Arabian Journal for Science and Engineering (2022) 47:9713–9725
https://doi.org/10.1007/s13369-021-06342-8

RESEARCH ARTICLE-COMPUTER ENGINEERING AND COMPUTER SCIENCE

A Heuristic Local-sensitive Program-Wide Diffing Method for IoT Binary Files

Lu Yu[1] · Yuliang Lu[1] · Yi Shen[1] · Zulie Pan[1] · Hui Huang[1]

Received: 21 April 2021 / Accepted: 24 October 2021 / Published online: 27 November 2021
© The Author(s) 2021

Abstract

Code reuse brings vulnerabilities in third-party library to many Internet of Things (IoT) devices, opening them to attacks such as distributed denial of service. Program-wide binary diffing technology can help detect these vulnerabilities in IoT devices whose source codes are not public. Considering the architectures of IoT devices may vary, we propose a data-aware program-wide diffing method across architectures and optimization levels. We rely on the defined anchor functions and call relationship to expand the comparison scope within the target file, reducing the impact of different architectures on the diffing result. To make the diffing result more accurate, we extract the semantic features that can represent the code by data flow dependence analysis. Earth mover distance is used to calculate the similarity of functions in two files based on semantic features. We implemented a proof-of-concept DAPDiff and compared it with baseline BinDiff, TurboDiff and Asm2vec. Experiments showed the availability and effectiveness of our method across optimization levels and architectures. DAPDiff outperformed BinDiff in recall and precision by 41.4% and 9.2% on average when making diffing between standard third-party library and the real-world firmware files. This proves that DAPDiff can be applicable for the vulnerability detection in IoT devices.

Keywords IoT vulnerability · Program-wide diffing · Feature extraction · Binary vulnerability · Data flow analysis

1 Introduction

With the rapid development of Internet of Things (IoT) technology, the security of IoT devices has attracted more attention than ever. The loose protection of IoT devices and the long-term existence of vulnerabilities make the security problem of Internet of Things more serious. Cui et al. [1] analyze about 4 million IoT devices and find that 540,435 of them had vulnerabilities. Many vulnerabilities in IoT devices are critical ones. Attacks on vulnerabilities in backbone services such as Domain Name Service(DNS) have vast implications. Mirai attack can use a large number of online IoT devices to implement distributed denial of service (DDoS) attacks against online services [2].

Vendors of IoT devices do not make the source code of their firmware images publicly available, making the analysis of IoT firmware files more difficult than that of open source files. In addition, vendors rely heavily on general-purpose packages and integrate such third-party software packages (such as OpenSSL and Busybox) in firmware images. Any vulnerability found in the third-party software packages may open the related devices to an attack. Furthermore, the security analysis of firmware files faces challenges from diverse underlying architectures. The file of IoT device can be compiled in MIPS or ARM architecture. The traditional vulnerability analysis methods of X86/X64 architecture cannot usually be directly applied to the vulnerability analysis of files in MIPS/ARM architecture. A feasible and effective way to detect the vulnerabilities of IoT devices is binary diffing technology across architectures.

Given two binary files without source code, the program-wide diffing method aims to discover and analyze similarities between the functions of the two files. There are a number of mature binary diffing tools, such as state-of-the-art Diaphora [3], BinDiff [4] and TurboDiff [5]. Diaphora compares two binary files according to features including function address, function hash, etc. TurboDiff takes the checksum of basic blocks (a straight-line sequence of code with only one entry point and only one exit) and the number of instructions as features for comparison. These features extracted by Diaphora and TurboDiff may vary due to slightly code

✉ Yuliang Lu
lulu071227@163.com

1 College of Electronic Engineering, National University of Defense Technology, Hefei 230007, China

changes (such as instruction reordering), which affects the accuracy of diffing results. BinDiff performs graph isomorphism detection on function pairs between binary files, and uses the small-primes-products (SPP) [6] to make the graph isomorphism fast in practice. However, the method based on small-primes-products is mainly designed to solve the slightly code changes, such as the instruction reordering, different register allocation and branch inversion. For binary files compiled in different architectures, their binary codes change greatly. Therefore, one challenge in binary diffing across architectures is how to make the diffing process less affected by architecture differences. Since the extracted features are the basis of comparison, the diffing result will be more accurate if the extracted features can better reflect the behavior of the code. The second challenge is to extract which features to represent code. Semantic features [7,8] in form of abstract syntax tree (AST) and control flow graph (CFG), etc., can better represent code behavior, making them good candidates for program diffing. With the wide application of machine learning technology in various fields, researchers begin to investigate the semantic feature extraction methods based on neural network. Ding et al. [9] apply a vector representation method by learning the latent semantic information without preliminary knowledge of X86 assembly code. However, this method can only support single-architecture diffing.

In this paper, we implement a program-wide binary diffing method across architectures and optimization levels to solve the above two challenges. The comparison is implemented based on the anchor functions and the call relationship between functions. We define similar functions in the two comparison files as anchor functions. In the initial anchor function selection process, unique features are used, such as string and integer constants, which remain unchanged for files compiled at different architectures and optimization levels. Based on the anchor functions and the call relationship, more and more functions are added to the comparison set from which new anchor functions are selected. This step-by-step comparison strategy is less affected by architecture and optimization level. In addition, it can divide the complete set of functions to be compared into multiple subsets, reducing the time complexity of diffing. To obtain more semantic features for comparison, we propose a local-data-sensitive feature extraction method inspired by the live variable analysis in data flow analysis technology. This method can record the variable transfer information between functions with call relationship. Then, comparison between functions is implemented by earth mover distance (EMD) [10] based mainly on the semantic features extracted by data flow analysis. According to the comparison results, more anchor functions are obtained. This process is iterated for several times to find more anchor functions to obtain the diffing result.

We implemented a DAPDiff (data-aware program-wide diffing) prototype and evaluated it with several experiments to measure its availability and effectiveness. DAPDiff was compared with the state-of-the-art tools Asm2vec [9], BinDiff [4] and TurboDiff [5] across architectures and optimization levels. The experimental results show that DAPDiff performs well not only across optimization levels but also across architectures. Among the three comparison tools, BinDiff performed better than Asm2vec and TurboDiff. However, DAPDiff outperformed BinDiff by 41.4% and 9.2% in recall and precision on average when making diffing between the standard third-party library and the binary file in real-world firmware. DAPDiff detected CVE vulnerability in 73 files of 93 real-world firmware files, proving the effectiveness of DAPDiff in the detection of vulnerability in IoT devices.

In summary, this paper makes the following contributions:

- We propose DAPDiff, a data-aware program-wide diffing method for binary files in IoT devices. Our comparison expansion strategy relies on features that are independent from architectures, and makes use of the call relationship between functions.
- To make the diffing result more accurate, we explore a local-data-sensitive feature extraction method to extract semantic features for diffing. The features extracted by this approach are combined with the earth mover distance (EMD) for comparison between functions.
- Extensive evaluations were conducted to examine the performance of DAPDiff. DAPDiff outperformed the state-of-the-art tools Asm2vec, BinDiff and TurboDiff, especially when making diffing across architectures. Experimental results prove that DAPDiff is applicable for the vulnerability detection in IoT devices.

The rest of this paper is organized as follows. Section 2 presents an overview of the system workflow. Section 3 introduces our anchor function selection strategy and the expansion strategy for the comparison function set. Section 4 presents how to extract data flow features and calculate EMD for comparison. Experiments are implemented in Sect. 5 to demonstrate the availability, efficiency and effectiveness of our method. The related work is discussed in Sect. 6 and the conclusion follows in Sect. 7.

2 System Overview

The program-wide binary diffing is to find the corresponding similar functions in the two binary files to be compared. We define similar functions in the two comparison files as anchor functions. Our method aims to find more anchor functions gradually according to the determined ones, and the system workflow is shown in Fig. 1. Initial anchor function selection is the first step of binary diffing. Unique features,

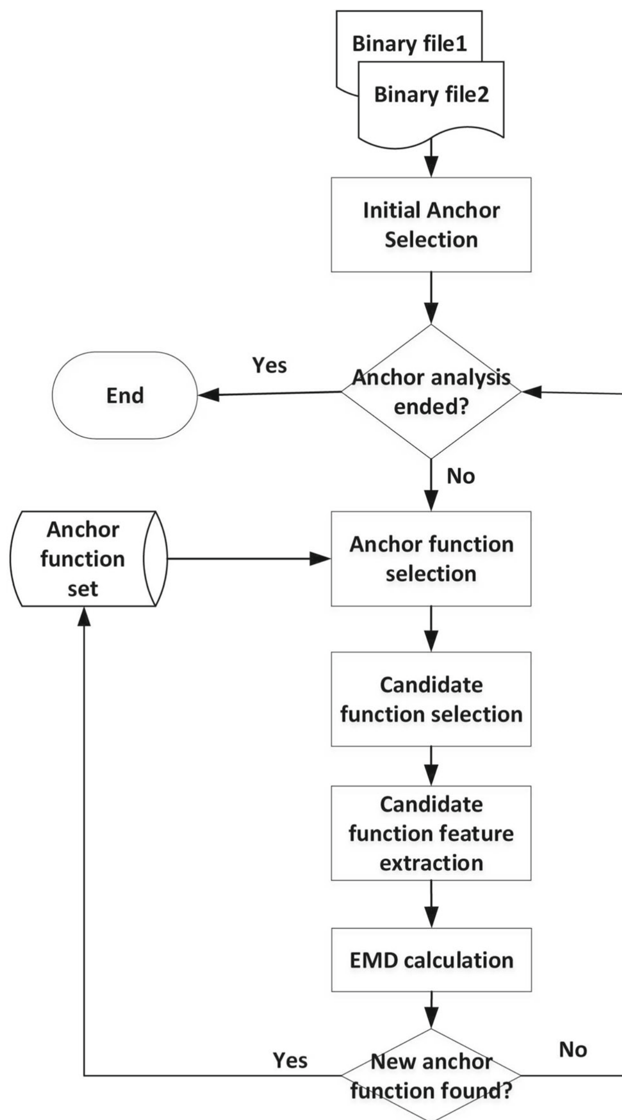


Fig. 1 System workflow

such as string constants and integer constants, are used to determine the initial anchor functions. Candidate functions refer to functions that have a call relationship with anchor functions. We group candidate functions that have the call relationship with the same anchor function into one comparison function subset. Then, new anchor functions are obtained by comparing the functions in the comparison function subset of two files. This comparison process relies on features extracted based on variable-liveness analysis technology and similarity result calculated by earth mover distance (EMD). To extract the semantic features of candidate functions, we explore a local-data-sensitive feature extraction method. After the feature extraction of the candidate functions, the earth mover distance (EMD) is applied to obtain the corresponding relationship between the candidate functions in the two comparison files, obtaining new anchor functions. The selection of anchor function and candidate function iterates continuously until the termination condition is reached.

3 Initial Anchor Function Selection and Expansion of Comparison Function Set

3.1 Initial Anchor Function Selection

To obtain the corresponding relationship between the functions in the two comparison files, we first need to find the comparison basis. This paper presents an anchor function-based comparison method. We define the anchor function in Definition 1.

Definition 1 Given two comparison files F containing functions $\{f_1, f_2, ..., f_m\}$ and F' containing functions $\{f'_1, f'_2, ..., f'_n\}$, function f in file F and f' in file F' are defined as anchor functions if f and f' are proved to be similar where $f \in \{f_1, f_2, ..., f_m\}$ and $f' \in \{f'_1, f'_2, ..., f'_n\}$.

For the two binary files to be compared, the selection of initial anchor function is critical for later comparison. The initial anchor function selection process depends on the features that are unique and easy to compare. The string constants referenced by functions are relatively unique and will not change according to different optimization levels and architectures. We study the functions with string constants in binary files and find that these functions account for more than 15% of the total functions, making string constant a feasible candidate feature for initial anchor function selection. However, some functions share the same string constant, such as the function *sendping_tail* and *echo_main* in OpenSSL library. Therefore, it is not enough to only take the string constant feature as the unique feature when selecting the anchor function. It is found the largest frequency of integer constant of *sendping_tail* and *echo_main* is 6 and 14, respectively. As a result, the frequency of integer constants also helps to distinguish different functions. In this paper, we consider three kinds of features when selecting the initial anchor functions, which are string constant, largest frequency of integer constant and the number of function parameters. Using these three kinds of features, we obtain the anchor function set containing the initial anchor functions.

The subsequent anchor function selection is different from the initial anchor function selection procedure. New anchor functions are added by calculating the EMD, which will be discussed in detail in Sect. 4.

3.2 Expansion Procedure of Comparison Function Set

The expansion procedure of the comparison function set begins with anchor functions. We add the functions that have

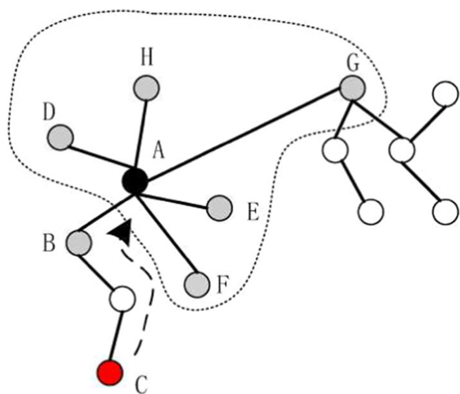


Fig. 2 The expansion procedure of comparison function set

call relationships with the anchor functions to the comparison function set. Then, new anchor functions are selected from the comparison function set. This process iterates until most functions are covered and compared. The selection of new anchor functions from comparison function set is discussed in detail in Sect. 4.3. In this section, we investigate the availability of comparison function expansion procedure based on the function call graph.

A function call graph [11,12] is a directed graph (and more specifically a flow graph [13]) that represents call relationships between functions in a computer program. Specifically, each node of the function call graph represents a function, and each edge (*f*,*g*) indicates the call relationship between function *f* and function *g*.

Figure 2 shows an example of a comparison function set expansion process based on a partial function call graph. In the function call graph, function node C has been selected as the anchor node. We use the function call graph to explore new comparison functions having call relationship with node C. After two iterations, node A can be added to the comparison function set and become a new anchor function. Nodes D to G are then added to the comparison function set in the next iteration.

The function expansion procedure benefits from the scale-free property [14] of the function call graph. Like traffic network and Internet, function call graph is a scale-free network [15]. That is, most nodes have fewer edges with others, while a few nodes connect with many other ones (with large degree). During the comparison expansion procedure, the earlier the function nodes with large degree are grouped into the comparison function set, the fewer iterations are required to cover most functions. Anchor function accounts for a relatively large proportion of the whole function set, which makes the anchor function set more likely to contain function nodes with large degrees. If the anchor function has higher degree, there will be more functions added into the comparison set. As the example shown in Fig. 2, anchor function B has more opportunities to find a new candidate function with large degrees, namely A. After selecting node A as the anchor function, nodes D to G can be grouped into the comparison function set. The expansion from node A can cover more function nodes, reducing the number of subsequent iterations. The scale-free property of function call graph can help cover most functions by adding them into the comparison function set in limited iterations.

4 Feature Extraction and EMD Calculation

As described in the previous section, the comparison function set contains functions that have call relationships with the anchor functions. The functions in the comparison function set can be selected as new anchor functions, so we call these functions candidate functions. We select new anchor functions from the candidate functions through feature extraction and EMD-based comparison.

4.1 Local-Data-Sensitive Feature Extraction

The features that can represent the function are the basis for finding new anchor functions by comparison. Inspired by data flow analysis, we implement a local-data-sensitive feature extraction method to obtain finer data dependencies between functions with call relationships. We focus on the variable liveness information in instructions from the caller function A to the function B called by A. Different from previous data flow dependency analysis method in basic block granularity, this extraction method works in instruction granularity, which is finer. The local-data-sensitive feature extraction algorithm is shown in Algorithm 1.

Algorithm 1 Local-data-sensitive feature extraction algorithm.

Require: function B, function A that calls B
1: *path=shortest_path(addr(A.entry),addr(call B)).reverse*
2: **for** $instruction\ i\ in\ path$ **do**
3: $IN[i] = F_i(OUT[i])$
4: $F_i(x) = USE(i) \cup (x - DEFS(i))$
5: **end for**
6: **for** $instruction\ j \in Prev(call\ B)$ **do**
7: **for** $variable\ r\ in\ USE[j]$ **do**
8: **if** $r\ in\ IN[B] \cap OUT(NEXT(A.entry))$: **then**
9: $vector[r] = 1$
10: **else**
11: $vector[r] = 0$
12: **end if**
13: **end for**
14: **end for**

The algorithm takes function B and its caller function A as input. To reduce the analysis overhead, we only analyze instructions from the function A entry address to the address of $call\ B$ in A. However, there may be multiple paths from

the entry of A to *call B* instruction. We choose the shortest path of the multiple paths (line 1). For instructions in this path, backward variable liveness analysis is applied to get the definition_use chain [16] (lines 2–5). Here, DEFS(i) denotes to the set of variables that are defined in instruction i and not used before instruction i. USE(i) records variables used in instruction i. We analyze the instructions in the entry of function A, denoted as NEXT(A.entry), and the instructions before the instruction of *call B*(X86 architecture) which is Prev(*call B*). Variables used in instruction set of Prev(*call B*) are judged whether they are defined in the NEXT(A.entry). If they are defined in NEXT(A.entry), the relevant value is 1, otherwise it is 0 (lines 6–14). Considering the binary analysis process is implemented without source code, the local-aware variable liveness analysis algorithm is implemented on the intermediate language (IL). Therefore, the variables in the algorithm are the related register values in the IL. We select IL registers and form the vector representing the liveness of variable in related instructions.

4.2 Extraction of Other Features

In addition to data flow features extracted through variable liveness analysis, we choose some lightweight semantic features. The out-degree and in-degree of each function node in the function call graph reflect the call relationship between functions, so they are added to the feature vector. Other features added include parameter number, return type and the number of API functions. These features are normalized to numbers and added to the feature vector.

4.3 Selection of New Anchor Functions

To select new anchor functions from comparison function set, we apply the earth mover distance (EMD) to make comparison based on the features extracted instead of the common-used graph edit distance method [17]. Earth mover distance is proposed by Rubner et al. [10] to measure the distance between two probability distributions in a specific area. It has been applied in the field of natural language processing (NLP) and Kusner et al. [18] proposed the word movement distance (WMD) to calculate the distance among documents. The structure of paragraph in natural languages is somewhat like that of binary code. Therefore, in this section, we apply the EMD to obtain the relaxed one-to-one mapping relationship between the function nodes in the comparison function set. To adapt to the generated feature vector, we make some modifications to EMD by replacing the distributed ground distance with cosine distance. Then, the flow matrix is obtained according to the distance between function nodes, which reflects the relaxed one-to-one mapping relationship between functions in candidate function set. The relaxed one-to-one mapping relationship means that there is probability that not all the nodes have strictly one-to-one relationships. This is because for the functions in the candidate function subset, there may be some functions which have close feature vector values, making the some one-to-one mapping relationship not precise sometimes. However, in most cases, the features of functions in candidate subset are not close, guaranteeing that the overall performance is relatively good. Functions with one-to-one mapping relationship are taken as new anchor functions.

After new anchor functions are selected, the calling functions of new anchor functions will be obtained, which is used to start a new iteration. The coverage rate of comparison function is high within limited iterations due to the scale-free property of function call graph (Sect. 3.2).

5 Evaluation

We implemented a proof-of-concept DAPDiff (data-aware program-wide diffing). To evaluate whether DAPDiff can make the program-wide diffing effectively across multiple optimization levels and different architectures, we would like to answer the following three research questions:

- $RQ1$: $Availability$. Is the expansion method based on comparison function set feasible?
- $RQ2$: *Efficiency and Effectiveness*. Can DAPDiff perform well across optimization levels and architectures with acceptable time overhead?
- $RQ3$: $Proportion\ of\ Anchor\ Functions$. Anchor functions play an important part in the expansion of comparison function set and influence the diffing result. Can we achieve a relatively high proportion of anchor functions in limited iterations?

In the experiment, we compiled Busybox, OpenSSL and Coreutils in different architectures (ARM/MIPS/ X86/X64) and optimization levels (O0-O3) like [19–21], taking the compiled binary files as our analysis target. In addition, real-world firmware files from Genius dataset [22] were also used, which contain third-party library. To verify the effectiveness of DAPDiff, we compared DAPDiff with the state-of-the-art tools Asm2Vec [9], BinDiff and TurboDiff.

5.1 Availability of the Expansion Method Based on Function Call Graph(RQ1)

To testify the availability of the anchor-function-based expansion method, we selected the file in real-world firmware R4500 [22] for analysis. Function call graph is the expansion basis, so it is needed to firstly discuss the scale-free property of function call graph. Then, the coverage of comparison

functions is addressed to testify the result of call-graph-based expansion method.

Scale-free property of function call graph. We constructed the function call graph of R4500 and recorded both the in-degree and out-degree of each function node in the function call graph. The cumulative distribution of degree is shown in Fig. 3, demonstrating that both the in-degree and out-degree distributions are in accordance with the power law distribution [14,23]. Meanwhile, we recorded the distribution of in-degree and out-degree in Fig. 4. The number of function nodes in R4500 with in-degree more than 4 is 111, accounting for 19.9% of the 559 functions. The proportion of nodes with out-degree greater than 4 is 22.4%. The maximum in-degree and out-degree are 106 and 88, while more than 77% of nodes have in-degree or out-degree less than 4. This means the connection between function nodes in function call graph has uneven distribution. In the function call graph, nodes with higher in-degree or out-degree values account for a small proportion of the total nodes. However, these nodes connect more other function nodes, making them key hubs. So the earlier such kinds of nodes are selected as anchor functions, the fewer iterations are required.

Coverage of functions based on function call graph. The diffing process starts with the initial anchor function. Using call relationship, more and more functions are added to the comparison function set from where new anchor functions are selected. Here, we are not concerned with the generation of new anchor functions according to our expansion strategy which will be discussed in Sect. 5.4, but with the coverage of comparison function set during iterations. For binary file in R4500, Fig. 5 is actually a reconstruction of the function call graph based on initial anchor functions. In the graph of

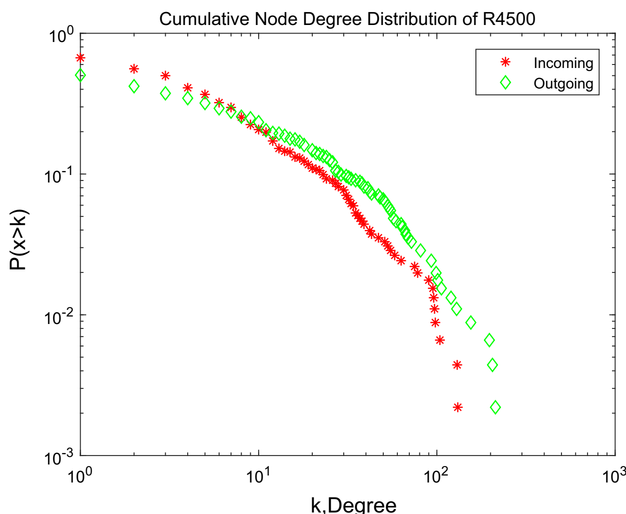
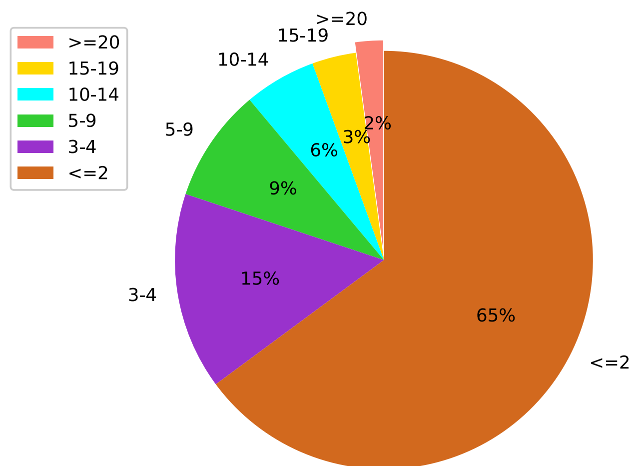
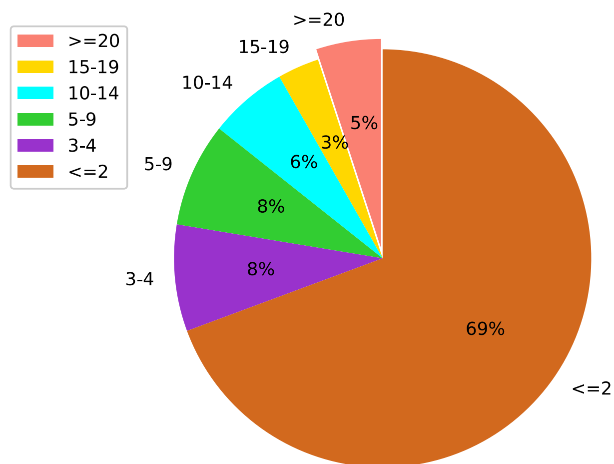


Fig. 3 Cumulative node degree distribution of R4500



(a) In-degree distribution of R4500



(b) Out-degree distribution of R4500

Fig. 4 Degree distribution of R4500

Fig. 5, only function nodes that have a direct or indirect call relationship with anchor functions have edges. Initial anchor function nodes are colored red. Other function nodes are colored differently according to the path length between them and the initial function nodes. The number of initial anchor function nodes in R4500 is 154, accounting for 27.55 percentage of all 559 functions. During the first round of function call relationship analysis, the number of newly covered nodes (having direct call relationship with initial anchor functions) is 184, increasing the coverage ratio to 60.5%.

In addition, there are some red nodes without edges in Fig. 5, meaning that these functions are initial anchor functions, but they do not have call relationships with other functions, such as function $SSL_get_version$. Furthermore, the nodes colored blue do not have edges, which means that they have no direct or indirect call relationship with the initial anchor functions. As a matter of fact, we later analyzed

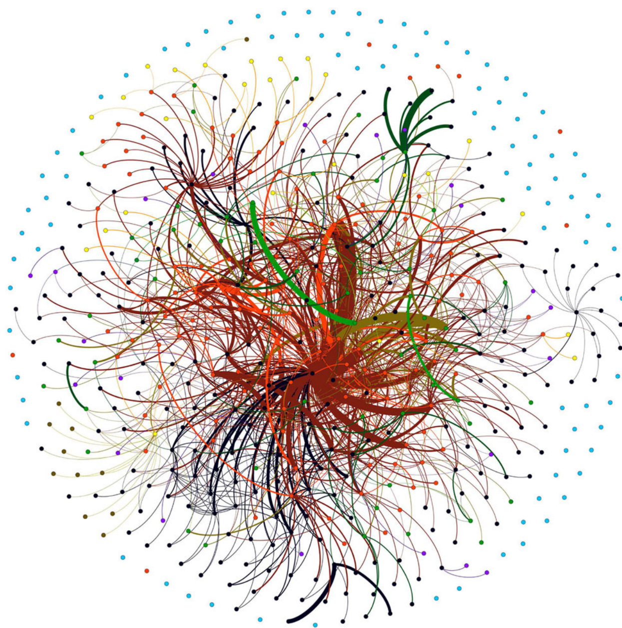


Fig. 5 Function nodes covered by function call graph in R4500

these functions and found that they usually have no call relationship with any other functions in the function call graph. We call them orphan functions. For example, the function $ssl3_alert_code$ is one of the orphan functions. The percentage of orphan nodes in R4500 is 17.2%. However, these orphan functions can be grouped into a separate comparison subset, and we can still extract features and compare them to based on EMD find new anchor functions (Sect. 5.4).

5.2 Analysis Efficiency (RQ2)

To testify the feasibility of the DAPDiff method, the time overhead should be discussed. The time cost of diffing is mainly related to three procedures, that is data flow-based feature extraction, statistical feature generation and the EMD calculation.

Data flow-based feature extraction time overhead. For function pair containing function A and function B called by A, data flow-based feature extraction time overhead refers to the time required to generate liveness-variable-related feature vector between A and B. We analyzed and recorded the variable liveness analysis (including vectorization) time of all function pairs with call relationship. The time cost of all function pairs in real-world firmware DAP-2360 is shown in Fig. 6. The maximum time cost of function pair analysis is 62.5 s. However, for most function pairs, the time overhead is less than 5 s and the average time cost is 9.63 s. In the actual analysis procedure, we only make data dependence analysis between anchor functions and functions that have call relationship with them, rather than all function pairs with call

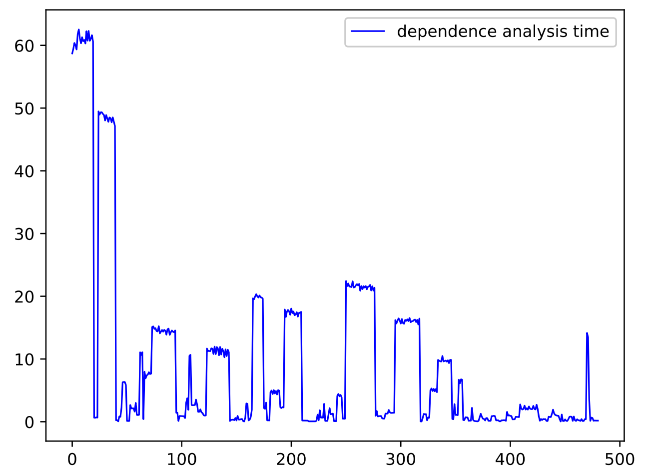


Fig. 6 Liveness variable analysis time of DAP-2360

relationship in binary file. In this way, we can significantly reduce the analysis overhead. This analysis strategy based on partial function pairs makes our data flow-based method relatively lightweight. The data flow analysis time cost of all the function pairs is 99.5 minutes. However, when making diffing between DAP-2360 and binary files compiled by OpenSSL, the actual time cost can be reduced to 16 minutes, which is relatively acceptable.

Statistical feature generation time. In addition to the data flow feature, we also select other features that we call statistical features, such as in-degree, out-degree and number of called API functions. We extracted these statistical feature by writing IDA plugin and recorded the time cost. Figure 7 shows the statistical feature extraction time cost of functions in OpenSSL binaries compiled in different architectures. The median line time of ARM/MIPS/X64/X86 architectures is 3.046e-6, 2.99e-6, 3.02e-6 and 3.0e-6 s, respectively. The maximum time cost is 2.5e-4 s. The time cost of statistical features generation is much less than that of data flow analysis.

EMD calculation time. We recorded the EMD calculation time cost when comparing the standard third-party library with the real-world firmware containing the library. We selected the OpenSSL library v1.0.1f and the firmware files of DIR-655 and R4500 containing the OpenSSL library, making diffing to record the EMD calculation time cost. The EMD calculation time cost when making diffing between DIR-655 and OpenSSL is shown in Fig. 8a, with 0.058 s as the maximum value and 0.007 s as the median line value. Figure 8b shows the corresponding time cost between R4500 and OpenSSL. The calculation time is less than 0.045 s and median line time is 0.008 s. EMD calculation time is also much less than data flow analysis time.

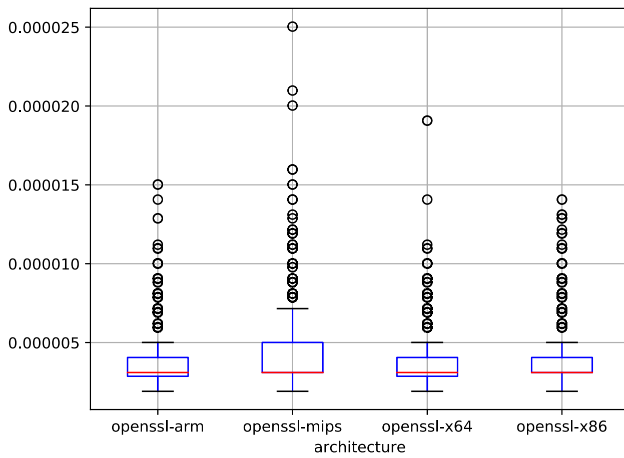
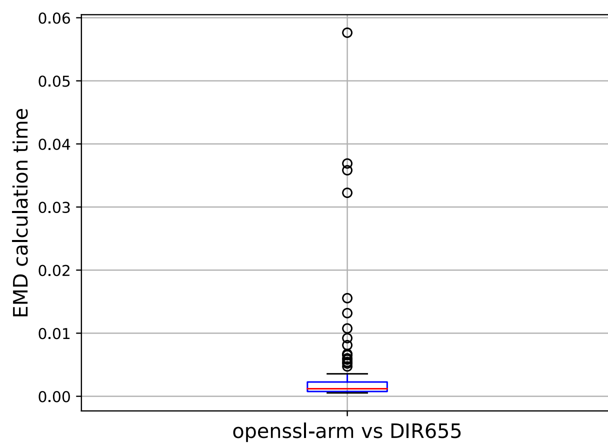
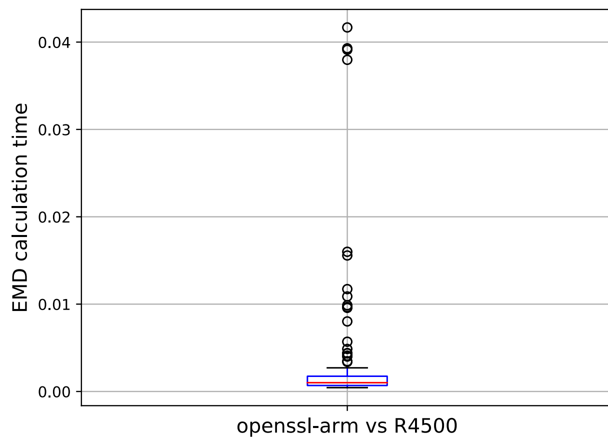


Fig. 7 Statistical feature extraction time of OpenSSL compiled in different architectures



(a) EMD calculation time in diffing DIR655 and OpenSSL.



(b) EMD calculation time in diffing R4500 and OpenSSL.

Fig. 8 EMD calculation time

The time cost of statistical feature extraction and EMD calculation is much less than that of data flow analysis. This is mainly because the variable liveness analysis procedure is implemented in instruction granularity which is much finer than other methods. Considering the time overhead caused by finer granularity, we make optimizations such as focusing on the variable within a limited number of instructions. In addition, we only apply data flow analysis to calling function pairs that contain anchor functions to improve efficiency.

5.3 Effectiveness (RQ2)

We evaluated the effectiveness of DAPDiff when making diffing between files compiled in different optimization levels and architectures. DAPDiff is compared with BinDiff, TurboDiff and Asm2vec by using precision and recall metrics [19–21]. To define the metrics of precision and recall, we use G to represent the ground truth information, which is the set of actually matched function pairs in the two binary files. The matched function pairs found by our method form a set represented by D, while unmatched function pairs form a set U. The three kinds of sets can be represented by Equation 1-3.

$$G = (g_1, g_1'), (g_2, g_2'), ...(g_s, g_s') \quad (1)$$
$$D = (d_1, d_1'), (d_2, d_2'), ...(d_n, d_n') \quad (2)$$
$$U = (u_1, u_1'), (u_2, u_2'), ...(u_t, u_t') \quad (3)$$

Then, $D \cap G$ is the set of correctly matched function pairs, and elements in $U - U \cap G$ refer to the unmatched function pairs detected by our method. Precision indicates the ratio of the correctly matched function pair number to the number of detected matched pairs. It is represented in Equation 4. Recall is the ratio of the correctly matched function pair number to the number of all correctly classified pairs shown in Equation 5.

$$\text{Precision} = \frac{||D \cap G||}{||D||} \quad (4)$$
$$\text{Recall} = \frac{||D \cap G||}{||D \cap G + U - U \cap G||} \quad (5)$$

To verity the effectiveness of DAPDiff compared with the other tools, we designed two scenarios: one is to make diffing between files compiled at different optimization levels and the other is to make diffing between files compiled in different architectures.

Binary diffing across optimization levels. During the procedure of diffing across optimization levels, we compared the performance between DAPDiff and state-of-the-art tools including Asm2vec, BinDiff and TurboDiff. We compiled Coreutils(v5.93), Busybox(v1.27.2) and OpenSSL(1.0.1h) in

Table 1 Recall and precision in different optimization levels

		Recall/Precision			
		Asm2vec	BinDiff	TurboDiff	DAPDiff
Coreutils	V5.93O0-O3	0.418/0.98	0.828/0.967	0.317/0.917	0.918/0.986
	V5.93 O1-O3	0.582/0.985	0.858/0.988	0.32/0.887	0.840/0.987
	V5.93 O2-O3	0.629/0.992	0.884/0.99	0.316/0.961	0.814/0.9836
	Average	0.543/0.986	0.856/0.982	0.318/0.922	0.857/0.985
Busybox	1.27.2 O0-O3	0.571/0.986	0.76/0.989	0.309/0.97	0.754/0.995
	1.27.2 O1-O3	0.638/0.99	0.96/0.983	0381/0.967	0.942/0.991
	1.27.2 O2-O3	0.677/0.987	0.983/0.992	0.427/0.974	0.972/0.989
	Average	0.629/0.988	0.901/0.988	0.372/0.97	0.889/0.992
OpenSSL	1.0.1h O0-O3	0.571/0.992	0.848/0.995	0.246/0.935	0.794/0.989
	1.0.1h O1-O3	0.644/0.991	0.917/0.991	0.232/0.942	0.781/0.989
	1.0.1h O2-O3	0.665/0.992	0.90/0.991	0.290/0.856	0.784/0.985
	Average	0.627/0.992	0.89/0.992	0.256/0.911	0.786/0.988

O0-O3 optimization levels and recorded the recall/precision results in Table 1.

Among the three tools Asm2vec, BinDiff and TurboDiff, BinDiff performed better than the other two tools in both recall and precision. However, the precision of the three tools is close. For example, the average diffing precision of Coreutils is 0.986, 0.982 and 0.922 for Asm2vec, BinDiff and TurboDiff, respectively, while the average recall value of the three tools was different. The average recall rate of Asm2vec is 0.543 and TurboDiff is 0.318. BinDiff, on the other hand, has an average recall of 0.856, which is higher than the other two tools. Higher precision and relatively lower recall relate to a higher false negative rate, meaning that a number of similar functions were not detected. When it comes to the comparison of BinDiff and DAPDiff, the precision and recall of Coreutils and Busybox are close. However, the recall of DAPDiff is lower than BinDiff when make diffing on OpenSSL binaries. This is mainly because that the orphan function ratio of OpenSSL is higher than that of Busybox and Coreutils, which affects the results of the anchor-based diffing method applied by DAPDiff. However, the orphan function ratio of real-world firmware files is not as high as that of standard third-party libraries, and the diffing results on firmware files discussed in the next paragraph are better than those on standard library files.

Binary diffing across architectures. To verify the effectiveness of DAPDiff across different architectures, we chose real-world firmware and the standard library as our diffing target. This diffing process actually deals with the comparison across versions, optimization levels and architectures because we usually do not know the version or optimization levels of the library file used in real-world firmware. Considering that Asm2vec can only make diffing in single architecture, we compared the performance of BinDiff, TurboDiff and DAPDiff. Real-world firmware files are obtained from DCS-1100, DIR 855L, DAP-2590 and DIR-636. DCS-1100 and DIR 855L contain OpenSSL library while DAP-2590 and DIR-636 contain Busybox. The diffing was made between the firmware files and the standard OpenSSL and Busybox libraries compiled in X86/MIPS/ARM architectures. Recall and precision values are listed in Table 2. TurboDiff had high precision and low recall value, meaning it makes wrong judgements on a number of similar functions. Although the performance of BinDiff was comparable to that of DAPDiff, and sometimes even better in the scenario of making diffing across optimization levels, DAPDiff outperformed BinDiff when making diffing across architectures. The average recall and precision of DAPDiff are 0.943 and 0.96, which are 41.4% and 9.2% higher than that of BinDiff. BinDiff uses small-primes-products method, relying on the basic blocks and edges of CFG. However, these features vary in the files compiled in different architectures. On the other hand, the features and call relationship used by DAPDiff are less affected by different architectures. Furthermore, the performance of DAPDiff in the real-world firmware diffing is better than that across optimization levels due to fewer orphan functions. The diffing across optimization levels is made on the standard third-party library, which contains all the functional modules. However, when vendors apply the third-party library to their firmware, due to the limitation of memory, they usually delete all the unnecessary codes including many orphan functions, resulting in a better performance.

5.4 Relationship Between Anchor Function Ratio and Iterations (RQ3)

The anchor function selection process will iterate for several times to find more functions to compare, from which new

Table 2 Recall and precision of diffing across architectures

	Recall/precision		
	BinDiff	TurboDiff	DAPDiff
OpenSSL_X86 vs DCS-1100	0.715/0.917	0.077/0.709	0.908/0.952
OpenSSL_ARM vs DCS-1100	0.737/0.685	0.151/0.776	0.867/0.979
OpenSSL_MIPS vs DCS-1100	0.726/0.942	0.137/0.922	0.92/0.963
OpenSSL_X86 vs DIR-855L 1.01	0.738/0.721	0.084/0.845	0.964/0.985
OpenSSL_ARM vs DIR-855L 1.01	0.444/0.604	0.11/0.847	0.944/0.943
OpenSSL_MIPS vs DIR-855L 1.01	0.854/0.949	0.094/0.889	0.973/0.973
Busybox_X86 vs DAP-2590	0.482/0.898	0.144/0.857	0.98/0.955
Busybox_ARM vs DAP-2590	0.48/0.962	0.192/0.819	0.969/0.948
Busybox_MIPS vs DAP-2590	0.498/0.936	0.191/0.902	0.952/0.936
Busybox_X86 vs DIR-636	0.787/0.979	0.39/0.982	0.928/0.97
Busybox_ARM vs DIR-636	0.767/0.970	0.354/0.952	0.947/0.972
Busybox_MIPS vs DIR-636	0.777/0.981	0.381/0.949	0.967/0.944
Average	0.667/0.879	0.192/0.87	0.943/0.96

anchor functions can be found. Due to the scale-free property of function call graph, most functions can be added to the comparison function set in limited iterations. However, the diffing result relies on the number of anchor functions found. In this section, we hope to explore how many iterations should be made to cover a high ratio of anchor functions and answer the question $RQ3$.

The anchor function ratio is the ratio of the anchor function number to the total function number in the binary files. The diffing was made between functions in standard third-party library and the real-world firmware files. We took DCS-1100 firmware containing OpenSSL and DAP-2590 firmware containing Busybox library as the diffing target. The file in DCS-1100 firmware was made diffing with OpenSSL v1.0.1f binary compiled in ARM/MIPS/X86/X64 architectures. DAP-2590 was compared with Busybox 1.27.2 binaries compiled in the above four architectures. The new anchor functions selection process was iterated four times. However, there are a number of orphan functions which have no call relationship with other functions, like the blue node shown in Fig. 5. After four iterations, we grouped all orphan nodes into one comparison subset, making feature extraction and EMD calculation like the new anchor function selection process. In this way, the anchor functions were selected from the orphan functions.

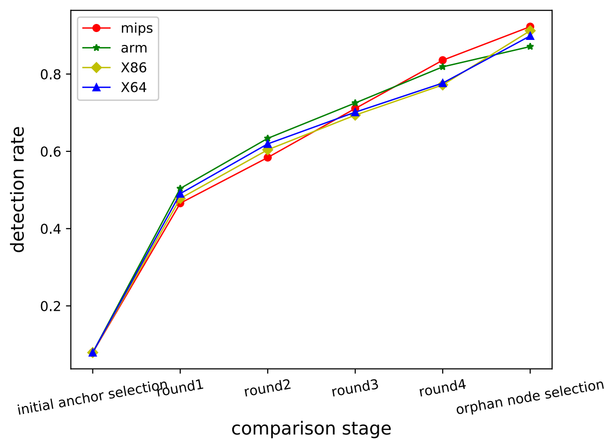
Figure 9a and b shows the anchor function ratio of firmware DCS-1100 and DAP-2590 when making diffing between them and the corresponding third-party library during different iterations. The first thing we need to mention is that although the anchor function ratio of each iteration is different for the four architectures, the difference is very small. This proves that both the anchor selection and the feature extraction strategies are not affected too much by the architecture. It can also be observed that the growth rate of anchor function becomes slower as the number of iterations increases. This is mainly because the anchor function is selected from comparison function set. However, as the number of iterations increases, many functions have been compared, resulting in fewer and fewer functions added to the comparison function set. After four iterations, we compared orphan functions using the same feature extraction and EMD calculation method to select anchor functions from orphan functions. This method is proved to be effective, increasing the average anchor function ratio to more than 90%.

Furthermore, we applied the diffing method to the vulnerability detection in the third-party library used by real-world firmware. It was found that of the chosen 93 real-world firmware files, 73 files were affected by CVE-2015-0204, indicating that this method can be applied to help detect vulnerabilities in IoT devices.

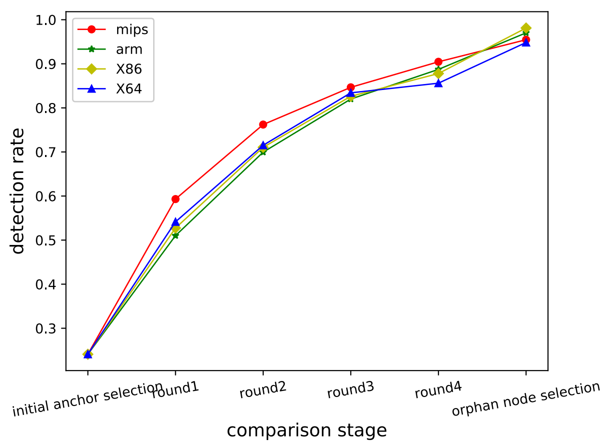
6 Related Work

Compared with the relatively mature open-sourced code diffing technology [24–30], binary diffing analysis method faces code optimization problems such as function inlining, redundancy elimination, instruction reordering and conversion in the compilation process. These problems make binary diffing more difficult than open-sourced files. However, many commercial software and files used in some fields (such as Internet of things devices) are not open-sourced, making binary diffing necessary and appropriate for vulnerability detection.

Researchers investigate the semantic equivalence of code by graph comparison [31], symbolic execution and theorem proving [32,33]. However, graph isomorphism is a NP-



(a) Anchor function ratio of diffing between DCS-1100 and OpenSSL



(b) Anchor function ratio of diffing between DAP-2590 and OpenSSL

Fig. 9 Anchor function ratio of real-world firmware

complete problem, leading to low efficiency, and the method based on theorem proving is not scalable. Later researches use the execution trace [34–37] or code signature [38] to measure the similarity of functions in binary files. However, they either have low coverage or are not robust to changes brought about by compiler optimization.

With the development of machine learning, especially deep learning, researchers begin to apply machine learning methods to binary diffing. Graph embedding [39–42] and graph neural network [43–45] models are applied to extract features to represent binary code. Genius [22] is one of the most outstanding solutions that transform the ACFG (attribute control flow graph) into feature vector to represent the functions and measure the similarity by bipartite graph matching algorithm. Gemini [46] relies on deep neural network to embed the ACFG graph of Genius into a matrix. The features extracted by Gemini and Genius are statistical, containing limited semantics. Vulseeker [47] generates labeled semantic flow graph(LSFG) to represent code feature. Redmond et al. [48] convert the binary code to intermediate language and record the input/output as signature for comparison across architectures. Zhang et al. [49] and Wang et al. [50] focus on the changes between patched and unpatched code and make similarity comparison of code snippet. Yu et al. [51] adopt the convolutional neural network to extract the order information as well as semantic information. They make diffing between source-code and binary code combining deep pyramid convolutional neural network (DPCNN) with graph neural network (GNN) [52]. There are also local preference methods to make binary function diffing. Kam1n0 [53] combines the subgraph matching and adaptive LSH to detect the code clone. Li et al. [54] propose a topology-aware hashing method by extracting graph signature of CFG as the comparison basis. Duan et al. [55] implement DEEPBINDIFF which combines the NLP(natural language processing) and TADW algorithm (Text-associated DeepWalk algorithm) [56] to obtain the semantic cross-function dependency feature. However, DEEPBINDIFF applies random walk in ICFGs (inter-procedural CFGs), which is relatively time-consuming. Besides, it only supports diffing in a single architecture.

7 Conclusion

In this paper, a data-aware program-wide diffing method is proposed to compare the binary files across architectures and optimization levels. Using the anchor functions and call relationship, this method expands the diffing scope step by step. To obtain more accurate diffing results, we extract semantic features by variable liveness analysis, and make comparisons using the extracted features and the modified EMD calculation method. Experiments show that our DAPDiff prototype performs well when making diffing across architecture and optimization levels, which proves that it is available for the vulnerability detection on IoT devices. This method can also be combined with other feature extraction and similarity calculation methods, such as those based on machine learning technology. However, there are improvements for the methods proposed in this paper. The accuracy of diffing results rely on the features extracted before the calculation of EMD. Currently, the features extracted mainly include the data-flow related variable liveness and the simple statistical ones such as in-degree and out-degree. These features guarantee a relatively stable and promising diffing precision and recall value. However, due to the inherent limitation of static method, features extracted cannot fully represent the behavior of binary code. In future work, we will explore ways to extract the dynamic execution related features that can represent code behavior more thoroughly, such as the relationship between

input and output values [47], and combine the feature extraction method with neural network model which can extract more semantic features that can represent the binary code [46,47].

Acknowledgements We thank the anonymous reviewers for the helpful comments. We thank JianGao for sharing his code of Vulseeker in github, which inspired us a lot when implementing our prototype.

Compliance with ethical standards

Funding This work is supported by the National Key Research and Development Program of China (No. 2017YFB0802900)

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Cui, A.; Stolfo, S.J.: A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In: Proceedings of the 26th Annual Computer Security Applications Conference, pages 97–106 (2010).
2. Herzberg, B.; Bekerman, D.; Zeifman, I.: Breaking down mirai: An iot ddos botnet analysis. Incapsula Blog, Bots and DDoS, Security (2016)
3. Diaphora. https://github.com/joxeankoret/diaphora. Accessed December 15, 2020.
4. Bindiff, Z.: https://www.zynamics.com/bindiff.html. Accessed September 14, 2020.
5. Turbodiff. https://www.coresecurity.com/core-labs/open-source-tools/turbodiff-cs. Accessed December 20, 2020.
6. Dullien, T.; Rolles, R.: Graph-based comparison of executable objects (english version). SSTIC **5**(1), 3 (2005)
7. Pewny, J.; Schuster, F.; Bernhard, L.; Holz, T.; Rossow, C.: Leveraging semantic signatures for bug search in binary programs. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 406–415. ACM (2014)
8. Karim, M.E.; Walenstein, A.; Lakhotia, A.; Parida, L.: Malware phylogeny generation using permutations of code. J. Comput. Virol. **1**(1), 13–23 (2005)
9. Ding, S.H.H.; Fung, B.C.M.; Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 472–489. IEEE (2019)
10. Rubner, Y.; Tomasi, C.; Guibas, L.J.: The earth mover's distance as a metric for image retrieval. Int. J. Comput. Vis. **40**(2), 99–121 (2000)
11. Callahan, D.; Carle, A.; Hall, M.W.; Kennedy, K.: Constructing the procedure call multigraph. IEEE Trans. Software Eng. **16**(4), 483–487 (1990)
12. Khedker, U.P.; Sanyal, A.; Karkare, B.: Data flow analysis: theory and practice. CRC Press, Cambridge (2017)
13. Jalote, P.: An integrated approach to software engineering. Springer Science & Business Media, Berlin (2012)
14. Barabási, A.; Bonabeau, E.: Scale-free networks. Sci. Am. **288**(5), 60–69 (2003)
15. Yu, L.; Shen, Y.; Pan, Z.: Structure analysis of function call network based on percolation. In: 2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC), pp. 350–354 IEEE (2018)
16. Stanier, J.; Watson, D.: Intermediate representations in imperative compilers: A survey. ACM Comput. Surv. (CSUR) **45**(3), 1–27 (2013)
17. Nair, A.; Roy, A.; Meinke, K.: funcgnn: A graph neural network approach to program similarity. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11 (2020)
18. Kusner, M.; Sun, Y.; Kolkin, N.; Weinberger, K.: From word embeddings to document distances. In: International Conference on Machine Learning, pp. 957–966. PMLR (2015)
19. Zabihimayvan, M.; Doran, D.: Fuzzy rough set feature selection to enhance phishing attack detection. In: 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), pp. 1–6. IEEE (2019)
20. Zabihimayvan, M.; Sadeghi, R.; Kadariya, D.; Doran, D.: Interaction of structure and information on tor. In: International Conference on Complex Networks and Their Applications, pp. 296–307. Springer (2020)
21. Sadeghi, R.; Banerjee, T.; Hughes, J.: Predicting sleep quality in osteoporosis patients using electronic health records and heart rate variability. In: 2020 42nd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC), pp. 5571–5574. IEEE (2020)
22. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 480–491, ACM (2016)
23. Zabihimayvan, M.; Sadeghi, R.; Doran, D.; Allahyari, M.: A broad evaluation of the tor english content ecosystem. In: Proceedings of the 10th ACM Conference on Web Science, pp. 333–342 (2019)
24. Ghaffarian, S.; Mohammad, S.; Hamid, R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv. (CSUR) **50**(4), 56 (2017)
25. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z.; Wang, S.; Wang, J.: Sysevr: a framework for using deep learning to detect software vulnerabilities. arXiv preprint arXiv:1807.06756 (2018)
26. Kronjee, J.; Hommersom, A.; Vranken, H.: Discovering software vulnerabilities using data-flow analysis and machine learning. In: Proceedings of the 13th international conference on availability, reliability and security, p. 6. ACM (2018)
27. Chernis, B.; Verma, R.: Machine learning methods for software vulnerability detection. In: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, pages 31–39. ACM (2018).
28. Pradel, M.; Sen, K.: Deep learning to find bugs. TU Darmstadt, Department of Computer Science (2017)
29. Harer, J.A.; Kim, L.Y.; Russell, R.L.; Ozdemir, O.; Kosta, L.R.; Rangamani, A.; Hamilton, L.H.; Centeno, G.I.; Key, J.R.; Ellingwood, P.M., et al.: Automated software vulnerability detection with machine learning. arXiv preprint arXiv:1803.04497 (2018)
30. Zou, Q.; Lu, L.; Yang, Z.; Gu, X.; Qiu, S.: Joint feature representation learning and progressive distribution matching for cross-project defect prediction. Inf. Softw. Technol. pp. 106588 (2021)

31. Flake, H.: Structural comparison of executable objects. In: DIMVA, vol. 46, pp. 161–173. Citeseer (2004)
32. Gao, D.; Reiter, M.K.; Song, D.: Binhunt: automatically finding semantic differences in binary programs. In: International Conference on Information and Communications Security, pp. 238–255. Springer (2008)
33. Ming, J.; Pan, M.; Gao, D.: ibinhunt: Binary hunting with interprocedural control flow. In: International Conference on Information Security and Cryptology, pp. 92–109. Springer (2012)
34. Zuo, F.; Li, X.; Young, P.; Luo, L.; Zeng, Q.; Zhang, Z..: Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706 (2018)
35. Alrabaee, S., Shirani, P., Wang, L., Debbabi, M.: Sigma: a semantic integrated graph matching approach for identifying reused.
36. Zhang, C.; Feng, C.; Li, R.H.: Locating vulnerability in binaries using deep neural networks. Ieee Access **7**, 134660–134676 (2019)
37. Hu, Y.; Wang, H.; Zhang, Y.; Li, B.; Gu, D..: A semantics-based hybrid approach on binary code similarity comparison. IEEE Trans. Softw. Eng. pp. 1–1 (2019)
38. Pewny, J.; Garmany, B.; Gawlik, R.; Rossow, C.; Holz, T.: Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy, pp. 709–724. IEEE (2015)
39. Tixier, A.J.-P.; Nikolentzos, G.; Meladianos, P.; Vazirgiannis, M.: Graph classification with 2d convolutional neural networks. In: International Conference on Artificial Neural Networks, pp. 578–593. Springer (2019)
40. Atamna, A.; Sokolovska, N.; Jean-Claude, C.: A simple permutation-invariant graph convolutional network. Spi-gcn (2019)
41. Wang, L.; Zong, B.; Ma, Q.; Cheng, W.; Ni, J.; Yu, W.; Liu, Y.; Song, D.; Chen, H.; Fu, Y.: Inductive and unsupervised representation learning on graph structured objects. In: ICLR (2020)
42. Liu, S.; Demirel, M.F.; Liang, Y.: N-gram graph: Simple unsupervised representation for graphs, with applications to molecules. arXiv preprint arXiv:1806.09206 (2018)
43. Li, Y.; Gu, C.; Dullien, T.; Vinyals, O.; Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. In: International Conference on Machine Learning, pp. 3835–3845. PMLR (2019)
44. Wang, R.; Yan, J.; Yang, X.: Learning combinatorial embedding networks for deep graph matching. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 3056–3065 (2019)
45. Jiang, B.; Sun, P.; Tang, J.; Luo, B.: Glmnet: graph learning-matching networks for feature matching. arXiv preprint arXiv:1911.07681 (2019)
46. Xu, X.; Liu, C.; Feng, Q.; Yin, H.; Song, L.; Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363–376. ACM (2017)
47. Gao, J.; Yang, X.; Fu, Y.; Jiang, Y.; Sun, J.: Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 896–899. ACM (2018)
48. Redmond, K.; Luo, L.; Zeng, Q.: A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. arXiv preprint arXiv:1812.09652 (2018)
49. Zhang, H., Qian, Z..: Precise and accurate patch presence test for binaries. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 887–902 (2018)
50. Wang, S.-C.; Liu, C.-L.; Li, Y.; Xu, W.-Y.: Semdiff: Finding semtic differences in binary programs based on angr. In: ITM Web of Conferences, vol. 12, pp. 03029. EDP Sciences (2017)
51. Zeping, Y.; Cao, R.; Tang, Q.; Nie, S.; Huang, J.; Shi, W.: Order matters: semantic-aware neural networks for binary code similarity detection. In: Proceedings of the AAAI Conference on Artificial Intelligence vol. 34, 1145–1152 (2020)
52. Yu, Z.; Zheng, W.; Wang, J.; Tang, Q.; Nie, S.; Wu, S.: Codecmr: cross-modal retrieval for function-level binary source code matching. Adv. Neural Inf. Process. Syst. 33 (2020)
53. Ding, S.H.H.; Fung, B.C.M.; Charland, P.: Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 461–470 (2016).
54. Li, Y.; Jang, J.; Ou, X.: Topology-aware hashing for effective control flow graph similarity analysis. In: International Conference on Security and Privacy in Communication Systems, pp. 278–298. Springer (2019)
55. Duan, Y.; Li, X.; Wang, J.; Yin, H..: Deepbindiff: Learning program-wide code representations for binary diffing (2020)
56. Yang, C.; Liu, Z.; Zhao, D.; Sun, M.; Chang, E.: Network representation learning with rich text information. In: Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)