LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries

Chaopeng Dong, Siyuan Li, Shouguo Yang, Yang Xiao, Yongpan Wang, Hong Li*, Zhi Li, Limin Sun
{dongchaopeng,lisiyuan,yangshouguo,xiaoyang,wangyongpan,lihong,lizhi,sunlimin}@iie.ac.cn
Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China

ABSTRACT

Open-source software (OSS) has been extensively employed to expedite software development, inevitably exposing downstream software to the peril of potential vulnerabilities. Precisely identifying the version of OSS not only facilitates the detection of vulnerabilities associated with it but also enables timely alerts upon the release of 1-day vulnerabilities. However, current methods for identifying OSS versions rely heavily on version strings or constant features, which may not be present in compiled OSS binaries or may not be representative when only function code changes are made. As a result, these methods are often imprecise in identifying the version of OSS binaries being used.

To this end, we propose *LibvDiff*, a novel approach for identifying open-source software versions. It detects subtle differences through precise symbol information and function-level code changes using binary code similarity detection. *LibvDiff* introduces a candidate version filter based on a novel version coordinate system to improve efficiency by quantifying gaps between versions and rapidly identifying potential versions. To speed up the code similarity detection process, *LibvDiff* proposes a function call-based anchor path filter to minimize the number of functions compared in the target binary. We evaluate the performance of *LibvDiff* through comprehensive experiments under various compilation settings and two datasets (one with version strings, and the other without version strings), which demonstrate that our approach achieves 94.5% and 78.7% precision in two datasets, outperforming state-of-the-art works (including both academic methods and industry tools) by an average of 54.2% and 160.3%, respectively. By identifying and analyzing OSS binaries in real-world firmware images, we make several interesting findings, such as developers having significant differences in their updates to different OSS, and different vendors may also utilize identical OSS binaries.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Software notations and tools**.

*corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623336

KEYWORDS

Open-source software, Version identification, Vulnerability detection, Firmware analysis

ACM Reference Format:
Chaopeng Dong, Siyuan Li, Shouguo Yang, Yang Xiao, Yongpan Wang, Hong Li*, Zhi Li, Limin Sun. 2024. LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623336

1 INTRODUCTION

Open-source software (OSS) has been extensively employed to expedite software development. The survey report of Forrester [4] shows that the average number of reused OSS has risen from 36% in 2015 to 75% in 2020. Furthermore, OSS is integrated by more than 90% of business software from Gatner's [3] study. However, the wide reuse of OSS also constantly exposes downstream software to potential risks. For instance, heartbleed vulnerability [1] in OpenSSL exposes millions of websites and servers to risk. Similarly, the Ghost vulnerability [2] within the glibc has affected a wide range of Linux systems, such as Debian, Ubuntu, and CentOS. The vulnerabilities of OSS are usually concealed in a specific version range and patched after a certain version. Identifying vulnerable OSS versions is a timely manner to mitigate the vulnerability exploitation due to OSS reuse. An effective version identification tool can not only detect existing software vulnerabilities but also promptly issue alerts via vulnerability impact range correlation when new vulnerabilities publish. To this end, various version identification methods have been proposed. The existing related methods can be broadly categorized into two groups based on the features they utilize.

G1: version string-based methods. These methods identify the OSS version through the detection of OSS version-related strings within a short time consumption. Pandora [54] and VES [35], utilize string patterns to extract the version strings (e.g., "shd version %s,%s") in the target binary, and employ control flow analysis and data flow analysis for real version number inference. OSSPolice [29] generates regular expressions of various OSS versions and determines the OSS version by counting the number of matched strings of each version. LibRARIAN [21] identifies the version by constructing heuristics regular expressions (e.g., "Libpng version 1(\.[0-9]1,)*") to match unique strings in binaries. Nevertheless, version string is not always a permanent and stable feature, since it could be lost or confused for several reasons, being a challenging problem referred as **P1**. First, not all OSS will retain version strings in their binaries (e.g., freetype). Based on our analysis of 79 OSS with 1,405 versions obtained from conan [9], only 42.5% of them contain version strings.

Second, version strings could be removed by developers for safety reasons or lost based on different compilation options [41]. Third, the true version can be confused with other similar strings. On the one hand, OSS will reuse other OSS and include their strings in their binaries. For example, libpng v1.4.15 reuses zlib v1.2.11 and both of these two version strings exist in the binary of libpng. On the other hand, different version strings of the same OSS may exist in a binary at the same time. For example, "OpenVPN 2.1" and "OpenVPN 2.3.4" both exist in OpenVPN 2.3.4.

G2: feature matching-based methods. In these methods, different types of features are extracted from source code or binaries as signatures, and the OSS version is identified based on the matching score generated by the number of matched features of different versions. B2SFinder [53] proposes seven different features (e.g., *Strings, Exports, Switch/case, etc*) extracted from source code and matches them in the target binary. To some extent, it mitigates the impact of version strings missing since more features have been utilized. Nevertheless, code updates may be subtle between OSS versions, some versions may only update part of functions or add/delete internal functions which have lost their names in stripped binaries, making it hard to identify the OSS version precisely (**P2**). LibvDiff is not the first one who introduced the binary code similarity detection (BCSD) technique for version identification, LibDB [41] combines basic features (i.e., exported function names and string literals) and function embeddings generated by Gemini [45] together to determine candidate OSS, followed by calculating the version scores based on measuring similarities between call graphs. However, it falls short of addressing the deteriorating precision in function retrieval, arising from an overabundance of functions. Additionally, feature matching-based methods have their own efficiency limitations, with an increasing number of candidate versions and the in the target binary, the time needed for version identification will significantly increase (**P3**).

To address the aforementioned problems, we introduce LibvDiff, an innovative approach based on version differences that can precisely and efficiently identify versions of OSS. Unlike methods that rely solely on version strings in OSS, LibvDiff takes a different approach by inspecting code changes across distinct versions of OSS. To this end, LibvDiff initially extracts changes in functions and string literals by comparing the source code across different versions. Subsequently, it confirms their existence in compiled binaries and employs them to establish the version differences, which consist of version-sensitive features (**for P1**). To enhance the efficiency of the identification process, we propose a candidate version filter (CVF) that rapidly filters out irrelevant versions based on a specially-designed version coordinate system and basic features (i.e., exported function names and string literals). The version coordinate system quantifies the gaps between versions by mapping them to a two-dimensional plane (**for P3**). To capture the subtle differences across versions (i.e., the change of function content), LibvDiff adopts the BCSD technique to retrieve version-sensitive functions in the target binary, and compare them with functions in candidate versions to determine the OSS version (**for P2**). Additionally, we propose an anchor path filter (APF) that utilizes the call relationship between the exported functions and other internal functions to minimize the number of functions required in the target binary. This filter offers two benefits: it saves time by avoiding the generation of features for unnecessary functions, and it enhances the precision of function retrieval since lots of irrelevant functions are filtered out (**for P3**).

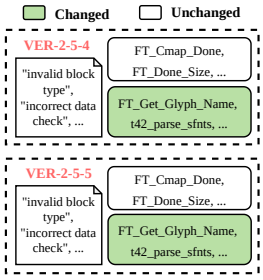
We conducted a comprehensive cross-compilation experiment to demonstrate the scalability of LibvDiff. We extracted version signatures from one compilation setting and identified versions from binaries compiled using different architectures (X86, X64, ARM, and PPC) and optimizations (O0, O1, O2, and O3). We divided the open-source software into two datasets, D1 and D2, with D1 containing version strings and D2 not containing them. Our results show that LibvDiff achieved exceptional precision in version identification across all compilation settings, outperforming the second-best baseline method B2SFinder by 54.6% in precision for D2. To demonstrate the improvement of LibvDiff's CVF and APF, we conducted an ablation experiment. The results demonstrate that they enhance precision by 29.5% and 25.9%, respectively, while also reducing identification time cost by 73.9% and 36.9%, respectively. We apply LibvDiff to the real-world firmware dataset with 351 distinct firmware images and conducted a comprehensive analysis of the identification results. 27,440 vulnerabilities caused by 286 CVEs are successfully detected, and 46.2% of them are still affecting the firmware even though the patches are available when the firmware images are released. In addition, we find that developers exhibit substantial variations in their updates to various OSS, while different vendors may also employ identical OSS binaries.

In summary, we have made the following contributions:

- We propose a novel approach LibvDiff that mainly utilizes version differences and the BCSD technique to identify the OSS version precisely even though version strings are absent. It has two effective modules CVF and APF, which can quickly filter out irrelevant versions and features to improve precision and efficiency. The code and dataset of LibvDiff are released at https://github.com/GentleCP/LibvDiff-public
- We conduct comprehensive experiments on 2,688 binaries across different compilation settings in two datasets. The experimental results show that LibvDiff achieves 94.5% and 78.7% precision in two datasets with only 12.15 seconds per binary, outperforming the state-of-the-art works by 54.2% and 160.3%, respectively.
- We performed version identification on 351 real-world firmware and detected 27,440 vulnerabilities. Additionally, by analyzing open-source software versions, we drew three conclusions that can inspire and guide firmware security analysis.

2 MOTIVATION EXAMPLE

We illustrate our motivation using the example of two freetype versions, *VER-2-5-4* and *VER-2-5-5*, as shown in Figure 1. Figure 1a provides a summary of the differences between the two versions, including changes in string literals and functions. Neither version contains any version-related strings, making it impossible to identify the version using such strings. Moreover, there are no apparent changes in simple features like string literals or function names between the two versions. Only two functions, *FT_Get_Glyph_Name* and *t42_parse_sfnts*, have been updated from *VER-2-5-4* to *VER-2-5-5*. We use the function shown in Figure 1b to illustrate the code update in the *FT_Get_Glyph_Name* function between the two versions.



(a) Version differences between VER-2-5-4 and VER-2-5-5.

```
FT_Get_Glyph_Name(...)
{
  ...
- if ( (FT_Long)glyph_index > face->num_glyphs ||
- !FT_HAS_GLYPH_NAMES( face ) )
+ if ( (FT_Long)glyph_index >= face->num_glyphs )
+ return FT_THROW( Invalid_Glyph_Index );
+
+ if ( !FT_HAS_GLYPH_NAMES( face ) )
+ return FT_THROW( Invalid_Argument );
  ...
}
```

(b) Changed code lines of FT_Get_Glyph_Name

Figure 1: A motivation example on freetype. Only some functions have been updated from VER-2-5-4 to VER-2-5-5

Traditional version identification approaches, even those that consider not only version strings but also function names in the dynamic table, fail to identify the software versions in this case. Therefore, it is necessary to detect code changes to differentiate between similar versions. To address this problem, we designed *LibvDiff*. Its goal is to differentiate between various versions by capturing the differences across them and verifying their presence in the target binary, allowing for appropriate identification of the true version.

3 METHODOLOGY OVERVIEW

The core idea of *LibvDiff* is to capture the differences across versions and verify their presence in the target binary to identify the OSS version. Figure 2 shows the workflow of *LibvDiff*, which consists of two primary stages: version signature generation, and OSS version identification.

During the stage of version signature generation, *LibvDiff* focuses on producing three categories of version signatures for the provided source code and compiled binaries of different versions: binary features, version differences, and version coordinates. Binary features are used to capture the code changes, version differences serve to document the altered features among versions, and version coordinates measure the gap between versions. Initially, *LibvDiff* generates the requisite binary features from the compiled binaries (§ 4.1). Then, it extracts version-sensitive features (VSFs) by contrasting the source code of multiple versions and verifying their existence in binaries (§ 4.2). Finally, *LibvDiff* constructs a version coordinate system and represents the gaps between versions by mapping them into the system (§ 4.3).

In the OSS version identification stage, for a given binary, *LibvDiff* identifies the version of a specific OSS by verifying the presence of the version signatures. From the beginning, *LibvDiff* generate basic features and version coordinates of the binary in the same way as the previous stage. Subsequently, candidate versions are obtained by selecting versions in proximity to the target binary based on the version coordinates and basic features (§ 5.1). Ultimately, the version is confirmed by comparing the candidate versions one by one with a well-designed RVG algorithm (§ 5.2).

4 VERSION SIGNATURE GENERATION

In order to distinguish between different versions of OSS, we generate three types of binary features initially, which can be used for lightweight filtering and or to improve efficiency and precision in version identification. The changes of functions and string literals between the various versions are summarized as *version difference*. Using the difference, we construct a version coordinate system where each OSS version is denoted by a unique coordinate.

4.1 Binary Feature Generation

To capture the code changes across various versions, *LibvDiff* primarily generates three types of features directly from the compiled binaries: basic features, function embeddings, and anchor paths.

4.1.1 Basic Feature Generation. The basic features we extract from the binary of v_i is represented as $BF_{v_i} = (S_f^{v_i}, S_s^{v_i}, S_e^{v_i})$, where the three sets are function names, string literals, and exported function names, respectively. Rather than extracting these directly from the source code, we obtain them from the compiled binary to ensure that all relevant features are included. This is because certain parts of the source code, such as test and example code, may not be compiled into the main binary [29]. These basic features are used to filter candidate versions in a coarse-grained manner compared to detecting code changes, as described in § 5.1.2.

4.1.2 Function Embedding Generation. Function embeddings are vector representations of functions generated by the BCSD tool, which are used for calculating function similarity. For version v_i, the function embeddings are represented as $FE_{v_i} = (h_1, h_2, ..., h_n)$, where h_j represents the vector for the j-th function. In *LibvDiff*, we use Asteria [49] as the BCSD tool to generate function embeddings which are used for calculating function similarity between target binaries and referenced OSS binaries (i.e., compiled binaries). This is because Asteria offers support for cross-optimization and cross-architecture function similarity detection and has been found to outperform other similar tools [49].

4.1.3 Anchor Path Generation. Using the BCSD tool to calculate similarities between all functions of two binaries can be time-consuming. Moreover, as the number of functions increases, the precision of finding homologous functions decreases. To address these challenges, we introduce the anchor path in this section and utilize it to eliminate irrelevant functions as in § 5.2.1.

The anchor path (AP) can be defined as a path that originates from an exported function and terminates at any internal (i.e., non-exported) function within the call graph of a stripped binary. It serves as a dependable mechanism for identifying the target (internal) functions by leveraging the sequential call relationships associated with exported functions. Its primary objective is to locate potential target functions by identifying similar sequential call relationships, thereby effectively reducing the search space for function retrieval during the process of version confirmation. This capability enables us to roughly establish function correspondences across different binary versions and is based on the fact that both the exported functions and the call graph retain their stability and integrity even in stripped binaries, as demonstrated in previous research works [29, 41, 48, 50, 53].

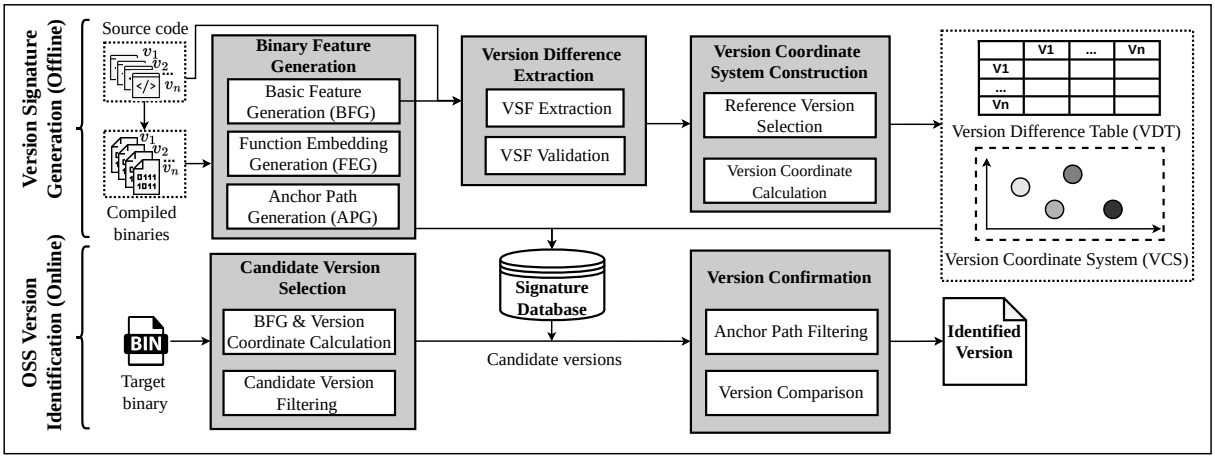


Figure 2: Workflow of LibvDiff

In our work, the call graph is precisely defined as a directed graph $CG = (V, E)$, where $V = \{f_1, ..., f_n\}$ represents the set of function nodes, and $E = \{(f_i, f_j) | f_i \in V, f_j \in V\}$ denotes the set of function call edges. Based on the call graph, the basic unit of AP named anchor path edge (APE) is defined as $(f_i, c, f_j), f_i \in V, f_j \in V, c \in \{e, r\}$, where c represents the call direction and e, r denote the callee direction and the caller direction, respectively. For instance, if function f_i calls function f_j, the APE from f_i to f_j is (f_i, e, f_j) (i.e., f_j is the callee function of f_i) and the APE from f_j to f_i is (f_j, r, f_i) (i.e, f_i is the caller function of f_j).

With these definitions in place, the AP originates from the exported function f_e^k (i.e., anchor node) and leads to a target function f_n is formally defined based on a sequence of APEs as follows:

$$AP = ((f_e^k, c_1, f_1), (f_1, c_2, f_2), ..., (f_{n-1}, c_n, f_n)), \quad f_i \in V, c_j \in \{e, r\}, f_e^k \in S_e \tag{1}$$

For example, in Figure 4d, the AP of 4_i-e-6_i-r-5_i is represented as $((4_i, e, 6_i), (6_i, r, 5_i))$. Algorithm 1 outlines the process of generating anchor paths using the provided exported functions and the call graph. For every exported function f_e^k, we recursively traverse its callee and caller until we encounter another exported function or exhaust all nodes in the graph. During the traversal process, we update the AP by appending the APE from current function f_{cur} to its callee/caller into the end of the AP from f_e^k to f_{cur}. Functions *GetCallee* and *GetCaller* at lines 7 and 13 obtain the callee and caller for a given function in the call graph separately, while the operator "+" at lines 11 and 17 append an APE to the AP.

Algorithm 1: Anchor path generation

Input: The set of exported function names $S_e^{v_i}$ and the call graph CG
Output: Anchor paths

```
1  Initialize an empty queue Q and a 2D array AP;
2  for f_e^k ∈ S_e^{v_i} do
3      S_visited ← ∅;
4      Q.push(f_e^k);
5      while Q do
6          f_cur ← Q.get();
7          for callee ∈ GetCallee (CG, f_cur) do
8              if callee ∉ S_e^{v_i} and callee ∉ S_visited then
9                  Q.push(callee);
10                 S_visited.push(callee);
11             AP_{f_e^k,callee} ← AP_{f_e^k,f_cur} + (f_cur, e, callee);
12         end
13         for caller ∈ GetCaller (CG, f_cur) do
14             if caller ∉ S_e^{v_i} and caller ∉ S_visited then
15                 Q.push(caller);
16                 S_visited.push(caller);
17             AP_{f_e^k,caller} ← AP_{f_e^k,f_cur} + (f_cur, r, caller);
18         end
19     end
20 end
21 return AP
```

4.2 Version Difference Extraction

Version differences denote the modified features (i.e., functions and string literals) across various versions. It is difficult to distinguish whether a feature is modified through version updating or compilation options from comparing binaries across versions directly, owing to varying compilation optimizations, architectures, etc. In contrast, it is easier to extract VSFs by analyzing the differences between two specific versions in the source code. Based on the above considerations, we extract VSFs from the source code and validate their presence based on function names and strings generated from compiled binaries.

4.2.1 VSF Extraction. To precisely extract VSFs in the source code, we analyze the git repository by parsing *git diff* results of two different versions and record features which have been modified as original VSFs. Furthermore, for the sake of extraction efficiency, we exclusively extract original VSFs across adjacent versions since the number of OSS version pairs can be numerous (e.g., OSS with 10 versions could have 45 (C_{10}^2) version pairs). For non-adjacent

versions, we construct their original VSFs by taking the union of original VSFs of all adjacent versions among them. To represent these version-sensitive features, we employ a set notation denoted as $\mathcal{F} = (S_f^{v_i,v_j}, S_s^{v_i,v_j}) = (\{f_1, f_2, ..., f_m\}, \{s_1, s_2, ..., s_n\})$. In this representation, $S_f^{v_i,v_j}$ and $S_s^{v_i,v_j}$ are the set of version-sensitive functions (e.g., f_i) and version-sensitive strings (e.g., s_j), respectively.

4.2.2 VSF Validation. To integrate information from both source and binary aspects, we adopt a straightforward approach by calculating the intersection and difference between the sets of features. In particular, we introduce the version difference between versions v_i and v_j as $VD_{v_i,v_j} = (S_{af}^{v_i,v_j}, S_{df}^{v_i,v_j}, S_{uf}^{v_i,v_j}, S_{as}^{v_i,v_j}, S_{ds}^{v_i,v_j})$ to represent the integrated information, where the five sets denote the added functions, deleted functions, updated functions, added strings and deleted strings, respectively. For every VSF in $\mathcal{F}$, we put it into separate sets by confirming its presence in the basic features of v_i and v_j, as described in Equation (2).

$$VD_{v_i,v_j} = \begin{cases} S_{af}^{v_i,v_j} = S_f^{v_i,v_j} \cap (S_f^{v_j} \setminus S_f^{v_i}) \\ S_{df}^{v_i,v_j} = S_f^{v_i,v_j} \cap (S_f^{v_i} \setminus S_f^{v_j}) \\ S_{uf}^{v_i,v_j} = S_f^{v_i,v_j} \cap (S_f^{v_j} \cap S_f^{v_i}) \\ S_{as}^{v_i,v_j} = S_s^{v_i,v_j} \cap (S_s^{v_j} \setminus S_s^{v_i}) \\ S_{ds}^{v_i,v_j} = S_s^{v_i,v_j} \cap (S_s^{v_i} \setminus S_s^{v_j}) \end{cases} \quad (2)$$

4.3 Version Coordinate System Construction

To narrow down the possible version range of the target binary and speed up the efficiency of version identification, we introduce a version coordinate system (VCS) to quantify the positional relationship among all versions. VCS is made up of reference versions and version coordinates of all versions. The gaps between all versions and the reference versions are reflected throughout their coordinates in VCS. For the target binary, we can quickly locate similar versions by mapping it into VCS and filtering out versions that are too far away. The construction of VCS consists of two parts: reference version selection and version coordinate calculation.

4.3.1 Reference Version Selection. Reference versions serve to provide version differences and build version axes. Initially, versions are sorted by their release dates, and then two versions that contain sufficient VSFs to detect modified features in other versions are selected. For the sake of convenience, the oldest version v_o and the newest version v_n are chosen as reference versions.

4.3.2 Version Coordinate Calculation. In VCS, the version coordinate of each version is a tuple, where the values indicate the gaps between the target binary and the two reference versions. For the sake of efficiency, only the exported function names are used as features for calculating version coordinates, as they are stable and will be preserved in the binary. The version coordinate of v_i with respect to the reference versions v_o and v_n can be calculated as follows:

$$VC_{v_i} = (|S_{af}^{v_o,v_n} \cap S_e^{v_i}|, |S_{df}^{v_o,v_n} \cap S_e^{v_i}|) \quad (3)$$

The matching number of added and deleted exported function names reflects the gap from v_i to v_o and v_i to v_n, respectively.

Example. Figure 3 gives a vivid example of VCS constructed for freetype, the versions range from $v2.4.0$ to $v2.8.1$. The coordinates

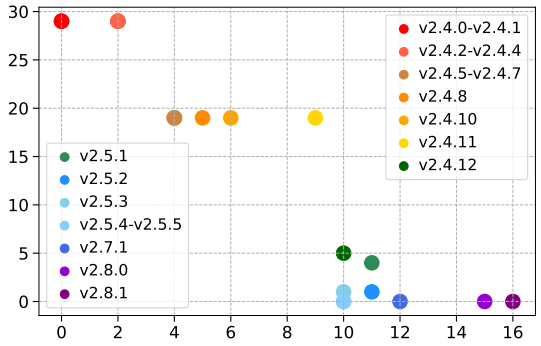


Figure 3: An example of version coordinates system for freetype, where v_o is $v2.4.0$, and v_n is $v2.8.1$

of each node in Figure 3 are calculated based on the Equation (3). As can be seen, the proximity of the release date correlates with the proximity of the version's location in VCS. For example, $v2.8.1$ is far away from $v2.4.0$ since many exported functions have been added or deleted from $v2.4.0$ to $v2.8.1$.

5 OSS VERSION IDENTIFICATION

OSS version identification aims to determine the version used in the target binary B. For the sake of efficiency, a large number of versions are filtered out by locating the position of the target binary in VCS, rather than directly comparing them with B. Subsequently, the remaining versions are considered candidate versions and are compared with B to determine the final identified version.

5.1 Candidate Version Selection

Candidate version selection selects the versions that are likely to be used by the target binary from all versions in the database. The key idea is filtering versions that are far away from the target binary B by mapping B into VCS and checking versions with basic features.

5.1.1 Basic Feature Generation & Version Coordinate Calculation. We generate the basic features (i.e., exported function names and string literals) and calculate the version coordinate of B (denoted as $VC_B = (X_B, Y_B)$) in the same manner as in § 4.1.1 and § 4.3.2.

5.1.2 Candidate Version Filtering. To narrow down the version scope, we propose a candidate version filter (CVF) that filters candidate versions with two steps. First, for a version v_i, whose version coordinate is $VC_{v_i} = (X_i, Y_i)$, it is kept when VC_{v_i} meets the conditions:

$$\frac{abs(X_i - X_B)}{X_B} \leq T_1 \; and \; \frac{abs(Y_i - Y_B)}{Y_B}) \leq T_1 \quad (4)$$

where T_1 denotes the threshold for similar versions locating. In other words, any version that is sufficiently close to B will be retained as a potential candidate. Second, we further filter versions with newly added and deleted basic features in version differences by comparing versions one by one. If two versions cannot be distinguished with added and deleted basic features directly, both of them will be kept as candidate versions. Otherwise, the version that matches fewer features is filtered out. Ultimately, candidate

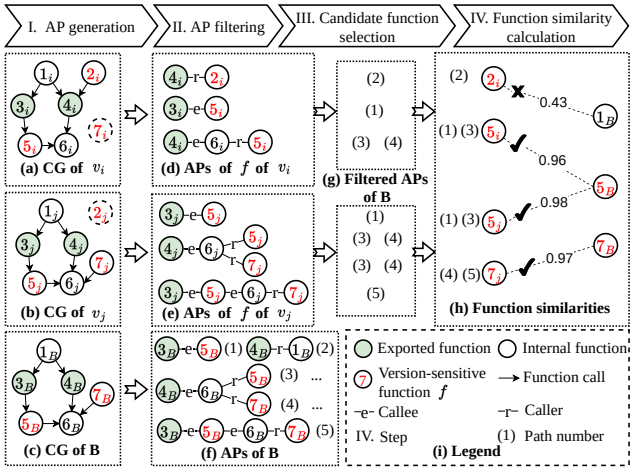


Figure 4: An example of RVG calculation

versions are sorted by release date order and forwarded to the subsequent phase for confirmation.

5.2 Version Confirmation

In this section, we further compare candidate versions with the target binary B to identify the OSS version precisely based on candidate versions filtered by CVF. The core idea of version confirmation is that the version used by the target binary can match more signatures than other versions. It has two phases: anchor path filtering and version comparison.

5.2.1 Anchor Path Filtering. The objective of anchor path filtering is to identify candidate functions within a provided version-sensitive function f. It is based on the anchor path filter (APF) whose core idea is that homologous functions have the same APs and can be used to eliminate potential functions from consideration. Since the internal function names are lost in stripped binaries, we further introduce the concept of call direction sequence (CDS) to find functions in the target binary B. The CDS ignores the names of the specific functions and comprises the exported function along with all call directions within the AP defined in Equation (1). It is defined as follows:

$$CDS = (f_e^k, (c_1, c_2, ..., c_n)), f_e^k \in S_e^{v_i}, c_i \in \{e, r\} \quad (5)$$

With the help of CDS, we are able to find the target function f at the end of the AP by starting from the exported function f_e^k and following the call directions. It should be noted that one function may have multiple APs starting from the different exported functions. In the first step, we generate all APs of B as described in § 4.1.3. Subsequently, for each f, we collect its APs and corresponding CDSs as two sets: $S_{AP} = \{AP_1, AP_2, ..., AP_n\}$ and $S_{CDS} = \{CDS_1, CDS_2, ..., CDS_n\}$. Afterward, APs of B whose CDS is not present in S_{CDS} will be filtered out. Eventually, we count the number of the end nodes of the remaining APs of B and choose the nodes with the highest frequency employed as the candidate functions for f.

Example. For version-sensitive function 5_i in Figure 4d, its anchor paths are 4_i-*e*-6_i-*r*-5_i and 3_i-*e*-5_i and the corresponding CDSs are $(4_i, (e, r))$ and $(3_i, (e))$, respectively. Thus, the filtered anchor paths of B are 3_B-*e*-5_B, 4_B-*e*-6_B-*r*-5_B, and 4_B-*e*-6_B-*r*-7_B since their CDSs meet the requirement. Finally, 5_B is selected as the candidate function of 5_i due to its frequency being higher than 7_B.

Algorithm 2: RVG algorithm

Input: Candidate versions v_i, v_j and the corresponding binaries B_{v_i}, B_{v_j} target binary B
Output: RVG from B to v_i, v_j, denoted as RVG_{B,v_i}, RVG_{B,v_j}

```
1  RVG_{B,v_i}, RVG_{B,v_j} ← 0, 0;
2  S_af^{v_i,v_j}, S_df^{v_i,v_j}, S_uf^{v_i,v_j} ← GetFuncDiff(v_i, v_j);
3  for f ∈ S_af^{v_i,v_j} do
4      if FuncRetrieval (f, B_{v_j}, B) ≥ T_2 then
5          RVG_{B,v_i} ← RVG_{B,v_i} + 1;
6  end
7  for f ∈ S_df^{v_i,v_j} do
8      if FuncRetrieval (f, B_{v_i}, B) ≥ T_2 then
9          RVG_{B,v_j} ← RVG_{B,v_j} + 1;
10 end
11 for f ∈ S_uf^{v_i,v_j} do
12     simi_{f_i} ← FuncRetrieval (f, B_{v_i}, B);
13     simi_{f_j} ← FuncRetrieval (f, B_{v_j}, B);
14     if simi_{f_i} ≥ T_2 or simi_{f_j} ≥ T_2 then
15         RVG_{B,v_i} ← RVG_{B,v_i} + (1 − simi_{f_i})
16         RVG_{B,v_j} ← RVG_{B,v_j} + (1 − simi_{f_j})
17 end
18 return RVG_{B,v_i}, RVG_{B,v_j}
19 Function FuncRetrieval(f_src, B_src, B_tgt):
20     S_f ← AnchorPathFiltering(f_src, B_src, B_tgt);
21     return MaxBCSD (f_src, S_f);
22 End Function
```

5.2.2 Version Comparison. We adopt the concept of reference version gap (RVG) to quantify version differences for two provided candidate versions, v_i and v_j, in relation to the target binary B. It allows us to determine which version is more proximate to B. By taking into account the version differences between v_i and v_j, we use the RVG algorithm 2 to estimate RVG from B to v_i and v_j. The underlying principle of this algorithm is to detect the added and deleted functions in B and evaluate the similarities of the updated functions between B and the provided candidate versions v_i and v_j. Based on the VD_{v_i,v_j}, we try to retrieve the added and deleted functions in B (line 2 to 10), and calculate the gap between the updated functions and two versions (line 11 to 18). The function *GetFuncDiff*, located at line 2, obtains the version difference of functions for two specified versions v_i and v_j. It first queries the version-sensitive functions and function names of two versions and then calculates the version difference of functions based on the Equation (2). The function *FuncRetrieval*, located at line 4, retrieves the target function (i.e., added, deleted, or updated function) in B for the given function in the source binary. It first uses the APF to filter candidate functions as described in 5.2.1 (i.e., function *AnchorPathFiltering* located at line 20) and then retrieves the function with the maximum similarity based on the BCSD tool (i.e., function *MaxBCSD* located at line 21).

Table 1: The composition of OSS version dataset

Category	OSS	Version (#)	Binary (#)
Document formatting	libxml2	10	160
	libexpat	14	224
Compression	c-blosc	19	304
	Zlib	13	208
SDK	aws-c-common	26	416
Encryption and TLS protocol	mbedtls	31	496
	OpenSSL	20	320
Font rendering	freetype	20	320
Image processing	libpng	15	240
Total	-	**168**	**2688**

Example. An illustrative example of RVG calculation is presented in Figure 4. Figure 4a to Figure 4c depict the call graph of v_i, v_j, and B, respectively. From the beginning (step I), we generate the APs of two versions and the target binary B as Figure 4d to Figure 4f shows. Afterwards (in step II), we select the APs of the target binary (B) for each version-sensitive function f by filtering those whose CDS fails to meet the condition outlined in 5.2.1, as illustrated in Figure 4g. Next (step III), candidate functions are selected for each f by considering the frequency of end nodes in the filtered APs. Each f and its corresponding candidate functions are $((2_i, 1_B), (5_i, 5_B), (5_j, 5_B), (7_j, 7_B))$ as in Figure 4h. Ultimately (step IV), the RVGs from B to v_i and v_j are calculated based on the similarities between version-sensitive functions and their corresponding candidate functions as in Figure 4h, which are $1 + (1 - 0.96) = 1.04$ and $1 - 0.98 = 0.02$, indicating v_j is more likely to be used in B.

With the aforementioned RVG algorithm, we initialize the first version v_1 as the current identified version v_{cur}. Then we traverse the remaining versions and compare them with v_{cur} one by one. For each version pair (v_i, v_{cur}) (e.g., (v_2, v_{cur})), we calculate RVG_{B,v_i} and $RVG_{B,v_{cur}}$, respectively. If RVG_{B,v_i} is less than $RVG_{B,v_{cur}}$, indicating that v_i is more proximate to B than v_{cur}, then we update the current identified version v_{cur} to v_i. Eventually, the version that holds the relative minimum RVG is regarded as the identified version for B.

6 EXPERIMENTAL SETUP

6.1 Dataset and Experiment Settings

We collected the source code of 9 widely used OSS from Github [13] and their official websites as Table 1 shows. These OSS can be categorized into different groups based on their functionality, such as document formatting, and compression. We compiled the source code into binaries using different compilation options, including 4 architectures (ARM, X86, X64, PPC) and 4 optimization levels (O0, O1, O2, O3) with GCC v9.4.0. In total, we obtained 168 distinct versions of all OSS, resulting in 2688 (168 * 16) binaries as shown in Table 1. After manual verification on arm-compiled binaries, we found that three of the collected OSS did not contain version strings, namely *libxml2* [15], *aws-c-common* [6], and *freetype* [16]. To evaluate our method's version identification ability, we constructed two datasets, **D1 comprising OSS with version strings** and **D2 comprising OSS without version strings**. There are two categories of binaries used in all version identification experiments: the source binaries (i.e., binaries with the source option) and the target binaries (i.e., binaries with the target option). The former is employed for the extraction of binary features, while the latter is utilized for version identification tests. We designed three experiment settings to test the scalability of each method across different compilation options. **1) CO (cross optimization).** The source binaries and the target binaries exhibit different optimizations (i.e., $4 * 3 = 12$ combinations) while sharing the same architecture. Since most binaries in IoT firmware are using ARM, we select ARM as the same architecture in CO experiment. **2) CA (cross architecture).** The source binaries and the target binaries exhibit different architectures (i.e., $4 * 3 = 12$ combinations) while sharing the same optimization. Since most default compilation optimization levels are O2, we select O2 as the same optimization level in the CA experiment. **3) CO+CA.** The source binaries and the target binaries are in different architectures and optimization simultaneously. More specifically, for CO and CA experiments, we use binaries in one compilation option (e.g., O0/X86) to extract features and identify binaries in the other three compilation options. For CO+CA experiment, we use binaries in arm O2 to extract features and identify binaries in other compilation options, namely PPC with O0, O1, and O3, X64 with O0, O1, and O3 and X86 with O0, O1, and O3, respectively. For each source and target option pair (e.g., O2 to O3), we employ the precision ($P = \frac{m}{N}$) as the metric to assess the effectiveness of each method, where N is the number of tested OSS binaries and m is the number of correct identification results. Ultimately, we calculated the final entry in Table 2 by taking the average precision of all compilation option pairs (e.g., ARM-O0 and ARM-O2) of all OSS (e.g., OpenSSL) in the specific experiment (e.g., CO)

6.2 Baselines

We evaluate *LibvDiff* against recent state-of-the-art methods, including three academic approaches and two industrial tools:

- **OSSPolice** [29]. A state-of-the-art approach that primarily relies on version string matching to identify OSS versions.
- **B2SFinder** [53]. A state-of-the-art source to binary OSS detection method by comparing 7 constant features between source code and binaries.
- **LibDB** [41]. A tool utilizes the BCSD technique and call graph to detect OSS and identify versions.
- **BinaryAI** [38]. A state-of-the-art commercial tool that supports software composition analysis.
- **Cybellum** [10]. A mature commercial software that supports software composition analysis.

For the above baselines, we use their original implementations and default settings. The version identification results of OSSPolice, B2SFinder, and LibDB rely on the number of matched features of different versions. The BCSD threshold in LibDB is set to the same value as indicated in their original paper, specifically 0.8. BinaryAI and Cybellum are commercial tools with predefined parameters that cannot be adjusted. To evaluate the contribution of CVF and APF, we further set up four configurations:

- **LibvDiff**: the original LibvDiff.
- **LibvDiff-C**: LibvDiff without CVF.
- **LibvDiff-A**: LibvDiff without APF.
- **LibvDiff-CA**: LibvDiff without CVF and APF.

Table 2: The precision of OSS version identification of two Dataset in three different experiments

Dataset	D1			D2		
Experiment	**CO**	**CA**	**CO+CA**	**CO**	**CA**	**CO+CA**
OSSPolice	0.98	0.84	0.84	0.00	0.17	0.22
B2SFinder	0.56	0.56	0.56	0.49	0.51	0.51
BinaryAI	0.97	0.85	0.85	0.20	0.21	0.21
Cybellum	0.80	0.70	0.71	0.44	0.47	0.49
LibDB	0.21	0.15	0.11	0.21	0.16	0.09
LibvDiff-CA	0.88	0.85	0.83	0.44	0.44	0.46
LibvDiff-C	0.94	0.92	0.89	0.74	0.68	0.73
LibvDiff-A	0.99	0.94	0.94	0.73	0.74	0.73
LibvDiff	**1.00**	**0.95**	**0.95**	**0.79**	**0.78**	**0.79**

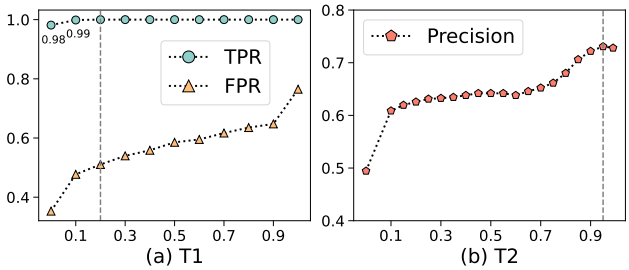


Figure 5: Threshold selection

6.3 Implementation

Environment and Tools. *LibvDiff* is implemented in 2,759 lines of python code. We employ IDA Pro 7.5 [5] to generate binary features and pydriller [11] to analyze the source code of OSS. We retrain the Asteria with the default settings in its paper. All the experiments were conducted on a server with Intel Xeon 128-core 3.0GHz CPU, 1 TB memory, and 4 GeForce RTX 3090.

Threshold Selection. We explored various values for T_1 and T_2, ranging from 0 to 1, to determine the optimal thresholds. For T_1, we use true positive rate $TPR = \frac{TP}{TP+FN}$ and false positive rate $FPR = \frac{FP}{FP+TN}$ as the metrics. The TP/FP represents the number of true/false versions that have been identified as candidate versions, while the FN/TN signifies the number of true/false versions that are filtered out by CVF. Through the optimization of both metrics, our objective is to achieve precise identification of the correct version (i.e., TPR is 1), while simultaneously maximizing the filtration of incorrect versions (i.e., low FPR). Based on the result in Figure 5a, we select a threshold value of 0.2 for T_1 as it achieves a TPR of 1 while maintaining a lower FPR. For T_2, we rely on the precision of the version identification as the metric, without implementing CVF. Based on the result in Figure 5b, the precision increases as T_2 rises due to the number of false positive functions decreasing and reaches the highest when T_2 is 0.95. However, when T_2 is close to 1.0, the precision starts to decline due to an increase in the number of false negative functions. Consequently, we employ T_2 as 0.95.

7 EVALUATION

Our evaluation aims to answer to the following research questions.

- **RQ1.** How precise is *LibvDiff* on different version identification experiment settings compared to related works? And to what extent can the precision of identification be improved by CVF and APF (i.e., ablation study)?
- **RQ2.** How efficient is *LibvDiff* for version identification compared to related works? And what are the effects of CVF and APF in improving efficiency (i.e., ablation study)?
- **RQ3.** What is the usage of different versions of OSS in real-world IoT devices? Will OSS be promptly updated to mitigate vulnerabilities?

RQ1 and **RQ2** compare *LibvDiff* with baselines and evaluate whether *LibvDiff* meets **P1, P2, P3** described in § 1. **RQ3** is an application of *LibvDiff*, aiming to analyze the OSS version usage with corresponding vulnerability and patching status in real-world firmware.

7.1 RQ1: Precision Evaluation

Table 2[1] shows the precision of version identification results in three experiments. The first column reports the names of methods, and the last six columns represent the precision in different experiment settings. *LibvDiff* outperforms all the baseline methods by considerable margins, especially when no version strings are available (i.e., precision on D2). For CO + CA experiment, *LibvDiff* outperforms the second-best method, BinaryAI and B2SFinder, by 11.7% and 54.6% on D1 and D2 separately, indicating better effectiveness for OSS version identification. The results demonstrate version string is an important factor affecting the effectiveness of version identification for most methods. The precision of OSSPolice, BinaryAI, and Cybellum rapidly decreases when there are no version strings available.

Ablation Results. As depicted in Table 2, *LibvDiff*-C and *LibvDiff*-A significantly enhance the precision by an average of 25.9% and 29.5%, respectively, in comparison to *LibvDiff*-CA. This is achieved by filtering out irrelevant functions and versions, respectively. Notably, for the **D2** dataset, *LibvDiff*-C and *LibvDiff*-A demonstrate an average precision improvement of 60.5% and 65.1%, respectively.

False Prediction Analysis. *LibvDiff*'s false identification results are mainly due to two reasons. *(1) No code changes between binary versions.* We find that some code changes between versions are not compiled into binaries (e.g., example, test code), or some versions are only updated to adapt to different platforms. The false identification for these versions is reasonable and it will not impact the discovery of vulnerabilities. *(2) Undetectable subtle changes.* Some versions with small modifications cannot be effectively identified due to the limitations of the BCSD tool. Specifically, non-syntactic structural changes, such as the modification of parameters and local data types between versions are challenging to identify in Asteria.

> **Answer to RQ1.** *LibvDiff* exhibits exceptional precision in all experiment settings, with an average increase in precision of 54.2% and 160.3% in two datasets in the CO+CA experiment. By employing CVF or APF, *LibvDiff* enhances its precision by 29.5% and 25.9%, in contrast to *LibvDiff* without them, respectively, and achieve the optimal results (0.95 in D1 and 0.79 in D2) when both utilized in tandem.

[1]The table in detail (i.e., with specific target option) is available at https://github.com/GentleCP/LibvDiff-public/blob/master/imgs/RQ1_data_detail.png

Table 3: The time cost of different methods. "FG" and "VI" denote feature generation and version identification, respectively.

Method	D1			D2		
	FG (s)	VI (s)	Total (s)	FG (s)	VI (s)	Total (s)
OSSPolice	**0.06**	0.10	**0.16**	**0.06**	**0.06**	**0.12**
B2SFinder	6.85	8.40	15.25	8.50	4.53	13.03
LibDB	35.72	118.60	154.30	34.36	137.60	171.96
BinaryAI	-	-	17.93	-	-	26.74
Cybellum	-	-	3.86	-	-	5.36
LibvDiff-CA	62.40	3.91	66.31	115.90	10.65	126.55
LibvDiff-C	40.80	2.74	43.55	72.57	5.54	78.11
LibvDiff-A	10.61	0.01	10.63	39.56	0.19	39.75
LibvDiff	7.75	**0.01**	7.76	16.46	0.07	16.53

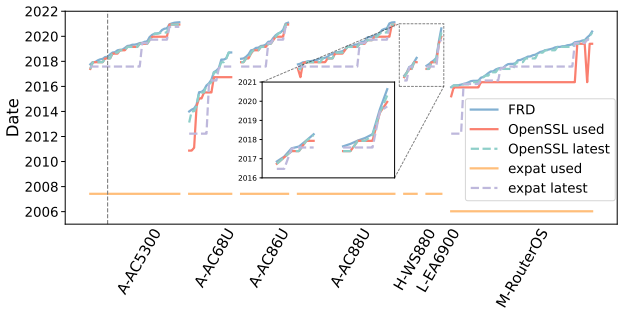
7.2 RQ2: Efficiency Evaluation

Table 3 shows the time cost for the two phases, feature generation and version identification, of each method. The time for feature generation in BinaryAI and Cybellum cannot be ascertained, hence we record the duration of the entire identification process directly. As can be seen, OSSPolice takes the least time cost in two datasets (0.161s, 0.117s), followed by Cybellum, *LibvDiff*, B2SFinder, BinaryAI, and LibDB in that order. The primary factor behind *LibvDiff*'s time consumption is the feature generation process, while LibDB also displays elevated costs in feature generation. The improved efficiency of version identification in *LibvDiff* can be attributed to version-sensitive features, which obviates the need for redundant features. In addition, the utilization of CVF and APF allows *LibvDiff* to efficiently eliminate irrelevant versions and select potential functions for individual comparison, resulting in a significant reduction in time cost for identifying OSS versions. More specifically, *LibvDiff* reduces time cost by 88.3% and 86.9% in D1 and D2 separately, compared to *LibvDiff*-CA which lacks CVF and APF.

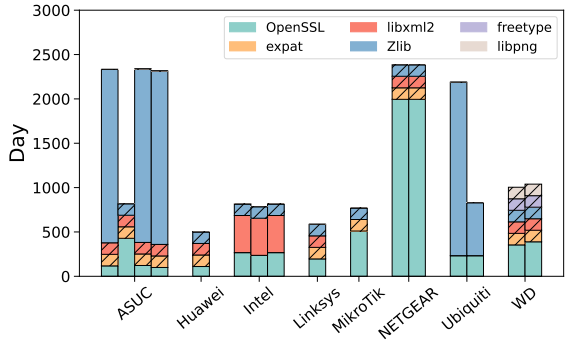
> **Answer to RQ2.** On average, *LibvDiff* takes 12.1 seconds to identify one OSS binary. While *LibvDiff* boasts supreme identification accuracy, it does come with a tradeoff of accuracy and time cost. Through the integration of CVF and APF, *LibvDiff* reduces an average time cost of 73.9% and 36.9%, respectively, compared to *LibvDiff* without either of them.

7.3 RQ3: Real-world Firmware Analysis

we collected firmware images from 8 IoT vendors and extracted binaries using binwalk [7]. To ensure precise identification of the version, we expanded the number of candidate versions in the database by conducting a manual analysis of OSS versions in both the oldest and latest firmware for each model. Additionally, we collected vulnerabilities related to OSS in the dataset from NVD[20] and official OSS websites[8, 12]. Our vulnerability dataset comprises a total number of 473 vulnerabilities, ranging from the year 2002 to 2022. For each identified firmware OSS version, we ascertain the impact of the vulnerability on the corresponding binary by verifying whether it falls within the scope of the affected version of the vulnerability.



(a) The version update trend of OpenSSL and expat in different models, where A is "ASUS", H is "Huawei", L is "Linksys", M is "MikroTik", and FRD denotes firmware release date



(b) The version update frequency of different OSS in different vendors, where WD is "Western-Digital"

Figure 6: The usage of OSS versions in different vendors and corresponding models

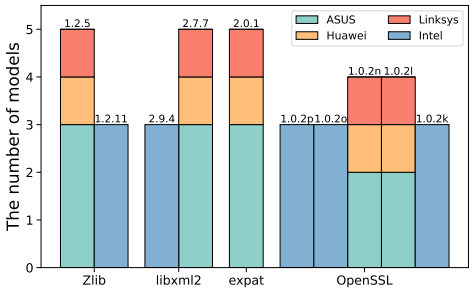


Figure 7: The top10 frequently used OSS binaries with their versions

The statistic results of the real-world dataset analysis are presented in Table 4[2]. The first column gives the statistical metrics which include the minimum (Min), the maximum (Max), and the average (Avg). The second and third columns show the quantity of firmware and binaries, while the fourth and fifth columns are the number of vulnerabilities that satisfy the corresponding conditions. Furthermore, the values of the sixth and seventh columns denote the proportion of vulnerabilities that are ignored even though patches are already available before firmware release and the proportion of vulnerabilities that have been patched during the updating of firmware in the same model. In the end, the eighth column reveals the patching delay. As can be seen, **(1) many outdated versions of**

[2]The table in detail (i.e., results of different vendors) is available at https://github.com/GentleCP/LibvDiff-public/blob/master/imgs/RQ3_data_detail.png

Table 4: The statistic results of the real-world dataset analysis. VDD, FRD, and PRD are abbreviations of vulnerability disclose date, firmware release date, and patch release date, respectively. $VDD < FRD$ denotes that the vulnerability was disclosed before the firmware release. X is "libxml2", E is "expat", S is "OpenSSL", Z is "Zlib", F is "freetype", P is "libpng".

	Firm (#)	Bin (#)	Vulnerability per firmware			Patching status[1]	
			$VDD < FRD$ and $PRD < FRD$ (#)	$VDD < FRD$ (#)	Prop (%)	Prop (%)	Delay (day)
Min	6	24	5.6	6.1	82.5	-	-6.6[2]
Max	59	182	133.2	136.4	98.7	-	837.5
Avg	22	93	47.2	48.7	94.9	X: 21.4, E: 0.0, P: 0.0, S: 81.9, Z: 0.0, F: 0.0	352.3

[1] We confirm whether the vulnerability has been patched by verifying the firmware of the same model but different versions, and only count vulnerabilities whose patches are available before the last firmware version.
[2] The vulnerability has been patched before it was published.

OSS are still in use even though the patches are already available, occupying an average proportion of 94.9%. Moreover, the high patching delay also exposes devices to long-period threats from attackers.

After that, we further investigate the usage of OSS versions in different vendors and corresponding models as in Figure 6. Figure 6a and Figure 6b have shown the version update trend and frequency of multiple OSS. The blue solid line in Figure 6a represents the firmware release date while the other four solid and dashed lines represent the release date of the OSS versions used in firmware and the latest OSS versions before firmware release, respectively. Take the position where the gray dashed line intersects the curve in the graph as an example, it means when *ASUS-AC5300 384.5.0* released on 2018/05/14, it used the latest OpenSSL version which was released on 2018/03/27, and it still used the expat which released on 2007/06/05, even if the latest expat was released on 2017/08/02. Additionally, lines within the corresponding segment (i.e., separated by white spaces) pertain to the identical model, such as ASUS-AC68U. Figure 6b shows the version update frequency of different OSS, counting by the average number of days. Each bar in Figure 6b denotes the different OSS version update frequencies in the same model of a specific vendor. For example, the first bar indicates that OpenSSL used in ASUC RT-AC5300 takes 117 days on average for version updates while Zlib takes 1955 days. Noted that any shadowed bar (i.e., bar with "//") represents that OSS has not been updated at all. Based on the analysis of the above two figures, we can find that **(2) developers have significant differences in their updates to different OSS.** OpenSSL demonstrates a heightened frequency of updates in contrast to other OSS. We infer this is because of its frequent interaction with external entities, which makes it more vulnerable to attacks.

Figure 7 represents the top-used binaries in the real-world dataset, each bar in Figure 7 denotes an identical binary (i.e., with the same hash value). We surprisingly observed that **(3) some identical OSS binaries are utilized across different models, even for different vendors.** We manually examine these models and find that all of them are based on the same project Asuswrt-Merlin [19], a third-party alternative firmware, which also explains why their version update trends are so similar. This also greatly increases the risk of different vendors being affected by the same OSS vulnerability.

Answer to RQ3. Analysis results demonstrate that Analysis results demonstrate that a significant number of firmware contain outdated OSS versions, despite the availability of patched versions, and different OSS have varying update priorities, as reflected in their respective update frequencies. Unfortunately, most vulnerabilities were either neglected or not addressed promptly, resulting in an average patch delay of 352 days.

8 DISCUSSION

Function Similarity Calculation. We acknowledge that despite the utilization of APF to mitigate the impact of irrelevant functions, it is still possible to encounter false negatives and false positives due to certain limitations in the BCSD tool. However, it is worth noting that the BCSD tool implemented in our framework can be substituted with a more effective method if one becomes available. Furthermore, the current implementation of the BCSD tool necessitates the use of binaries of different versions, which poses challenges when binaries are not available. To address this limitation, we propose two potential solutions. The first approach involves acquiring and downloading binaries from third-party websites such as Fedora [14], Libraries [17], and Pakgs [18]. The second approach involves utilizing a source-to-binary BCSD tool, such as CodeCMR [52], which only requires the source code of the OSS.

More Programming Languages. *LibvDiff* currently supports C/C++-based binaries because of their popularity (i.e., widely used in firmware). However, the core idea of *LibvDiff* which involves extracting version differences and comparing them with the target, is applicable to other programming languages as well. When adapting *LibvDiff* to other programming languages, it is necessary to make adjustments to the tools used for feature generation and function similarity calculation in order to accommodate the characteristics of those languages. We leave the support for more programming languages for future work.

Threats to Validity. (1) The first threat is the two thresholds used for identification. Since it is inevitable to eliminate all the false negatives and false positives due to the minor difference across OSS versions and binary functions, we selected the thresholds through a reasonable experimental design to minimize the threat. (2) The second threat arises from the code obfuscation [25, 40, 42–44]. The semantics of functions are significantly impacted by code obfuscation, leading to a decrease in the precision and recall of BCSD. We acknowledge that addressing this threat remains challenging and leave it for future work.

9 RELATED WORK

In this section, we review the closely related works in two directions: OSS version identification and binary code similarity detection.

OSS Version Identification. Various approaches have been proposed to identify the version of OSS binaries. In the early stage, most prior works rely on version strings or related information in

target binary to identify OSS versions. Pandora[54] and VES [35] utilize string patterns to locate version strings in the target binary and recover the version number with control flow analysis and data flow analysis, respectively. OSSPolice [29] and LibRARIAN [21] generate regular expressions to match version strings. These methods heavily rely on the existence of version strings in target binaries and will fail when there are no version strings available. Works after that mainly focus on utilizing more features extracted from source code or binaries of OSS. B2SFinder[53] utilizes seven kinds of code features that are traceable in both binary and source code to detect OSS in binaries. LibDB[41] and FirmSec [55] introduce the BCSD tool and combine basic features to filter out candidate versions and determine the OSS version with the call graph matching score and the number of matched features, respectively.

These works either rely heavily on version strings or treat different versions as different OSS for version identification. Therefore, they may generate false positives when there are no version strings available or the changes between versions are subtle. Our work only focuses on the different parts across versions and captures fine-grained version changes by introducing the BCSD technique and useful function filter. Thus, it can significantly improve the precision of identification results.

Binary Code Similarity Detection. Binary code similarity detection has drawn much attention for its wide range of applications, such as bug search [26, 27, 36], malware detection [22, 23, 37], and patch analysis[30, 33, 46]. DCC [39] normalize the binary instructions and calculate the function similarity by cutting the function with a sliding window. TRACY [28] divides the function with tracelet to address the basic block merging problem. DiscovRE [31] computes the similarity between functions by combining structural and numeric features. Genius [32] proposes attributed CFG (ACFG) and utilizes a clustering algorithm to encode functions. BinGo [24] and BinGo-E [47] use multiple kinds of features (i.e., syntax, semantics, and emulation features) to find similar functions. With the development of deep learning and natural language processing (NLP), the BCSD methods which integrate these technologies outperform other traditional schemes. Gemini [45] first integrates ACFG used in Genius and a neural network to embed functions as vectors. VulSeeker[34] add the data flow information by labeling the write and read memory instructions to ACFG to achieve better performance. Yu [51] applies the most popular NLP technique BERT to pretrain the binary code and use three kinds of semantic-aware, structural-aware, and order-aware features (o.e., semantic, structural, and order) to embed the function.

The flourishing development of BCSD technology provides support for us to capture functional semantic changes between versions. They can efficiently and effectively retrieve similar functions in binaries. However, using the BCSD tool directly also brings challenges in precision since non-homologous functions may mislead the retrieval results, which drives us to propose APF.

10 CONCLUSION

This work proposes a version difference-based OSS version identification approach, named *LibvDiff*. It optimally exploits version-sensitive features to discriminate between versions and employs the binary code similarity detection (BCSD) technique to capture the functional semantics. To improve the precision and efficiency, *LibvDiff* proposes two effective filters, namely CVF and APF that reduce the quantity of compared versions and functions for retrieval. The experimental results show that *LibvDiff* achieves a high precision with relatively little time consumption. Our analysis of real-world dataset drew reveals three conclusions that can inspire and guide firmware security analysis.

11 ACKNOWLEDGEMENT

We appreciate all the anonymous reviewers for their invaluable comments and suggestions. This work is partly supported by National Key Research and Development Program of China (No.2022YFB3103 904), National Natural Science Foundation of China (No.62202462, No.61931019), and the Chinese Young Scientists Fund of the National Natural Science Foundation (No.62002342). Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 2014. CVE-2014-0160. https://nvd.nist.gov/vuln/detail/cve-2014-0160.
[2] 2015. CVE-2015-0235. https://nvd.nist.gov/vuln/detail/cve-2015-0235.
[3] 2020. Market Guide for Software Composition Analysis. https://www.gartner.com/doc/reprints?id=1-26OO2IJ6&ct=210630&st=sb.
[4] 2021. The Forrester Wave: Software Composition Analysis, Q3 2021. https://reprints2.forrester.com/assets/2/679/RES176091/report.
[5] 2022. The best-of-breed binary code analysis tool, an indispensable item in the toolbox of world-class software analysts, reverse engineers, malware analyst and cybersecurity professionals. https://hex-rays.com/ida-pro/.
[6] 2022. Core c99 package for AWS SDK for C. Includes cross-platform primitives, configuration, data structures, and error handling. https://github.com/awslabs/aws-c-common.
[7] 2022. A fast, easy to use tool for analyzing, reverse engineering, and extracting firmware images. https://github.com/ReFirmLabs/binwalk.
[8] 2022. The official PNG reference library. http://www.libpng.org/pub/png/libpng.html.
[9] 2022. The open source, decentralized and multi-platform package manager to create and share all your native binaries. https://conan.io/.
[10] 2022. A platform supports manage cybersecurity and cyber compliance across the entire lifecycle. https://tomato.groov.pl/.
[11] 2022. Python Framework to analyse Git repositories. https://github.com/ishepard/pydriller.
[12] 2022. TLS/SSL and crypto library. https://github.com/openssl/openssl.
[13] 2023. A code hosting platform where over 100 million developers shape the future of software, together. https://github.com/.
[14] 2023. The Fedora Project is an independent project[2] to coordinate the development of Fedora Linux. https://admin.fedoraproject.org/mirrormanager/.
[15] 2023. A free replacement for Adobe's enscript program. https://gitlab.gnome.org/GNOME/libxml2.
[16] 2023. A freely available software library to render fonts. https://gitlab.freedesktop.org/freetype/freetype.
[17] 2023. Libraries.io monitors 6,373,004 open source packages across 32 different package managers, so you don't have to. https://libraries.io/.
[18] 2023. Packages for Linux and Unix. https://pkgs.org/.
[19] 2023. A third party alternative firmware based on Asuswrt-Merlin project, for different routers. https://xvtx.ru/xwrt/.
[20] 2023. The U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP). https://nvd.nist.gov/.
[21] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. 2021. Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 1347–1359.
[22] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *International Conference on Detection of intrusions and malware, and vulnerability assessment.*
[23] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2014. Control Flow-Based Malware VariantDetection. *IEEE Transactions on Dependable and Secure Computing* 11 (2014), 307–317.
[24] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary

search. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).
[25] Christian S. Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. 2012. Distributed application tamper detection via continuous software updates. In *Asia-Pacific Computer Systems Architecture Conference*.
[26] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017).
[27] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018).
[28] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014).
[29] 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 *CCS*. 2169–2185.
[30] Thomas Dullien. 2005. Graph-based comparison of Executable Objects.
[31] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *NDSS*.
[32] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-Based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.
[33] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *International Conference on Information, Communications and Signal Processing*.
[34] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *ASE*. 896–899.
[35] Xulun Hu, Weidong Zhang, Hong Li, Yan Hu, Zhaoteng Yan, Xiyue Wang, and Limin Sun. 2020. VES: A Component Version Extracting System for Large-Scale IoT Firmwares. In *WASA*, Dongxiao Yu, Falko Dressler, and Jiguo Yu (Eds.), Vol. 12385. 39–48.
[36] He Huang, Amr M. Youssef, and Mourad Debbabi. 2017. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017).
[37] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *International Symposium on Recent Advances in Intrusion Detection*.
[38] Tencent Security Keen Lab. 2022. BinaryAI. https://www.binaryai.cn/.
[39] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis*.
[40] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. *2009 30th IEEE Symposium on Security and Privacy* (2009), 94–109.
[41] 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 *MSR*. arXiv:2204.10232
[42] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo García Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. *2015 IEEE Symposium on Security and Privacy* (2015), 659–673.
[43] Irfan ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. 2017. Malware Lineage in the Wild. *Comput. Secur.* 78 (2017), 347–363.
[44] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018).
[45] 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 *CCS*. 363–376.
[46] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020).
[47] Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan, and Yang Liu. 2019. Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation. *IEEE Transactions on Software Engineering* 45 (2019), 1125–1149.
[48] 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. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)* (2022), 1393–1405.
[49] Shouguo Yang. 2021. Asteria: Deep Learning-Based for Cross-Platform Binary Code Similarity Detection. In *DSN*. 13.
[50] 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. *ArXiv* abs/2301.00511 (2023).
[51] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. In *AAAI*, Vol. 34. 1145–1152.
[52] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. In *NeurIPS*. 12.
[53] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xuey, and Wei Huo. 2019. B2SFinder: Detecting Open-Source Software Reuse in COTS Software. In *ASE*. 1038–1049.
[54] Weidong Zhang, Yu Chen, Hong Li, Zhi Li, and Limin Sun. 2018. PANDORA: A Scalable and Efficient Scheme to Extract Version of Binaries in IoT Firmwares. In *2018 IEEE International Conference on Communications (ICC)*. 1–6.
[55] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem A. Beyah. 2022. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in IoT firmware. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022).