Ban *et al. Cybersecurity* (2021) 4:21
https://doi.org/10.1186/s42400-021-00085-7

Cybersecurity

RESEARCH Open Access

B2SMatcher: fine-Grained version identification of open-Source software in binary files

Gu Ban[1,2], Lili Xu[1*], Yang Xiao[1], Xinhua Li[1], Zimu Yuan[1] and Wei Huo[1]

Abstract

Codes of Open Source Software (OSS) are widely reused during software development nowadays. However, reusing some specific versions of OSS introduces 1-day vulnerabilities of which details are publicly available, which may be exploited and lead to serious security issues. Existing state-of-the-art OSS reuse detection work can not identify the specific versions of reused OSS well. The features they selected are not distinguishable enough for version detection and the matching scores are only based on similarity.

This paper presents B2SMatcher, a fine-grained version identification tool for OSS in commercial off-the-shelf (COTS) software. We first discuss five kinds of version-sensitive code features that are trackable in both binary and source code. We categorize these features into program-level features and function-level features and propose a two-stage version identification approach based on the two levels of code features. B2SMatcher also identifies different types of OSS version reuse based on matching scores and matched feature instances. In order to extract source code features as accurately as possible, B2SMatcher innovatively uses machine learning methods to obtain the source files involved in the compilation and uses function abstraction and normalization methods to eliminate the comparison costs on redundant functions across versions. We have evaluated B2SMatcher using 6351 candidate OSS versions and 585 binaries. The result shows that B2SMatcher achieves a high precision up to 89.2% and outperforms state-of-the-art tools. Finally, we show how B2SMatcher can be used to evaluate real-world software and find some security risks in practice.

Keywords: Version Indentification, Binary-to-Source Mapping, Component Analytics, Code Features, One-Day Risks

Introduction

During the modern software development process, developers often use the rich functions provided by open source software (OSS) to shorten the development cycle, spending more time on personalized development. In recent years, the number of OSS is growing at an exponential rate. Up to now, there are over 44 million repositories on Github (Repo Statistics on Github 2020). Such a large amount of OSS has brought great convenience to software development. However, improper use of OSS can cause potential serious security risks. Wang et al. (Wang et al. 2020) analyzed 806 software and pointed out that the use of outdated OSS is a common phenomenon, software containing outdated OSS is more likely to be exploited. For example, a severe security vulnerability called Heartbleed (Heartbleed 2020) was found in version 1.0.1 before 1.0.1g of OpenSSL, a popular cryptographic software library. For software that used vulnerable versions of this library, attackers could steal private information such as the names and passwords of users. It affected much famous software such as LibreOffice (LibreOffice 2020) from version 4.2.0 to 4.2.2 and VMware Workstation 10 (VMware Workstation Pro 2020).

*Correspondence: xulili@iie.ac.cn
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
Full list of author information is available at the end of the article

As shown in the previous examples, vulnerabilities appear in specific versions of OSS. By correctly identifying the version of reused OSS in COTS software, we can not only get the release time and determine whether it is an outdated version, but also check whether this version is a vulnerable version. The aim of this paper is to implement a fine-grained version identification tool for OSS reused in COTS binary files.

Our task of OSS version identification is based on the result of reused OSS candidates generated by code reuse detection methods. In the following, we briefly review three classic approaches of code reuse detection and discuss which one suits our scenario. The three main approaches are source-to-source comparison, binary-to-binary comparison, and source-to-binary comparison. Since we target COTS software, of which source code is inaccessible, the source-to-source comparison method (Kamiya et al. 2002; Li et al. 2006) is trivially excluded. Binary-to-binary analysis (Tang et al. 2020; Hemel et al. 2011) measures the similarity between the target binary and the binaries built from all candidate OSS versions. However, an OSS often contains a large number of versions, for example, LibPNG contains 1615 versions. To the best of our knowledge, there is no way to directly collect the binary files of all versions. Too much labor is required to manually configure OSS dependencies and find suitable compiler flags, therefore it is unacceptable to spend tremendous amounts of time to build versions of a significant amount of OSS (Yuan et al. 2019; Duan et al. 2017).

Therefore, we choose to directly perform the binary-to-source comparison. For each given binary file, we extract features from this binary and compare them with the features extracted in advance from the candidate OSS source codes. When adopting prior work, B2SFinder (Yuan et al. 2019) and OSSPolice (Duan et al. 2017), to determine the version of reused OSS, the experimental results in "Evaluation" section show that their accuracy is not satisfactory. Features extracted by them such as constant numerical array are not discriminative enough to distinguish between versions, and the comparison methods they adopt are mainly designed to detect OSS code reuse.

In order to accurately and effectively identifying the versions of reused OSS, the following three problems need to be solved.

1. How to select version-sensitive code features? A natural idea of code feature selection would be that of first evaluating those features already used in the prior work. As evaluated in "Evaluation" section, global arrays and control-flow related features considered by B2SFinder (Yuan et al. 2019) bring high false positives in the setting of version identification. Code features like string literals and exported function names selected by B2SFinder (Yuan et al. 2019) and OSSPolice (Duan et al. 2017) are not informative enough to distinguish all versions. We are motivated to explore more fine-grained code features that are traceable in both source code and binary files, and enjoy a high degree of discrimination among different versions. Such features are defined as version-sensitive features.

2. How to effectively extract code features? Code features help us measure the similarity between target binary and candidate OSS versions. The following two aspects need to be considered when performing the extraction of code features. (1) From the point of view of precision improvement: Partial build, which means that most source files are not compiled into the target binary, is common in the OSS compilation process. For example, we find that only 7.3% source files of zlib are used in the build process and obviously only the code features of compiled source files can be found in the resulting binaries. For the purpose of precisely calculating the similarity between source code and binary code, we need to explore effective methods to determine the source files participating in the compilation process and extract features from these files. (2) From the point of view of scalability improvement: Note that in general, the entire source code of an OSS is developed iteratively each time the OSS is updated, thus using all the source code of each version during the comparison would bring in redundant cost at the parts that are not notably modified among different versions. Identifying code changes across versions and eliminating redundant comparison on the common parts can improve the efficiency of version identification.

3. How to design detection methods to improve the precision of version identification? In code comparison, code similarity is commonly used as an indicator to predict code reuse. We give a realistic example to illustrate the limitation of only relying on code similarity in our scenario. The library SQLite3.dll is reused by a multimedia player Storm Codec (Storm Codec 7 2020) and has the highest similarity with SQLite version 3-6-15. But some code features on higher versions of SQLite can also be found in this dll. More specifically, we find in this dll some special functions which only exist in SQLite version 3-8-8-3. Through manual analysis, we find this binary file reuses SQLite version 3-6-15 and 3-8-8-3, in a way of patching on 3-6-15 subsequently for compatibility. It inspires us to propose an appropriate detection method such as combining matched features and similarity.

To solve the aforementioned challenges, we propose an effective binary-to-source comparison technique, B2SMatcher, for precisely identifying the version of reused OSS. By evaluating the discrimination degree of the code features considered in the previous work (Duan et al. 2017;

Yuan et al. 2019), B2SMatcher decides to use string literals and exported function names as program-level code features. Usually, adjacent versions may differ only in the implementation of some functions, using the program-level code features is not sensitive enough to distinguish them. We are motivated to add function-level code features to capture slight changes among neighbor OSS versions. We propose a two-stage identification approach, of which the first stage uses program-level features for a rough matching and identifying reuse types, the second stage uses function-level features for precise matching. For the reuse types, we identify two types of reuse relationship, namely, *single-version reuse* and *multi-version reuse*.

One way of obtaining the list of compiled source files is to hook the build process (Duan et al. 2019). Taking into account the low success rate of auto-build (Shahkar 2016; Yuan et al. 2019; Duan et al. 2017) and the very time-consuming manual build, we inventively use clustering analysis (K-means Clustering 2020) and decision tree (Decision tree 2020) to predict compiled source files and extract features from them. For the sake of scalability, we adopt abstraction and normalization techniques to eliminate redundant functions that appear in multiple versions.

In order to evaluate the effectiveness of our work, we crawled in total 6351 versions of source code from popular and historically-vulnerable open source libraries on Github. We construct a ground truth dataset that contains 585 binaries, consisting of manually compiled binaries and the ones collected from real-world open source software. We compare B2SMatcher with the two most closely related approaches OSSPolice and B2SFinder, and one commercial tool Cybellum (Cybellum 2020). The experiments show that B2SMatcher achieves a precision of 89.2%, which notably outperforms B2SFinder, and is 8% and 7% higher than the ones of OSSPolice and Cybellum, respectively. We also evaluate our work on software collected from Tencent Application Center (Tencent Software Download Official Version 2020) and find that some popular applications like Zoom and TeamViewer reuse vulnerable versions of OSS.

In summary, we make the following contributions.

- We introduce a binary-to-source comparison approach for fine-grained version identification of reused OSS. The key technical contributions include using machine learning methods for predicting compiled source files and precisely extracting source code features, selecting representative functions of OSS versions by abstraction and normalization methods, and adopting a two-stage identification approach to precisely recognize reused OSS versions.
- We define a new concept of reuse type for patching cases (*multi-version reuse*) and utilize it to better discover security risks (discussed in "Multi-version reuse" section)
- We develop a prototype implementation called B2SMatcher, which achieves a precision of 89.2% when identifying the reused version between 6351 candidate OSS versions and 585 binaries. B2SMatcher is also shown to be capable of reporting security risks for software in the real world.

The remainder of the paper is organized as follows. "Overview" section uses a practical example to briefly introduce the workflow of B2SMatcher. "Design" section presents more implementation details, including how to select version-sensitive features, how to predict compiled files, how to eliminate redundant functions and how to perform a two-stage identification. We evaluate B2SMatcher in "Evaluation" section and discuss the limitation of our work in "Discussion" section. Related work is in "Related work" section and we conclude in "Conclusion" section.

Overview

In this section, we first describe the assumption and the goal of B2SMatcher. Then we walk through a motivating example to illustrate how B2SMatcher detects the specific version of a reused OSS.

Assumption and goal

In order to discover the potential security risks of a binary file brought in by reusing vulnerable OSS, as discussed in "Introduction" section, it is crucial to detect the specific version of a reused OSS. Given a binary B_1, we assume that the reused open source software in B_1 are known, for simplicity, they are oss_1, oss_2, oss_3. The aim of this paper is to find the specific versions of oss_1, oss_2, oss_3. Furthermore, those potential security bugs introduced in certain reused OSS versions can be obtained and developers could be informed about the security risks in this binary.

We use two tools, one open source project B2SFinder and a commercial tool Cybellum, to obtain the reused libraries in a binary and assume that their results are correct. To perform a version-level OSS reuse detection, we need to extract code features from a binary. We use a binary analysis tool IDA Pro and assume that features extracted by it are accurate.

A motivating example

We use a font rendering library freetype-VER-2-6.so (compiled from its corresponding source code) as an example to intuitively describe the workflow of B2SMatcher. The key steps in version identification are to

figure out which source code files and which code features should be considered.

Extracting code features from all source files, regardless of whether they are compiled into binary files or not, may cause false positives. Take freetype-VER-2-6.so as an example, if we use the code feature of string literals and extract feature instances simply from all source files, the version which has the maximum number of exact matches with the shared library would wrongly be freetype version 2-9. This is due to a source file Ftobjs.c, which, however, is actually not compiled into the binary. Some strings in this file, such as "FT_Property_Get" and "FT_Property_Set", affect the similarity. Therefore, to perform a precise version identification, we only extract features from the compiled source files.

For the feature selection, we first take into account all these code features adopted by B2SFinder and OSSPolice. More specifically, they are string literals, exported function names, global arrays and constants in control-related statements. We evaluate the effect of all these features and the comparison results are shown in Table 1: For each feature in the first column, an exact match between the binary file and different freetype versions' source codes is used. Those versions with the maximum number of exact matches are listed in the second column. It can be seen that the candidate versions matched by using string literals and exported function names correctly include the target version of this freetype library, i.e. 2-6.

For the rest features in Table 1, they can not recognize the target version 2-6. Constant branches in switch/case statements and if/else statements appear only in a few versions of freetype. And freetype does not have global string arrays and global enumeration arrays in the source code at all, as can be seen from Row 6 and 7 in Table 1. In the case of the global integer array, we find that it is easy to distinguish different global integer arrays from the point of view of source code, but it is difficult to distinguish them in binary files. Using it would mismatch version 2-3-6 as a candidate, as can be seen from the last row in Table 1. We detail its mismatch process as follows.

Table 1 Comparison of code features when identifying freetype-VER-2-6.so

Feature Name	Matched Versions
String	[freetype-VER-(2-6, 2-6-1, 2-6-2 ... 2-8-1)]
Export	[freetype-VER-(2-6, 2-6-1, 2-6-2)]
Consts in switch/case	[freetype-VER-(2-6-4, 2-6-5, 2-7)]
Consts in if/else	[freetype-VER-2-6-4]
Global string array	[]
Global enum array	[]
Global integer array	[freetype-VER-2-3-6]

In the source code of freetype version 2-3-6, there is a global integer array *ft_extra_glyph_unicodes* and its content is shown in the second column of the third row in Table 2. After being compiled to a shared file, it exists in the binary as a little-endian bitstream, which is shown in the third column of the third row in Table 2.

In another version of freetype, i.e. version 2-6, there is also a global integer array *ft_extra_glyph_unicodes*. It's content in source code and related little-endian bitstream are shown in the second row of Table 2. We can find that it does not have the last two integers compared with the previous one in version 2-3-6.

As shown in Fig. 1, the bitstreams of the previous two arrays can both be found in freetype-VER-2-6.so, highlighted with a red and a blue rectangle, respectively. Since a numerical array exists in a binary file as a sequence of bytes, usually it is difficult to accurately determine the boundary of the array. This eventually makes the overall number of exact matches of freetype version 2-3-6 the maximum one, and misleadingly regards version 2-3-6 as the candidate.

Therefore, we decide to choose string literals and exported function names to perform version identification. We consider them as program-level features and use them to do a rough match in the first stage. The rough match can determine the reused range of OSS versions. For example, as can be seen from the third row in Table 1, the rough match results of freetype-VER-2-6.so are versions 2-6, 2-6-1 and 2-6-2.

Taking into account that the function will change between different versions, we explore function-level features such as constants extracted from assignments, use them in the second stage for a precise match, and finally determine the reused OSS version. As can be seen from the left side of Fig. 3, the constants extracted from assignments in a decompiled function of freetype-VER-2-6.so, namely, [0, 0, 2, 6, 0, 1], are the same as the constants defined in the corresponding function in version 2-6 shown in the first box on the right side, while different from the ones in version 2-6-1 and 2-6-2, which are [0, 0, 2, 6, 1, 1] and [0, 0, 2, 6, 2, 1], respectively. This helps to determine that version 2-6 is the reused OSS version.

Workflow of B2SMatcher

As shown in Fig. 2, B2SMatcher contains three major steps.

(1) Features Selection. As shown in the motivating example, appropriate code features need to be selected for precisely identifying reused version. We select both program-level features and function-level features. The detailed feature selection approach will be presented in "Feature selection" section.

(2) Features Extraction. After the step of feature selection, we need to extract the selected features effectively.

Table 2 Global integer array in source code and binary file

Version	ft_extra_glyph_unicodes	
	Source code	Binary file
ver-2-6	{0x0394, 0x03A9, 0x2215, 0x00AD, 0x02C9, 0x03BC, 0x2219, 0x00A0, 0x021A, 0x021B}	94030000 a9030000 15220000 ad000000 c9020000 bc030000 19220000 a0000000 1a020000 1b020000
ver-2-3-6	{0x0394, 0x03A9, 0x2215, 0x00AD, 0x02C9, 0x03BC, 0x2219, 0x00A0}	94030000 a9030000 15220000 ad000000 c9020000 bc030000 19220000 a0000000

Not all source files will be compiled into the final binary files, we extract features only from compiled source files to better check out which version of the source code can match the binary file with the maximum number of matched features. Considering that there are massive duplicate functions between versions, we focus on distinct functions of each OSS version for improving efficiency in comparison.

(3) Two-stage detection: The program-level code features are used for a rough identification, and its results are divided into two reuse types: single-version reuse (such as the above-mentioned freetype-VER-2-6.so) and multi-version reuse (such as SQLite.dll in "3. How to design detection methods to improve the precision of version identification?" section). Function-level features are used in the precise identification stage. The results of the two stages are combined to determine the reused version. For each target binary, we detect the OSS versions reused by it and form a reuse relationship report. This report can help track potential security risks, which will be discussed in "Real-world software exploration" section.

Design

In this section, we present the detailed design of B2SMatcher for version detection in COTS software.

Feature selection

Though there exist considerable differences between binary and source code representations, we try to find uniform features between them for comparison. For the purpose of version detection in COTS software, we propose the following two selection criteria. First, the code features should exist in both source code and binary file, and the representation of features does not vary a lot during the compilation. Second, these features should have a high degree of discrimination between different OSS versions. Features that meet the above criteria are defined as version-sensitive features. The features we choose can be divided into two types: program-level features and function-level features.

Program-level features

Table 3 lists a total of 7 candidate code features, covering all the features used in existing binary-to-source comparison techniques (Li et al. 2017; Hemel et al. 2011; Duan et al. 2017; Yuan et al. 2019).

First, we evaluate at the source code level which code features enjoy a relatively high discrimination degree among OSS versions. Below, we present a formula for computing the discrimination degree of a code feature. The larger the discrimination degree is, the more versions this code feature can distinguish at the source code level. Take the code feature string literals as an example, for each version in a given project, we extract string literals from source code, sort them in an alphabetically-ascending order and store them in a file. A hash value is generated for the entire file and we denote it by *feature_hash*. If the hash value of a version is unique, i.e., it is different from the rest, it means that this version has string literals that do not appear in other versions, thus this version can be distinguished. Overall, the discrimination degree for a code feature, denoted by *discrimination*, is obtained according to Formula 1. We use $\#(distinct(feature\_hash))$ to represent the total number of unique *feature_hash* among all versions. The notation k represents the total number of versions in this project.

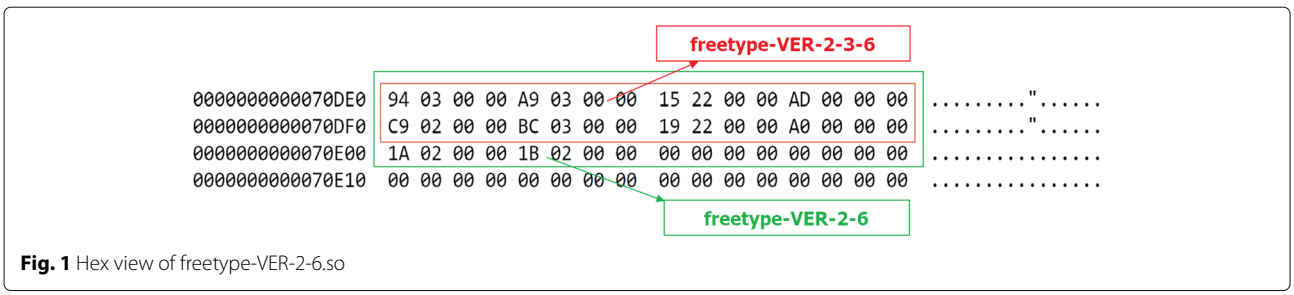

Fig. 1 Hex view of freetype-VER-2-6.so

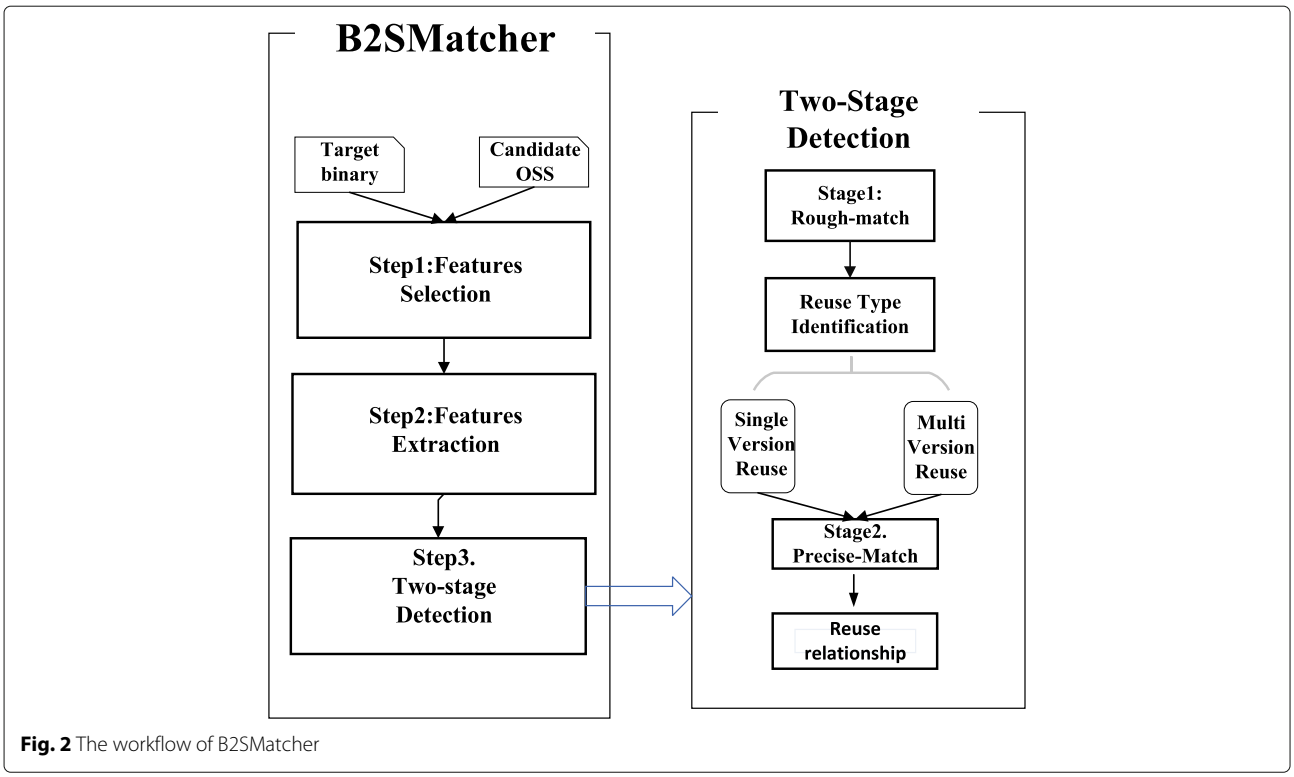

Fig. 2 The workflow of B2SMatcher

$$discrimination = \frac{\#(distinct(feature\_hash))}{k} \quad (1)$$

To give a flavor of the discrimination degree that a code feature enjoys, we select three OSS projects FreeType, LibTIFF, and SQLite as examples. The discrimination degree of each code feature is computed for the three projects according to Formula 1 and the result is shown in Table 3. The "Average" column shows the average value of discrimination degrees of the three projects. As can be seen from Table 3, the average discrimination degrees of string literals, exported function names and global integer arrays are all over 0.5. We decide to use the above top three features as program-level code features for the moment.

Next, we evaluate at the binary code level whether these features can effectively distinguish different versions of binary files. String literals and exported function names stay the same after the compilation, but a global integer array exists in binary files as a sequence of bytes. We can not exactly restore the boundaries of an array. As already discussed in "A motivating example" section, using this type of code feature can be misleading in identifying the correct version of binary files. In conclusion, we choose string literals and exported function names as the program-level features.

Function-level features

Fine-grained function-level code features also need to be extracted to increase the accuracy of version pinpointing. Prior works on binary-to-source comparison, B2SFinder and OSSPolice, do not consider function-level

Table 3 Candidate code features

Feature Class	Feature Name	Discrimination				Selected?
		freetype	SQLite	libTIFF	Average	
Program-level	String	0.7246	0.8889	1.0	0.8712	✓
	Export	0.7536	0.5	0.7778	0.5624	✓
	Consts in switch/case	0.2778	0.1726	0.3551	0.2685	
	Consts in if/else	0.3551	0.1726	0.2778	0.2684	
	Global string array	0	0	0	0	
	Global enum array	0.0145	0.0089	0.0279	0.0170	
	Global integer array	0.6493	0.5513	0.4944	0.5650	✓

code features. We turn to works on binary-to-binary comparison for inspirations of function-level code features. Numerical features in functions are widely used in binary-to-binary comparison works (Eschweiler et al. 2016; Feng et al. 2016; Xu et al. 2017), such as the constants in comparison instructions, the total number of parameters and local variables. But they are heavily affected during the compilation by optimization, which violates our criteria. Through manual analysis, we find that two types of constants are affected relatively slightly during the compilation, which are the constant sequence of assignments arranged in an ascending order and the constant parameters in function calls. We depict their meanings below.

We first illustrate the constant sequence of assignments in an ascending order using code snippets in Fig. 3. Basically, the feature is obtained by extracting all constants in assignments of a function, which are further arranged in an ascending sequence. As shown in Fig. 3, the code snippet on the left side is taken from the shared file freetype-VER-2-6.so, and the code snippet on the upper right corner is taken from the same version of the source code. We find that the constant sequence in assignments are the same in the two code snippets, which are both [0, 0, 2, 6, 0, 1] (highlighted in red in the figure). The corresponding ascending order is [0, 0, 0, 1, 2, 6]. While the constant sequences in assignments in the same function of the two other versions 2-6-1 and 2-6-2 of freetype are [0, 0, 2, 6, 1, 1] and [0, 0, 2, 6, 2, 1], respectively. Their corresponding ascending orders are [0, 0, 1, 1, 2, 6] and [0, 0, 1, 2, 2, 6], respectively, which are different from the one of version 2-6. This type of constant-related code feature is selected and named briefly as *constants in assignments*.

Next, we use the function openDatabase in sqlite-version-3.7.14 to illustrate another constant-related code feature, namely, the constant parameters in function calls. The relevant source code and disassembled code snippets of openDatabase are shown in the following two listings, respectively. In Listing 1, at line 9, function sqlite3MisuseError is called by openDatabase with a constant parameter 113824. We can see clearly in the corresponding disassembled version in Listing 2, at line 6, the parameter of the callee sqlite3MisuseError is also the constant 113824, correctly restored after being disassembled.

Similar functions share similar call graphs. We consider previous works on binary function similarity detection. DiscovRE (Eschweiler et al. 2016) uses call graph as a code feature. αdiff (Liu et al. 2018) uses as the inter-function feature the numbers of callers and callees of a function, i.e., the in-degree and out-degree of the function on call graph. Inspired by them, we also take into account the code feature related to call graph, namely, the in/out-degrees of a function.

```
1. static int openDatabase(
2. const char *zFilename,
3. sqlite3 **ppDb,
4. unsigned int flags,
5. const char *zVfs
6. ){
7. ...
8. if( ((1<<(flags&7)) & 0x46)==0 )
9. return sqlite3MisuseError(113824);
10. ...
11. }
constant type parameter in the function call:{113824}
```

Listing 1 openDatabase (source code).

```
1. __int64 __fastcall openDatabase
2. (__int64 a1, char *a2, int a3, __int64 a4)
3. {
4. ...
5. if ( !((70 >> (v9 & 7)) & 1) )
6. return sqlite3MisuseError(113824LL);
7. ...
8. }
constant type parameter in the function call:{113824}
```

Listing 2 openDatabase (binary).

Overall, we select five kinds of code features, including two kinds of program-level features: string literals and exported function names, and three kinds of function-level features: constants in assignments, constant parameters in function calls and in/out-degrees of a function on call graph. We will discuss the effectiveness of these features in "Effectiveness of the two-stage identification" section through experiments.

Feature extraction

After the selection of code features, we need to extract feature instances from binary code and source code in an accurately and effectively way. For the feature extraction in source code, as discussed in "Introduction" section, restricting the analysis within compiled source files and eliminating comparison on duplicate functions can improve the accuracy and effectiveness, respectively. We detail in the following why and how to use machine learning methods to obtain the compiled source files, and how to use function abstraction and normalization methods to eliminate redundant functions and focus on "unique" functions across versions.

Compilation-related files

OSS source package contains source files that will be compiled into the binary, and source files that will not be compiled into the binary. Taking the project *zlib-v1.2.8* as an example, there are 42 source files with .c suffix in the source code package, but only 15 of them are compiled into binary. We call this phenomenon "selective build". We define compiled source files as compilation-related files and the rest as compilation-irrelevant files.

Compilation-related files can be obtained by hooking the build process as done in autopatch (Duan et al. 2019).

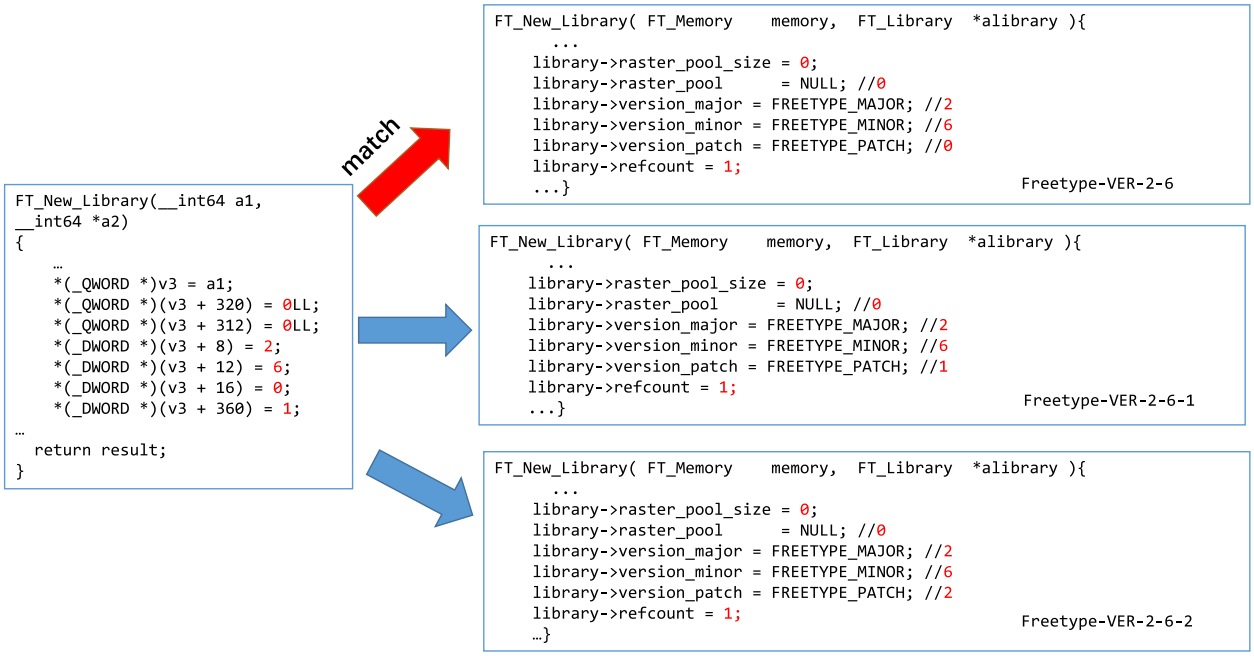

Fig. 3 Function-level code feature instance: constants in assignments

For the purpose of version identification, we need to build multiple OSS projects. However, an experiment performed by B2SFinder (Yuan et al. 2019) shows that only about a quarter of 2189 OSS projects can be automatically built without manual intervention. In order to successfully build OSS, a large amount of manual work is needed such as finding appropriate external OSS dependencies. For the purpose of reducing manual participation, we are inspired to propose a static method. More specifically, for versions of an OSS project which can not be built successfully, we use compilation-related files obtained from successfully built versions to predict compilation-related files of versions that are not successfully built.

For an OSS project, we find that those versions of a project that share similar lists of source files usually have similar lists of compilation-related files as well. In order to seek the correlation between them, we do the following experiments.

We manually build in total 210 mainline versions of three libraries zlib (30), freetype (67) and libxml2 (113), and construct "a version vector" for each built binary. Basically, for each binary, we traverse all source files (with c/cpp suffix) of all the built versions, use the names of these files to form a vocabulary, adopt the one-hot (One-hot Embedding 2020) approach, and construct a version vector, i.e., putting 1/0 in the corresponding position, according to the compilation-related files.

First, we use t-SNE (t-Distributed Stochastic Neighbor Embedding 2020), a tool for visualizing high dimensional vectors, to plot the version vectors generated by the compilation-related files of zlib's built versions, and the result is shown in Fig. 4. A quick inspection shows that the built versions are visibly divided into several clusters according to compilation-related files, which means that the lists of compilation-related files in some versions are very similar.

Second, we cluster the versions of the three libraries according to the version vectors generated by compilation-related files and all source files, respectively. We calculate the similarity of the above two clustering results. We depict below how the similarity degree between two clustering results is obtained. The similarity degree between two sets S and T is obtained as follows, where the notation # denotes the number of items in a set.

$$SetSim(S,T) = \frac{\#(S \cap T)}{max\{\#S, \#T\}}$$

The similarity degree between two clustering results $\mathbb{S}$ and $\mathbb{T}$, where $\mathbb{S} = \{S_1, S_2, ..., S_n\}$, $\mathbb{T} = \{T_1, T_2, ..., T_n\}$ and n is the number of clusters, is obtained as follows:

$$ClusterSim(\mathbb{S},\mathbb{T}) = \frac{\sum_{i\in[1,n]} max_{j\in[1,n]}\{SetSim(S_i,T_j)\}}{n}$$

For each binary, the similarity degrees of the two clustering results in the settings where the numbers of clusters range from 3 to 7 are calculated and shown in Fig. 5. As can be seen from the figure, for all binaries, the similarity degrees under all settings are over 80%. The result means that if two versions of a project have similar lists of source code files in the source code packages, then their

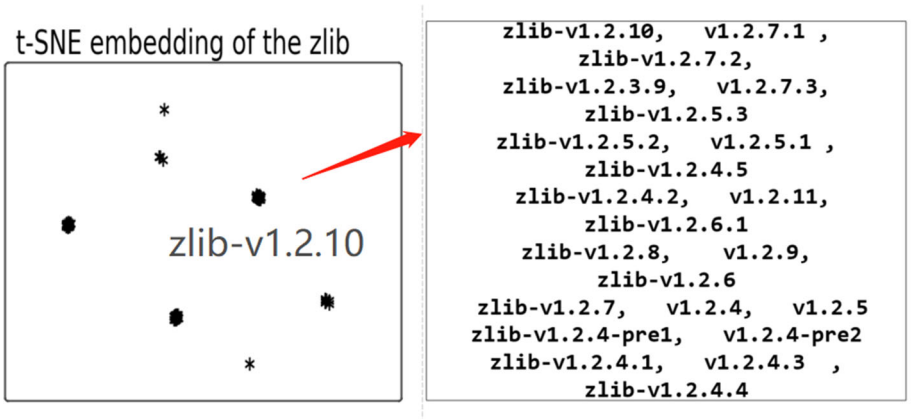

Fig. 4 T-SNE of version vectors constructed by compilation-related files

compilation-related files are also similar, which means that they tend to build similar files into binary.

Based on the two observations, we predicate compiled files as follows:

- Version clustering. For each OSS project, we form the version vector corresponding to each version and then use cluster analysis to group similar versions. Specifically, *native k-means* clustering (K-means Clustering 2020) algorithm is used.
- Obtaining compilation-related files. For each version in a group, we execute plainly the auto-build file in the source code folder, such as Makefile, under the default compilation option. If none version in a group can be successfully compiled, we choose the latest version in this group and compile it manually. In this way, for each group, there is at least one version compiled. Hence, for each group, the compilation-related files of at least one version are known.
- Predicting compilation-related files. We use compilation-related files in successfully built binaries to construct a decision tree (Decision tree 2020) in each version group. After that, we use this decision tree to classify each source file of a binary that is not successfully built in the same group, and determine whether it is a compilation-related file.

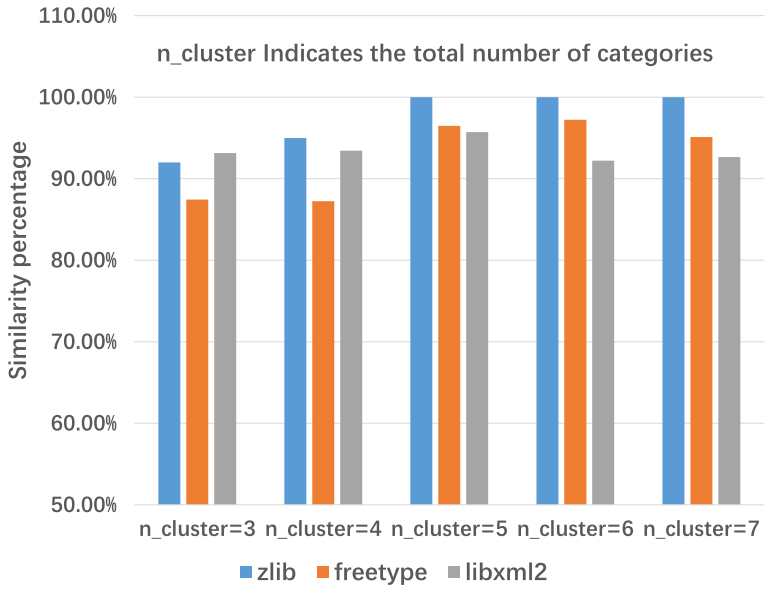

Fig. 5 Clustering result of libxml, freetype, zlib

Note that, an OSS project may generate more than one library file (with .a / .so suffix). We will predict for each library the compilation-related files contained in it.

Unique functions

If functions have the same source code and they are shared by different versions, we define such functions as repeated functions. If a function is implemented only in one version, we define such a function as a unique function. Identifying whether the type of a function is a repeated one or a unique one can help reduce duplicate comparisons across OSS versions and improve efficiency. Taking cURL as an example, the ratio of repeated functions between adjacent versions is 93%. Moreover, unique functions play a crucial role in recognizing unique versions at the precise match stage. We detail below how to obtain unique functions among a given set of OSS versions.

Some functions have the same name but different implementations in each version. To improve the robustness of function types identification, inspired by Vuddy (Kim et al. 2017), we also use abstraction to normalize functions and eliminate the difference between functions caused by the modification of types, identifiers, comments, and whitespace. We consider each normalized function as a text string and compute a hash value for it.

To get unique functions which can be used to identify unique versions, we use function name + function hash value (such as "deflateInit293f14310d6527296cfc24691a 576ab1c") as a key field, the versions which have the function with the same name and same hash value as a value field, to create a direct (inverted) mapping of functions to OSS versions. We look up for functions whose total number of OSS versions is one in this mapping and select them as unique functions. Note that, each unique function is essentially contained by only one version, while each version may contain several unique functions.

The detailed normalization process of a function is shown as follows, we also exemplify the normalization result using a toy function in Listing 3.

- *Local variables abstraction*: For each local variable in function body, we replace it with the symbol *local*. For each member in struct, we replace it with the symbol *member*.
- *Parameters abstraction*: We use symbol *param* to represent parameters in function header and function body.
- *Type abstraction of all variables*: Regardless of the type information of function parameters and local variables in the function, we replace it with the symbol *type*.
- *Function hash generation*: We remove all whitespaces and comments in a function, and then generate the hash value of the function.

```
int ZEXPORT inflateBackEnd(strm)
{
    z_streamp strm;
    if (strm == Z_NULL || strm->state == Z_NULL
        || strm-zfree == free_func)0)
        return Z_STREAM_ERROR;
    ZFREE(strm, strm->state);
    strm->state = Z_NULL;
    return Z_OK;
}
=====================================================
intinflatebackend(strm)(param)typeparam;
if(param==0||param->member==0
||param->member==(type)0)return(-2);
(*((param)->member))
((param)->member,(type)(param->member));
param->member=0;return0;
```

Listing 3 Abstraction on a simple function.

Two-stage identification

For performing a fine-grained version recognition, we design a two-stage identification approach. Program-level features are used at the rough match stage for quickly determining the version range. According to the results of the rough match stage, we identify the reuse type of the binary and use different matching methods for different types at the precise match stage.

From here on, the feature extraction of an OSS source code is performed by default from the compilation-related files obtained from the previous section, source files mentioned below are actually referred to the compilation-related files as well.

Rough match

At the rough match stage, we use the code features string literals and exported function names as program-level features. Considering that the two kinds of code features always keep the same even after compilation. Therefore, for those feature instances of string literals and exported function names extracted from binary and source files, only when they are exactly the same can they be considered to be a match. We use the following equation (2) to score each OSS version with a *match_score*, and versions with the highest *match_score* are the potential reused versions for a target binary.

$$\begin{aligned} match\_score &= weighted\_matched\_features - loss \\ &= \sum_{f \in (BIN \cap OSS)} \frac{1}{n(f)} - \frac{N_{src} - N_f}{N_{src}} \end{aligned} \tag{2}$$

In Eq. (2), we use *BIN* and *OSS* to denote the feature instances extracted from the target binary and the OSS version to be scored, respectively. To measure the contribution of each feature instance for version recognition, B2SMatcher weighs each feature instance according to its frequency in different versions and calculates *weighted_matched_features*. The notation $n(f)$ denotes

the total number of versions in which a feature instance f appears. The similarity is also taken into consideration, and is defined as the notation *loss* in Eq. (2). For each version to be scored, N_{src} denotes the total number of feature instances extracted from source code and N_f denotes the total number of matched feature instances.

Identify reuse type

In general, a high *match_score* means true reuse, and these binaries are defined as single-version reuse. Sometimes only relying on *match_score* may not be enough. For example, unique version string "3.8.5" can be found in SQLite.dll of matlab (MATLAB 2020) and the *match_score* computed by Eq. (2) also shows SQLite.dll reuse SQLite-3.8.5. However, we also found that SQLite.dll has program-level feature instances occurring in SQLite versions 3.8.8.1, 3.8.8.2, 3.8.8.3, 3.8.8.5 and 3.8.8.8, while these feature instances do not occur in version 3.8.5. Such examples are defined as multi-version reuse and we discuss how to identify this scenario as follows.

We use $mf_g(b, AV)$ to represent the total number of matched program-level feature instances between a target binary b and all versions AV of an OSS. The notation $mf_g(b, CV)$ represents the total number of matched program-level feature instances between the binary b and the candidate versions CV (with the highest *match_score*) obtained from the rough match stage. If the value of $mf_g(b, AV)$ and the value of $mf_g(b, CV)$ are close, this means that all matched feature instances can be found in the candidate versions. We define the reuse type of the target binary b in this case as single-version reuse. If the value of $mf_g(b, AV)$ and the value of $mf_g(b, CV)$ are obviously different, we define the reuse type of the target binary b in this case as multiple-version reuse. We use the following equation to identify different reuse types, where the threshold is set empirically.

$$\begin{cases} \frac{mf_g(b,AV)-mf_g(b,CV)}{mf_g(b,CV)} \gg threshold, & \text{multiple-version reuse} \\ \frac{mf_g(b,AV)-mf_g(b,CV)}{mf_g(b,CV)} \ll threshold, & \text{single-version reuse} \end{cases} \tag{3}$$

Precise match

Previously in the feature selection step, we select three kinds of function-level features: *constants in assignments*, *constant parameters in function calls*, and in/out-degrees of a function on call graph, to further characterize the similarity between a binary function and a source function. We first define below how to calculate two types of similarity degrees based on the first two kinds of code features, and based on the third kind of code feature, respectively, and use the summation of the two similarity degrees to determine the final reused OSS versions.

For the code features *constants in assignments* and *constant parameters in function calls*, given a binary function *BinFunc*, we find its similarity degree $constants_{similarity}$ with a source function *SrcFunc* calculated by Eq. (4) as follows. We use $BinFunc_{const}$ and $SrcFunc_{const}$ to represent the aforementioned two kinds of feature instances extracted from the target binary function and the source function, respectively. We use the notation # to denote the number of items in a set. If the total number of constants in a binary function is much more than that in a source function, only considering the match ratio may cause a false positive. So we also add the ratio of the binary feature instances number to the source feature instances number to the equation.

$$\begin{aligned} &constants_{similarity}(BinFunc, SrcFunc) \\ &\quad = \frac{\#(BinFunc_{const} \cap SrcFunc_{const})}{\#SrcFunc_{const}} \\ &\quad \times \frac{\#BinFunc_{const}}{\#SrcFunc_{const}} \end{aligned} \tag{4}$$

Next, we show how to compute the similarity degrees between binary and source functions based on in/out-degrees of a function on call graph. We define in/out-degree vectors of a function as an ascending sequence of its numbers of incoming/outgoing edges to/from the function, respectively. For example, if function A is called by function B one time and function C two times[1], and function A calls function D three times and function E one time, the in-degree vector of this function is (1, 2) and the out-degree vector of this function is (1, 3). If two vectors do not have the same number of dimensions, we pad the shorter vector with 0 from its front to make them have the same length. Thus we use Euclidean distance (Euclidean Distance 2020) to compute similarity as follows, where the subscripts "*in*/*out_degree*" are used to represent the corresponding in/out-degree vectors.

$$\begin{aligned} &callgraph_{similarity}(BinFunc, SrcFunc) \\ &\quad = distEclud(BinFunc_{in\_degree}, SrcFunc_{in\_degree}) \\ &\quad + distEclud(BinFunc_{out\_degree}, SrcFunc_{out\_degree}) \end{aligned} \tag{5}$$

Given a target binary function *BinFunc*, a source function *SrcFunc* is considered to be matched with *BinFunc*, if it has the highest score ($constants_{similarity} + callgraph_{similarity}$) with *BinFunc*.

Given a target binary file, to determine the OSS version for single-version reuse type, B2SMatcher compares all the binary functions of the target binary file with all unique source functions of candidate versions obtained from the rough match stage. The candidate version that

[1] All feature instances in this article are extracted statically, i.e., without the need of dynamically running programs. Here means that there are two call sites in C's function body that both call A.

matches the maximum number of unique source functions with the target binary file is considered to be the reused version.

For multi-version reuse type, we match binary functions with unique source functions of all OSS versions, and then decide the version range of OSS to be those versions that have at least one matched unique function with a function of the target binary file.

Architecture

Figure 6 shows B2SMatcher's architecture, it contains three main modules: Collecter, Extractor and Detector. The Collecter module performs information collection. By taking advantage of API provided by github (GitHub 2020) and CVEdetails (CVEDetails 2020), we obtain OSS source code packages and details of vulnerabilities. We use a powerful scraping framework Scrapy[75] to collect real-world software.

The Extractor module extracts features from source code with the help of ANTLR (Parr and Quong 1995). It is difficult and time-consuming to find a proper build environment to complete the build process for each version of a project. We extract the code features from compilation-related files by developing some static analysis tools with the parser generator ANTLR 4.5.3 (Parr and Quong 1995). ANTLR parses C/C++ code based on the concept of fuzzy parsing and does not require a build environment. For each source file, it parses as much symbols as it can. For each binary to be analyzed, the Extractor module extracts relevant features by using IDAPython (IDAPython 2020).

Detector module identifies reused OSS versions by the following steps:

1. It performs a rough match by applying the matching method on program-level features.
2. It identifies reuse type based on the match score and matched features obtained from Step 1.
3. Detector module performs a precise match stage based on reuse types and function-level features.

Having matched OSS versions, and obtaining known vulnerability information of these versions by Collector, Detector finally highlights 1-day security risks for the target binaries.

Evaluation

In this section, we evaluate the precision and efficiency of B2SMatcher. We also explore a large number of real-world software and find that some commercial software like Teamviewer (TeamViewer 2020) and Zoom (Zoom 2020) reuse vulnerable OSS versions.

Dataset

We construct three datasets to evaluate B2SMatcher including one candidate OSS dataset and two binary file datasets.

Candidate OSS dataset (S)

This dataset includes not only commonly used OSS, but also OSS with known vulnerabilities. First, we get all versions of the top 10 frequently reused libraries according to the reuse detection result given by B2SFinder and OSSPolice. For the purpose of reporting potential vulnerabilities caused by OSS reuse, we obtain vulnerability information from CVEdetails (CVEDetails 2020) and collect all versions of those third-party libraries which contain at least 30 previously disclosed vulnerabilities.

Up to now, this dataset contains 243 open source libraries, in a total of 6351 versions. We have categorized these open source libraries as shown in Table 4. The complete list of projects in this dataset is available at (Detailed datasets used in this paper 2020).

Binaries with labeled versions (B1)

Because there is no publicly available ground-truth dataset, we manually label 585 binaries to evaluate the precision of our work in identifying versions of OSS. B1 contains ELF files compiled from multiple OSS, types of which include font rendering (e.g. freetype), image

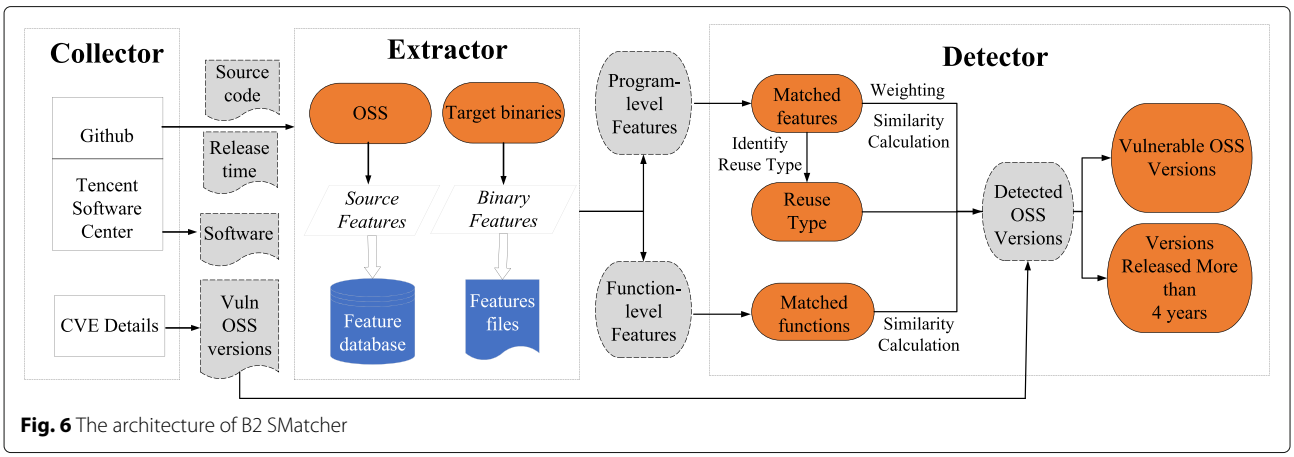

Fig. 6 The architecture of B2 SMatcher

Table 4 The categories of candidate OSS

Project type	Examples
Image processing	libpng, openjpeg, libjpeg
Encryption and decryption	openssl, botan
Sound/video processing	ffmpeg, libsndfile
Document formatting	libxml2, tinyxml, poppler
Font processing	freetype, libtiff
Protocol	openssh, libssh, dnsmasq
Database	sqlite
Compression	bzip2, zlib, unrar
Other	libvnxcserver, opencv

processing (e.g. LibPNG), and document processing (e.g. libxml2). In order to evaluate B2SMatcher on software in the real world, we obtain the versions of open source libraries used in open source software, according to the license and version description files in their source code package. We check all reused versions extracted from the source and verified them in binaries. These software cover different areas, such as video parsing (e.g. VLC), PDF rendering (e.g. SumatraPDF) and audio editing (e.g. Audacity). Binary files of these open source software are also contained in B1.

All binary files in B1 are shown in Table 5. Take the second row as an example, we manually compile 67 versions of freetype from source code. As shown in the penultimate row, we download VLC from its official website and obtain libxml2.dll from the VLC installation package. All sqlite.dll are multi-version reuse cases, we obtain them from real world commercial software. The rest binary files are single-version reuse cases.

Real-world commercial software (B2)

This dataset contains 217 commercial closed source software which is crawled from Tencent Application Center. There are overall 3889 binary files in this dataset. Due to space limit, partial software are presented in Table 6. The complete list of B2 is available at (Detailed datasets used in this paper 2020).

Precision

We manually label 585 pairs of version-level reuses, denoted as B1. We query binaries in B1 against our OSS projects dataset S, which contains 6351 OSS versions. In this part, the accuracy of B2SMatcher is evaluated from two aspects. First, B2SMatcher is compared with state-of-the-art tools B2SFinder and OSSPolice. Second, B2SMatcher is compared with a mature commercial tool Cybellum of which functionality is to create a detailed genome map of software components. We also evaluate the effectiveness of the two-stage identification in this section.

Table 5 The list of binary files in B1

Project name	All versions	Data source	Format
freetype	67	Compiled from source code	ELF
libjpeg-turbo	22	Compiled from source code	ELF
libjpeg	23	Compiled from source code	ELF
openjpeg	11	Compiled from source code	ELF
jbig2dec	8	Compiled from source code	ELF
libpng	210	Compiled from source code	ELF
libxml2	112	Compiled from source code	ELF
sqlite	113	Compiled from source code	ELF
zlib	1	Audacity.exe	PE
freetype.dll	6	Official websites	PE
freetype.dll	4	Mupdf, VLC, SamatraPDF, GIMP	PE
libfreetype.dll	1	GIMP	PE
libpng.dll	1	VLC	PE
libxml2.dll	1	VLC	PE
sqlite.dll	5	Matlab, QQMusic, TIM, QQPCMGR, Storm Codec	PE
total	585		

In "Compilation-related files" section, we have designed an approach based on clustering and decision tree to predict the compilation-related files. At the end of this section, we evaluate the accuracy of this approach.

Comparison with B2SFinder and OSSPolice

The experimental results of B2SMatcher compared with B2SFinder (Yuan et al. 2019) and OSSPolice (Duan et al. 2017) are shown in Table 7.

B2SFinder does not directly support version identification, we make a little improvement for comparison. Since B2SFinder computes a matching score for each type of

Table 6 The partial list of binary files in B2

Project name	Brief introduction
Teamviewer	Remote access tool
Sunlogin	Remote access tool
Zoom	Enterprise video communication
Tencent meeting	Enterprise video communication
Duet display	External display software
Tencent PC manager	Freeware antivirus protection software
QQmusic	Music player software

Table 7 Comparison of B2SMatcher with other tools

	Versions	TP[1]	FP[1]	P[2]
B2SMatcher		**522**	**63**	**89.2%**
B2SFinder (Top 1)		**7**	581	1.1%
B2SFinder (Top 3)	585	**44**	541	7.5%
OSSPolice		**475**	110	81.2%
Cybellum		**481**	104	82.2%

[1]TP: True Positive. FP: False Positive
[2]P: Precision

code feature, we sum up the matching scores of all code features for each version and rank the versions in descending order. We also introduce two criteria for B2Sfinder: (1) the version with the highest score is the correct version (top1), (2) the versions with top 3 scores contain the correct version (top3). OSSPolice supports version identification, thus we can directly compare with it.

As shown in Table 7, we find that B2SMatcher outperforms OSSPolice and B2SFinder in all settings. We first analyze why B2SFinder performs so badly. In order to analyze the effectiveness of each code feature used in B2Sfinder, we use the matching score of one feature each time for version identification and the result is shown in Fig. 7. As can be seen from Fig. 7, the code feature of constant branches in if/else statements has the highest false positive, where 95% of versions can not be correctly identified. We briefly analyze how B2SFinder computes a matching score. We use BIN to denote the set of code feature instances, i.e., concrete feature objects, extracted from target binaries, use OSS to denote the set of feature instances extracted from open source software. The calculation method of B2SFinder can be interpreted into $\frac{\#(BIN \cap OSS)}{\#OSS}$, where the notation # represents the number of instances in a set.

To give a concrete example, we found the numbers of the matched feature instances, i.e., $\#(BIN \cap OSS)$, between *freetype-VER-2-5-0-1.so* and the freetype versions of 2-5-0-1, 2-5-1, 2-5-2, 2-5-3, 2-5-5, 2-5-0 are the same. However, version 2-5-3 has the maximum matching score calculated by B2SFinder because the total number of features, i.e. $\#OSS$, in version 2-5-3 is the least. Therefore *freetype-VER-2-5-0-1.so* is wrongly identified as version 2-5-3 by B2SFinder. We recognize versions based on both the total number of matched features and similarity, and the result presented in Table 7 shows that the total numbers of false positives have been significantly reduced.

Our tool can correctly identify 8% more OSS versions than OSSPolice. It detects fewer OSS versions mainly for two reasons. (1) OSSPolice claims that it identifies versions mainly through version strings such as "inflate 1.2.5.3 Copyright 1995-2011 Mark Adler" in *zlib-v1.2.5.3*. However, some OSS, e.g. *freetype* and *jbig2dec*, does not have version strings among versions. Besides that, common commercial software tends to strip away such kinds of strings. (2) OSSPolice uses "NormScore" (a similarity calculation formula defined by OSSPolice) to obtain matched versions if no version string can be used. NormScore can also be interpreted into the way of $\frac{\#(BIN \cap OSS)}{\#OSS}$, of which limitation is already discussed in the case of B2SFinder.

For multi-version reuse cases, B2SFinder identifies versions relying on maximum similarity and OSSPolice recognizes versions mainly relying on version strings. Therefore, they can not handle multi-version reuse cases well. Take the library SQLite3.dll in the software of Tencent PC Manager with version 13.5.20525 as an example, this binary file has the version string "3.7.5". OSSPolice considers that the version of this binary is SQLite-v3.7.5. The identification result of B2SFinder is also SQLite-v3.7.5. However, with the help of function-level features,

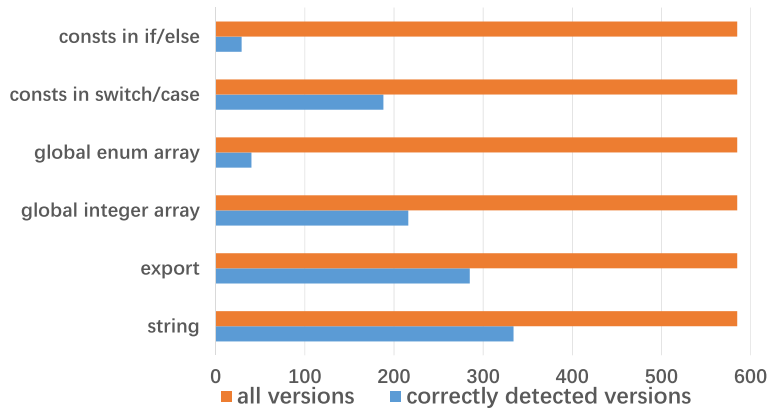

Fig. 7 Effectiveness of each feature in B2SFinder

B2SMatcher finds that this binary contains functions which are in versions of {v3.8.8, v3.8.8.1, v3.8.8.2, v3.8.8.3}. Thus the final identification results given by B2SMatcher are {v3.7.5, v3.8.8, v3.8.8.1, v3.8.8.2, v3.8.8.3}.

In conclusion, B2SMatcher not only combines program-level features and function-level features, but also improves the similarity calculation by considering the total matched features. Besides that, B2SMatcher can identify multi-version reuse cases and perform better than OSSPolice and B2SFinder.

Comparison with Cybellum

We run Cybellum against binaries in B1 and compute the numbers of its true positives and false positives. As can be seen from Table 7, the precision of Cybellum is 82.2%, which is 7% lower than the one of B2SMatcher.

In the identification of some OSS versions, B2SMatcher performs much better than Cybellum. For example, B2SMatcher can correctly identify 96.4% of libxml2 versions in B1. The precision of the Cybellum is only 63.1%. B2SMatcher can correctly identify 74.6% freetype versions in B1 and the precision of Cybellum is only 48.1%. For multi-version cases, Cybellum can not identify the correct versions.

Effectiveness of the two-stage identification

B2SMatcher employs a two-stage version identification approach. It uses program-level features at the rough match stage and uses function-level features at the precise match stage. We use B2SMatcher to identify the version of the binary file in B1 and evaluate the contribution of each stage. As shown in Table 8, B2SMatcher obtains a precision of 82.1% by using the rough match. By adding a precise match stage, B2SMatcher is able to correctly identify 42 more versions and improve the precision from 82.1% to 89.2%.

We further evaluate the effectiveness of each function-level feature. A total of 6405 unique functions are selected from different versions of freetype as a testcase set. As shown in Table 9, we present the precisions of function matching based on *function in/out-degree* and constant type features in the second and the third row, respectively. The precision of function matching based on the *function in/out-degree* feature is above 80%, the precision of function matching based on the constant type feature is nearly 80%. We think that the corresponding precision rates are acceptable.

Table 8 Effectiveness of two stage detection

B2SMatcher	Versions	TP[1]	FP[1]	P[2]
Rough match	585	480	105	82.1%
Rough+Precise match		522	63	89.2%

[1] TP: True Positive. FP: False Positive
[2] P: Precision

Table 9 Effectiveness of function level features

Feature name	Total	TP[1]	FP[1]	P[2]
In/out-degrees of a function callgraph	6409	5384	1025	84%
Constant type features		5072	1337	79.1%

[1] TP: True Positive. FP: False Positive
[2] P: Precision

Effectiveness of the compilation-related file prediction

In "Compilation-related files" section, we have designed an approach based on clustering and decision tree to predict the compilation-related files. Here, we perform an evaluation on the accuracy of correctly predicted complation-related files using our approach, which is carried on 4 projects containing 5 libraries and 280 mainline versions in total. The result of the evaluation shows that nearly 97% of truly compiled files are correctly predicted. We describe the detailed experiment below.

We manually compiled in total 280 mainline versions of zlib(30), freetype(67), libxml2(113) and openssl(70). An OSS project may generate more than one library. For example, the project OpenSSL generates two dynamic libraries libssl.so and libcrypto.so. After the build process, for each library, the list of real compilation-related files is defined as true_compiled_files = $\{tf_1, tf_2, ..., tf_m\}$, the list of predicted compilation-related files is defined as predicted_compiled_files = $\{pf_1, pf_2, ..., pf_n\}$. We use the following three indicators for evaluation.

- TP: If predicted_compiled_files and true_compiled_files are exactly the same, we think it is a true positive.
- FP: If predicted_compiled_files and true_compiled_files have different files, we consider it as a false positive.
- F_Ratio: It is an indicator defined by us. We use the following equation to calculate F_ratio and use the notation # to denote the number of items in a set, the set operation $\triangle$ to denote the symmetric difference between two sets. F_Ratio represents the difference between predicted_compiled_files and true_compiled_files.

$$F_{Ratio} = \frac{\#(true\_compiled\_files \triangle predicted\_compiled\_files)}{\#true\_compiled\_files}$$

We evaluate our approach on the 4 OSS mentioned above and the result is shown in Table 10. The second column shows the name of dynamic libraries generated after the build process. The third column represents the total number of versions predicted. Although the TP of some OSS is relatively low, the F_ratio of all OSS

Table 10 The effectiveness of prediction method

Name	Dynamic library	Total number	TP	FP	F_Ratio
zlib	libz.so	25	25	0	0
freetype	libfreetype.so	43	30	13	0.0325
libxml2	libxml2.so	85	84	1	0.03
openssl	libssl.so	58	17	41	0.033
	libcrypto.so	58	6	52	0.0155

shows that there is not much difference between predicted_compiled_files and true_compiled_files. We think that the result is promising and shows the effectiveness of our approach.

Efficiency

Our experiments are mainly run on a machine equipped with 8 cores of Intel(R) Xeon(R) Gold 6133 CPU @ 2.50GHz, 16 GB memory and 300GB disks.

For the purpose of improving the efficiency of B2SMatcher, we employ machine learning methods to solve selective build and obtain compilation-related files mentioned in "Compilation-related files" section. Moreover, we eliminate repeated functions among versions. We randomly select 146 OSS source code and show the quantitative relationship between the number of compilation-related files and all source files in Fig. 8. On average, only 32.5% of source files are compiled. Extracting features only in compilation-related files can therefore effectively reduce the total number of files to be analyzed. Besides, we count the numbers of unique functions and all the functions for each source code package. The statistic result is shown in Fig. 9. The functions that need to be extracted and compared are effectively reduced from 2784 to 656 on average.

To evaluate the identification time of B2SMatcher, for each binary in B1, we spent on average 145.12 seconds on features matching, of which the program-level and the function-level feature matching processes contribute to 2.27% and 97.73%, respectively.

Real-world software exploration

We use B2SMatcher to conduct a large-scale analysis between 6351 OSS source code from dataset S and 3889 binary files from dataset B2. By analyzing the experimental results, we get the following findings. First, *multi-version reuse* is a common phenomenon, the proportion of it is up to 16.3%. We find that binary files that reuse LibTIFF and SQLite are more likely to be multi-version reuse cases.

Second, we make an overall security risk assessment for the binaries in B2, based on version-level reuses results obtained by B2SMatcher. We select the top 3 OSS projects which are most frequently reused, i.e., Zlib, SQLite and libxml2, and show their version distributions in Fig. 10. The horizontal axis shows the release time of each identified version. We are surprised to find that 100% reuses of Zlib and more than 50% reuses of libxml2 and SQLite contain the versions that are released at least 3 years ago (earlier than 2017).

We identify those versions with at least one previously disclosed vulnerability according to CVEdetails (CVEDetails 2020) as a vulnerable version and show vulnerable version distribution in Fig. 11. The versions of an OSS

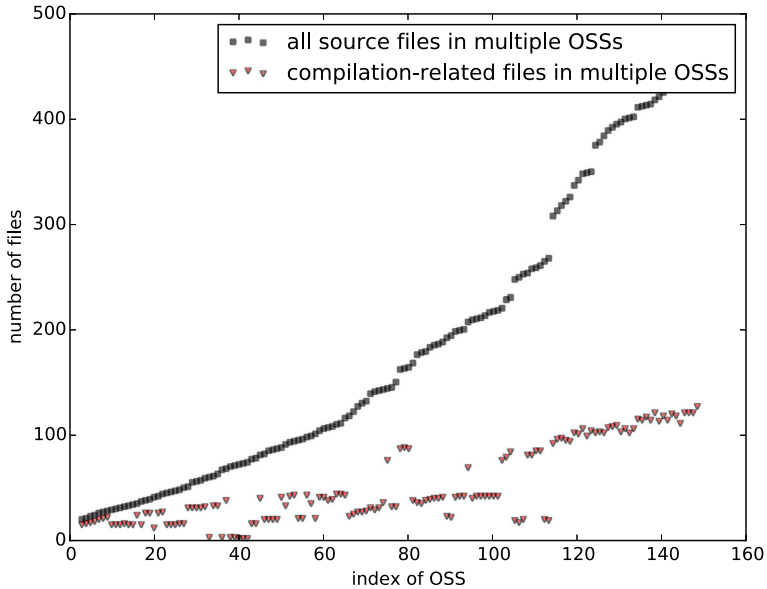

Fig. 8 All source files vs compilation-related files in multiple OSS

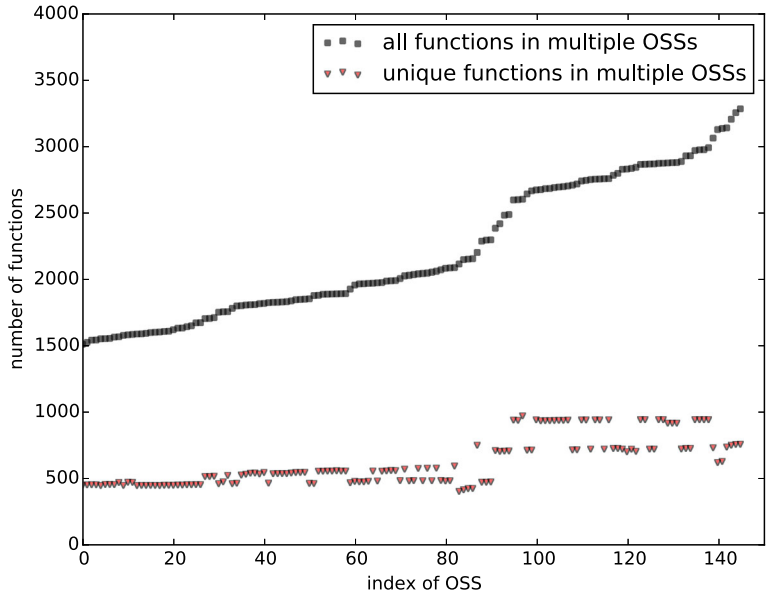

Fig. 9 All functions vs unique functions in multiple OSS

that have the same number of disclosed vulnerabilities are grouped into one column. The vulnerability numbers are decorated on the yellow curve in Fig. 11. We find that most reuses contain vulnerable OSS versions. For example, as can be seen from the highest column in Fig. 11, more than 60% reuses of SQLite use the version that contains a disclosed vulnerability.

Case study

In the end, we show some analysis results of recently released and popular software. According to an annual analysis report OSSRA (2020 Open Source Security and Risk Analysis Report 2020) published by Synopsys, using obsolete components (released 4 years ago or not developed in recent two years) or using a version of OSS that contains at least one CVE brings security risk. After getting the version of reused OSS, we classify it as a vulnerable version if it meets any of the following criteria. First, the OSS of this version has been disclosed at least one vulnerability. Second, this version is released at least 4 years ago (before 2016).

Then we show how to find security risks in real world software and give examples to illustrate software with different reuse types.

Single-version reuse

We select three well-known software Zoom, Teamviewer, and Tencent PC Manager. B2SMatcher shows in Table 11 that these software contain some vulnerable OSS versions.

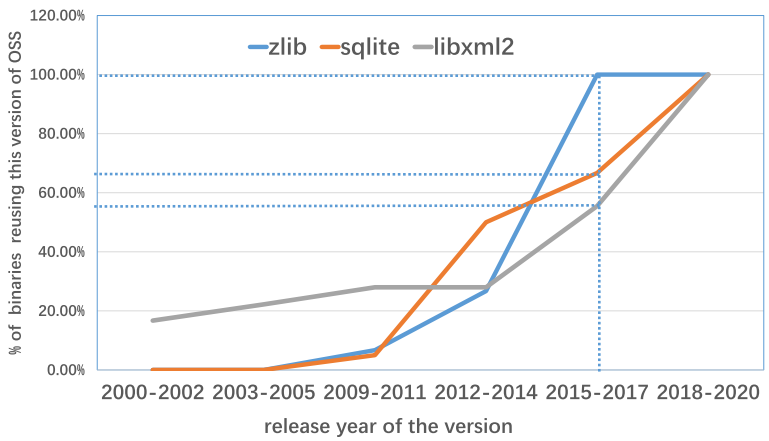

Fig. 10 Version release time of zlib, sqlite, libxml2

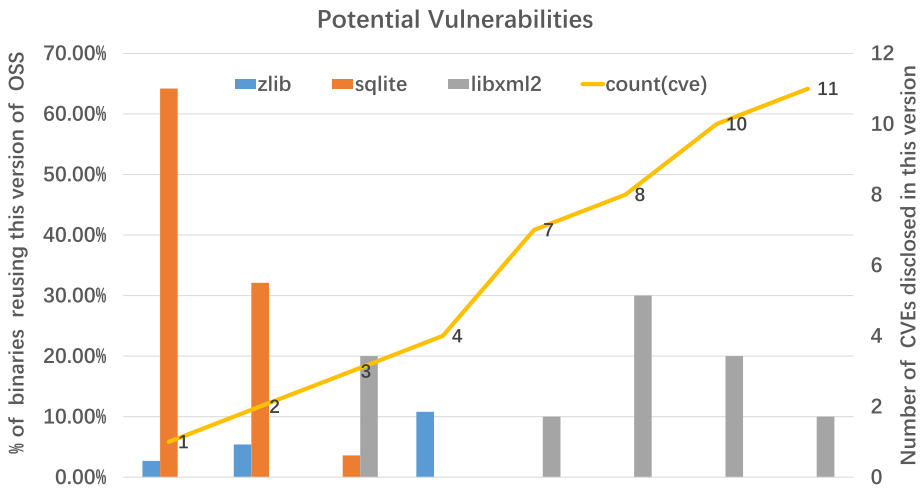

Fig. 11 Ratio of vulnerable versions in zlib, sqlite, libxml2

Nevertheless, B2SMatcher finds that some versions of reused OSS are almost the latest ones and there are not yet any disclosed vulnerabilities on these versions. For example, TeamViewer with version 15.10.5.0 uses libvpx-1.8.2 (released at Mon, 09 Dec 2019 23:09:20 GMT); Zoom with version 5.1.28656.0709 reuses openssl-1.1.1e (released at Tue, 17 Mar 2020 14:31:17 GMT) and libjpeg-turbo-2.0.4 (released at Tue, 31 Dec 2019 07:10:30 GMT).

Multi-version reuse

We take an anonymous database management software "A" as an example, of which name can not be disclosed for the moment, as the discussion and negotiation with the developer are not yet completed. B2SMatcher identifies that A reuses SQLite-v3.7.14 in the rough matching stage by locating the unique string "3.7.14" which only appears in SQLite version 3.7.14. Since there are some unique functions, such as sqlite3OsDlSym and sqlite3_enable_load_extension, which are contained only in SQLite versions of {v3.8.8, v3.8.8.1, v3.8.8.2, v3.8.8.3}, B2SMatcher identifies {v3.8.8, v3.8.8.1, v3.8.8.2, v3.8.8.3} in the precise stage. Combining the results of the two stages, we identify that A may reuse {v3.7.14, v3.8.8, v3.8.8.1, v3.8.8.2, v3.8.8.3}. For each version in the above list, we check whether it is a vulnerable version and obtain related CVEs. Finally, we find that the vulnerable function sqlite3VXPrintf reported by CVE-2015-3416 (disclosed in SQLite-v3.8.8.3) can be found in this software.

Through the above example, by categorizing a binary into different reuse types, we can discover more security risks.

Discussion

In this section, we discuss the limitations of our work and potential solutions.

Version analysis improvements

More OSS

Up to now, we have collected 6351 versions of source code to form our public OSS dataset. It would be worthwhile to collect more open source libraries for analyzing real-world software. Some commonly used open source libraries such as *libevent*, *harfbuzz* are not included in the current dataset, because these libraries contain less than 5 previously disclosed vulnerabilities. Considering that each repository in GitHub has *stargazers count* and *fork count*,

Table 11 Vulnerable versions used in software

Software	Software version	Reused OSS	Reused OSS version	Historical CVE/ Release time
TeamViewer	15.10.5.0	libcurl	7.65.3	CVE-2019-5481, CVE-2019-5842
		zlib	1.2.5	Sun, 18 Dec 2011 18:39:45 GMT
		libjpeg-turbo	8b	Mon, 27 Jul 2015 18:48:40 GMT
Tencent PC Manager	13.5.20525.234	zlib	1.2.8	CVE-2016-9840, CVE-2016-984, CVE-2016-9842, CVE-2016-9843
		libpng	1.4.17	CVE-2015-8472, CVE-2015-8540, CVE-2016-10087
Zoom	5.1.28656.0709	zlib	1.2.5	Sun, 18 Dec 2011 18:39:45 GMT
		sqlite	3.27.2	CVE-2019-8457, CVE-2019-9936, CVE-2019-9937

potentially indicating its popularity, it would be beneficial for improving B2SMatcher's availability to include repositories with more than 200 stargazers or the number of fork count over 50.

More function-level code features

We find that some OSS such as *unrar* and *bzip* have few version-sensitive program-level features, thus their version identifications mainly depend on function-level code features. The function-level features we currently choose are not perfect, they can still be influenced during the compilation. For example, the constant parameters in function calls can be influenced by constant propagation. The in/out-degrees of a function on the call graph will change due to function inlining. However, the result of the precision evaluation of function matching based on these function-level features shown in "Effectiveness of the two-stage identification" section is acceptable. It is also explainable thanks to the way of our similarity degree computation. We compare functions using similarity degree rather than exact match, which is able to tolerate a certain degree of the influence induced during the compilation.

Nonetheless, it is worth exploring more robust function-level features as one of the future work. Inspired by binary similarity analysis work, e.g. DeepBinDiff (Duan et al. 2020) and discovRE (Eschweiler et al. 2016), functions can be regarded as special text and NLP techniques can be used to capture the semantic information of functions. It would be interesting to explore the combination of manually selected code features and the semantic features provided by NLP techniques for achieving better identification results.

Compilation-related files

In this paper, we build a project and make predictions of compilation-related files plainly based on the project's default compilation setting. However, the functions and source code files compiled into a binary depend on compilation options such as macro definition. If the configuration in the default build is significantly different from the one used in the target binary program, there will be indeed a difference between the features extracted from binary and the features extracted from source code. It is worthwhile to explore configuration-aware source-to-binary analysis for further improving the precision.

Vulnerability confirmation

The final goal of our work is to discover potential 1-day vulnerabilities caused by misusing vulnerable versions of OSS, a lot of work needs to be done to confirm whether the vulnerability really exists. First, we need to locate the vulnerable function, then, analyze whether this vulnerable function has been patched. VulDeePecker (Li et al. 2018) uses vulnerable code fragments to train BLSTM and applies this BLSTM to classify code gadgets in the target program. FIBER (Zhang and Qian 2018) generates signatures from patches and adopts symbolic execution to obtain semantic information of the function to be detected, it develops a match engine for searching patch signature in the target binary and verifies the presence of patches in functions based on the matching result. In the future, we would like to investigate more methods for determining whether the vulnerabilities really exist.

Related work

Improper use of OSS will bring security issues, OSS reuse detection in COTS software has become imperative. The previous work related to us can be categorized into the following lines of work.

Security risk by using OSS

Among the top 10 security risks of OWASP (OWASP Top 10 Application Security Risks 2020), components that use known vulnerabilities are the ninth-highest risk. Synopsys (2020 Open Source Security and Risk Analysis Report 2020), based on its product Black Duck's audit results of closed-source software, believes that 96% of the software contains open source components, and most of the components are out-of-date and contain many known vulnerabilities. Some articles track and study the specific scenarios of using components with known vulnerabilities. Wang et al. (Wang et al. 2020) study usages, updates and risks of OSS in JAVA projects, and provide experiences on maintaining them as third-party libraries. Cadariu et al. (Cadariu et al. 2015) point out that the use of third-party components may introduce security vulnerabilities in the software system. They present a tool, Vulnerability Alert Service(VAS), to track known vulnerabilities in software systems.

Serena et al. (Ponta et al. 2018) mainly focus on whether the vulnerability in OSS can be exploited, they combine dynamic and static analysis to determine the accessibility of a vulnerable part.

Code clone detection

Considering the relevance to our work, we mainly focus on the following two analysis methods.

Binary-to-Binary

Binary code analysis measures the similarity between two binaries. It is widely used for code clone detection. Tang et al. (Tang et al. 2020) introduce a tool Libdx, they use contents in read-only data segments as features and propose a concept of logic feature block to cope with the feature duplication. Some other features are also taken into consideration, such as numeric features in instruction (Eschweiler et al. 2016), callgrah (Gao et al. 2008;

Liu et al. 2018; Wang et al. 2009) and control flow graph (Eschweiler et al. 2016; Xu et al. 2017; Feng et al. 2016), and semantic related information including IO behavior (Pewny et al. 2015) and execution traces (Chandramohan et al. 2016). Besides manually selected features, some work automatically extracts semantic information with the help of NLP techniques (Ding et al. 2019; Duan et al. 2020; Yu et al. 2020).

Binary-to-Source

Binary-to-source comparison usually extracts feature instances from source files in advance, and detects potential reused OSS projects for target binaries. We give a brief introduction to the following work. BinPro (Miyani et al. 2017) extracts 7 types of code features. After using SVM to assign weights to features, it uses a combinatorial optimization algorithm to calculate the similarity between source functions and binary functions. The Binary Analysis Tool (BAT) (Hemel et al. 2011) applies string literals as the only code feature and proposes a data compression method to compute the similarity score. OSSPolice (Duan et al. 2017) exploits string literals and exported function names as code features, and devises a hierarchical indexing scheme to detect OSS reuse. B2SFinder (Yuan et al. 2019) utilizes 7 kinds of code features and designs multiple methods to solve different types of reuse. Karta (Karta 2020) is an IDA Python plugin, it mainly uses string and numeric features and employs a heuristic matching algorithm for matching open source projects in binaries.

Conclusion

In this paper, we implement a fine-grained version identification tool B2SMatcher for OSS reused in COTS binary files. Overall, B2SMatcher selects five kinds of version-sensitive code features, forming the program-level and function-level features, and design a two-stage identification approach. For the purpose of precisely extracting features, B2SMatcher uses the K-means algorithm and decision tree to predict compilation-related files. For improving efficiency, B2SMatcher eliminates comparisons on duplicate functions between versions. We evaluate B2SMatcher with 6351 versions of source code and 585 labeled binaries, the experiment result shows that B2SMatcher outperforms the two most closely related approaches and one commercial tool. With B2SMatcher, we find that popular software in the real world, such as TeamViewer and Zoom, reuses vulnerable OSS versions.

Acknowledgments

The authors would like to thank Yican Yao for preparing partial experiments' environment and conditions.

Authors' contributions

GB proposed the technical route. GB and LX performed the experiments and drafted the paper. YX and WH made crucial contributions on the technical route and revised the article. XHL and ZMY revised the article. The author(s) read and approved the final manuscript.

Funding

This research was supported (in part) by the National Natural Science Foundation of China (Grant No. 61802394, U1836209), Key Program of the National Natural Science Foundation of China (Grant No. 62032010).

Availability of data and materials

All public dataset sources are as described in the paper.

Declarations

Competing interests

The authors declare that they have no competing interests.

Author details

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.

Received: 25 January 2021 Accepted: 29 March 2021
Published online: 02 August 2021

References

2020 Open Source Security and Risk Analysis Report (2020). https://www.synopsys.com/zh-cn/software-integrity/resources/reports/2020-open-source-security-risk-analysis.html. Accessed 10 Apr 2021

Cadariu M, Bouwers E, Visser J, van Deursen A (2015) Tracking known security vulnerabilities in proprietary software systems. In: Proceedings of 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Software Analysis, Evolution, and Reengineering, New York. pp 516–519

Chandramohan M, Xue Y, Xu Z, Liu Y, Cho CY, Tan HBK (2016) Bingo: Cross-architecture cross-os binary search. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. USENIX Association, Kyoto. pp 678–689

CVEDetails (2020) Free CVE security vulnerblity database source. https://www.cvedetails.com/. Accessed 10 Apr 2021

Cybellum (2020) Uncover the Software Components Inside Your Vehicles and Identify All Vulnerabilities. https://cybellum.com/. Accessed 10 Apr 2021

Decision tree (2020). https://en.wikipedia.org/wiki/Decision_tree. Accessed 10 Apr 2021

Detailed datasets used in this paper (2020). https://github.com/summerban/B2SMatcher-cybersecurity. Accessed 10 Apr 2021

Ding SH, Fung BC, Charland P (2019) Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). Springer, Kyoto. pp 472–489

Duan R, Bijlani A, Ji Y, Alrawi O, Xiong Y, Ike M, Saltaformaggio B, Lee W (2019) Automating patching of vulnerable open-source software versions in application binaries. In: Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)

Duan R, Bijlani A, Xu M, Kim T, Lee W (2017) Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 2169–2185

Duan Y, Li X, Wang J, Yin H (2020) Deepbindiff: Learning program-wide code representations for binary diffing. In: Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20). Springer, Shanghai

Eschweiler S, Yakdan K, Gerhards-Padilla E (2016) discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In: Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS). The Internet Society, London

Euclidean Distance (2020). https://en.wikipedia.org/wiki/Euclidean_distance. Accessed 10 Apr 2021

Feng Q, Zhou R, Xu C, Cheng Y, Testa B, Yin H (2016) Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC

Conference on Computer and Communications Security. Springer, Beijing. pp 480–491
Gao D, Reiter MK, Song D (2008) Binhunt: Automatically finding semantic differences in binary programs. In: Proceedings of the International Conference on Information and Communications Security. The Internet Society, London. pp 238–255
GitHub (2020) Where the World Builds Software. https://github.com/. Accessed 10 Apr 2021
Heartbleed (2020). https://en.wikipedia.org/wiki/Heartbleed. Accessed 10 Apr 2021
Hemel A, Kalleberg KT, Vermaas R, Dolstra E (2011) Finding software license violations through binary code clone detection. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp 63–72
IDAPython (2020). https://www.hex-rays.com/products/ida/support/idapython_docs/. Accessed 10 Apr 2021
K-means Clustering (2020). https://en.wikipedia.org/wiki/K-means_clustering. Accessed 10 Apr 2021
Kamiya T, Kusumoto S, Inoue K (2002) Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670
Karta (2020). https://github.com/CheckPointSW/Karta. Accessed 10 Apr 2021
Kim S, Woo S, Lee H, Oh H (2017) Vuddy: A scalable approach for vulnerable code clone discovery. In: Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland). IEEE, San Jose. pp 595–614
Li Z, Lu S, Myagmar S, Zhou Y (2006) Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans Softw Eng 32(3):176–192
Li M, Wang W, Wang P, Wang S, Wu D, Liu J, Xue R, Huo W (2017) Libd: scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE. pp 335–346
Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681 13:266–267
LibreOffice (2020) A Free and Open-source Office Suite, a Project of The Document Foundation. https://en.wikipedia.org/wiki/LibreOffice. Accessed 10 Apr 2021
Liu B, Huo W, Zhang C, Li W, Li F, Piao A, Zou W (2018) αdiff: cross-version binary code similarity detection with dnn. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. The Internet Society, San Diego. pp 667–678
MATLAB (2020). https://en.wikipedia.org/wiki/MATLAB. Accessed 10 Apr 2021
Miyani D, Huang Z, Lie D (2017) Binpro: A tool for binary source code provenance. arXiv preprint arXiv:1711.00830 Suppl 3:149–170
One-hot Embedding (2020). https://en.wikipedia.org/wiki/One-hot. Accessed 10 Apr 2021
OWASP Top 10 Application Security Risks (2020). https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. Accessed 10 Apr 2021
Parr TJ, Quong RW (1995) Antlr: A predicated-ll (k) parser generator. Softw Pract Experience 25(7):789–810
Pewny J, Garmany B, Gawlik R, Rossow C, Holz T (2015) Cross-architecture bug search in binary executables. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P). Springer, Kyoto
Ponta SE, Plate H, Sabetta A (2018) Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In: Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). International Conference on Software Maintenance and Evolution, New York. pp 449–460
Repo Statistics on Github (2020). https://octoverse.github.com. Accessed 10 Apr 2021
Shahkar A (2016) On matching binary to source code. PhD thesis, Concordia University
Storm Codec 7 (2020) A Video Codec Pack. https://storm-codec-7.en.uptodown.com/windows. Accessed 10 Apr 2021
t-Distributed Stochastic Neighbor Embedding (2020). https://lvdmaaten.github.io/tsne/. Accessed 10 Apr 2021
Tang W, Luo P, Fu J, Zhang D (2020) Libdx: A cross-platform and accurate system to detect third-party libraries in binary code. In: Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). Software Analysis, Evolution, and Reengineering, New York. pp 104–115
TeamViewer (2020). https://www.teamviewer.com/en/. Accessed 10 Apr 2021
Tencent Software Download Official Version (2020). https://pc.qq.com/. Accessed 10 Apr 2021
VMware Workstation Pro (2020). https://en.wikipedia.org/wiki/VMware_Workstation. Accessed 10 Apr 2021
Wang Y, Chen B, Huang K, Shi B, Xu C, Peng X, Liu Y, Wu Y (2020) An empirical study of usages, updates and risks of third-party libraries in java projects. arXiv preprint arXiv:2002.11028 Suppl 3:149–170
Wang X, Jhi Y-C, Zhu S, Liu P (2009) Detecting software theft via system call based birthmarks. In: Proceedings of the 2009 Annual Computer Security Applications Conference. The Internet Society, San Diego. pp 149–158
Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Springer, Beijing. pp 363–376
Yu Z, Cao R, Tang Q, Nie S, Huang J, Wu S (2020) Order matters: Semantic-aware neural networks for binary code similarity detection. Proc AAAI Conf Artif Intell 34:1145–1152
Yuan Z, Feng M, Li F, Ban G, Xiao Y, Wang S, Tang Q, Su H, Yu C, Xu J, et al. (2019) B2sfinder: detecting open-source software reuse in cots software. In: Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE. pp 1038–1049
Zhang H, Qian Z (2018) Precise and accurate patch presence test for binaries. In: Proceedings of the 27th USENIX Security Symposium (Security). Springer, Oakland. pp 887–902
Zoom (2020). https://zoom.us/. Accessed 10 Apr 2021

Publisher's Note