Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge

SHOUGUO YANG, CHAOPENG DONG, YANG XIAO, YIRAN CHENG, ZHIQIANG SHI, ZHI LI, and LIMIN SUN, Institute of Information Engineering, Chinese Academy of Sciences, China and School of Cyber Security, University of Chinese Academy of Sciences, China

Widespread code reuse allows vulnerabilities to proliferate among a vast variety of firmware. There is an urgent need to detect these vulnerable codes effectively and efficiently. By measuring code similarities, *AI-based binary code similarity detection* is applied to detecting vulnerable code at scale. Existing studies have proposed various function features to capture the commonality for similarity detection. Nevertheless, the significant code syntactic variability induced by the diversity of IoT hardware architectures diminishes the accuracy of binary code similarity detection. In our earlier study and the tool *Asteria*, we adopted a Tree-LSTM network to summarize function semantics as function commonality, and the evaluation result indicates an advanced performance. However, it still has utility concerns due to excessive time costs and inadequate precision while searching for large-scale firmware bugs.

To this end, we propose a novel deep learning-enhancement architecture by incorporating domain knowledge-based pre-filtration and re-ranking modules, and we develop a prototype named Asteria-Pro based on *Asteria*. The pre-filtration module eliminates dissimilar functions, thus reducing the subsequent deep learning-model calculations. The re-ranking module boosts the rankings of vulnerable functions among candidates generated by the deep learning model. Our evaluation indicates that the pre-filtration module cuts the calculation time by 96.9%, and the re-ranking module improves MRR and Recall by 23.71% and 36.4%, respectively. By incorporating these modules, Asteria-Pro outperforms existing state-of-the-art approaches in the bug search task by a significant margin. Furthermore, our evaluation shows that embedding baseline methods with pre-filtration and re-ranking modules significantly improves their precision. We conduct a large-scale real-world firmware bug search, and Asteria-Pro manages to detect 1,482 vulnerable functions with a high precision 91.65%.

CCS Concepts: • **Security and privacy → Software security engineering;**

S. Yang and C. Dong contributed equally to this work.

This research was supported in part by National Key R&D Program of China (Grant No. 2022YFB3103904), the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDC02020100), Joint Fund Cultivation Project of National Natural Science Foundation of China (Grant No. U1636120), the Science and Technology Project of State Grid Corporation of China (Grant No. 5700-202258499A-3-0-ZZ), National Natural Science Foundation of China (Grant No. U1766215), and Chinese National Natural Science Foundation (Grant No. 62202462). Any opinions, findings, and conclusions in this article are those of the authors and do not necessarily reflect the views of the funding agencies.

Authors' addresses: S. Yang, C. Dong, Y. Xiao (corresponding author), Y. Cheng, Z. Shi (corresponding author), Z. Li, and L. Sun, Institute of Information Engineering, Chinese Academy of Sciences, No. 19 Shucun Road, Haidian District, Beijing, China, 100085 and School of Cyber Security, University of Chinese Academy of Sciences, No. 1 Yanqihu East Rd, Huairou District, Beijing, China, 101408; emails: {yangshouguo, dongchaopeng, xiaoyang, chengyiran, shizhiqiang, lizhi, sunlimin}@iie.ac.cn.

1049-331X/2023/11-ART1 $15.00

https://doi.org/10.1145/3604611

Additional Key Words and Phrases: Binary code similarity detection, pre-fitering, re-ranking, abstract syntactic tree, graph neural network

ACM Reference format:
Shouguo Yang, Chaopeng Dong, Yang Xiao, Yiran Cheng, Zhiqiang Shi, Zhi Li, and Limin Sun. 2023. Asteria-Pro: Enhancing Deep Learning-based Binary Code Similarity Detection by Incorporating Domain Knowledge. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 1 (November 2023), 40 pages.
https://doi.org/10.1145/3604611

1 INTRODUCTION

Code reuse is very popular in IoT firmware to facilitate its development [63]. Unfortunately, code reuse also introduces vulnerabilities concealed in the original code into a variety of firmware [22]. The security and privacy of our lives are seriously threatened by the widespread use of these firmware [64]. Even though the vulnerabilities have been publicly disclosed, there are a large number of firmware versions that still contain them due to delayed code upgrades or code compatibility issues [18]. Recurring vulnerabilities, often referred to as "N-day vulnerabilities," cannot be detected through symbol information such as function names, because this type of information is usually removed during firmware compilation. Additionally, the source code of firmware is typically unavailable as IoT vendors only provide binary versions of their firmware.

To this end, **binary code similarity detection (BCSD)** is applied to quickly find homologous vulnerabilities in a large amount of firmware [23]. The BCSD technique focuses on determining the similarity between two binary code pieces. As for the vulnerability search, BCSD looks for other vulnerable functions that are similar to one that is already known to be vulnerable. In addition to the vulnerability search, BCSD has been widely used for other security applications such as code plagiarism detection [16, 48, 56], malware detection [41, 42], and patch analysis [28, 34, 62]. Despite many existing research efforts, the diversity of IoT hardware architectures and software platforms poses challenges to BCSD for IoT firmware. There are many different **instruction set architectures (ISA)** for IoT firmware, such as ARM, PowerPC, X64, and X86. The instructions are different, and the rules, such as the calling convention and the stack layout, also differ across different ISAs. It is non-trivial to find homologous vulnerable functions across various architectures.

BCSD methods can be generally classified into two categories: (i) Dynamic analysis-based methods and (ii) static analysis-based methods. The methods based on dynamic analysis capture the runtime behavior as function semantic features by running target functions, where the function features can be I/O pairs of function [54] or system calls during program execution [29], and so on. They are not scalable for large-scale firmware analysis, since running firmware requires specific devices and emulating firmware is also difficult [20, 35, 72]. The methods based on static analysis mainly extract statistical features from assembly code. An intuitive way is to calculate the edit distance between assembly code sequences [24]. They cannot be directly applied across architectures, since instruction sets are totally distinct. Architecture-independent statistical features of functions are proposed for similarity detection [31]. These features are less affected across architectures such as the number of function calls, strings, and constants. Furthermore, the **control flow graph (CFG)** at the assembly code level is utilized by conducting a graph isomorphism comparison for improving the similarity detection [31, 33]. Based on statistical features and CFG, Gemini [65] leverages the graph embedding network to encode functions as vectors for similarity detection. With the application of deep learning models in programming language analysis, various methods have recently appeared to employ such models to encode binary functions in different forms and calculate function similarity based on function encoding [46, 50, 53, 61]. Static analysis-based methods are faster and more scalable for large-scale firmware analysis but often produce false positives due to the lack of semantic information. Since homologous vulnerable functions in different architectures usually have the same semantics, a cross-architecture BCSD should be able to capture the semantic information about functions in a way that can be scaled.

In our previous work *Asteria* [70], we first utilized the Tree-LSTM network to encode the AST in an effort to capture its semantic representation. In particular, Tree-LSTM is trained using a *Siamese* [37] architecture to understand the semantic representation by feeding homologous and non-homologous function pairs into the Tree-LSTM network. Consequently, the Tree-LSTM network learns function semantic representations to distinguish between homologous and non-homologous functions. To further improve the accuracy, we also use the call graph to calibrate the AST similarity. Precisely, we count callee functions of target functions in the call graph to measure the difference in function calls. The final function similarity is determined by calibrating the AST similarity with the disparity in function calls. In our previous evaluation, *Asteria* outperformed the available state-of-the-art methods, *Gemini* and *Diaphora*, in terms of accuracy. The evaluation results demonstrate the superiority of function semantic extraction by encoding AST with the Tree-LSTM model. However, encoding the AST incurs a clear temporal cost for *Asteria*. According to our earlier research [70], the entire AST encoding process takes about one second. When *Asteria* is applied to vulnerability detection, where there are numerous functions to perform similarity calculations given a vulnerable function, the time cost becomes unacceptable. Since the majority of candidate functions are non-homologous, there is room for enhancing the efficiency of *Asteria*. In other words, non-homologous candidate functions differ from vulnerable functions in certain characteristics that we can exploit to skip the majority of non-homologous functions more effectively. In addition, the evaluations do not align with the approaches used in the majority of real-world vulnerability detection efforts [33, 45, 51, 65, 73], including our prior study *Asteria*. Vulnerability detection involves retrieving homologous (vulnerable) functions from a large pool of functions. Consequently, their performance in detecting vulnerabilities is insufficiently described. It is necessary to evaluate the performance of *Asteria* on the vulnerability search task. Moreover, according to the result in the real world vulnerability detection [70], *Asteria* suffers from high false positives, which affects its effectiveness in reality.

There are two main challenges that hinder *Asteria* from being practical for large-scale vulnerability detection:

- **Challenge 1 (C1).** It is challenging to filter out the majority of non-homologous functions before encoding ASTs, while retaining the homologous ones, to speed up the vulnerability-detection process.
- **Challenge 2 (C2).** It is challenging to distinguish similar but non-homologous functions. Despite *Asteria*'s high precision in homologous and non-homologous classification, it still yields false positives when distinguishing functions with similar ASTs.

We design Asteria-Pro by introducing domain knowledge as two answers, **A1** and **A2** to overcome these two challenges. Our fundamental concept is that introducing inter-functional domain knowledge will helps Asteria-Pro achieve greater precision combined the intra-functional semantic knowledge deep learning model learned. Asteria-Pro consists of three modules: **(1) Domain Knowledge-based (DK-based)** pre-filtration, **(2) Deep Learning-based (DL-based)** similarity detection, and **(3)** DK-based re-ranking, among them DL-based similarity detection is basically based on *Asteria*. Domain knowledge is fully exploited for different purposes in DK-based pre-filtration and re-ranking. In pre-filtration module, Asteria-Pro aims to skip as many as possible non-homologous function by comparing lightweight robust features (**A1**). Meanwhile, filtration is required to retain all homologous functions. To this end, we conducted a preliminary study into the filtering performance of several lightweight function features. According to the findings of the study, we propose a novel algorithm that successfully employs three distinct function features in the filter. In the re-ranking module, Asteria-Pro confirms the homology of functions by comparing call relationships (**A2**), based on the assumption that functions designed for distinct purposes have different call relationships.

Our evaluation indicates that Asteria-Pro significantly outperforms existing state-of-the-art methods in terms of both accuracy and efficiency. Compared with *Asteria*, Asteria-Pro successfully cuts the detection time of *Asteria* by **96.90%** by incorporating DK-based pre-filtration module. In the vulnerability-search task, Asteria-Pro has the shortest average search time than other baseline methods. By incorporating DK-based re-ranking, Asteria-Pro manages to enhance the MRR and Recall@Top-1 by 23.71% and 36.4%, to **90.8%** and **89.6%**, respectively. We have also applied our enhancement framework to embed baseline methods, and the evaluation results demonstrate a significant improvement in the precision of these methods. Asteria-Pro identifies 1,482 vulnerable functions with a high precision of 91.65% by conducting a large-scale real-world firmware vulnerability detection utilizing 90 CVEs. Moreover, the detection results of CVE-2017-13001 demonstrate that Asteria-Pro has an advanced capacity to detect inlined vulnerable code.

Our contributions are summarized as follows:

- We conduct a preliminary study to demonstrate the effectiveness of various simple function features in identifying non-homologous functions.
- To the best of our knowledge, it is the first work to propose incorporating domain knowledge before and after deep learning models for vulnerability detection optimization. We implement the domain knowledge-based pre-filtration and re-ranking algorithms and equip Asteria with them.
- The evaluation indicates the pre-filtration module significantly reduces the detection time, and re-ranking module improves the detection precision by a fairly amount. The Asteria-Pro outperforms existing state-of-the-art methods in terms of both accuracy and efficiency. In evaluation 8.5, we find that the performance of distinct BCSD methods may vary widely in different usage scenarios.
- We demonstrate the utility of Asteria-Pro by conducting a large-scale, real-world firmware vulnerability detection. Asteria-Pro manages to find 1,482 vulnerable functions with a high precision of 91.65%. We analyze the vulnerability distribution in widely-used software from various IoT vendors to illustrate our inspiring findings.

2 BACKGROUND

We first briefly describe the AST structure adopted in this work, followed by a demonstration of the AST holding a more stable structure than CFG across architectures. Then we introduce the Tree-LSTM model utilized in AST encoding. Finally, the broad problem definition for the application of BCSD to bug search is given.

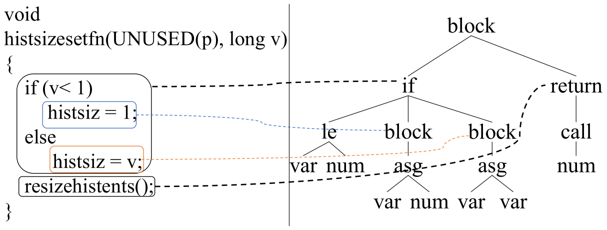
2.1 Abstract Syntax Tree

2.1.1 AST Description. An AST is a tree representation of the abstract syntactic structure of code in the compilation and decompilation processes. This work focuses on the ASTs extracted by decompiling binary functions. Different subtrees in an AST correspond to different code scopes in the source code. Figure 1 shows a decompiled AST corresponding to the source code of function `histsizesetfn` in `zsh v5.6.2` on the left. The `zsh` is a popular shell software designed for interactive use, and the function `histsizesetfn` sets the value of a parameter. The lines connecting the source code and AST in Figure 1 show that a node in the AST corresponds to an expression or a statement in the source code. A variable or a constant value is represented by a leaf node in AST. We group nodes in an AST into two categories: (i) Statement nodes and (ii) expression

Table 1. Statements and Expressions in ASTs

	Node Type	Label	Note
Statement	if	1	if statement
	block	2	instructions executed sequentially
	for	3	for loop statement
	while	4	while loop statement
	switch	5	switch statement
	return	6	return statement
	goto	7	unconditional jump
	continue	8	continue statement in a loop
	break	9	break statement in a loop
Expression	asgs	10∼17	assignments, including assignment, assignment after or, xor, and, add, sub, mul, div
	cmps	18∼23	comparisons including equal, not equal, greater than, less than, greater than or equal to, and less than or equal to.
	ariths	24∼34	arithmetic operations including or, xor, addition, subtraction, multiplication, division, not, post-increase, post-decrease, pre-increase, and pre-decrease
	other	34∼43	others including indexing, variable, number, function call, string, asm, and so on.

We count the statements and expressions for nodes in ASTs after the decompilation by `IDA Pro` and list the common statements and expressions. This table can be extended if new statements or expressions are introduced.

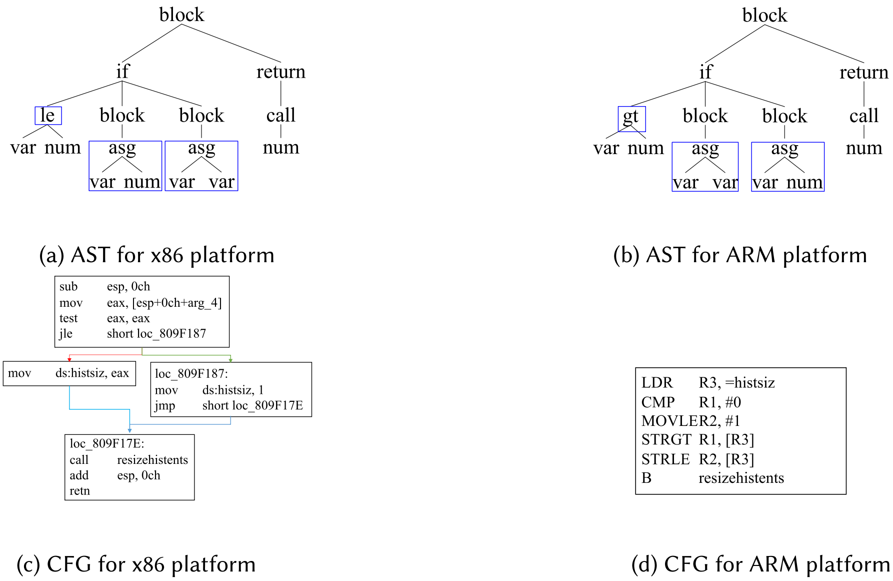


```
void
histsizesetfn(UNUSED(p), long v)
{
 if (v< 1)
    histsiz = 1;
 else
     histsiz = v;
 resizehistents();
}
```

Fig. 1. Source code of function *histsizesetfn* and the corresponding decompiled AST of x86 architecture.

nodes according to their functionalities shown in Table 1. Statement nodes control the function execution flow while expression nodes perform various calculations. Statement nodes include *if, for, while, return, break*, and so on. Expression nodes include common arithmetic operations and bit operations.

2.1.2 AST Structure Superiority. Both CFG and AST are structural representations of a function. The CFG of a function contains the jump relationships between basic blocks that contain straight-line code sequences [38]. Though CFG has been used for similarity measurement in BCSD [31], David et al. [24] demonstrated that CFG structures differ significantly across different architectures. We observe that AST shows better architectural stability across architectures compared with CFG. It is because AST is generated from the machine-independent intermediate presentations, which are disassembled from assemble instructions during the decompilation process [21]. Figure 2 depicts the evolution of ASTs and CFGs for the x86 and ARM architectures, respectively. For the CFGs from x86 to ARM, we observe that the number of basic blocks changes from 4 to 1, and the number of assembly instructions has changed a lot. However, the ASTs, which are based on a higher-level intermediate representation, differ very slightly between x86 and ARM, with the



(a) AST for x86 platform

(b) AST for ARM platform

```
sub     esp, 0ch
mov     eax, [esp+0ch+arg_4]
test    eax, eax
jle     short loc_809F187
```

```
mov     ds:histsiz, eax
```

```
loc_809F187:
mov     ds:histsiz, 1
jmp     short loc_809F17E
```

```
loc_809F17E:
call    resizehistents
add     esp, 0ch
retn
```

```
LDR     R3, =histsiz
CMP     R1, #0
MOVLER2, #1
STRGT   R1, [R3]
STRLE   R2, [R3]
B       resizehistents
```

(c) CFG for x86 platform

(d) CFG for ARM platform

Fig. 2. ASTs and CFGs of the function *histsizesetfn* under different architectures.

differences highlighted by blue boxes. In addition, AST maintains the semantics of functionality, making it an ideal structure for cross-platform similarity detection.

2.2 Tree-LSTM Model

In natural language processing, **Recursive Neural Networks (RNN)** are widely applied and perform better than Convolutional Neural Networks [71]. RNNs take sequences of arbitrary lengths as inputs considering that a sentence can consist of any number of words. However, standard RNNs are not capable of handling long-term dependencies due to the gradient vanishing and gradient exploding problems. As one of the variants of RNN, **Long Short-Term Memory (LSTM)** [39] has been proposed to solve such problems. LSTM introduces a gate mechanism including the input, forget, and output gates. The gates control the information transfer to avoid the gradient vanishing and exploding (calculation details in Section 6.1). Nevertheless, LSTM can only process sequence input but not structured input. Tree-LSTM is proposed to process tree-structured inputs [58]. The calculation by Tree-LSTM model is from the bottom up. For each non-leaf node in the tree, all information from child nodes is gathered and used for the calculation of the current node. In sentiment classification and semantic relatedness tasks, Tree-LSTM performs better than a plain LSTM structure network. There are two types of Tree-LSTM proposed in the work [59]: Child-Sum Tree-LSTM and Binary Tree-LSTM. Researchers have shown that Binary Tree-LSTM performs better than Child-Sum Tree-LSTM [59]. Since the Child-Sum Tree-LSTM does not take into account the order of child nodes, while the order of statements in AST reflects the function semantics, we use the Binary Tree-LSTM for our AST encoding.

2.3 Function Call Compilation Optimization

There are two main types of function call compilation optimization that can impact binary code similarity analysis: Function inline and intrinsic functions.

Function inline. Function inline is a compiler optimization technique where the code of a called function is inserted directly into the calling function, rather than making a separate function call.

This can improve program performance by reducing the overhead of function calls and improving cache utilization. The decision to inline a function is typically made by the compiler based on various factors such as function size, frequency of calls, and available register space.

Intrinsic function. Intrinsic functions (also known as built-in functions) are special functions that are implemented by the compiler itself and are mapped to a single instruction or a sequence of instructions in the target architecture. These functions provide low-level access to the hardware and are used to implement various low-level operations, such as arithmetic, bit manipulation, and memory access. Intrinsic functions are often used in performance-critical code, where the use of low-level instructions can lead to significant speedups compared to equivalent code written in a higher-level language.

3 PRELIMINARY STUDY

This study aims to assess and uncover accessible function features that are effective at identifying non-homologous functions to guide our pre-filtration design. To evaluate the features, we prepare the code base and incorporate a number of metrics (Section 3.1). We focus primarily on evaluating and comparing prevalent conventional features present in existing remarkable works (Section 3.2).

3.1 Evaluation Benchmark

3.1.1 Dataset. To derive robust features, we compile a large collection of binaries from 184 **open source software (OSS)**, including widely used OpenSSL, FFmpeg, Binutils, and so on. Since our tool aims to conduct similarity detection across different architectures, we compile these open source software for four common architectures: X86, X64, ARM, and PowerPC. In addition, we align the default compilation settings during compilation with real-world usage. After compilation, numerous test binaries with “test” or “buildtest” as a prefix or suffix are generated to test the software’s functionality. These test binaries are removed from the collection, because (1) their functions are simple and comprise only a few lines of code, and (2) they do not participate in the real execution of software function. After removal, the binary collection retains 1,130 binaries, or 226 for each architecture.

We create a large dataset consisting of pairs of homologous and non-homologous functions based on their function names. Function names are retained in the software after compilation, allowing us to construct the dataset. To create homologous function pairs, we select binary functions with the same function names within the same software. However, functions with different names were considered non-homologous. For example, if function F is present in the source code, then compilation would generate four versions of binary functions for different instruction set architectures: F_{x86}, F_{x64}, F_{arm}, and F_{ppc}. These variants of functions are considered homologous to each other. We extract a total of 529,096 binary functions, comprising 132,274 unique functions for each architecture. To avoid overfitting in final evaluation, we randomly selected 40,111 functions from each architecture. Among them, we randomly chose n functions as source functions to evaluate the filtering capability of diverse features. For each source function F^BX, we constructed a pool of candidate functions consisting of M randomly selected binary functions and three homologous functions of F_X^B. As a result, each source function F_X^B forms three homologous pairs and M non-homologous pairs.

3.1.2 Metrics. **True-positive rate (TPR)** and **false-positive rate (FPR)** are utilized to evaluate the filtering capability of various features. TPR demonstrates the feature’s capacity to retain homologous functions, while FPR demonstrates its capacity to exclude non-homologous functions. In the subsequent filtering phase, our goal is *to identify features that can filter out non-homologous functions as effectively as possible (low FPR) while maintaining all homologous functions (very high TPR).*

For a source function F_X^B, all function pairs in candidate function pool are measured by various feature similarity scores. The function pairs with similarity scores below a threshold value T are filtered. In the remaining function pairs, the homologous function pairs are regarded as true positives TP while the non-homologous function pairs are regarded as false positives FP. The following equations illustrate how we calculate these three metrics for various features:

$$TPR = \frac{\sum_{i=1}^{n} TP_i^p}{3 \times n}, \tag{1}$$

$$FPR = \frac{\sum_{i=1}^{n} FP_i}{n \times (M \times 4 - 4)}. \tag{2}$$

3.2 Candidate Features Evaluation

We aim to identify the most efficient and effective filter features by evaluating existing features proposed in previous studies and their variants. Based on the evaluation results, we select and improve candidate features to meet the filter requirements, which is to remove as many non-homologous functions as possible while retaining all homologous ones.

3.2.1 Feature Selection. We gather basic features from prior research [31, 65, 66, 70] and categorize them into two groups: CFG-family features and AST-family features.

The CFG-family features include four types of numeric features: The number of instructions (*No. Instruction*), arithmetic instructions (*No. Arithmetic*), call instructions (*No. Callee*), and logical instructions (*No. Logic*), along with two constant features: String constants (*String Constant*) and numeric constants (*Numeric Constant*) [31]. We also introduce a newly proposed feature called the *named callee list* (*NCL*) to capture the text sequence information of callee functions that retain their function names due to dynamic linking. In particular, *NCL* is designed to be a list of callee functions that are either imported or exported functions. These functions retain their original names as they are used as identifiers to reference the functions in other parts of the code.

Since AST is necessary for model encoding calculation (Section 6), we summarize three syntactic features as AST-family features:

- *No. AST Nodes*: The number of AST nodes.
- *AST Node Cluster*: The number of different node types in the AST. For example, in Figure 1, the AST node cluster is denoted as $[block : 3, if : 1, return : 1, call : 1, num : 3, block : 2, asg : 2, var : 4, le : [1]$.
- *AST Fuzzy Hash*: We first generate a node sequence by traversing the AST preorder. Then, we apply the fuzzy hash algorithm [44] to generate the fuzzy hash of the AST.

3.2.2 Feature Similarity Calculation. The format of features divides them into two types with distinct similarity calculations: value type and sequence type. Value type features consist of *No. Instruction*, *No. Arithmetic*, *No. Logic*, *No. Callee*, and *No. AST nodes*. Sequence type features consist of *Numeric Constant*, *String Constant*, *AST Node Cluster*, *AST Fuzzy Hash*, and *NCL*. For value type features, we use the relative difference ratio (*RDR*) as shown below for similarity calculation:

$$RDR(V_1, V_2) = 1 - \frac{abs(V_1 - V_2)}{max(V_1, V_2)}, \tag{3}$$

where V_1, V_2 are feature values. For each sequence-type feature, we first sort the feature's items and then concatenate them into a single sequence. Then, we employ the **common sequence ratio (CSR)** based on the **longest common sequence (LCS)** as follows:

$$CSR(S_1, S_2) = \frac{2 \times LCS(S_1, S_2)}{len(S_1) + len(S_2)}, \tag{4}$$

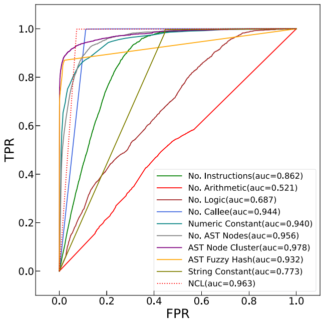


Fig. 3. ROC curves for features.

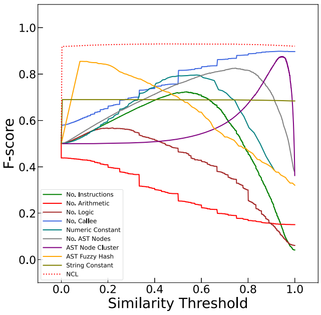


Fig. 4. F_{score} of features.

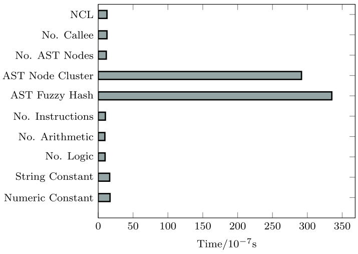


Fig. 5. Time costs of similarity calculation for different features.

where S_1, S_2 are feature sequences, and function $LCS(\cdot, \cdot)$ returns the length of the longest common sequence between S_1, S_2. The above two equations are used for similarity calculation of various features.

3.2.3 Evaluation Results. In the evaluation, the values for n and M in Section 3.1.2 are set to 1,000 and 20,000, respectively. As depicted in Figure 3, TPRs and FPRs calculated for each feature under various thresholds are presented as a **receiver operating characteristic (ROC)** [74] curve. Additionally, we compute the **area under the ROC curve (AUC)**, which reflects the feature's ability to distinguish between homologous and non-homologous functions. The AUC values of the features extracted from AST (i.e., *No. AST Nodes*, *AST Node Cluster*, and *AST Fuzzy Hash*) are high, as presented in the Figure 3. However, when the TPR is high, they generate a high FPR.

We observe in Figure 3 that at high TPR (0.996), the *No. Callee* feature produces a relatively lower FPR (0.111). Recalling the requirement of the filtering phase, we aim to select features with a low FPR at a very high TPR. Features with high AUC do not necessarily meet our objective. For example, the feature *AST Node Cluster* has a higher FPR (0.47) than the feature *No. Callee* (FPR = 0.111) under the same TPR (0.996), even though the feature *AST Node Cluster* has a higher AUC (0.978) than the feature *No. Callee* (AUC = 0.944). In this regard, we propose a new metric, F_{score}, which indicates a high TPR and a lower FPR:

$$F_{score} = \frac{1}{\frac{1}{TPR} + FPR}. \tag{5}$$

Figure 4 plots the F_{score} curves of various features at different similarity thresholds. The results indicate that the "*NCL*" feature has the highest F_{score} of 0.92 among all the candidate features. It achieves a high true-positive rate at a low false-positive rate, with a relatively high AUC score of 0.963. The "*No. Callee*" feature performs slightly worse, with an AUC score of 0.944 and an F_{score} of 0.902. The "*String Constant*" feature shows a relatively high F_{score} at a very low threshold (e.g., 0.01), since it decisively determines the homology of functions. In particular, if two functions have the same strings, then they are highly likely to be homologous. Although the F_{score} does not increase as the threshold increases, it is because some functions do not include string constants, which limits the number of true-positive pairs. Based on the filtering performance of the candidate features, we have decided to use *NCL* along with *No. Callee* and *String Constant* for our prefiltering design. Figure 5 depicts the time costs associated with similarity calculations for various features. Clearly, sequence type features require more time than value type features. Nonetheless, their time consumption falls within an acceptable range of magnitudes. At least 10^5 exact calculations can be completed every second.

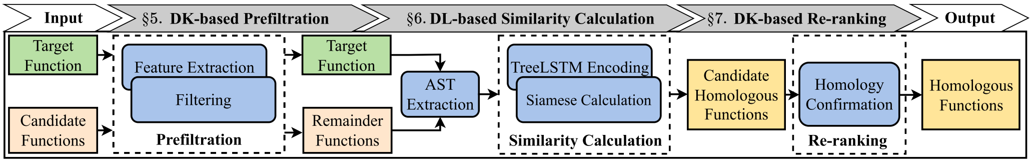


Fig. 6. Workflow of Asteria-Pro. DK stands for domain knowledge. DL stands for deep learning.

4 METHODOLOGY OVERVIEW

Asteria-Pro consists of three primary modules: **DK-based Prefiltration**, **DL-based Similarity Calculation**, and **DK-based Re-ranking**, as shown in Figure 6. Here, DK stands for Domain Knowledge, and DL stands for Deep Learning. The DK-based prefiltration module utilizes syntactic features to filter out dissimilar functions from the candidate functions in a lightweight and efficient manner (see Section 5). The DL-based similarity calculation module encodes ASTs into representation vectors using the Tree-LSTM model and determines the similarity score between the target function and the remaining functions using a Siamese network (see Section 6). The DK-based re-ranking module reorders the candidate homologous functions produced by the DL-based similarity calculation module using lightweight structural features, such as the function call relationship. By integrating these three modules, Asteria-Pro efficiently and effectively detects homologous functions across architectures.

5 DK-BASED PREFILTRATION

At this stage, Asteria-Pro aims to incorporate an efficient and effective filter. To achieve this goal, we have summarized the challenges associated with fully exploiting the *NCL* feature. Based on these challenges, we have developed a novel algorithm that overcomes these obstacles and enables us to construct the filter.

5.1 Exploitation Challenges

We have manually examined the false-negative cases where homologous functions were filtered out by both the *NCL* and *No. of callee* features. Through this examination, we have identified the challenges associated with appropriately exploiting the *NCL* and *No. of callee* features to address these false-negative cases.

- **Exploitation Challenge 1 (EC1).** Decorated callee function name. A decorated callee function name is the result of a function name being decorated by the compiler using various techniques [8]. One such technique is name mangling, which is used by the C++ compiler to encode the function name with additional information about its parameters and return type to facilitate function overloading. The name of the function after decoration may differ from its original name in the source code and will be distinct across different architectures, particularly X86 and X64, due to their differing return types.
- **Exploitation Challenge 2 (EC2).** In certain functions, commonly referred to as leaf nodes in a call graph, there are no callee functions present. These functions are self-contained and do not call any other functions within their code. As a result, the leaf nodes do not possess distinguishable *NCL* and *No. of callee* features.
- **Exploitation Challenge 3 (EC3).** Function calls in binary target functions might not always be consistent with source code. Function calls may be added or deleted due to compiler optimization. The reasons for the function call change are function inline, intrinsic function replacement, instruction replacement for optimization, that behave differently in different architectures. These challenges are introduced in Section 2.3.

To overcome exploitation challenges, we improve feature *ECL* and propose a novel algorithm **UpRelation**.

5.2 Definition of *NCL*

This section provides a formal definition of *NCL* to enhance clarity and precision. *NCL* is built upon the call graph of the software. The *call graph CG* can be defined by representing all functions as nodes and the call relationships between them as edges: $CG = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v|v \text{ is a function}\}$ denotes the node collection and $\mathcal{E} = \{(u, v)|u \text{ calls } v\}$ denotes the edge collection. For any edge $(u, v) \in \mathcal{E}$, we say that function v is a callee function of function u. To facilitate linking, function names in the dynamic symbol table *DST* (i.e., import and export table) are preserved [36]. For instance, if a target function calls an external function such as "strcpy," then the callee function name "strcpy" remains in the import table, rather than being removed after binary stripping. The *NCL* of a target function f is defined as $NCL_f = \{v|v \in \mathcal{V}, v \in DST\ (f, v) \in \mathcal{E}, \}$, where v is sorted by its call instruction address.

To address **EC1**, we employ two strategies to recover the original function names. First, for C++ decorated names, we use the recovery tool *cxxfilt* [5] to recover the function names. Second, for other decorated functions, we define heuristic rules to recover the function names. For example, we recover the function call to "_gets" by replacing it with "gets," by removing the underscore at the beginning. In cases where a function calls the same function multiple times, we keep multiple identical function names.

5.3 Filtration Algorithm

To address the additional two challenges, we propose a callee similarity-based algorithm called **UpRelation**. This algorithm leverages context information in the call graph to overcome challenges **EC2** and **EC3**. Specifically, the algorithm utilizes parent nodes of leaf nodes in the call graph to match similar leaf nodes and address challenge **EC2**. In the algorithm, we adopt a drill-down strategy that combines three features: *NCL*, *No. Callee*, and *String Constants*, based on their information content. The *No. Callee* of function f is denoted by $Callee_f$, and the set of *String Constants* for function f is denoted by $StrCons_f$.

Given a vulnerable function f_v, Algorithm 1 aims to eliminate most non-homologous functions while retaining the vulnerable candidate functions in a list (VFL) from the target function list (TFL). The code from lines 2 to 6 performs filtering when the feature NCL_{fv} of f_v is not empty. Specifically, the algorithm calculates the callee similarity ratio (*CSR*) between NCL_{fv} and NCL_f of all candidate functions in line 4. It then filters out functions whose *CSR* is less than a threshold T_{NCL}. Similarly, when the number of callee functions ($Callee_{fv}$) of f_v is not zero, the algorithm filters out functions by calculating the Relevance Distance Ratio (*RDR*) score from line 7 to 11. The most crucial portion of the algorithm is in lines 12 to 19, where it matches the leaf functions to address **EC2**. All caller functions of f_v are first visited, and the algorithm employs $UpRelation$ to discover all functions that are similar to the caller function *caller* in line 14. For each similar function $caller'$, the algorithm considers all its callee functions as vulnerable candidate functions at line 16. Matching the same leaf functions by locating the same caller functions introduces some extraneous (leaf) functions that share the same caller function but are not the same as the leaf function. To remove these extraneous functions, the algorithm utilizes string similarity at line 22. After filtering by callees and strings, the algorithm finally obtains the expected vulnerable candidate function list VFL.

Leaf Node Calculation Illustration. When the *Strings*, *No. Callee*, and *NCL* are non-empty, the similarity calculation in our algorithm is straightforward. The arduous aspect of the algorithm lies in managing leaf functions that do not call other functions. To provide a clearer illustration,

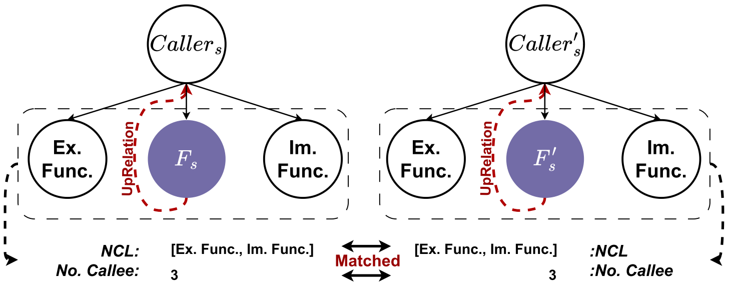


Fig. 7. The leaf function F'_s is matched by its homologous function F_s (rather than being filtered out). **Ex. Func.** is short for "Exported Function." **Im. Func.** is short for "Imported Function."

ALGORITHM 1: UpRelation

```
Input: Vulnerable Function $fv$, Target Function List $TFL$, Thresholds $T_{NCL}, T_{callee}, T_{string}$
Output: Vulnerable Candidate Function List $VFL$
$VFL \leftarrow TFL$;
if $NCL_{fv}$ is not null then
    for $f \in TFL$ do
        $s$ = CSR($NCL_{fv}, NCL_f$);
        if $s < T_{NCL}$ then $VFL$.pop($f$);
    end
else if $Callee_{fv} > 0$ then
    for $f \in TFL$ do
        $s$ = RDR($Callee_{fv}, Callee_f$);
        if $s < T_{callee}$ then $VFL$.pop($f$);
    end
else
    $FL' = \emptyset$;
    for $caller \in$ GetCallers($fv$) do
        for $caller' \in UpRelation(caller, VFL)$ do
            $FL'$.add(GetCallees($caller'$));
        end
    end
    $VFL = FL'$;
if $StrCons_{fv}$ is not null then
    for $f \in VFL$ do
        $s$ = CSR($StrCons_{fv}, StrCons_f$);
        if $s < T_{string}$ then $VFL$.pop($f$);
    end
else
    return
end
$VFL$;
```

we have employed an example depicted in Figure 7 to demonstrate why homologous functions of leaf function F_s are preserved after pre-filtration. In this example, we presume that leaf function F_s does not comprise any strings. The algorithm proceeds to lookup its caller and collate its NCL as $[Ex., Func., Im., Func.]$, where **Ex. Func.** is an abbreviation for "Exported Function," and **Im.**

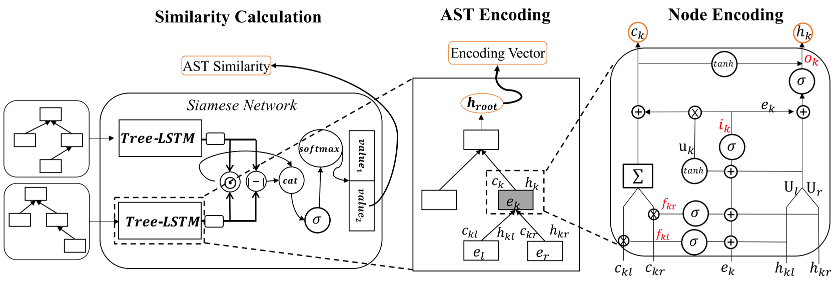


Fig. 8. Siamese architecture and Tree-LSTM encoding.

Func. is an abbreviation for "Imported Function." Similarly, the algorithm collects the *NCL* of caller of F_s' and attempts to correlate between the two *NCLs*. We postulate that homologous functions from the same software have equivalent callers, signifying that caller functions *caller* and *caller′* invoke the same exported and imported functions. Consequently, the *NCL* of two caller functions comprise the same elements $[Ex., Func., Im., Func.]$. Upon the successful correlation of the *NCL* of caller function $Caller_s'$, the algorithm preserves all its offspring nodes, encompassing F_s', **Im. Func.**, and **Ex. Func.**, and eliminates all other functions. As a result, homologous function F_s' of F_s is conserved after pre-filteration.

6 DL-BASED SIMILARITY CALCULATION

This module calculates the similarity between two function ASTs by encoding them into vectors and applying the Siamese architecture to calculate similarity between encoded vectors. Figure 8 depicts the calculation flow.

6.1 Tree-LSTM Encoding

Given an AST, Tree-LSTM model encodes it into a representation vector. Tree-LSTM model is firstly proposed to encode the tree representation of a sentence and summarize the semantic information in natural language processing. Tree-LSTM model can preserve every property of the plain LSTM gating mechanisms while processing tree-structured inputs. The main difference between the plain LSTM and the Tree-LSTM is the way to deal with the outputs of predecessors. The plain LSTM utilizes the output of only one predecessor in the sequence input. We utilize Tree-LSTM to integrate the outputs of all child nodes in the AST for calculation of the current node. To facilitate the depiction of the Tree-LSTM encoding, we assume that node v_k has two child nodes v_l and v_r. The Tree-LSTM encoding of node v_k takes three types of inputs: Node embedding e_k of v_k, hidden states h_{kl} and h_{kr}, and cell states c_{kl} and c_{kr} as illustrated in Figure 8. The node embedding e_k is generated by using the pre-trained model CodeT5 to embed the node v_k to a high-dimensional representation vector. h_{kl}, h_{kr}, c_{kl}, and c_{kr} are outputs from the encoding of child nodes. During the node encoding in Tree-LSTM, there are three gates and three states that are important in the calculation. The three gates are calculated for filtering information to avoid gradient explosion and gradient vanishing [58]. They are input, output, and forget gates. There are two forget gates f_{kl} and f_{kr}, filtering the cell states from the left child node and right child node separately. As shown in **Node Encoding** in Figure 8, the forget gates are calculated by combining h_{kl}, h_{kr}, and e_k. Similar to the forget gates, the input gate, and the output gate are also calculated by combining h_{kl}, h_{kr}, and e_k. The details of the three types of gates are as follows:

$$f_{kl} = \sigma\left(W^f e_k + \left(U_{ll}^f h_{kl} + U_{lr}^f h_{kr}\right) + b^f\right), \tag{6}$$

$$f_{kr} = \sigma\left(W^f e_k + \left(U_{rl}^f h_{kl} + U_{rr}^f h_{kr}\right) + b^f\right), \tag{7}$$

$$i_k = \sigma\left(W^i e_k + \left(U_l^i h_{kl} + U_r^i h_{kr}\right) + b^i\right), \tag{8}$$

$$o_k = \sigma\left(W^o e_k + \left(U_l^o h_{kl} + U_r^o h_{kr}\right) + b^o\right), \tag{9}$$

where i_k and o_k denote the input gate and the output gate, respectively, and the symbol σ denotes the `sigmoid` activation function. The weight matrix W, U, and bias b are different corresponding to different gates. After the gates are calculated, there are three states u_k, c_k, and h_k in Tree-LSTM to store the intermediate encodings calculated based on inputs h_{kl}, h_{kr}, and e_k. The cached state u_k combines the information from the node embedding e_k and the hidden states h_{kl} and h_{kr} (Equation (10)). And note that u_k utilizes `tanh` as the activation function rather than *sigmoid* for holding more information from the inputs. The cell state c_k combines the information from the cached state u_k and the cell states c_{kl} and c_{kr} filtered by forget gates (Equation (11)). The hidden state h_k is calculated by combining the information from cell state c_k and the output gate o_k (Equation (12)). The three states are computed as follows:

$$u_k = tanh\left(W^u e_k + \left(U_l^u h_{kl} + U_r^u h_{kr}\right) + b^u\right), \tag{10}$$

$$c_k = i_k \odot u_k + \left(c_{kl} \odot f_{kl} + c_{kr} \odot f_{kr}\right), \tag{11}$$

$$h_k = o_k \odot tanh(c_k), \tag{12}$$

where the $\odot$ means Hadamard product [40]. After the hidden state and input state are calculated, the encoding of the current node v_k is finished. The states c_k and h_k will then be used for the encoding of v_k's parent node. During the AST encoding, Tree-LSTM encodes every node in the AST from bottom up as shown in **Tree-LSTM Encoding** in Figure 8. After encoding all nodes in the AST, the hidden state of the root node is used as the encoding of the AST.

6.2 Siamese Calculation

This step uses Siamese architecture, which integrates two identical Tree-LSTM models to calculate similarity between encoded vectors. The details of the Siamese architecture $\mathcal{M}(T_1, T_2)$ are shown in Figure 8. The Siamese architecture consists of two identical Tree-LSTM networks that share the same parameters. In the process of similarity calculation, the Siamese architecture first utilizes Tree-LSTM to encode ASTs into vectors. We design the Siamese architecture with subtraction and multiplication operations to capture the relationship between the two encoding vectors. After the operations, the two resulting vectors are concatenated into a larger vector. Then the resulting vector goes throughÂăa layer of *softmax* function to generate a two-dimensional vector. The calculation is defined as

$$\mathcal{M}(T_1, T_2) = softmax(\sigma(cat(|\mathcal{N}(T_1) - \mathcal{N}(T_2)|, \mathcal{N}(T_1) \odot \mathcal{N}(T_2)) \times W))), \tag{13}$$

where W is a $2n \times 2$ matrix, the $\odot$ represents Hadamard product [40], $|\cdot|$ denotes the operation of making an absolute value, the function $cat(\cdot)$ denotes the operation of concatenating vectors. The softmax function normalizes the vector into a probability distribution. Since W is a $2n \times 2$ weight matrix, the output of Siamese architecture is a 2×1 vector. The format of output is $[dissimilarity\ score, similarity\ score]$, where the first value represents the dissimilarity score and the second represents the similarity score. During the model training, the input format of Siamese architecture is $< T_1, T_2, label >$. In our work, the label vector $[1, 0]$ means T_1 and T_2 are from non-homologous function pairs and the vector $[0, 1]$ means homologous. The resulting vector and the label vector are used for model loss and gradient calculation. During model inference, the second

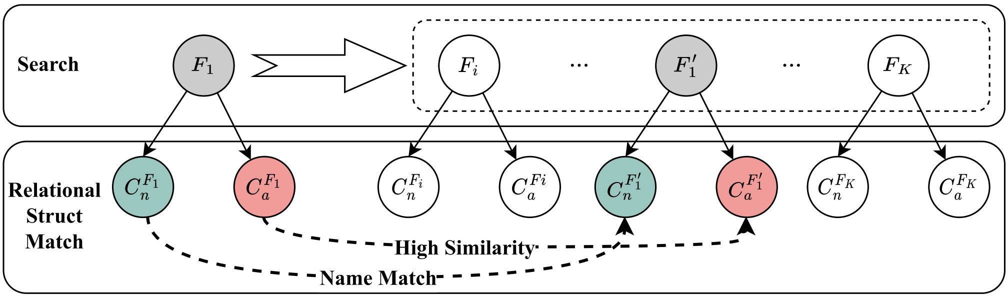


Fig. 9. The re-ranking motivation example. In the rectangular box with dashed line are the top K candidate homologous functions of F_1 produced by the search (i.e., DL-based similarity detection). Solid line arrows indicate the function call relationship (e.g., F_1 calls $C_n^{F_1}$). The dotted line arrows indicate the callee function match in re-ranking.

value in the output vector is taken as the similarity of the two ASTs, and the similarity of ASTs is used in re-ranking.

7 DK-BASED RE-RANKING

This module seeks to confirm the homology of the top k candidate functions output by the Tree-LSTM network by re-ranking them. In the prior phase, the Tree-LSTM network infers the semantic information from the AST, which is an intra-functional feature. The knowledge gained from the AST is insufficient to establish the homology of functions. In this phase, function call relationships are used as domain knowledge to compensate for the lack of knowledge regarding the inter-functional features of the Tree-LSTM. To this end, we design an algorithm called **Relational Structure Match**. In contrast to the callee application in the pre-filtering module, this module uses more extensive information from callee relationships to show the degree of homology of candidate functions.

7.1 Motivated Example

Our algorithm is based on a conforming observation to an intuitive law: *If a function F_1 calls function C^{F_1}, then its homologous function F_1' will also call the homologous function $C^{F_1'}$ of C^{F_1}*. As depicted in Figure 9, we have F_1 calls C^{F_1}, and F_1' calls $C^{F_1'}$. Assume that the search process for F_1 yields the top K functions containing the target homologous function F_1'. We then employ the call relations of F_1 and F_1' to conduct precise callee function matching for re-ranking. In particular, callee functions of F_1 are divided into two categories, named callees $C_n^{F_1}$ and anonymous callees $C_a^{F_1}$. For named callees, their names are utilized to match callees of functions between the source function F_1 and the candidate top K functions. For anonymous callees, we employ DL-based similarity detection to calculate the similarity between callees of functions between the source function F_1 and the candidate top K functions. Recalling the observation, the homologous function F_1' of F_1 holds the most matched callees. After re-ranking the candidate functions based on the matched callees, F_1' is re-ranked in the first place.

7.2 Relational Structure Match Algorithm

The algorithm aims to rescore each candidate function by leveraging the call relationship between the target function and the candidate functions. The relational structure refers to the call relations between the target function and all its callee functions, as illustrated in Section 7.1. To match the relational structure, the algorithm performs one of two distinct operations (O_1 and O_2) based on whether the source function has callee functions or not.

O_1: When the source function F_1 has one or more callee functions, the algorithm extracts all callee functions of F_1 to build a ***mixed callee function set*** **(*MCFS*)**. (details are described below). Using *MCFS*, the algorithm calculates similarities between the target function and the candidate functions, resulting in new match scores. It re-ranks all candidate functions by combining the original **Asteria** scores (Equation (5)) with the newly calculated match scores. The details of *MCFS* and match score calculation are described in the subsequent sections.

O_2: When the source function F_1 has no callee functions, the algorithm removes all candidate functions that have one or more callee functions. The remaining candidate functions are then re-ranked based on their original **Asteria** scores.

7.2.1 Mixed Callee Function Set. The *MCFS* of function F consists of two types of callee functions: Named callee functions and anonymous callee functions. Named callee functions refer to functions whose names have been preserved. These functions are typically imported or exported functions, and their function names are necessary for external linking purposes. However, anonymous callee functions are a type of function for which the function name has been removed for security reasons. These functions are typically anonymized to protect sensitive information. We denote the *MCFS* of function F as $CS_F = C_{n1}^F, \ldots, C_{nj}^F, C_{a1}^F, \ldots, C_{aj}^F$, where C_{nj}^F represents a named callee function and C_{aj}^F represents an anonymous callee function. The set CS_F includes both types of callee functions for function F.

7.2.2 Match Score Calculation. The algorithm performs two types of matches to calculate the match score M for each candidate function, utilizing the *MCFS*s of the target function F_1 and all candidate functions.

Named Callee Match: For all named callees $C_{nj}^{F_1}$ in CS_{F_1}, the algorithm matches them with the named callees of each candidate function based on function names. If a named callee $C_n^{F_1}$ in F_1 has the same function name as a named callee $C_n^{F_1'}$ in a candidate function F_1', then they are considered a match. The number of matched functions in candidate function F_i is denoted as $\mathcal{N}_n^{F_i}$.

Anonymous Callee Match: For all anonymous callees $C_{ai}^{F_1}$ in CS_{F_1}, the algorithm utilizes DL-based similarity detection to calculate similarity scores between all anonymous callees of the target function and the anonymous callees of all candidate functions. For each anonymous callee $C_{aj}^{F_i}$ in a candidate function F_i, the algorithm calculates the maximum similarity score between $C_{aj}^{F_i}$ and all anonymous callees of the target function. This maximum similarity score is denoted as $\mathcal{S}_{aj}^{F_i}$.

After matching all callee functions of the candidate functions, the match score M_{F_i} of candidate function F_i is calculated as follows:

$$M_{F_i} = \mathcal{N}_n^{F_i} + \sum \mathcal{S}_{aj}^{F_i}, \tag{14}$$

where $\mathcal{S}_{aj}^{F_i}$ represents the similarity score between an anonymous callee $C_{aj}^{F_i}$ in candidate function F_i and the anonymous callees of the target function F_i. The sum is taken over all anonymous callees in CS_{F_i}, the *MCFS* of candidate function F_i.

7.2.3 Match Score-based Re-ranking. The re-ranking score of candidate function F_i is obtained by combining the match score M_{F_i} and its DL-based similarity score $\mathcal{M}F_i$ using Equation (15). The algorithm calculates a new score $S^{re-rank}F_i$ for each candidate function F_i as follows:

$$S_{F_i}^{re-rank} = \alpha \times \mathcal{M}_{\mathcal{F}_{\rangle}} + \beta \times M_{F_i}. \tag{15}$$

Here, α and β are weight coefficients that satisfy $\alpha + \beta = 1$. The new score $S^{re-rank}F_i$ combines the DL-based similarity score $\mathcal{M}F_i$ and the match score M_{F_i}, emphasizing their importance according to the weights. A higher re-ranking score indicates a higher degree of homology.

After calculating the re-ranking scores for all candidate functions, the algorithm sorts them in descending order based on their new scores $S^{re-rank}_{F_i}$. This ranking allows for the identification of candidate functions with higher homology, as those with higher scores are prioritized.

8 EVALUATION

We aim to conduct a comprehensive practicality evaluation of various state-of-the-art function similarity detection methods for bug search. To this end, we adopt eight different metrics to depict the search capability of different methods in a more comprehensive way. Furthermore, we construct a large evaluation dataset, in a way that is closer to practical usage of bug search.

8.1 Research Questions

In the evaluation experiments, we aim to answer following research questions:

- **RQ1.** How does Asteria-Pro compare to baseline methods in cross-architecture and cross-compiler function similarity detection?
- **RQ2.** What is the performance of Asteria-Pro, compared to baseline methods for bug search purpose?
- **RQ3.** How much do DK-based filtration and DK-based re-ranking improves in accuracy and efficiency for Asteria-Pro? How do their performance compare to other baseline methods when integrated together?
- **RQ4.** How do different configurable parameters affect the accuracy and efficiency?
- **RQ5.** How does Asteria-Pro perform in a real-world bug search?

8.2 Implementation Details

We utilize `IDA Pro 7.5` [10] and its plugin *Hexray Decompiler* to decompile binary code and extract ASTs. The current version of the *Hexray Decompiler* supports x86, x64, PowerPC (PPC), and ARM architectures. For the encoding of leaf nodes in Equations (6)–(11), We assign zero vectors to the state vectors h_{kl}, h_{kr}, c_{kl}, and c_{kr}. During model training, we use the **binary cross-entropy loss function (*BCELoss*)** to measure the discrepancy between the labels and the predictions. The *AdaGrad* optimizer is utilized for gradient computation and weight-matrix updating after the losses are computed. Due to the dependency of Tree-LSTM computation steps on the AST shape, parallel batch computation is not possible. Therefore, the batch size is always set to 1. The model is trained for 60 epochs. Our experiments are conducted on a local server with two Intel Xeon CPUs E5-2620 v4 @ 2.10 GHz, each with 16 cores, 128 GB of RAM, and 4 TB of storage. The Asteria-Pro code runs in a Python 3.6 environment. We compile the source code in our dataset using the `gcc v5.4.0` compiler and utilize `buildroot-2018.11.1` [3] for dataset construction. We use the binwalk tool [2] to unpack firmware and obtain the binaries for further analysis. In the **UpRelation** algorithm of the filtering module, we set the threshold values $T_{NCL}, T_{callee}, T_{string}$ to 0.1, 0.8, and 0.8, respectively, based on their F_{score}. The crucial threshold T_{NCL} is discussed in Section 8.8.1. In Equation (15), we set $\alpha = 0.1$ and $\beta = 0.9$ to emphasize the role of callee function similarities in the re-ranking process. The sensitivity analysis of these weights is presented in Section 8.8.2.

8.3 Comprehensive Benchmark

To compare BCSD methods in a comprehensive way, we build an extensive benchmark based on multiple advanced works [50, 61, 65]. The benchmark comprises of two datasets, two detection tasks, and five measure metrics.

8.3.1 Dataset. The functions not involved in the prefiltering test (see Section 3) are divided into two datasets for model training and testing and evaluation. The evaluation dataset consists of two sub-datasets, each of which is used for a different detection task.

Model Dataset Construction. The model dataset is constructed for training and testing the Tree-LSTM model. It consists of a total of 31,940 functions extracted from 1,944 distinct binaries. From these functions, 314,852 pairs of homologous functions and 314,852 pairs of non-homologous functions are created. To ensure a fair evaluation of the model's performance, the dataset is divided into a training set and a testing set using an 8:2 ratio. This means that 80% of the function pairs are used for training the model, while the remaining 20% are used for testing and evaluating the model's performance. The dataset construction allows the Tree-LSTM model to learn and generalize from a diverse set of functions, including both homologous and non-homologous pairs. By dividing the dataset into training and testing sets, the model's performance can be assessed on unseen data to measure its effectiveness in identifying homologous functions.

Evaluation Dataset Construction. The dataset construction process involves creating two sub-datasets: the g-dataset and the v-dataset. These datasets are used for different evaluation tasks: Classification test and bug search test. The **g-dataset** is constructed for the classification test, which evaluates the model's ability to classify homologous and non-homologous function pairs. It consists of tuples in the form $(F, (F_h, F_n))$, where F is the source function and (F_h, F_n) represents a function set containing a homologous function F_h and a non-homologous function F_n. Each tuple in the g-dataset represents a pair of functions to be classified as homologous or non-homologous. However, the **v-dataset** is constructed for the bug search test, which evaluates the model's ability to identify non-homologous functions among a larger set of candidates. The tuples in the v-dataset are of the form $(F, (F_h, F_{n1}, \ldots, F_{ni}, \ldots, F_{n10000}))$. Here, F_h represents a homologous function, and F_{n1} to F_{n10000} represent non-homologous functions. In this case, the P_{set} contains a larger number of non-homologous functions to simulate the bug search scenario. For both datasets, the source function F is matched with all the functions in the P_{set} for evaluation. The g-dataset focuses on evaluating the model's accuracy in classifying homologous and non-homologous pairs, while the v-dataset assesses the model's performance in identifying non-homologous functions among a larger pool of candidates.

8.3.2 Metrics. We choose five distinct metrics for comprehensive evaluation from earlier works [53, 61, 70]. In our evaluation, the similarity of a function pair is calculated as a score of r. Assuming the threshold is β, if the similarity score r of a function pair is greater than or equal to β, then the function pair is regarded as a positive result; otherwise, it is regarded as a negative result. For a homologous pair, if its similarity score r is greater than or equal to β, then it is a true positive **(TP)**. If a similarity score of r is less than β, then the calculation result is a false negative **(FN)**. For a non-homologous pair, if a similarity score r is greater than or equal to β, then it is a false positive **(FP)**. When the similarity score r is less than β, it is a true negative **(TN)**. These metrics are described as follows:

- **TPR.** TPR is short for true-positive rate. TPR shows the accuracy of homologous function detection at threshold β. It is calculated as $TPR = \frac{TP}{TP+FN}$.
- **FPR.** FPR is short for false-positive rate. FPR shows the accuracy of non-homologous function detection at threshold β. It is calculated as $FPR = \frac{FP}{FP+TN}$.
- **AUC.** AUC is short for area under the curve, where the curve is termed Receiver Operating Characteristic **(ROC)** curve. The ROC curve illustrates the detection capacity of both homologous and non-homologous functions as its discrimination threshold β is varied. AUC is a quantitative representation of ROC.

- **MRR.** MRR is short for mean reciprocal rank, which is a statistic measure for evaluating the results of a sample of queries, ordered by probability of correctness. It is commonly used in retrieval experiments. In our bug retrieval-manner evaluation, it is calculated as $MRR = \frac{1}{|P_{set}|} \sum_{F_{hi} \in P_{set}} \frac{1}{Rank_{F_{hi}}}$, where $Rank_{F_{hi}}$ denotes the rank of function F_{hi} in pairing candidate set P_{set}, and $|P_{set}|$ denotes the size of P_{set}.
- **Recall@Top-k.** It shows the capacity of homologous function retrieve at top k detection results. The top k results are regarded as homologous functions (positive). It is calculated as follows:

$$g(x) = \begin{cases} 1 & \text{if } x = True \\ 0 & \text{if } x = False \end{cases},$$

$$Recall@k = \frac{1}{|F|} \sum g(Rank_{f_i^{gt}} \le k).$$

To demonstrate the reliability of the ranking results, we adopt Recall@Top-1 and Recall@Top-10.

8.3.3 *Detection Tasks.* The two function similarity detection tasks based on BCSD applications are as follows:

Task-C (Classification Task): This task focuses on evaluating the ability of methods to classify function pairs as either homologous or non-homologous. It involves performing binary classification on the **g-dataset**, which contains tuples of the form $(F, (F_h, F_n))$, where F_h represents a homologous function and F_n represents a non-homologous function. The task evaluates the performance using three metrics: TPR, FPR, and AUC of the ROC curve. TPR and FPR are commonly used to measure the performance of binary classification models, while AUC provides an overall measure of the model's discriminative ability.

Task-V (Bug/Vulnerability Search Task): This task focuses on evaluating the ability of methods to identify homologous functions from a large pool of candidate functions. It uses the **v-dataset**, which contains tuples of the form $(F, (F_h, F_{n1}, \ldots, F_{ni}, \ldots, F_{n10000}))$, where F_h represents a homologous function and F_{ni} represents non-homologous functions. The task involves calculating function similarity between a source function F and all functions in the P_{set}. The functions in P_{set} can then be sorted based on similarity scores. The task evaluates the performance using three metrics: MRR, Recall@Top-1, and Recall@Top-10. MRR measures the rank of the first correctly identified homologous function, while Recall@Top-1 and Recall@Top-10 measure the proportion of cases where the correct homologous function is included in the top-1 and top-10 rankings, respectively.

These tasks provide a comprehensive evaluation of the methods' performance in distinguishing between homologous and non-homologous functions and identifying homologous functions from a large pool of candidates.

8.4 Baseline Methods

We choose various representative cross-architectural BCSD works, that make use of AST or are built around deep learning encoding. These BCSD works consist of *Diaphora* [7], *Gemini* [65], *SAFE* [51], and *Trex* [53]. Moreover, we also use our previous conference work *Asteria* as one of baseline methods. We go over these works in more details below.

Diaphora. *Diaphora* performs similarity detection also based on AST. *Diaphora* maps nodes in an AST to primes and calculates the product of all prime numbers. Then it utilizes a *difference function* to calculate the similarity between the prime products. We download the *Diaphora* source code from github [7], and extract *Diaphora*'s core algorithm for AST similarity calculation for comparison. Noting that it would take a significant amount of time (several minutes) to compute a pair of functions with extremely dissimilar ASTs, we add a filtering computation before the prime difference. The filtering calculates the AST size difference and eliminates function pairs with a significant size difference. We publish the improved Diaphora source code on our website [1].

Gemini. *Gemini* encodes **attributed CFGs (ACFGs)** into vectors with a graph embedding neural network. The ACFG is a graph structure where each node is a vector corresponding to a basic block. We have obtained *Gemini*'s source code and its training dataset. Notice that in Reference [65] the authors mentioned it can be retrained for a specific task, such as the bug search. To obtain the best accuracy of *Gemini*, we first use the given training dataset to train the model to achieve the best performance. Then, we re-train the model with the part of our training dataset. *Gemini* supports similarity detection on X86, MIPS, and ARM architectures.

SAFE. *SAFE* works directly on disassembled binary functions, does not require manual feature extraction, is computationally more efficient than *Gemini*. In their vulnerability search task, *SAFE* outperforms *Gemini* in terms of recall. *SAFE* supports three different instruction set architecture X64, X86, and ARM. We retrain *SAFE* based on the official code [51] and use retrained model parameter for our test. In particular, we select all appropriate function pairs from the training dataset, whose instruction set architectures are supported by *SAFE*. Then, we extract the function features for all function pairs selected and discard the function pairs whose features *SAFE* cannot extract. After feature extraction, 27,580 function pairs of three distinct architecture combinations (i.e., X86-X64, X86-ARM, and X64-ARM) are obtained for training. Next, We adopt the default model parameters (e.g., embedding size) and training setting (e.g., training epoches) to train *SAFE*.

Trex. *Trex* is based on pretrained model [53] of the state-of-the-art NLP technique, and micro-traces. It utilizes a dynamic component to extract micro-traces and use them to pretrain a masked language model. Then it integrates pretrained ML model into a similarity detection model along with the learned semantic knowledge from micro-traces. It supports similarity detection of ARM, MIPS, X86, and X64.

8.5 Comparison of Similarity Detection Accuracy (RQ1)

In the evaluation of cross-architecture scenarios, the focus was on assessing the detection capability of different approaches in two tasks separately, which is commonly encountered in vulnerability search scenarios. Additionally, the evaluation also considered the performance in cross-compiler scenarios involving three different combinations of compilers: gcc-clang, gcc-icc, and clang-icc.

8.5.1 Cross-architecture Evaluation. In the evaluation of the two distinct tasks, it is important to note that the baseline methods may not be capable of detecting function similarities for all four instruction set architectures. As a result, the detection results for certain architecture combinations may be empty, indicating that the baseline methods were unable to provide any meaningful results. For each task, the evaluation measured the performance of various approaches in terms of the defined metrics. The specific outcomes and results of the evaluation for each task were analyzed and discussed.

Comparison on task-C. In Task-C, all approaches were evaluated by conducting similarity detection on all supported architectural combinations. The evaluation results were used to calculate the three metrics (TPR, FPR, and AUC) for each approach. These results are presented in Table 2 and visualized in Figure 10, where each subplot represents the ROC curve for a specific architecture combination. The x-axis represents the FPR (False-positive Rate), and the y-axis represents the TPR (True-positive Rate). By examining the ROC curves in Figure 10, it can be observed that methods with performance curves closer to the upper-left corner generally exhibit superior performance.

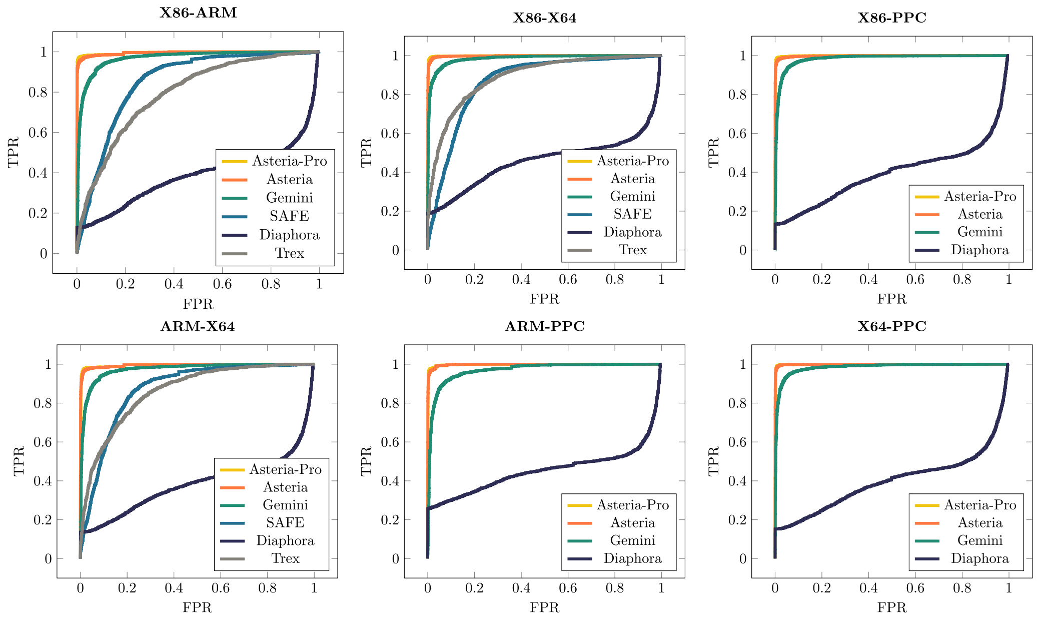


Fig. 10. ROC curves on all cross-architecture combination detection.

Table 2. AUCs in Task-C

Methods	X86-ARM	X86-X64	X86-PPC	ARM-X64	ARM-PPC	X64-PPC	Average
Asteria-Pro	**0.996**	**0.998**	**0.995**	**0.998**	**0.998**	**0.999**	**0.997**
Asteria	0.995	0.998	0.998	0.995	0.998	0.999	0.997
Gemini	0.969	0.984	0.984	0.973	0.968	0.984	0.977
SAFE	0.851	0.867	—	0.872	—	—	0.863
Trex	0.794	0.891	—	0.861	—	—	0.849
Diaphora	0.389	0.461	0.397	0.388	0.455	0.400	0.415

In particular, the ROC curves of Asteria-Pro and Asteria are almost indistinguishable across all architectural combinations, indicating that they possess equivalent classification performance in Task-C. Furthermore, the AUC values presented in Table 2 provide a quantitative measure of the approaches' ability to distinguish between homologous and non-homologous functions. It is noted that Asteria-Pro and Asteria demonstrate nearly identical performance in this regard. However, the AUC values of Asteria-Pro are consistently greater than those of the other baseline techniques for all architectural combinations. This suggests that Asteria-Pro exhibits superior discriminative capability between homologous and non-homologous functions in Task-C. These findings highlight the strong performance of Asteria-Pro in the classification task and its ability to outperform the baseline methods in distinguishing between homologous and non-homologous functions across various architecture combinations.

Comparison on task-V. Table 3 presents the results of calculating MRR, Recall@Top-1, and Recall@Top-10 for different architectural combinations. These metrics evaluate the performance of the methods in the bug (vulnerability) search task. Recall@Top-1 measures the ability to accurately detect homologous functions, while Recall@Top-10 assesses the capability to rank homologous functions within the top ten positions. In the table, the first column represents the metrics, and the second column lists the names of the methods. The third through eighth columns display

Table 3. MRR and Recall of Different Methods

Metrics	Methods	X86-X64	X86-ARM	X86-PPC	X64-ARM	X64-PPC	ARM-PPC	Avg
MRR	**Asteria-Pro**	**0.934**	**0.887**	**0.931**	**0.879**	**0.919**	**0.903**	**0.908**
	Asteria	0.776	0.724	0.731	0.708	0.713	0.750	0.734
	Trex	0.414	0.206	—	0.309	—	—	0.310
	Gemini	0.478	0.250	0.325	0.336	0.357	0.256	0.334
	Safe	0.029	0.007	—	0.009	—	—	0.015
	Diaphora	0.023	0.019	0.020	0.019	0.020	0.021	0.020
Recall@Top-1	**Asteria-Pro**	**0.917**	**0.868**	**0.912**	**0.879**	**0.899**	**0.903**	**0.896**
	Asteria	0.706	0.648	0.652	0.627	0.631	0.675	0.657
	Trex	0.274	0.110	—	0.192	—	—	0.192
	Gemini	0.405	0.180	0.242	0.261	0.279	0.229	0.266
	Safe	0.004	0.002	—	0.002	—	—	0.003
	Diaphora	0.021	0.016	0.017	0.016	0.017	0.018	0.018
Recall@Top-10	**Asteria-Pro**	**0.961**	**0.921**	**0.962**	**0.913**	**0.952**	**0.932**	**0.940**
	Asteria	0.902	0.867	0.882	0.857	0.866	0.890	0.877
	Trex	0.710	0.452	—	0.575	—	—	0.579
	Gemini	0.615	0.383	0.482	0.478	0.502	0.468	0.488
	Safe	0.022	0.010	—	0.014	—	—	0.015
	Diaphora	0.029	0.024	0.026	0.026	0.025	0.027	0.026

the metric values for the different architectural combinations, while the last column shows the mean value across all architectures. It can be observed that Asteria-Pro and *Asteria* consistently outperform the baseline approaches by a significant margin across all architecture configurations. Asteria-Pro achieves an impressive average MRR of 0.908, indicating a substantial improvement of up to 23.71% compared to *Asteria*. Even after retraining, *Safe* demonstrates poor performance in properly recognizing small functions. In terms of Recall@Top-1, both Asteria-Pro and *Asteria* achieve relatively high average precisions of 0.89 and 0.65, respectively, which are 237% and 146% higher than the best result (0.26). Notably, Asteria-Pro shows a 36.4% improvement in Recall@Top-1 compared to *Asteria*. Regarding Recall@Top-10, both Asteria-Pro and *Asteria* continue to exhibit superior performance compared to the other methods. While other methods show a significant increase in recall compared to Recall@Top-1, their values remain below Asteria-Pro. Overall, these results demonstrate that Asteria-Pro outperforms the baseline methods, including *Asteria*, in terms of MRR, Recall@Top-1, and Recall@Top-10 across different architecture combinations. The recall of other methods, such as *Trex*, increases significantly from Recall@Top-1 to Recall@Top-10, indicating their ability to rank homologous sequences more accurately. However, they still fall short compared to Asteria-Pro.

Indeed, the performance of BCSD approaches can vary significantly between different evaluation tasks, as demonstrated by the differences in Task-V performance compared to the similar ROC curve performance in Task-C. In the case of *Gemini*, despite having a high AUC score similar to *Asteria*, its MRR performance is relatively poor compared to both *Asteria* and Asteria-Pro. This indicates that evaluating BCSD approaches in a single experiment setting, such as Task-C, may not provide a comprehensive understanding of their real-world applicability and behavior. Task-V, which focuses on bug (vulnerability) search, simulates the scenario of identifying homologous functions from a pool of candidate functions. In this task, the ability to accurately rank and identify homologous functions becomes crucial. While ROC curves and AUC scores provide information about the ability to discriminate between homologous and non-homologous functions, they may not reflect the performance in ranking and retrieving homologous functions accurately. Therefore, it is important to consider multiple evaluation tasks, such as Task-C and Task-V, to assess the overall performance and effectiveness of BCSD approaches. The results obtained from different tasks can provide a more comprehensive understanding of the strengths and limitations of each method and their suitability for real-world applications.

```
1  CK_RV proxy_C_DigestInit(...){
2      /*
3      Variable Initialization.
4      */
5      v5 = (Proxy *)self[1].
           C_GetSlotInfo;
6      v7 = handle;
7      result =
           map_session_to_real(v5
           ,&v7,&map,V3);
8      if (!result)
9          result = map.funcs->C_
               DigestInit(v7,
               mechanism);
10     return result;
11 }
```

```
1  CK_RV proxy_C_DigestKey(...){
2      /*
3      Variable Initialization.
4      */
5      v5 = (Proxy *)self[1].
           C_GetSlotInfo;
6      v7 = handle;
7      result =
           map_session_to_real(v5
           ,&v7,&map,V3);
8      if (!result)
9          result = map.funcs->C_
               DigestKey(v7,
               mechanism);
10     return result;
11 }
```

Fig. 11. Two proxy functions with only distinctions highlighted in red.

False-positive Analysis. The false-positive outcomes of Asteria-Pro can be attributed to two primary causes:

Cause 1: Similar Syntactic Structures of Proxy Functions—Proxy functions exhibit similar syntactic structures, which can lead to similar semantics. This can make it challenging for Asteria-Pro to differentiate between proxy functions, since their semantics are alike. Figure 11 provides an illustration of two proxy functions that differ only on line 9. Due to their similar semantics, it becomes difficult to confirm the actual callees, especially when symbols are lacking or when indirect jump tables are involved.

Cause 2: Compiler-Specific Intrinsic Functions—Compilers for different architectures utilize various intrinsic functions, which substitute libc function calls with optimized assembly instructions. For example, the gcc-X86 compiler may replace the memcpy function with several memory operation instructions that are specific to the architecture. As a result, the memcpy function may be absent from the list of callee functions used by Asteria's filtering and re-ranking modules. This lack of complete callee function information can lead to a loss of precision in the scoring calculation.

Both causes contribute to the false-positive outcomes in Asteria-Pro, highlighting the challenges in accurately detecting function similarity across different architectures and handling variations in compilers' optimization techniques. Addressing these causes and improving the precision of function similarity detection in such scenarios is an ongoing area of research and development in the field of BCSD.

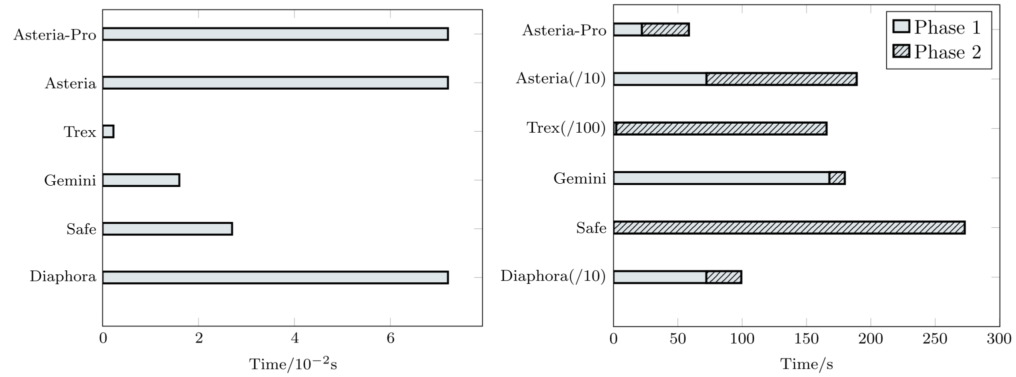
8.5.2 Cross-compiler Evaluation. In the cross-compiler evaluation, we conducted experiments using three different compilers: gcc, icc (Version 2021.1 Build 20201112_000000), and clang (10.0.0), all for the x86 architecture. The evaluation results are presented in Table 4. We evaluated the performance of different methods using metrics such as MRR and Recall in the three cross-compiler settings: gcc-clang, gcc-icc, and clang-icc. The average values for all three settings are also provided in the last column of the table. Our new tool, Asteria-Pro, consistently outperforms the baseline methods by significant margins across all three compiler combinations. Compared to Asteria, Asteria-Pro achieves an average improvement of 47.6% in MRR and Recall, demonstrating its superior performance. The improvements compared to other baseline tools such as Trex, Gemini, Safe, and Diaphora are even more substantial, with average improvements of 596.6%,

Table 4. Cross-Compiler Evaluation Results

Metrics	Methods	gcc-clang	gcc-icc	clang-icc	Avg.
MRR	**Asteria-Pro**	**0.755**	**0.560**	**0.564**	**0.626**
	Asteria	0.624	0.319	0.328	0.424
	Trex	0.148	0.063	0.093	0.101
	Gemini	0.234	0.121	0.080	0.145
	Safe	0.058	0.187	0.076	0.107
	Diaphora	0.727	0.370	0.384	0.494
Recall@Top-1	**Asteria-Pro**	**0.694**	**0.479**	**0.486**	**0.553**
	Asteria	0.541	0.244	0.256	0.347
	Trex	0.099	0.031	0.040	0.057
	Gemini	0.164	0.079	0.048	0.097
	Safe	0.027	0.152	0.031	0.070
	Diaphora	0.662	0.312	0.330	0.435
Recall@Top-10	**Asteria-Pro**	**0.864**	**0.706**	**0.711**	**0.760**
	Asteria	0.783	0.466	0.469	0.573
	Trex	0.257	0.124	0.075	0.152
	Gemini	0.368	0.196	0.137	0.234
	Safe	0.101	0.239	0.149	0.163
	Diaphora	0.844	0.476	0.497	0.606

331.7%, 485.0%, and 26.7%, respectively. It is worth noting that Diaphora achieves surprisingly high precision in the gcc-clang setting, particularly compared to the cross-architecture setting. This may be attributed to the fact that compilers gcc and clang employ similar compilation optimization algorithms, resulting in similar assembly code and **abstract syntax tree (AST)** structures. However, since Asteria is not trained on a cross-compiler dataset, it exhibits relatively lower precision compared to Diaphora. Although the precision performances of the methods vary in different compiler combination settings, a consistent trend can be observed. Specifically, higher precision is observed in the gcc-clang setting, while lower precision is observed in the gcc-icc and clang-icc settings, except for Safe. This can be attributed to the fact that the icc compiler employs more aggressive code optimizations, resulting in dissimilar assembly code compared to the other compilers. Overall, the results of the cross-compiler evaluation demonstrate the effectiveness of Asteria-Pro in detecting function similarity across different compilers and highlight its superior performance compared to the baseline methods.

Answer to RQ1: Asteria-Pro Demonstrates superior accuracy in both Task-C and Task-V. In Task-C, dominant model in Asteria-Pro demonstrates the best classification performance by producing the highest AUC (0.997). Regarding Task-V, Asteria-Pro outperforms other baseline methods by a large margin in MRR, Recall@Top-1, and Recall@Top-10. In particular, Asteria-Pro has 172%, 236%, 147% higher MRR, Recall@Top-1, Recall@Top-10 than the best baseline methods. Compared with *Asteria*, Asteria-Pro manages to improve it for Task-V with 23.71% higher MRR, 36.4% higher Recall@Top-1, and 7.2% higher Recall@Top-10. In a cross-compiler setting, Asteria-Pro continues to outperform baseline methods by a significant margin. All BCSD methods exhibit higher accuracy in the gcc-clang pairing compared to the gcc-icc or clang-icc pairings, likely because the icc compiler tends to emit highly optimized assembly code.



(a) Average Feature Extraction Time for One Function

(b) Average Time for One Search

Fig. 12. Performance comparison of all methods on task-V.

8.6 Performance Comparison (RQ2)

In this section, the detection time of function similarity for all baseline approaches and Asteria-Pro are measured. Since the DK-based prefiltration and DK-based re-ranking modules are intended to enhance performance in Task-V, we only count the timings in Task-V. In Task-V, given a source function, methods extract the function features of source and all candidate functions, which is referred to as *phase 1*. Next, the extracted function features are subjected to feature encoding and encoding similarity computation to determine the final similarities, which is referred to as *phase 2*.

As shown in Figure 12(a), we calculate the average feature extraction time for each function. The x-axis depicts extraction time, while the y-axis lists various extraction methods. During feature extraction for one single function, Asteria-Pro, Asteria, and Diaphora all execute the same operation (i.e., AST extraction), resulting in the same average extraction time. Since AST extraction requires binary disassembly and decompilation, it requires the most time compared to other methods. Trex requires the least amount of time for feature extraction, which is less than 0.001 s per function, as code disassembly is the only time-consuming activity.

Figure 12(b) illustrates the average duration of a single search procedure for various methods. The phases 1 and 2 of a single search procedure are denoted by distinct signs. Due to its efficient filtering mechanism, Asteria-Pro requires the least amount of time (58.593 s) to complete a search. Due to its extensive pre-training model encoding computation, Trex is the most time-consuming algorithm. Asteria-Pro cuts search time by 96.90%, or 1831.36 s, compared to Asteria (1889.96 s).

Answer to RQ2: Asteria-Pro Costs the least average time to accomplish task-V. Compared with Asteria, Asteria-Pro cuts search time by 96.90% by introducing the filtering module.

8.7 Ablation Experiments (RQ3)

To demonstrate the progresses made by different modules of DK-based filtration and DK-based re-ranking, we conduct ablation experiments by evaluating the different module combinations in Asteria-Pro. The module combinations are *Pre-filtering + Asteria* and *Asteria + Re-ranking*. The two module combinations performs Task-V and the results are shown in Table 5. For *Asteria + Re-ranking*, the top 20 similarity detection are re-ranked by the Re-ranking module.

Table 5. Accuracy of Different Module Combination

Module Combination	**MRR**	**Recall@Top-1**	**Recall@Top-10**	**Average Time (s)**
Pre-filtering + Asteria	0.824	0.764	0.929	57.8
Asteria + Reranking	0.882	0.864	0.910	1889.8

Table 6. Integration of Pre-Filtration and Re-Ranking Modules with Alternative BCSD Techniques

Methods	Trex	**Trex-I**	Gemini	**Gemini-I**	Safe	**Safe-I**	Diaphora	**Diaphora-I**	Asteria	**Asteria-Pro**
MRR	0.310	**0.547**	0.334	**0.775**	0.015	**0.533**	0.020	**0.772**	0.734	**0.908**
Recall@Top-1	0.192	**0.377**	0.266	**0.722**	0.003	**0.484**	0.018	**0.711**	0.657	**0.896**
Recall@Top-10	0.579	**0.881**	0.488	**0.865**	0.015	**0.603**	0.027	**0.878**	0.877	**0.940**

We denote the integration with BCSD method X as X-I.

8.7.1 Filtration Improvement. Compared to Asteria, the integration of pre-filtering improves MRR, Recall@Top-1, and Recall@Top-10 by 12.26%, 16.29%, and 5.93%, respectively. In terms of efficiency, it cuts search time by 96.94%. The *Pre-filtering + Asteria* combination performs better than *Asteria + Re-ranking* in terms of *Recall@Top-10* and time consumption. It generates a greater *Recall@Top-10*, because it filters out a large proportion of highly rated non-homologous functions.

8.7.2 Re-ranking Improvement. Compared to Asteria, the integration of Re-ranking module improves MRR, Recall@Top-1, and Recall@Top-10 by 20.16%, 31.51%, and 3.76%, respectively. In terms of efficiency, it costs average additional 0.13 s for re-ranking, which is negligible. Compared to *Pre-filtering + Asteria*, re-ranking module contributes to an increase in *MRR* and *Recall@Top-1* by enhancing the rank of homologous functions.

8.7.3 Embedding Baseline Methods. We demonstrate the generalizability of innovative BCSD enhancement framework, by integrating our two new components, pre-filtering and re-ranking, with other baseline BCSD methods. Specifically, we apply all the baseline methods to compute the similarity scores between the remaining functions and the source function after pre-filtering. Subsequently, we rank the remaining functions in descending order based on their similarity scores and select the top 50 functions for re-ranking. The final similarity score is obtained by combining the re-ranking score and the score generated by the baseline methods. Using the final similarity score, we determine the rankings of the top 50 candidate functions and calculate three metrics, namely, MRR, Recall@Top-1, and Recall@Top-10. Table 6 presents a comparison of the original and integrated versions of the baseline methods, with the baseline method names listed in the first row and their corresponding integrated versions in the next column, such as **Trex-I** for the integrated version of Trex. The second to fourth rows provide the values of different metrics, namely, MRR, Recall@Top-1, and Recall@Top-10.

The accuracy of the baseline methods is significantly improved by the addition of our two components, with **Diaphora-I** in particular showing a substantial increase in MRR from 0.02 to 0.772. We manually analyzed the outputs of Diaphora and **Diaphora-I** to understand the reason for the improved ranking of homologous functions. We found that while Diaphora tends to assign high similarity scores to homologous functions, it also assigns high scores to numerous non-homologous functions, which lowers the ranking of homologous functions. Specifically, we found that the average score difference between the highest score (i.e., score of top 1) and the score assigned to the homologous function is only 0.11. By incorporating reranking scores into the final scores, **Diaphora-I** places a higher emphasis on homologous functions, resulting in improved ranking. If homologous functions are present in the top 50 before re-ranking, then they are mostly ranked at the top. **Safe-I** also shows improved accuracy, although not as substantial as

Table 7. Capacity to Filter of Various Filtering Thresholds

T_{NCL}	# Filtered Function	Recall
0.1	9666.7	0.9813
0.2	9734.1	0.9808
0.3	9777.4	0.9791
0.4	9793.5	0.9773
0.5	9805.5	0.9737

Diaphora-I, as Safe tends to rank homologous functions outside the top 50, reducing the impact of reranking.

The enhancement framework also effectively enhances the accuracy of Asteria. Specifically, Asteria-Pro achieves very high MRR and Recall@Top-1, with a notable margin compared to other integrated versions of baseline methods. The high accuracy of Asteria-Pro enables it to generate more reliable search results, which can significantly reduce the efforts required for vulnerability confirmation when applied to bug search tasks.

Answer to RQ3: The filtering significantly cuts the calculation time by 96.94%, and increase precision slightly. Re-ranking improves MRR, Recall@Top-1, and Recall@Top-10 by 20.16%, 31.51%, and 3.76%, respectively, with negligible time costs. Our enhancement framework, which embeds the BCSD method with pre-filtering and re-ranking modules, has demonstrated a significant improvement in the accuracy of other baseline methods as well.

8.8 Configurable Parameter Sensitivity Analysis (RQ4)

Asteria-Pro has two sets of configurable parameters: The filtering threshold T_{NCL} in the pre-filtering algorithm, and the weight values in Equation (15) for the final precision score. In our evaluation, we analyze the impact of these parameters on Asteria-Pro's performance by testing different values of T_{NCL} for pre-filtering and varying weight combinations for the final precision score.

8.8.1 Different Filtering Threshold. In Algorithm 1, the threshold T_{NCL} determines the number of functions that are filtered out. We evaluate the efficacy of the filtering module by utilizing various T_{NCL} values, and the results are presented in Table 7. The threshold values range from 0.1 to 0.5 in the first column, where a higher threshold value suggests a more severe selection of the similarity function. The second column indicates the number of functions omitted by the filter, while the third column displays the recall rate in the filteration results. As the threshold value increases, the recall rate declines and the number of filtered-out functions grows. We use 0.1 as our threshold value for two key reasons: (a) The high recall rate of filtering results is advantageous for subsequent homologous function detection, and (b) there is no significant difference in the number of functions that are filtered out.

8.8.2 Weights in Re-ranking. We conducted a sensitivity analysis of different weight values in Equation (16). The evaluation results are presented in Table 8. The first two columns display the combinations of two distinct weights, α and β, from Equation (16). The last three columns give the values of various metrics, including MRR, Recall@Top-1, and Recall@Top-10.

We did not test all weight combinations as the accuracy metrics consistently decreased with an increase in α. As shown in the table, when $\alpha = 0.1$ and $\beta = 0.9$, Asteria-Pro has the best accuracy.

Table 8. Accuracy of Asteria-Pro with Various Weight Combinations in Equation (15)

α	β	MRR	Recall@Top-1	Recall@Top-10
0.0	1.0	0.901	0.890	0.930
0.1	**0.9**	**0.908**	**0.896**	**0.940**
0.2	0.8	0.905	0.893	0.938
0.3	0.7	0.902	0.890	0.937
0.4	0.6	0.900	0.889	0.935
0.5	0.5	0.899	0.887	0.934
1.0	0.0	0.824	0.764	0.929

Table 9. Vulnerability Dataset

Software	CVE #	Disclosure Years	Vulnerable Version Range
OpenSSL	22	2013~2016	[1.0.0, 1.0.0s]
			[1.0.1, 1.0.1t]
			[1.0.2, 1.0.2h]
Busybox	10	2015~2019	[0.38, 1.29.3]
Dnsmasq	14	20{15,17,20,21}	[2.42, 2.82], 2.86
Lighttpd	10	20{08,10,11,13,14,15,18}	[1.3.11, 1.4.49]
Tcpdump	36	2017	[3.5.1, 4.9.1]

The last column sets β to 0.0, meaning the re-ranking score is not included in the final similarity calculation. Therefore, the results are consistent with the combination "Pre-filtering + Asteria" in Section 8.7.

Answer to RQ4: The pre-filtering module performs best with a filtering threshold value of 0.1. This threshold allows Asteria-Pro to filter out 96.67% of non-homologous functions per search, while achieving a recall rate of 98.13%. Regarding the re-ranking score, we found that non-zero weight values have a relatively small effect on the final accuracy of Asteria-Pro. However, incorporating the re-ranking score significantly improves the precision of the tool.

8.9 Real World Bug Search (RQ5)

To assess the efficacy of Asteria-Pro, we conduct a massive real-world search for bugs. To accomplish this, we obtain firmware and compile vulnerability functions to create a firmware dataset and a vulnerability dataset. Utilizing vulnerability dataset, we then apply Asteria-Pro to detect vulnerable functions in the firmware dataset. To confirm vulnerability in the resulting functions, we design a semi-automatic method for identifying vulnerable functions. Through a comprehensive analysis of the results, we discover intriguing facts regarding vulnerabilities existed in IoT firmware.

8.9.1 Dataset Construction. In contrast to our prior work, we expand both the vulnerability dataset and the firmware dataset for a comprehensive vulnerability detection evaluation.

Vulnerability Dataset. The prior vulnerability dataset of seven CVE functions is enlarged to **90**, as shown in Table 9. Vulnerability information is primarily gathered from the NVD website [11]. As shown in the first column, the vulnerabilities are collected from widely used

Table 10. Firmware Dataset and Its Software Statistics

Vendor	Firmware Dataset			Software Statistics				
	Firmware #	Binary #	Function #	OpenSSL	Busybox	Dnsmasq	Lighttpd	Tcpdump
Netgear	548	984	2,627,143	349	512	85	14	24
TP-Link	95	177	427,795	66	90	11	3	7
Hikvision	90	92	279,299	55	35	0	0	2
Cisco	29	66	60,396	23	26	10	5	2
Schneider	10	20	31,228	7	9	2	2	0
Dajiang	7	16	57,275	7	7	1	0	1
All	779	1,355	3,483,136	507	679	109	24	36

# denotes number.

open-source software in IoT firmware, including OpenSSL, Busybox, Dnsmasq, Lighttpd, and Tcpdump. In the second column, the number of software vulnerabilities is listed. In the third column, the timeframe or specific years of the disclosure of the vulnerability are listed. The final column describes the software version ranges affected by vulnerabilities. Note that the version ranges are obtained by calculating the union of all versions mentioned in the vulnerability reports. As a result, Asteria-Pro is expected to generate vulnerability detection results for all software versions falling within the specified ranges.

Firmware Dataset. We download as much firmware from six popular IoT vendors as we could, consisting of Netgear [12], Tp-Link [15], Hikvision [9], Cisco [4], Schneider [14], and Dajiang [6] as shown in first column of Table 10. These firmware are utilized by routers, IP cameras, switches, and drones, all of which play essential parts in our life. The second column shows the firmware numbers, which range from 7 to 548. The third and fourth columns give numbers of binaries and functions after unpacking firmware by using binwalk. Note that the binary number is the number of software selected to be in the vulnerability dataset. The fifth column to ninth column gives the five software numbers in all firmware vendors. OpenSSL and Busybox are widely integrated in these IoT firmware as their numbers are close to those of the firmware. Through querying their official websites for device type information, we find that the majority of Hikvision vendor firmware is for IP cameras, whereas Cisco vendor firmware is for routers. In particular, IP camera firmware incorporates less software than router firmware, because routers offer more functionality. For example, the firmware of the Cisco *RV340* router includes OpenSSL, Tcpdump, Busybox, and Dnsmasq, whereas the majority of IP camera firmware only include OpenSSL. Similarly, the majority of the firmware of Netgear and Tp-Link consists of routers, while Schneider and Dajiang'firmware include specialized devices such as Ethernet Radio and Stabilizers.

8.9.2 Large Scale Bug Search. Asteria-Pro is employed to identify vulnerable homologous functions among 3,483,136 firmware functions by referencing 90 functions from the vulnerability dataset. Specifically, to expedite the detection process, vulnerability detection is restricted to the same software between firmware dataset and vulnerability dataset. For instance, the vulnerable functions disclosed in OpenSSL are utilized to detect vulnerable homologous functions in OpenSSL in the firmware dataset. For each software S, we first extract features (i.e., ASTs and call graphs) of all functions in firmware dataset and vulnerable functions in vulnerability dataset. For each vulnerability disclosed in S, the pre-filtration module uses the call graph to filter out non-homologous functions, followed by the Tree-LSTM model encoding all remaining functions as vectors. Asteria-Pro then computes the AST similarity between the vulnerable function vectors and the firmawre function vector. Asteria-Pro computes reranking scores based on the top 20 of AST similarities based on similarity scores, since the evaluation demonstrates a very high recall in the top 20. As a final step, Asteria-Pro generates 20 candidate homologous functions for each

Table 11. Numbers of Vulnerable Functions, Software, Firmware in Confirmation Results

vendor	Vulnerable Function #						Vulnerable Software #						Vulnerable
	OpenSSL	Busybox	Dnsmasq	Lighttpd	Tcpdump	All	OpenSSL	Busybox	Dnsmasq	Lighttpd	Tcpdump	All	Firmware #
Netgear	367	0	31	0	26	424	133	0	7	0	10	150	145 (26.46%)
TP-Link	394	9	0	2	5	410	36	3	0	2	5	46	36 (37.89%)
Hikvision	553	0	0	0	12	565	52	0	0	0	1	53	53 (58.89%)
Cisco	0	0	0	0	2	2	0	0	0	0	2	2	2 (6.90%)
Schneider	10	0	0	0	0	10	1	0	0	0	0	1	1 (10.00%)
Dajiang	70	0	0	0	1	71	7	0	0	0	1	8	7 (100.00%)
Total	1,394	9	31	2	46	1,482	229	3	7	2	19	260	244 (31.32%)

S as a **bug search result** for each vulnerability. To further refine the bug search results, we compute the average similarity score of homologous functions in Section 8.5 and use it to eliminate non-vulnerable functions. In particular, the average similarity score of 0.89 is used to eliminate 3,987 of 5,604 results. We perform heuristic confirmation of vulnerability for the remaining 1,617 results.

Vulnerability Confirmation Method. We devise a semi-automatic method for confirming the actual vulnerable functions from the candidate homologous functions. The method makes use of the symbols and string literals within the firmware binaries of the target. Specifically, we use unique regular expressions to match version strings for each software and to extract function symbols from the software. The method is then comprised of two distinct operations that correspond to two distinct vulnerable circumstances VC_1, VC_2.

- VC_1**.** In this circumstances, the target binary contains version string (e.g., "OpenSSL 1.0.0a") and the symbol of target function is not removed.
- VC_2**.** Target binary contains version strings whereas the symbol of vulnerable homologous function is removed.

The versions of software listed in Table 9 are easy to extract using version strings [27]. The descriptions of the two confirmation operations CO_1 and CO_2 are as follows:

- CO_1**.** For VC_1, we confirm the vulnerable function based on the version and name of the target software. In particular, a vulnerable function is confirmed when the following two conditions are met: (1) Software version is in vulnerable version range, (2) the vulnerable function name retains after elimination with average similarity score.
- CO_2**.** For VC_2, if the software versions are in the range of vulnerable versions, then we manually compare the code between the CVE functions and remaining functions to confirm the vulnerability.

Results Analysis. In Table 11, We tally the number of vulnerable functions, software, and firmware upon vulnerability confirmation. The first column contains the names of different vendors. The second through sixth columns show the amount of vulnerable functions in various software, while the seventh column indicates the total number of vulnerable functions across all vendors. The eighth through twelfth columns display the amount of vulnerable software binaries in various software, while the thirteenth column provides the total number of vulnerable software binaries. According to the seventh column of Table 10, there are a total of **1,482** vulnerable functions. 1456 are confirmed by CO_1, whereas 26 are confirmed by CO_2. For a total of 1,456 CO_1 vulnerable functions, 1,377 vulnerable functions rank first and 79 vulnerable functions rank second. CO_2 is performed on 47 detection results, of which 26 are confirmed. the 21 unconfirmed detection results can be attributed to two reasons. First, 18 of them were due to the fact that the target binaries detected did not contain any of the target vulnerable functions. For example, we were unable to detect the vulnerable function "EVP_EncryptUpdate" of CVE-2016-2106 from the "libssl.so" library

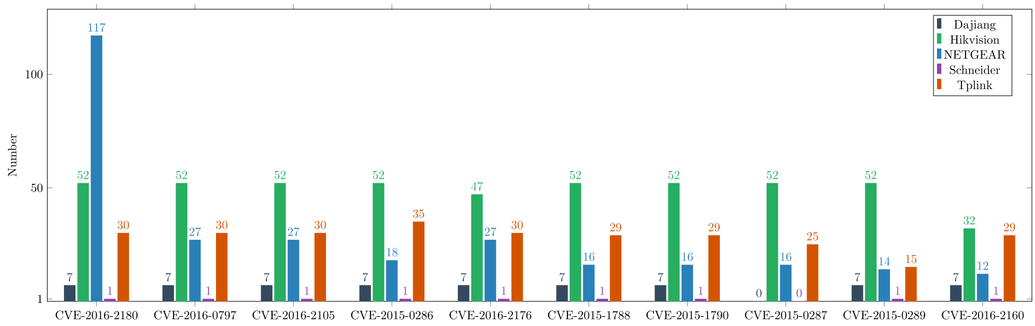
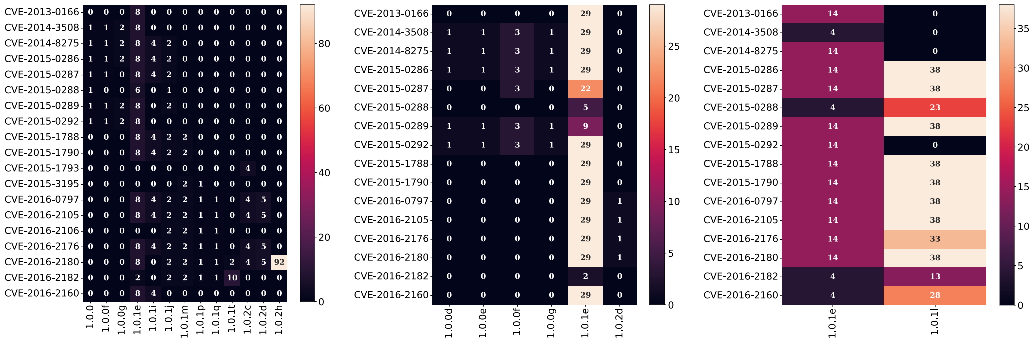


Fig. 13. Number of vulnerable functions detected from five vendors.

of OpenSSL, since it exists in the "libcrypto.so" library. Second, three of the unconfirmed results were ranked in the top 20 but were subsequently filtered out by the similarity threshold used in real world bug search setting. A large proportion of vulnerable functions are found in the OpenSSL software used by the three vendors. The number of vulnerable software is consistent with this circumstance. The final column shows the number of firmware containing at least one vulnerable function, together with its proportion of total firmware. Every Dajiang firmware contains at least one CVE vulnerability, because all OpenSSL components used in firmwareÂăare vulnerable. In addition, Hikvision is detectedÂăto have a largeÂăproportion of vulnerable firmware (58.89%). To inspect the CVE vulnerable function distribution, we plot the top 10 CVEs and their distributions in five vendors except Cisco in Figure 13, since Cisco takes additional two CVEs.

- **Top 10 CVE Analysis.** Figure 13 demonstrates the top 10 CVE distribution in various vendors. The total number of discovered CVE vulnerabilities decreases from left to right along the x-axis. Except for CVE-2015-0287, all of the top 10 CVE vulnerabilities are discovered in every *Dajiang* firmware. This is because Dajiang utilizes an outdated version of OpenSSL 1.0.1h that contains numerous vulnerable functions [13]. Although Hikvision firmware has the third largest number of firmware, it has the most vulnerable functions in our experiment settings. The reason for this is that Hikvision firmware heavily uses OpenSSL-1.0.1e (184) and OpenSSL-1.0.1l (401) versions, both of which contain a large number of vulnerabilities. **Finding:** Since they typically adopt the same vulnerable software version, it is highly plausible that firmware from the same vendor and released at the same period contains identical vulnerabilities. Security analysts can quickly narrow down the vulnerability analysis based on the firmware release date.
- **CVE and Version Analysis.** Figure 14 Depicts the distribution of vulnerable OpenSSL versions for various CVEs from various vendors. Where the x-axis represents the version and the y-axis represents the CVE ID associated with the vulnerability. Each square in each subfigure indicates the number of OpenSSL versions that are vulnerable and contain the corresponding CVE along the y-axis. The number is greater the lighter the red colour. The left subfigure demonstrates that OpenSSL 1.0.2h is widely used by Netgear, resulting in a significant number of CVE-2016-2180 vulnerabilities (92). Additionally, OpenSSL version 1.0.1e exposes the majority of CVEs listed on the y-axis, which may increase the device's attack surface. The TP-Link firmware incorporates OpenSSL version 1.0.1e, resulting in brighter hues. Hikvision firmware utilizes versions 1.0.1e and 1.0.1l, which are vulnerable to a number of CVEs. Comparing vulnerability distribution in OpenSSL version 1.0.1e among different vendors reveals inconsistencies in the existence of vulnerabilities. For instance, CVE-2016-2106 is present in OpenSSL 1.0.1e from Hikvision but not from Netgear and



(a) Netgear (b) TP-Link (c) Hikvision

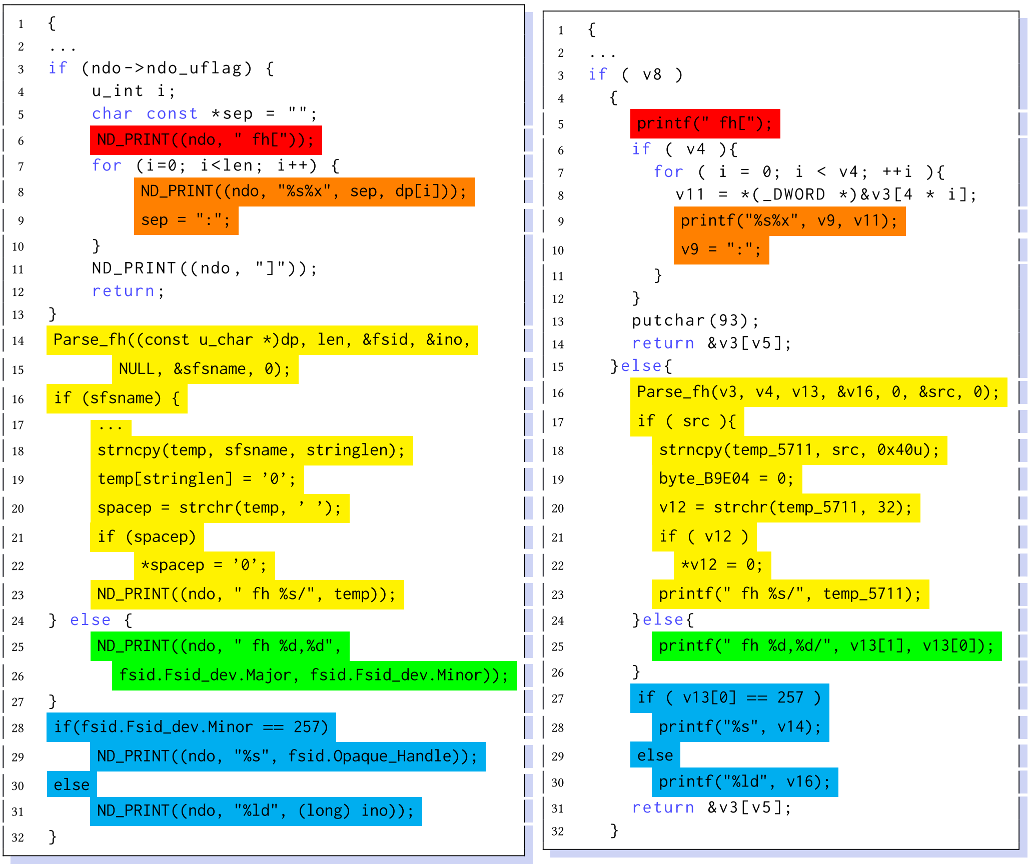
Fig. 14. Distribution of CVEs for different OpenSSL versions in vendors (a) Netgear, (b) TP-Link, and (c) Hikvision.

TP-Link. **Finding:** Despite using the same version of software, various vendor firmware behaves differently in terms of vulnerability, since they can tailor the software to the device's specific capabilities.

- **CVE-2016-2180 Analysis.** The CVE-2016-2180, which is a remote *Denial-of-Service* flaw caused by received forged time-stamp file, impacting OpenSSL 1.0.1 through 1.0.2h, exists in 207 firmware. NETGEAR is responsible for 117 of these, as it deploys 92 OpenSSL 1.0.2h out of a total of 548 firmware. NETGEAR incorporated an extra nine OpenSSL 1.0.2 series software and sixteen OpenSSL 1.0.1 series software. The vulnerable version 1.0.2h was released in May 2016, and by comparing their timestamps, we determined that OpenSSL 1.0.2h was integrated into firmware between 2016 and 2019. **Finding:** Even after their vulnerabilities have been discovered, the vulnerable versions of software continue to be used for firmware development.

Based on the confirmation results, Asteria-Pro manages to detect 1,482 vulnerable functions out of 1,617 bug search results, indicating that Asteria-Pro achieves a high vulnerability detection precision of **91.65%** under our experiment settings. By randomly selecting 1,000 of 5,604 bug search results, we manually validate the existence of vulnerabilities in software binaries to calculate the recall. Among 1,000 bug search results, 205 target functions are confirmed to be vulnerable by checking software versions and the vulnerable functions. Targeting 205 vulnerable functions, Asteria-Pro detects 53 of them, representing a recall rate of 25.85%.

Finding Inlined Vulnerable Code. During the analysis of mismatched cases, in which the target homologous functions are not in the top ranking position, we observe that the top-ranked functions contain the same vulnerable code. We use CVE-2017-13001 as an illustration of inlined vulnerable code detection. CVE-2017-13001 is a buffer over-read vulnerability in the Tcpdump *nfs_printfh* function prior to version 4.9.2. After a confirmation operation CO_2, Asteria-Pro reports a single function, *parsefh* as being vulnerable. We manually compare the decompiled code of the *parsefh* function to the source code of *nfs_printfh* in tcpdump version 4.9.1 (i.e., vulnerable version). Figure 15 demonstrates that the source code of *nfs_printfh* (on the left) and the partial code of *parsefh* (on the right) are consistent. We designate codes with apparently identical semantics with distinct backdrop hues. In other words, during compilation, function *nfs_printfh* is inlined into function *parsefh*. As a result, the function *parsefh* contains CVE-2017-13001 vulnerable code, and Asteria-Pro manages to identify the inlined vulnerable code. Asteria-Pro



```c
{
...
if (ndo->ndo_uflag) {
    u_int i;
    char const *sep = "";
    ND_PRINT((ndo, " fh["));
    for (i=0; i<len; i++) {
        ND_PRINT((ndo, "%s%x", sep, dp[i]));
        sep = ":";
    }
    ND_PRINT((ndo, "]"));
    return;
}
Parse_fh((const u_char *)dp, len, &fsid, &ino,
         NULL, &sfsname, 0);
if (sfsname) {
    ...
    strncpy(temp, sfsname, stringlen);
    temp[stringlen] = '0';
    spacep = strchr(temp, ' ');
    if (spacep)
        *spacep = '0';
    ND_PRINT((ndo, " fh %s/", temp));
} else {
    ND_PRINT((ndo, " fh %d,%d",
        fsid.Fsid_dev.Major, fsid.Fsid_dev.Minor));
}
if(fsid.Fsid_dev.Minor == 257)
    ND_PRINT((ndo, "%s", fsid.Opaque_Handle));
else
    ND_PRINT((ndo, "%ld", (long) ino));
}
```

(a) Source Code of Vulnerable Function.

```c
{
...
if ( v8 )
  {
    printf(" fh[");
    if ( v4 ){
      for ( i = 0; i < v4; ++i ){
        v11 = *(_DWORD *)&v3[4 * i];
        printf("%s%x", v9, v11);
        v9 = ":";
      }
    }
    putchar(93);
    return &v3[v5];
  }else{
    Parse_fh(v3, v4, v13, &v16, 0, &src, 0);
    if ( src ){
      strncpy(temp_5711, src, 0x40u);
      byte_B9E04 = 0;
      v12 = strchr(temp_5711, 32);
      if ( v12 )
        *v12 = 0;
      printf(" fh %s/", temp_5711);
    }else{
      printf(" fh %d,%d/", v13[1], v13[0]);
    }
    if ( v13[0] == 257 )
      printf("%s", v14);
    else
      printf("%ld", v16);
    return &v3[v5];
  }
```

(b) Part of Decompiled Code in Detected Function.

Fig. 15. Inlined vulnerable function in detected function. the semantics of the code with the same background color are same.

has detected an additional eight instances of inlined vulnerable code out of 20 functions in vulnerable circumstance VC_2.

The preceding analysis and conclusions are constrained by the dataset we constructed, which offers security analysts some recommendations for the security analysis of firmware.

Answer to RQ5. We employ 90 CVE vulnerabilities to search for bugs in 3,483,136 real firmware functions. Asteria-Pro detects 1,482 vulnerable functions with a high level of precision of 91.65%. In addition, the capability of Asteria-Pro to identify inlined vulnerable code is stated and illustrated in detail. In conclusion, Asteria-Pro generates bug search results with a high degree of confidence, thereby reducing analysis labor by a substantial margin.

9 THREATS TO VALIDATION

Threats to internal validity come from these aspects.

- We use vulnerable version ranges collected from the NVD website to aid vulnerability confirmation in a real-world bug-search experiment. The vulnerable version ranges may be inaccurate, and vulnerabilities may be missed or incorrectly stated. We will conduct additional verification of the susceptible version ranges by confirming the existence of vulnerable code.
- We adopt IDA Pro to decompiling and generate ASTs from the functions in binaries. As pointed by study [47], accurate decompiling and binary analysis is not easy. The errors in AST generation may affect the AST similarity calculation and further affect the results.

Threats to external validity rise from following issue.

- In practice, firmware binaries may be compiled with distinct compiler (e.g., clang), compiler version, and optimization level for special-purpose compilation. Different compilation configurations alter the AST structures and call graphs, sometimes leading in lower scores for homologous functions.

10 DISCUSSION

10.1 How does the re-ranking module solve the function inline issues?

Inlining functions is a common optimization technique used by compilers to improve the performance of code execution. The decision of whether or not to inline a function is based on various factors, including the size of the function, the frequency of function calls, and the complexity of the code. In general, inlining smaller functions tends to be more beneficial than inlining larger functions.

When smaller functions being inlined, the re-ranking module in Asteria-Pro will be capable of handling inline issues by considering both function similarities and the match of callee relational structures. Specifically, the module matches all callee functions between the source function and target functions, which allows for high similarity even if one callee function is inlined to the target function. This is because the target function can still maintain a relatively high similarity with the source function even after incorporating inlined code, and the un-inlined callee functions still contribute to the final similarity score. As a result, Asteria-Pro exhibits high metric values (i.e., recall and MRR) in our evaluation based on the contribution of the target function code and all its callee functions. Through manual analysis of search results in real-world bug detection, we have also demonstrated that Asteria-Pro is capable of finding homologous vulnerable functions that contain inlined function code.

10.2 What is the Design Difference between Pre-Filtering, Re-Ranking and SCA Tools

Some **software composition analysis (SCA)** tools have adopted similar feature when comparing to the pre-filtering and re-ranking module of Asteria-Pro. For example, Modx [69] matches string literals and whole call graph between two libraries, and LibDB [60] adopts string literals and exported function names to measure the similarity of libraries. The usages of string literals and call graphs are quite straightforward.

However, we would like to highlight some conceptual-level differences between our approach and these prior works. While the use of string literals and call graphs is indeed straightforward, it can be challenging to apply them to function matching, particularly when functions lack string literals or are leaf nodes in call graphs. Additionally, callee functions of a target function may not be exported functions, meaning that function names are removed and cannot be used for matching.

To address these challenges, our approach differs from SCA methods in several key ways. First, we utilize the local context extracted from call graphs in both the pre-filtering and re-ranking modules to efficiently remove non-homologous functions and confirm homologous ones. Second, we introduce an algorithm called "UpRelation" to utilize caller relations from call graphs in pre-filtering. The algorithm leverages the *genealogist* of parent nodes to identify potential homologous functions. It achieves this by matching the *genealogist* of parent nodes and retaining the child nodes of the matched parent nodes. This approach is particularly useful when the target function is a leaf node in the call graph and does not contain any string literals. Last, our re-ranking module considers both structural and semantic similarities of functions, resulting in more accurate ranking of homologous functions. Specifically, the re-ranking module uses Asteria to calculate similarities when callee functions are not exported functions.

10.3 What Will Asteria-Pro Perform on Cross-Optimization Settings?

Although we did not evaluate the performance of Asteria-Pro in cross-optimization settings, it is worth discussing the potential impact of such settings on the performance of our method. Cross-optimization refers to the situation where the training and testing sets are compiled with different optimization settings. This is a common scenario in practice as different developers may use different optimization flags, or the same developer may use different optimization levels for different releases. Previous studies have shown that cross-optimization can significantly affect the accuracy of BCSD methods, as the semantic features extracted from the binary code may change depending on the optimization settings. For instance, in a study by Liu et al. [46], the accuracy of a state-of-the-art BCSD method dropped from 95.3% to 46.2% when tested in cross.

In the case of our method, **Asteria-Pro**, which is based on the Tree-LSTM architecture, the impact of cross-optimization on its performance is likely to be substantial. This is because Tree-LSTM model is sensitive to AST structure and summarizes semantics by identifying structure patterns. Therefore, if the source and target functions are compiled with different optimization settings, then the Tree-LSTM may not be capable to summarize the expected semantic from substantial AST structure transformation and thus produce inaccurate results.

Moreover, training our model on cross-optimization settings would require significant computational resources and time, which may not be feasible in practice. Therefore, we have not evaluated our method on cross-optimization settings in this study. Nevertheless, we acknowledge that cross-optimization is an essential consideration for evaluating **Asteria-Pro**'s generalizability, and we encourage future studies to investigate this aspect further.

11 RELATED WORKS

11.1 Feature-based Methods

When considering the similarity of binary functions, the most intuitive way is to utilize the assembly code content to calculate the edit distance for similarity detection between functions. Khoo et al. concatenated consecutive mnemonics from assembly language into the N-grams for similarity calculation [43]. David et al. proposed Trecelet, which concatenates the instructions from adjacent basic blocks in CFGs for similarity calculation [24]. Saebjornsen et al. proposed to normalize/abstract the operands in instructions, e.g., replacing registers such as eax or ebx with string "reg", and conduct edit distance calculation based on normalized instructions [55]. However, binary code similarity detection methods based on disassembly text can not be applied to cross-architecture detection, since the instructions are typically different in different architectures. The works in References [19, 31, 54, 66] utilize cross-architecture statistical features for binary code similarity detection. Eschweiler et al. [31] defined statistical features of functions such as the number of instructions, size of local variables. They utilized these features to calculate and filter out candidate functions. Then they performed a more accurate but time-consuming calculation with the graph isomorphism algorithm based on CFGs. Although this method takes a pre-filtering mechanism, the graph isomorphism algorithm makes similarity calculation extremely slow. To improve the computation efficiency, Feng et al. proposed Genius, which utilizes machine learning techniques for function encoding [33]. Genius uses the statistical features of the CFG proposed in Reference [31] to compose the ACFG. Then it uses a clustering algorithm to calculate the center points of ACFGs and forms a codebook with the center points. Finally, a new ACFG is encoded into a vector by computing the distance with ACFGs in the codebook and the similarity between ACFGs is calculated based on the encoded vectors. But the codebook calculation and ACFG encoding in Genius are still inefficient. Xu et al. proposed *Gemini* based on Genius to encode ACFG with a graph embedding network [65] for improving the accuracy and efficiency. However, the large variance of binary code across different architectures makes it difficult to find architecture-independent features [26].

11.2 Semantic-based Methods

For more accurate detection, semantic-based features are proposed and applied for code similarity detection. The semantic-based features model the code functionality, and are not influenced by different architectures. Khoo et al. applied symbolic execution technique for detecting function similarity [49]. Specifically, they obtained input and output pairs by executing basic blocks of a function. But the input and output pairs can not model the functionality of the whole function accurately. Ming et al. leveraged the deep taint and automatic input generation to find semantic differences in inter-procedural control flows for function similarity detection [52]. Feng et al. proposed to extract conditional formulas as higher-level semantic features from the raw binary code to conduct the binary code similarity detection [32]. In their work, the binary code is lifted into a platform-independent **intermediate representation (IR)**, and the data-flow analysis is conducted to construct formulas from IR. Egele et al. proposed the blanket execution, a novel dynamic equivalence testing primitive that achieves complete coverage by overwriting the intended program logic to perform the function similarity detection [30]. These semantic-based features capture semantic functionalities of a function to reduce the false positives. Pei et al. proposes Trex [53], applying a transfer-learning-based framework to automate learning execution semantics from functions' micro-traces, which are forms of under-constrained dynamic traces. However, the methods above depend heavily on emulation or symbolic execution, which are not suitable for program analysis in large-scale IoT firmware, since the emulation requires peripheral devices [20, 35, 72] and symbolic execution suffers from the problems of path explosion.

11.3 AST in Source Code Analysis

Since the AST can be easily generated from source code, there has been research work proposed to detect source code clone based on AST. Baxter et al. proposed to hash ASTs of functions to buckets and compare the ASTs in the same bucket [17] to find clones. Because the method proposed in Reference [17] is similar to *Diaphora*, which hash ASTs, we only perform a comparative evaluation with *Diaphora*. In addition to the code clone detection, AST is also used in vulnerability extrapolation from source code [67, 68]. To find vulnerable codes that share a similar pattern, Fabian et al. [68] encoded AST into a vector and utilized the latent semantic analysis [25] to decompose the vector to multiple structural pattern vectors and compute the similarity between these pattern vectors. Yusuke Shido et al. proposed an automatic source code summary method with extended Tree-LSTM [57].

12 CONCLUSION

In this work, we present Asteria-Pro, a domain knowledge-enhanced BCSD tool designed to detect homologous vulnerable functions on a broad scale in efficient and accurate manner.

Asteria-Pro introduces domain knowledge before and after deep learning model-based function encoding to eliminate a large proportion of non-homologous functions and score homologous functions higher, separately. The pre-filtering module makes extensive use of function name information prior to function encoding to accelerate the function encoding. The function call structure is utilized by the re-ranking module following function encoding to calibrate the encoding similarity scores. Asteria-Pro is capable of finding homologous functions rapidly and precisely, according to a comprehensive comparison with existing state-of-the-art research. Furthermore, Asteria-Pro manages to find 1,482 vulnerable functions in the real-world firmware bug search experiment with high precision of 91.65%. The search results for CVE-2017-13001 demonstrate Asteria-Pro successfully finds inlined vulnerable code. Asteria-Pro can aid in detecting vulnerabilities from large-scale firmware binaries to mitigate the attach damage on IoT devices.

REFERENCES

[1] Asteria-BCSD. 2022. Asteria-Pro Code Repository. Retrieved from https://github.com/Asteria-BCSD/Asteria-Pro

[2] Microsoft Security. 2022. Binwalk. Retrieved from https://www.refirmlabs.com/binwalk/

[3] Buildroot. 2022. Buildroot Making Embedded Linux Easy. Retrieved from https://buildroot.org

[4] Cisco. 2022. Cisco website. Retrieved from https://www.cisco.com/c/en_hk/index.html

[5] cxxfilt. 2022. cxxfilt. Retrieved from https://pypi.org/project/cxxfilt/

[6] Dajiang. 2022. Dajiang Website. Retrieved from https://www.dji.com/

[7] Diaphora. 2022. Diaphora. Retrieved from https://github.com/joxeankoret/diaphora

[8] Wikibooks. 2022. Function decoration. Retrieved from https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions#Note_on_Name_Decorations

[9] Hikvision. 2022. hikvision website. Retrieved from https://www.hikvision.com/sg/

[10] Hex rays. 2022. IDA Pro. Retrieved from https://www.hex-rays.com/products/ida/index.shtml

[11] NVD. 2022. National Vulnerability Database. Retrieved from https://nvd.nist.gov/

[12] NetGear. 2022. NetGear Download. Retrieved from http://support.netgear.cn/download.asp

[13] OpenSSH. 2022. OpenSSH 1.0.1h Vulnerability Statistics. Retrieved from https://www.cvedetails.com/version/524410/Openssl-Openssl-1.0.1h.html

[14] Schneider. 2022. Schneider website. Retrieved from https://www.se.com/ww/en/

[15] TP-Link. 2022. TP-Link website. Retrieved from https://www.tp-link.com/sg/

[16] Hamid Abdul Basit and Stan Jarzabek. 2005. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 156–165. https://doi.org/10.1145/1095430.1081733

[17] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 368–377.

[18] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Trans. Softw. Eng. Methodol.* 30, 4 (2021), 1–56.

[19] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.

[20] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'16).*

[21] Cristina Cifuentes and K. John Gough. 1995. Decompilation of binary programs. *Softw.: Pract. Exper.* 25, 7 (1995), 811–829.

[22] Ang Cui, Michael Costello, and Salvatore Stolfo. 2013. When firmware modifications attack: A case study of embedded exploitation. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'13).*

[23] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Not.* 53, 2 (2018), 392–404.

[24] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14).* ACM, New York, NY, 349–360. https://doi.org/10.1145/2594291.2594343

[25] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *J. Amer. Soc. Info. Sci.* 41, 6 (1990), 391–407.

[26] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The correctness-security gap in compiler optimization. In *Proceedings of the IEEE Security and Privacy Workshops*. IEEE, 73–87.

[27] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2169–2185.

[28] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects (English version). *SSTIC* 5, 1 (2005), 3.

[29] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. 303–317.

[30] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. USENIX Association, San Diego, CA, 303–317. Retrieved from https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/egele

[31] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*.

[32] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*. 346–359.

[33] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.

[34] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the International Conference on Information and Communications Security*. Springer, 238–255.

[35] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christopher Kruegel et al. 2019. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'19)*. 135–150.

[36] Laune C. Harris and Barton P. Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Comput. Architect. News* 33, 5 (2005), 63–68.

[37] Anfeng He, Chong Luo, Xinmei Tian, and Wenjun Zeng. 2018. A twofold Siamese network for real-time object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4834–4843.

[38] John L. Hennessy and David A. Patterson. 2011. *Computer architecture: A Quantitative Approach*. Elsevier.

[39] Sepp Hochreiter and JÃijrgen Schmidhuber. [n.d.]. Long short-term memory. *Neural Comput.* 9, 8 ([n.d.]), 1735–1780.

[40] Roger A. Horn. 1990. The Hadamard product. In *Proceedings of the Symposium on Applied Math*, Vol. 40. 87–169.

[41] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. 611–620.

[42] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. 2013. Mutantx-s: Scalable malware clustering based on static features. In *Presented as Part of the USENIX Annual Technical Conference (USENIX ATC'13)*. 187–198.

[43] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 329–338.

[44] Amanda Lee and Travis Atkison. 2017. A comparison of fuzzy hashes: Evaluation, guidelines, and future suggestions. In *Proceedings of the SouthEast Conference*. 18–25.

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

[46] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 667–678.

[47] Zhibo Liu and Shuai Wang. 2020. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 475–487.

[48] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 389–400.

[49] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 389–400. https://doi.org/10.1145/2635868.2635900

[50] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2099–2116. Retrieved from https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[51] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329.

[52] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the International Conference on Information Security and Cryptology*. Springer, 92–109.

[53] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. Retrieved from https://arxiv.org/abs/2012.08680

[54] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 709–724.

[55] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. ACM, New York, NY, 117–128. https://doi.org/10.1145/1572272.1572287

[56] Andrew Schulman. 2005. Finding binary clones with opstrings function digests: Part III. *Dr. Dobb's J.* 30, 9 (2005), 64.

[57] Yusuke Shido, Yasuaki Kobayashi, Akihiro Yamamoto, Atsushi Miyamoto, and Tadayuki Matsumura. 2019. Automatic source code summarization with extended tree-LSTM. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'19)*. IEEE, 1–8.

[58] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. Retrieved from https://arXiv:1503.00075

[59] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. Retrieved from http://arxiv.org/abs/1503.00075

[60] Wei Tang, Yanlin Wang, Hongyu Zhang, Shi Han, Ping Luo, and Dongmei Zhang. 2022. LibDB: An effective and efficient framework for detecting third-party libraries in binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 423–434.

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

[62] Zheng Wang, Ken Pierce, and Scott McFarling. 2000. Bmat-A binary matching tool for stale profile propagation. *J. Instruct.-Level Parallel.* 2 (2000), 1–20.

[63] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the correct origin of publicly reported software vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. 3041–3058.

[64] J. Wurm, K. Hoang, O. Arias, A. Sadeghi, and Y. Jin. 2016. Security analysis on consumer and industrial IoT devices. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC'16)*. 519–524.

[65] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 363–376.

[66] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2018. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Trans. Softw. Eng.* 45, 11 (2018), 1125–1149.

[67] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*. 13–13.

[68] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 359–368.

[69] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: Binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405.

[70] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. 2021. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 224–236.

[71] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. 2017. Comparative study of CNN and RNN for natural language processing. Retrieved from https://arXiv:1702.01923

[72] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. 1–16.

[73] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. Retrieved from arXiv:1808.04706.

[74] Mark H. Zweig and Gregory Campbell. 1993. Receiver-operating characteristic (ROC) plots: A fundamental evaluation tool in clinical medicine. *Clin. Chem.* 39, 4 (1993), 561–577.

Received 16 December 2022; revised 15 March 2023; accepted 2 May 2023