1	3 → 1	4 → 1
		Bash 4.3.30	1 → 1	1 → 1	4 → 1	1 → 1
ffmpeg	2015-6826	ffmpeg 2.6.4	1 → 1	5 → 1	1 → 1	1 → 1
Clobberin' Time	2014-9295	ntpd 4.2.7	1 → 1	1 → 1	1 → 1	1 → 1
		ntpd 4.2.8	1 → 1	1 → 1	1 → 1	1 → 1
Heartbleed	2014-0160	OpenSSL 1.0.1e	1 → 1	1 → 1	1 → 1	4 → 1
		OpenSSL 1.0.1f	1 → 1	1 → 1	1 → 1	1 → 1
		OpenSSL 1.0.1g	1 → 1	1 → 1	1 → 1	3 → 1
wget	2014-4877	wget 1.8	1 → 1	1 → 1	1 → 1	3 → 1
ws-snmp	2011-0444	Wireshark 1.12.8	1 → 1	1 → 1	1 → 1	1 → 1

The numbers before and after the arrow represent the top-k ranking. For instance, "1" means an input software with a vulnerability can be matched with correct vulnerability samples in the databse [26] at top-1. Improvement enabled by `BinUSE` *is shown as "→".*

TABLE 14
Augmenting Vulnerability Search of `PalmTree` (Mean/GGNN) Using `BinUSE`

Vulnerability	CVE	Software/Version	`-sub`	`-bcf`	`-fla`	`-hybrid`
Shellshock #1	2014-6271	Bash 4.3	1 → 1	41 → 1	2283 → 1	2392 → 1
		Bash 4.3.30	1 → 1	1 → 1	731 → 1	1564 → 1
Shellshock #2	2014-6271	Bash 4.3	1 → 1	2 → 1	2145 → 1	2329 → 1
		Bash 4.3.30	1 → 1	1 → 1	2193 → 1	2544 → 1
ffmpeg	2015-6826	ffmpeg 2.6.4	1 → 1	235 → 2	32 → 2	34 → 2
Clobberin' Time	2014-9295	ntpd 4.2.7	1 → 1	2 → 1	582 → 1	1538 → 1
		ntpd 4.2.8	1 → 1	4 → 1	428 → 1	396 → 1
Heartbleed	2014-0160	OpenSSL 1.0.1e	1 → 1	578 → 1	724 → 1	808 → 1
		OpenSSL 1.0.1f	1 → 1	421 → 1	2141 → 1	2095 → 1
		OpenSSL 1.0.1g	1 → 1	81 → 1	2183 → 1	2297 → 1
wget	2014-4877	wget 1.8	1 → 1	738 → 11	2593 → 15	1946 → 9
ws-snmp	2011-0444	Wireshark 1.12.8	1 → 1	4 → 1	143 → 1	419 → 6

The numbers before and after the arrow represent the top-k ranking. For instance, "1" means an input software with a vulnerability can be matched with correct vulnerability samples in the databse [26] at top-1. Improvement enabled by `BinUSE` is shown as "→".

real-world functions in a very effective manner. In contrast, many programs in `coreutils` and `binutils` contain functions with closely related semantics. For instance, `quote_mem`, `quote_n_mem`, `quoteargs_n_mem` functions in typical `coreutils` programs share nearly identical callsite patterns. This imposes an extra challenge for function matching of `BinUSE`. In contrast, vulnerable function search evaluation in this section reveals that `BinUSE` can exhibit even better performance when processing real-world software.

In short, evaluation in this section reveals highly encouraging results when using `BinUSE` in analyzing real-world applications for security purposes. We also interpret that evaluation in this section justifies the necessity of considering fine-grained callsite information when performing function matching.

8.7 Extension of `BinUSE`

As defined in Section 2.1, our proposed method compares a target function f_t to every function in a repository RP of assembly functions. While the customized USE adopted by `BinUSE` is shown as efficient (Section 8.3), it is still costly in general. Hence, comparing f_t with every function $f \in RP$ is expensive. In this section, we study a possible extension of `BinUSE`; we aim to reduce the cost, by comparing f_t with top-k functions $RP_k \subset RP$ first returned by DNN-based binary matching tools.

Setup. At this step, we benchmark a large-scale and challenging dataset, Linux `binutils`. Each program in `binutils` contains about 1,765.0 functions. That is, while analyzing each program in `coreutils` forms RP with about one hundred functions, analyzing each `binutils` faces an RP with about 15 × more functions. Table 16 reports the top-1 accuracy over `binutils` test cases using two DNN-based tools. At this step, `asm2vec` is not included, given that its local client crashed when processing `binutils` test cases [72]. As for `ncc`, its employed binary lifter fails or generates broken LLVM IR code over `binutils` programs; we thus skip evaluating `ncc`.

In general, Table 16 illustrates that both DNN models, particularly `PalmTree`, can be improved over `binutils` test cases. However, given that `BinUSE` by default needs to

TABLE 15
Augmenting Vulnerability Search of `PalmTree` (HBMP/GGNN) Using `BinUSE`

Vulnerability	CVE	Software/Version	`-sub`	`-bcf`	`-fla`	`-hybrid`
Shellshock #1	2014-6271	Bash 4.3	1 → 1	1 → 1	198 → 1	110 → 1
		Bash 4.3.30	1 → 1	2 → 1	92 → 1	77 → 1
Shellshock #2	2014-6271	Bash 4.3	1 → 1	1 → 1	149 → 1	287 → 1
		Bash 4.3.30	1 → 1	17 → 1	132 → 1	238 → 1
ffmpeg	2015-6826	ffmpeg 2.6.4	1 → 1	387 → 2	1232 → 2	1345 → 2
Clobberin' Time	2014-9295	ntpd 4.2.7	1 → 1	1 → 1	80 → 1	95 → 1
		ntpd 4.2.8	1 → 1	1 → 1	54 → 1	132 → 1
Heartbleed	2014-0160	OpenSSL 1.0.1e	1 → 1	549 → 1	1430 → 1	1536 → 1
		OpenSSL 1.0.1f	1 → 1	103 → 1	1413 → 1	1454 → 1
		OpenSSL 1.0.1g	1 → 1	65 → 1	1425 → 1	2743 → 1
wget	2014-4877	wget 1.8	1 → 1	67 → 1	944 → 14	801 → 9
ws-snmp	2011-0444	Wireshark 1.12.8	1 → 1	33 → 1	425 → 4	533 → 5

The numbers before and after the arrow represent the top-k ranking. For instance, "1" means an input software with a vulnerability can be matched with correct vulnerability samples in the databse [26] at top-1. Improvement enabled by `BinUSE` is shown as "→". We have manually confirmed that after enhancement of `BinUSE`, all cases that are not *in the top-1 (e.g., wget under `-hybrid`) are because multiple cases, including the true positive case itself, are in a tie at top-1.*

TABLE 16
MRR and Top-1 Accuracy Comparison With the State-of-the-Art (DNN-Based) Binary Code Diffing and Search Tools Over binutils Programs

Comparison Setting	Obfuscation	BinaryAI		PalmTree (mean/GGNN)		PalmTree (HBMP/GGNN)	
		MRR	Top-1	MRR	Top-1	MRR	Top-1
gcc -O0 vs. gcc -O3	NA	0.694	0.632	0.497	0.392	0.560	0.454
gcc -O0 vs. clang -O3	-sub	0.728	0.657	0.425	0.310	0.509	0.394
gcc -O0 vs. clang -O3	-bcf	0.570	0.479	0.142	0.089	0.198	0.125
gcc -O0 vs. clang -O3	-fla	0.565	0.465	0.017	0.009	0.021	0.008
gcc -m32 -O0 vs. gcc -O3	NA	0.652	0.579	0.059	0.031	0.081	0.046
gcc -m32 -O0 vs. clang -O3	-sub	0.691	0.616	0.052	0.026	0.071	0.035
gcc -m32 -O0 vs. clang -O3	-bcf	0.543	0.451	0.026	0.013	0.045	0.019
gcc -m32 -O0 vs. clang -O3	-fla	0.529	0.421	0.004	0.001	0.010	0.005
arm -O0 vs. gcc -O3	NA	0.673	0.618	NA	NA	NA	NA
arm -O0 vs. clang -O3	-sub	0.717	0.654	NA	NA	NA	NA
arm -O0 vs. clang -O3	-bcf	0.560	0.475	NA	NA	NA	NA
arm -O0 vs. clang -O3	-fla	0.569	0.472	NA	NA	NA	NA

compare the target function f_t with every function in RP, we see it as necessary to optimize the usage of BinUSE, by confining BinUSE's analysis toward the top-K functions ranked by DNN models. Note that this extension requires to change our re-order algorithm defined in Section 6 only slightly: in our current experiments, we pick K as 100. Once DNN models have decided the top-100 matched functions with the target function f_t, BinUSE is used to compare these 100 ranked functions with f_t and adjusts their ranking, using our re-order strategy presented in Section 6. This way, BinUSE's comparison is reduced from the size of RP ($|RP|$) to only 100, reducing its cost when analyzing binutils programs. Nevertheless, since BinUSE only accesses and re-orders the top-100 assembly functions ranked by the DNN model, the enhanced top-k accuracy (where $k \leq 100$) is bounded by the DNN model's top-100 accuracy. In other words, if the target DNN model is of low accuracy even for top-100, the chance of enhancing it is slim.

Results. We report enhancement over BinaryAI and PalmTree in Table 17. Note that in this table, we evaluate 12 comparison settings which are consistent with our previous evaluation setups. However, we clarify that, as disclosed by the BinaryAI authors, binutils programs are in the training dataset of BinaryAI following three comparison settings: gcc -O0 versus gcc -O3, gcc -m32 -O0 versus gcc -O3, and arm -O0 versus gcc -O3. It generally explains BinUSE's relatively low enhancement over these three settings: for instance, for the gcc -m32 -O0 versus gcc -O3 comparison setting, BinUSE leads to even negative enhancement. We manually looked at these cases, and confirmed that they are due to false positives of BinUSE. Similarly, BinUSE delivers low enhancement toward BinaryAI under three comparison settings using the -sub obfuscation. Our manual study shows that -sub obfuscation does not primarily influence the control flow structures and imposes less challenge to BinaryAI. As a result, these three -sub comparison settings are mostly similar to their unobfuscated comparison settings which are in the training dataset of BinaryAI, thus becoming hard to be further enhanced by BinUSE. Nevertheless, for the other two obfuscation settings (-bcf and -fla), BinUSE delivers general higher enhancement, despite the fact that these two obfusction methods heavily changed the control flow structures.

BinUSE achieves higher enhancement toward PalmTree comparing to that of BinaryAI. This is primarily due to the relatively lower accuracy of PalmTree on binutils test cases, leaving more chances for enhancement. On the other hand, compared with evaluations on the coreutils dataset, BinUSE achieves lower enhancement. In addition to the general difficulty of analyzing binutils functions, we clarify that for this evaluation, BinUSE only analyzes the top-100 functions returned by DNN-based tools. According to our observation, some true matchings are not even within the top-100 functions. To further explore a higher degree of accuracy enhancement, users may consider leveraging top-150 or top-200 functions returned by the DNN-based tools.

Other Possible Extensions. Along with the extension proposed and evaluated in this section, we envision other ways to extend the usage of BinUSE. In short, recent advances in explainable Artificial Intelligence (XAI) techniques [73] (e.g., through the use of attention mechanisms [74]) have enabled the identification of the most influential code components in terms of DNN models' decision making. As a result, we anticipate to extend BinUSE and deliver a "post-verification" pipeline in which we first use XAI techniques to flag influential code fragments c_1 and c_2, which are principally responsible for the DNN models' decision to match assembly functions f_1 and f_2. We can then launch BinUSE toward those flagged c_1 and c_2 to check their semantics equivalence. Note that such critical code fragments c_1 and c_2 should usually be much smaller than the whole assembly functions f_1 and f_2, substantially lowering the cost of BinUSE. We leave this for future exploration; the primary difficulty would be properly delineating the code boundary of c_1 and c_2, as slight drifting in the formed symbolic constraints could flip its decision from sat to unsat, or vice versa.

9 Discussion

Soundness and Completeness. BinUSE's strength is in its ability to perform practical binary function-level equivalence checking with a low rate of false alarms and high speed. However, Our evaluation found that when BinUSE is used to analyze varied sets of real-world binary code, an average FP rate of 25.0% and an average FN rate of 4.2% may occur (see Table 7).

Besides over-approximating legitimate input space, our USE implementation, BinUSE, is not sound due to some engineering challenges (e.g., cross-architecture comparison). Also, the underlying SE engine, angr [51], has a lightweight but unsound memory model. In addition, a number of functions do not contain external callsites and are therefore not analyzable by BinUSE. They also contribute to the errors of BinUSE.

TABLE 17
Boosting DNN-Based Tools With `BinUSE` Over `binutils` Programs

Comparison Setting	Obfuscation	`BinaryAI(%)`				`PalmTree (HBMP/GGNN)(%)`			
		MRR	Top-1	Top-3	Top-5	MRR	Top-1	Top-3	Top-5
`gcc -O0` versus `gcc -O3`	NA	+1.0	+0.5	+1.7	+1.7	+4.6	+5.5	+4.3	+3.4
`gcc -O0` versus `clang -O3`	`-sub`	+1.0	+0.8	+1.5	+1.4	+5.4	+6.7	+5.3	+4.0
`gcc -O0` versus `clang -O3`	`-bcf`	+4.2	+4.9	+4.0	+3.7	+8.2	+9.2	+8.3	+7.7
`gcc -O0` versus `clang -O3`	`-fla`	+5.8	+6.6	+6.1	+5.1	+3.0	+3.0	+3.4	+3.2
`gcc -m32 -O0` versus `gcc -O3`	NA	-1.7	-2.2	-1.1	-0.9	+5.1	+5.0	+5.8	+5.5
`gcc -m32 -O0` versus `clang -O3`	`-sub`	-1.9	-2.2	-1.6	-1.5	+5.7	+5.7	+6.5	+5.8
`gcc -m32 -O0` versus `clang -O3`	`-bcf`	+1.0	+1.5	+0.9	+0.4	+4.0	+4.0	+4.6	+4.5
`gcc -m32 -O0` versus `clang -O3`	`-fla`	+2.1	+2.7	+2.3	+1.3	+1.7	+1.6	+2.1	+2.0
`arm -O0` versus `gcc -O3`	NA	+0.8	+0.6	+1.2	+1.2	NA	NA	NA	NA
`arm -O0` versus `clang -O3`	`-sub`	+0.4	+0.4	+0.7	+0.6	NA	NA	NA	NA
`arm -O0` versus `clang -O3`	`-bcf`	+3.9	+4.8	+3.8	+2.9	NA	NA	NA	NA
`arm -O0` versus `clang -O3`	`-fla`	+5.9	+7.2	+5.9	+4.6	NA	NA	NA	NA

For this enhancement evaluation, `PalmTree` *(HBMP/GGNN) is used because it has much better accuracy comparing with that of* `PalmTree` *(mean/GGNN), as shown in Table 16.*

However, unlike prior SE-based binary clone detection techniques that analyze either basic blocks or execution traces [4], [20], `BinUSE` is scalable and efficient to analyze whole `coreutils` programs. To provide a practical USE design for function equivalence checking, the design of `BinUSE` trades completeness for speed. Note that the function search problem is difficult in that the true positive rate in the population of all the functions is quite low ($\frac{1}{103.7} \approx 1.0\%$ given that each `coreutils` executable in our dataset contains about 103.7 functions). Along with FPs and FNs, Table 7 also reports that analyzing 13.8% `coreutils` functions are terminated due to the underlying symbolic execution engine or the reverse engineering platform exceptions.

In the rest of this section, we discuss all issues that can induce false alarms of `BinUSE` in Sections 9.1 and 9.2. Section 9.3 further lists issues that can lead to the failure of `BinUSE`. We clarify that many issues are also pointed out by previous SE tools [75]. `BinUSE` will become sound after fixing these (implementation-level) issues, which is highly challenging and beyond the consideration of this work and many other "sound" SE engines (e.g., due to unsound memory model; see discussions Section 9.2).

9.1 External Callsites Inconsistency

To compare a target assembly function f_t with another function f_s, `BinUSE` is designed to extract semantics signatures from external function call inputs for equivalence checking. In other words, we assume that compiling a C function source code should not generate two assembly functions with inconsistent external callsites. While these assumptions generally hold for even challenging settings (e.g., cross-architecture or obfuscation) evaluated in this research, we listed all corner cases violating our assumptions in this section.

External Callsite Renaming. To construct constraints and check the semantics equivalence of two callsites, we first decide if they are referring to the same external function. Apparently, an external callsite of `fopen` should not be matched to a callsite of `fwrite`.

However, compiler optimization could rename certain C library functions. For instance, we find that when compiling `coreutils` programs with optimizations enabled, function calls to C library function `dcgettext` could be optimized into `gettext` but does not change the semantics. For the current implementation, we manually map a library function to functions that could be a possible replacement of it due to optimizations. This way, we consider `dcgettext`, `dgettext`, and `gettext` are identical library functions. However, we admit that our map is not complete, which could induce mismatching of certain callsites referring to different C library calls. We have discussed this issue and listed all renaming cases we constructed in Section 5.3. To our knowledge, it subsumes all possible C library replacements that could be found in our test cases.

External Callsite Elimination Due to Optimization. Compiler optimizations could replace standard C library calls into buildin function calls, and further inline the buildin function. For instance, we find that `gcc`, with `-O3` optimization enabled, could inline a buildin version of C library functions such as `memset` and `memcpy`. These would save the extra cost of function calls and returns. In contrast, `clang` seems to retain those library calls even with full optimizations enabled.

This could cause inconsistency and FNs when comparing assembly functions compiled by `gcc` and `clang`, since certain library callsites (e.g., `memset`) in assembly functions compiled by `clang` would never find its matched callsite in an assembly function compiled by `gcc`.

Zero External Callsites in a Function. When comparing two functions f_t and f_s, it is possible that neither function's expanded CFG contains any external calls. For such cases, our current implementation has to skip the comparison. Two functions are very unlikely to have identical functionality in case one function's expanded CFG contains external calls and the other does not. Nevertheless, FNs could still be introduced, in case all external function calls are eliminated by compiler optimizations from the expanded CFG, although the likelihood is extremely low. Furthermore, if no external callsites can be found in both functions, we rely on the similarity results yielded by DNN-based tools for comparison.

No Parameters in an External Callsite. Certain C library calls do not have parameters (e.g., `time`). For such cases, our current implementation instead extracts the path conditions of this function. In other words, our solution for such external callsites are aligned with our optimization introduced in Section 5.5. As mentioned in Section 5, path constraints are collected along with the symbolic execution and used to form the symbolic constraint. Nevertheless, if no path condition can be constructed, we skip comparing this callsite.

9.2 Others

In addition to inconsistency of external callsites, we also list some other issues that may undermine the analysis of BinUSE. They are primarily due to the unsound memory model of the angr and some corner cases raised by the cross-architecture comparisons.

angr Unsound Memory Model. angr couples concrete execution with symbolic execution to reduce the overhead [76]. Before dereferencing a symbolic pointer, angr would replace the symbolic memory address with a concrete address. This concolic design practical reduces the burden of performing complex points-to analysis. Nevertheless, adopting an unsound memory model makes implementing a sound BinUSE *infeasible*.

angr Tampering Global Memory Region. We also find that when starting to perform BinUSE at the entry point of a function, certain memory locations could have been somewhat initialized with concrete values. Those concrete values, when being used to compute branch conditions, can enforce angr to take only one path instead of exploring all feasible paths. As a result, angr can be impeded from finding paths to certain external callsites. Neglecting external callsites can lead to FNs as well.

IDA-Pro Failures. To compare a pair of assembly functions f and f', we conduct USE and constraint solving which is generally expensive. The implementation of BinUSE indeed adopts a practical early-stop condition criteria: f and f' will not be rigorously compared, in case they do not share even one identical external callee. Instead of using angr to collect external callees during USE, For the current implementation, we use IDA-Pro to statically disassemble binary code, reconstruct CFG, and recursively collect external callsites of f and f'. However, we report that IDA-Pro might sometimes throw reverse engineering failures and stop analyzing such functions, particularly for highly optimized and obfuscated binary code samples (recall we compile our test cases with full optimization and various obfuscation methods). IDA-Pro failed to analyze about 1.0% binary samples on all of our test cases; for these cases, we have to rely on the predictions of DNN-based tools.

Cross-Architecture Challenges. Our evaluation involves some cross-architecture settings, by comparing functions compiled on one architecture with functions compiled on different architectures. angr performs symbolic execution by first lifting binary code into VEX, a RISC-like intermediate language. While VEX itself is deemed "platform-independent," we still encountered a number of cross-architectural inconsistencies impeding foolproof equivalence checking.

64-bit Registers versus 32-bit Registers. We find plenty of cases, where a 64-bit register on x86 64-bit architectures are represented by two 32-bit registers on x86 32-bit architectures. When forming constraints for equivalence checking, it becomes very obscure, if at all possible, to match the 64-bit register with its corresponding two 32-bit registers.

64-bit Constant versus 32-bit Constant. We also find that certain constants are changed in 32-bit and 64-bit machine codes. For instance, a constant 0x55555554 in 32-bit x86 machine code is extended into 0x5555555555555554 in 64-bit machine code. While this should not affect execution on CPU, we note that such inconsistencies induce erroneous results for symbolic execution and equivalence checking.

9.3 Failures of BinUSE

This section lists failures thrown by the underlying symbolic engine and reverse engineering tools. As mentioned in Section 8, these issues contribute to the failures of BinUSE when analyzing coreutils binary code (on average 13.8% cases failed; as reported in Table 7).

Symbolic Code Pointer. As aforementioned in Section 9.2, angr implements a unsound memory model. When dereferencing a symbolic pointer, angr replaces it with a concrete address. When performing symbolic execution, we indeed observed a number of pointer deference failures thrown by angr. With further investigation, we find that before performing pointer dereference, certain symbolic code pointers cannot be concretized into concrete addresses. We have reported the issue to the angr developers. As noted by the angr developers, this issue demends specific concretizing strategy. Currently when encountering such dereference failure, we have to skip analyzing this function.

Unsupported Instructions When analyzing certain binary codes, angr may throw "unsupported instructions," leading to the failure of analyzing this function. For instance, angr failed to process x86 instruction pavgusb, which is a commonly used instruction to optimize image and video processing software like FFmpeg.

10 Related Work

Section 3 has reviewed DNN-based binary code similarity research and explored their limits. While recent DNN-based approaches have shown decent support for difficult settings like cross-optimization and cross-architecture [14], [15], [16], [23], [77], our research proposes to use low-cost functionality checking to compensate the inherent limitation of DNN-based approaches further. We have also reviewed the theory proving-based binary code equivalence checking in Section 2.2, and compare BinUSE with SE-based and random sampling-based binary code matching in Fig. 6.

Tracy [78] decomposes assembly functions into tracelets for comparison. The extracted features, mostly on the syntactic level, might suffer from cross optimization or cross compiler comparison settings. Two follow-up works, Esh [26] and GitZ [79], extract strands (i.e., data-flow slices of basic blocks) for comparison. Both methods operate at the boundaries of a basic block. Some challenging settings which break the integrity of basic blocks may reduce the accuracy of these two methods. GitZ lifts binary code into VEX IR, and then converts the lifted VEX IR into LLVM IR for analysis. However, their VEX to LLVM converter is not available for use or comparison.

Some conventional techniques leverage program syntactic features for similarity analysis, such as distributions of instructions, opcodes, system calls, or some control-flow graph-level features [9], [71], [80], [81], [82]. BinDiff, as the industrial standard tool, identifies similar code components through graph isomorphism comparison [71]. It has been pointed out that BinDiff is not robust toward obfuscation methods such as control flow flattening which largely changes the control flow structures [14], [31].

Several works leverage dynamic analysis or random input sampling for similarity analysis [83], [84], [85], [86], [87], [88], [89]. However, as compared in Fig. 6, a prominent issue is the low coverage of test targets (e.g., functions), rendering it generally unsuitable for production scenarios. Section 4 has clarified that while random sampling can be used to promptly generate input-output relations of two executables for comparison, they might exhibit potentially low completeness and result in FPs [21], [27], [30], [31], [32]. Recent works [90], by enhancing I/O features with structural features and high-level semantics features, manifest much better empirical results and outperform static tools like BinDiff and Tracy.

11 Conclusion and Future Works

This paper identifies common limitations in DNN-based binary code search and proposes to enhance DNN models with low-cost equivalence checking. In particular, we design BinUSE, a practical static analysis framework on the basis of under-constrained symbolic execution (USE) to check the equivalence of assembly functions. BinUSE incorporates a variety of optimizations to alleviate overhead incurred by path explosion and costly constraints. This way, BinUSE can be used to flag and shave assembly functions with semantic deviations compared with the target function, thus effectively enhancing the accuracy of DNN-based binary code matching. Our empirical results show that the proposed approach enables a general and highly effective improvement of cutting-edge DNN models in this field, making assembly function search more practical in production. We demonstrate the capability of BinUSE by matching programs from the Linux coreutils and binutils test suites, as well as the feasibility of largely augmenting vulnerable function search over complex real-world software such as OpenSSL, Wireshark, and FFmpeg.

As aforementioned, we have released the source code of BinUSE and evaluation data for reproducibility at [22]. In the future, we will maintain BinUSE to benefit research comparison and extension. On the basis of BinUSE, we envision many downstream applications may be explored and further developed. For instance, Section 8.7 has clarified that in addition to launching BinUSE to reorder only top-K matched functions, BinUSE can be used to verify each individual decision made by DNN: we first use recent advances in XAI techniques (e.g., neural attention) to scope critical code fragments contributing to the decision of DNN-based binary code matching, and then use BinUSE to verify the semantics equivalence of those scoped code fragments (which are usually much smaller than entire assembly functions).

Along with the vulnerable function search demonstrated in this paper, we intend to integrate BinUSE with de facto DNN-based binary matching to expedite the search for unknown vulnerabilities in real-world closed-source (commercial) software. BinUSE contains a high engineering effort directed at developing a cross-architecture solution capable of analyzing binary executables on both x86 and ARM (aarch64) platforms. With the proliferation of legacy software and third-party libraries on embedded and Internet of Things (IoT) devices, it is anticipated that BinUSE's cross-architecture, cross-compiler, and obfuscation-resistant binary code search capabilities will accelerate the process of matching vulnerable executable files.

Acknowledgments

We are grateful to the associate editor and anonymous reviewers for their valuable comments. All correspondence should be addressed to Shuai Wang at the address shown on the first page of this paper.

References

[1] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 81–96.

[2] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 1–12, 2008.

[3] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 887–902.

[4] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 389–400.

[5] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 111–121.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning distributed representations of code," in *Proc. ACM Program. Lang.*, 2019, pp. 1–29.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2018, *arXiv:1808.01400*.

[8] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019.

[9] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla , "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Netw. Distrib. Syst. Secur. Symp.*, 2016.

[10] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 896–899.

[11] S. Luan, D. Yang, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," , 2018. [Online]. Available: http://arxiv.org/abs/1812.01158

[12] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 363–376.

[13] T. Ben-Nun , A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, 2018, pp. 3589–3601.

[14] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 472–489.

[15] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 1145–1152.

[16] Y. Duan, X. Li, J. Wang, and H. Yin, "DEEPBINDIFF: Learning program-wide code representations for binary diffing," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[17] B. Liu *et al.*, "diff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2018, pp. 667–678.

[18] X. Li, Q. Yu, and H. Yin, "PalmTree: Learning an assembly language model for instruction embedding," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 3236–3251.

[19] D. A. Ramos and D. Engler, “Under-constrained symbolic execution: Correctness checking for real code,” in *Proc. 24th USENIX Conf. Secur. Symp.*, 2015, pp. 49–64.

[20] D. Gao, M. K. Reiter, and D. Song, “BinHunt: Automatically finding semantic differences in binary programs,” in *Proc. 10th Int. Conf. Inf. Commun. Secur.*, 2008, pp. 238–255.

[21] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, “Cross-architecture bug search in binary executables,” in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 709–724.

[22] Research Artifacts of BinUSE, 2021. [Online]. Available: https://github.com/computer-analysis/BinUSE

[23] I. U. Haq and J. Caballero, “A survey of binary code similarity,” 2019, *arXiv:1909.11424*.

[24] W. Jin *et al.*, “Binary function clustering using semantic hashes,” in *Proc. 11th Int. Conf. Mach. Learn. Appl.*, 2012, pp. 386–391.

[25] J. Ming, D. Xu, Y. Jiang, and D. Wu, “BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking,” in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 253–270.

[26] Y. David, N. Partush, and E. Yahav, “Statistical similarity of binaries,” in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 266–280.

[27] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, “BinGo: Cross-architecture cross-OS binary search,” in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 678–689.

[28] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM: A practical concolic execution engine tailored for hybrid fuzzing,” in *Proc. 27th USENIX Conf. Secur. Symp.*, 2018, pp. 745–761.

[29] S. Wang and D. Wu, “In-memory fuzzing for binary code similarity analysis,” in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 319–330.

[30] L. Jiang and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 81–92.

[31] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 303–317.

[32] D. McKee, N. Burow, and M. Payer, “Software ethology: An accurate and resilient, and cross-architecture binary analysis framework,” 2019, *arXiv:1906.02928*.

[33] C. Cadar *et al.*, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, pp. 209–224.

[34] F. Busse, M. Nowack, and C. Cadar, “Running symbolic execution forever,” in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 63–74.

[35] C. A. R. Hoare, “How did software get so reliable without proof?,” in *Proc. Int. Symp. Formal Methods Eur.*, 1996, pp. 1–17.

[36] E. Gunnerson, “Defensive programming,” in *A Programmer’s Introduction to C#*, New York, NY, USA: Apress, 2001.

[37] M. Stueben, “Defensive programming,” in *Good Habits for Great Coding*, New York, NY, USA: Apress, 2018.

[38] D. Brumley, P. Poosankam, D. Song, and J. Zheng, “Automatic patch-based exploit generation is possible: Techniques and implications,” in *Proc. IEEE Symp. Secur. Privacy*, 2008, pp. 143–157.

[39] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “SoK: Automated software diversity,” in *Proc. IEEE Symp. Secur. Privacy*, 2014, pp. 276–291.

[40] Translation with gettext, 2022. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Translation-with-gettext.html

[41] __printf_chk, 2022. [Online]. Available: https://refspecs.linuxbase.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/libc—printf-chk-1.html

[42] IDA Pro, 2021. [Online]. Available: https://hex-rays.com/ida-pro/

[43] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “ByteWeight: Learning to recognize functions in binary code,” in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 845–860.

[44] Fast Library Identification and Recognition Technology, 2021. [Online]. Available: https://www.hex-rays.com/products/ida/tech/flirt/

[45] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *Proc. 24th USENIX Conf. Secur. Symp.*, 2015, pp. 611–626.

[46] Y. Lin and D. Gao, “When function signature recovery meets compiler optimization,” in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 36–52.

[47] S. Wang, P. Wang, and D. Wu, “Reassembleable disassembling,” in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 627–642.

[48] R. Wang *et al.*, “Ramblr: Making reassembly great again,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017.

[49] E. Bauman, Z. Lin, and K. W. Hamlen, “Superset disassembly: Statically rewriting x86 binaries without heuristics,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[50] L. D. Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[51] Y. Shoshitaishvili *et al.*, “SOK: (State of) the art of war: Offensive techniques in binary analysis,” in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.

[52] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-LLVM: Software protection for the masses,” in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, 2015, pp. 3–9.

[53] T. J. McCabe , “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[54] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” 2018, *arXiv:1810.04805*.

[55] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, “Gated graph sequence neural networks,” 2015, *arXiv:1511.05493*.

[56] Y. Goldberg and O. Levy, “word2vec explained: Deriving mikolov *et al.*’s negative-sampling word-embedding method,” 2014, *arXiv:1402.3722*.

[57] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry,” 2017, *arXiv:1704.01212*.

[58] Q. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.

[59] RetDec, 2021. [Online]. Available: https://github.com/avast/retdec

[60] T. of Bits, McSema, 2018. [Online]. Available: https://github.com/lifting-bits/mcsema

[61] Microsoft, LLVM-mctoll, 2020. [Online]. Available: https://github.com/Microsoft/llvm-mctoll

[62] A. Talman, A. Yli-Jyrä, and J. Tiedemann, “Sentence embeddings in NLI with iterative refinement encoders,” *Natural Lang. Eng.*, vol. 25, no. 4, pp. 467–482, 2019.

[63] D. Gunning, Palmtree, 2021. [Online]. Available: https://github.com/palmtreemodel/PalmTree

[64] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 303–317.

[65] Z. Li *et al.*, “Unleashing the power of compiler intermediate representation to enhance neural program embeddings,” in *Proc. Int. Conf. Softw. Eng.*, 2022.

[66] KLEE Execution Time, 2021. [Online]. Available: https://www.dropbox.com/s/s1atcgwvby5q2hb/klee_table.pdf

[67] MoKLEE Execution Time, 2021. [Online]. Available: https://www.dropbox.com/s/im9jo6lpxrgzwcp/moklee_table.pdf

[68] Y. Hu, H. Wang, Y. Zhang, B. Li, and D. Gu, “A semantics-based hybrid approach on binary code similarity comparison,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 6, pp. 1241–1258, Jun. 2021.

[69] J. Jiang *et al.*, “Similarity of binaries across optimization levels and obfuscation,” in *Proc. Eur. Symp. Res. Comput. Secur.*, 2020, pp. 295–315.

[70] FunctionSimSearch, 2021. [Online]. Available: https://github.com/googleprojectzero/functionsimsearch

[71] H. Flake, “Structural comparison of executable objects,” in *Proc. Int. GI Workshop Detection Intrusions Malware Vulnerability Assessment*, 2004, pp. 161–174.

[72] Client Crash, 2021. [Online]. Available: https://github.com/McGill-DMaS/Kam1n0-Community/issues/73

[73] D. Gunning, “Explainable artificial intelligence (XAI),” *Defense Adv. Res. Projects Agency, nd Web*, vol. 2, no. 2, 2017.

[74] S. Woo, J. Park, J.-Y. Lee, and I. So Kweon , “CBAM: Convolutional block attention module,” in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 3–19.

[75] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, May 2018.

[76] F. Gritti *et al.*, “SYMBION: Interleaving symbolic with concrete execution,” in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2020.

[77] K. Redmond, L. Luo, and Q. Zeng, “A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis,” 2018, *arXiv:1812.09652*.

[78] Y. David and E. Yahav, “Tracelet-based code search in executables,” in *Proc. 35th ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2014, pp. 349–360.

[79] Y. David, N. Partush, and E. Yahav, “Similarity of binaries through re-optimization,” in *Proc. 38th ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2017, pp. 79–94.

[80] X. Hu, T.-C. Chiueh, and K. G. Shin, “Large-scale malware indexing using function-call graphs,” in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 611–620.

[81] T. Dullien and R. Rolles, “Graph-based comparison of executable objects (English version),” in *Proc. Symp. Secur. Technol. Inf. Commun.*, 2005, pp. 1–13.

[82] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, “Detecting code clones in binary executables,” in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 117–128.

[83] G. Myles and C. Collberg, “Detecting software theft via whole program path birthmarks,” in *Proc. Int. Conf. Inf. Secur.*, 2004, pp. 404–415.

[84] D. Schuler, V. Dallmeier, and C. Lindig, “A dynamic birthmark for Java,” in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 274–283.

[85] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, “Detecting software theft via system call based birthmarks,” in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2009, pp. 149–158.

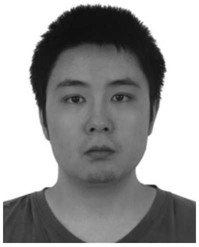
[86] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, “Value-based program characterization and its application to software plagiarism detection,” in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 756–765.

[87] S. Cesare and X. Yang, *Software Similarity and Classification*. Berlin, Germany: Springer, 2012.

[88] Y. C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, “Program characterization using runtime values and its application to software plagiarism detection,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 925–943, Sep. 2015.

[89] Y. Hu, Y. Zhang, J. Li, and D. Gu, “Cross-architecture binary semantics understanding via similar code comparison,” in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 57–67.

[90] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, “Accurate and scalable cross-architecture cross-os binary code search with emulation,” *IEEE Trans. Softw. Eng.*, vol. 45, no. 11, pp. 1125–1149, Nov. 2018.



Huaijin Wang received the bachelor’s degree in software engineering from Nanjing University, Jiangsu, China, in 2018. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include software security, reverse engineering, and program analysis.

Pingchuan Ma received the bachelor’s degree in information management and information system from Beijing Electronic Science and Technology Institute, Beijing, China, in 2020. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include software security and data mining.

Yuanyuan Yuan received the bachelor’s degree in computer science from Fudan University, Shanghai, China, in 2020. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include software security, reliability of AI systems, and software engineering.

Zhibo Liu received the bachelor’s degree in information security from Nankai University, Tianjin, China, in 2019. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include reverse engineering, software engineering, and computer security.

Shuai Wang (Member, IEEE) received the PhD degree in informatics from Pennsylvania State University, State College, PA, USA, in 2018. He is currently an assistant professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong. His research interests include cybersecurity and software engineering.

Qiyi Tang received the bachelor’s degree from the Beijing University of Posts and Telecommunications, Beijing, China. He is currently a researcher with KEEN Security Lab, Tencent. He is currently focuses on researching how to use AI algorithms to solve computer security problems.

Sen Nie received the master’s degree from Shanghai Jiao Tong University, Shanghai, China. He is currently a security researcher with KEEN Security Lab, Tencent. His current research focuses on AI-powered software composition analysis.

Shi Wu received the bachelor’s degree from Fudan University, Shanghai, China. He currently leads the Keen Lab, a world-class security research lab in Tencent. He has had many years’ research experiences on static analysis, fuzzing, and other vulnerability detection technologies.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.