B2SFinder: Detecting Open-Source Software Reuse in COTS Software

Muyue Feng, Zimu Yuan*, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang,
Qian Tang, He Su, Chendong Yu, Jiahuan Xu, Aihua Piao, Jingling Xue†, and Wei Huo

Institute of Information Engineering, Chinese Academy of Sciences
†School of Computer Science and Engineering, University of New South Wales
{fengmuyue, yuanzimu, lifeng, bangu, xiaoyang, wangshiyang, tangqian, suhe, yuchendong,
xujiahuan, piaoaihua, huowei}@iie.ac.cn, †jingling@cse.unsw.edu.au

***Abstract*—COTS software products are developed extensively on top of OSS projects, resulting in OSS reuse vulnerabilities. To detect such vulnerabilities, finding OSS reuses in COTS software has become imperative. While scalable to tens of thousands of OSS projects, existing binary-to-source matching approaches are severely imprecise in analyzing COTS software products, since they support only a limited number of code features, compute matching scores only approximately in measuring OSS reuses, and neglect the code structures in OSS projects.**

We introduce a novel binary-to-source matching approach, called B2SFinder[1], to address these limitations. First of all, B2SFinder can reason about seven kinds of code features that are traceable in both binary and source code. In order to compute matching scores precisely, B2SFinder employs a weighted feature matching algorithm that combines three matching methods (for dealing with different code features) with two importance-weighting methods (for computing the weight of an instance of a code feature in a given COTS software application based on its specificity and occurrence frequency). Finally, B2SFinder identifies different types of code reuses based on matching scores and code structures of OSS projects. We have implemented B2SFinder using an optimized data structure. We have evaluated B2SFinder using 21991 binaries from 1000 popular COTS software products and 2189 candidate OSS projects. Our experimental results show that B2SFinder is not only precise but also scalable. Compared with the state of the art, B2SFinder has successfully found up to 2.15x as many reuse cases in 53.85 seconds per binary file on average. We also discuss how B2SFinder can be leveraged in detecting OSS reuse vulnerabilities in practice.

***Index Terms*—COTS Software, OSS, Code Reuse, One-Day Vulnerability, Code Feature, Binary-to-Source Matching**

I. Introduction

With the widespread adoption of structural design patterns and the pressing need for shortening time-to-market, more and more commercial off-the-shelf (COTS) software products are being developed on top of open-source software (OSS) projects. Such a rapid application development leads to several undesirable problems, including licence violations [1, 2] and security issues [2, 3]. Among these problems, OSS reuse vulnerabilities are one of the most severe issues [4, 5].

OSS vulnerabilities can be introduced into COTS software when some vulnerable OSS code is integrated into and thus reused in the software. Such OSS vulnerabilities, which are referred to as *OSS reuse vulnerabilities*, are ubiquitous and can have a serious impact on the security of COTS software. For example, Adobe Reader [6] and Windows Defender [7] were both found to be vulnerable as both had used some vulnerable versions of open-source projects, `Libxslt` and `UnRAR`. In fact, most of OSS reuse vulnerabilities remain in COTS software even if their vulnerable OSS versions have been patched already. According to a Synopsis report [8], 96% of the COTS products audited have reused OSS projects as their components, containing unpatched OSS vulnerabilities in OSS projects released six years ago, on average.

To detect OSS reuse vulnerabilities, it is imperative to identify the OSS projects included in COTS software as precisely as possible. We are therefore motivated to address the underlying *OSS Reuse Detection for COTS software* problem in this paper. While the number of mobile applications keeps increasing, the traditional COTS products that run on desktop computers and servers are still widely used. So we focus on COTS software in this paper. A COTS product usually consists of tens of stripped binaries, most of which are either in the portable executable (PE) format (for Windows) or the executable and linkable format (ELF) (for Linux).

Given a binary file for a target COTS software product and a set of candidate OSS projects, there are two representative approaches to OSS reuse detection. One approach is to compute the similarity between the binary of a given target COTS product and a compiled binary of a candidate OSS project. Despite a lot of prior work on binary similarity detection [9–11], we still see two challenges ahead. First, the fully-automatic compilation for all the candidate OSS projects is nontrivial and usually requires manual work to find appropriate compiler flags to enable their successful compilation. In an experiment that we performed on a total of 2189 OSS projects crawled from Ubuntu Packages [12], we found that only approximately a quarter can be compiled automatically. Second, any binary similarity analysis can be expensive. For a moderate COTS software product, approximately billions of comparisons are needed for finding code reuses from many candidate OSS

* Zimu Yuan is the Corresponding author. The authors from the Institute of Information Engineering, Chinese Academy of Sciences, are also affiliated with Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing Key Laboratory of Network Security and Protection Technology, and School of Cyber Security, University of Chinese Academy of Sciences.

[1] Open-sourced at https://github.com/1dayto0day/B2SFinder.

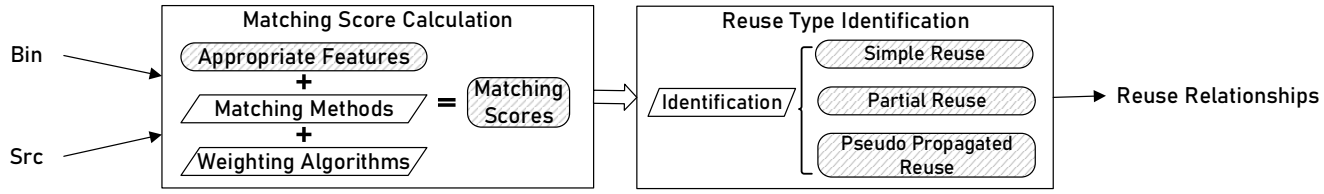

Figure 1: The workflow of B2SFINDER.

projects, making the analysis unscalable (if done naively).

The other approach is to compare the target binary of a COTS software product with the source code of a candidate OSS project directly. In the absence of the compiler flags used for producing the COTS software binary, the prior work on binary-to-source matching [1, 2, 13] relies heavily on *code features* including string-type features like function names and string literals, which are not usually correlated with the compiler flags used. As a result, the possibility of code reuses is measured as the number of common *feature instances* present in both the target binary and a given candidate OSS, where a feature instance denotes a concrete feature object like a function name `jpeg_start_decompress`. While being scalable to tens of thousands of candidate OSS projects [2], applying it directly to OSS reuse detection does not deliver the precision desired due to the lack of code features considered.

In this paper, we address the following three problems:

1. How can we select as many code features as possible while ensuring that all the selected features are traceable in compiled binaries? The key to binary-to-source matching lies in the code features considered. For example, BAT [1] considers string literals only and thus misses 39.7% of code reuses that are not related to string literals (as evaluated later). OSSPolice [2], which considers not only string literals but also exported function names, performs well in the ELF-formatted libraries. However, the PE files for COTS software are usually stripped of these clues, making it impossible for OSSPolice to perform code matching needed in stripped binaries.

2. How can we precisely compute the matching scores with respect to different code features and their feature instances? The prior work [1, 2, 10, 11, 14] usually measures the degree of feature matching by computing a matching score. However, their score computation is imprecise for two reasons: (1) different kinds of features are usually matched by the same process, and (2) different feature instances of the same feature are assumed to contribute equally in feature matching.

3. How can we exploit the code structures of OSS projects to improve reuse identification? In general, a higher matching score does not always imply a higher possibility for code reuse, and vice versa. For example, as shown in the right side of Figure 3, every feature in `LibPNG` is matched with that of `libopenjp2-7.dll`, resulting in a high matching score. However, `libopenjp2-7.dll` only reuses `OpenJPEG` rather than `LibPNG`. This suggests that the complex code structures of OSS projects should also be considered in order to decrease the number of false reuse identifications reported and increase the number of true reuse identifications found.

To address the three aforementioned issues, we propose a novel binary-to-source matching approach, B2SFINDER, for detecting OSS reuses. As shown in Figure 1, B2SFINDER proceeds in two phases, "Matching Score Calculation" and "Reuse Type identification", which are discussed briefly below.

In order to compute matching scores in the first phase precisely, we have aggressively selected seven kinds of stable code features, of which four kinds are not affected by compilation and three kinds are affected slightly during the compilation. By dividing seven kinds of features into three types, *string-type*, *integer-type* and *control-flow-type*, we have designed three corresponding matching methods, exact matching, search-based matching and semantics-based matching. In order to describe the relative importance of different matched feature instances, we introduce two feature instance attributes, *specificity* and *occurrence frequency*. The specificity attribute indicates that a special matched feature instance, e.g., `0x6a09e667`, is more helpful to distinguish one OSS project from others than a trivial one, e.g., `0x0001`. The occurrence frequency attribute denotes the appearance of a matched feature instance across all the candidate OSS projects. The lower the occurrence frequency is, the more significant it is in identifying the reused OSS projects. The effects of these two attributes on code reuses can be captured in a weight computation by combining a bitstream entropy algorithm and a TF-IDF-like weighting algorithm. Overall, we propose a novel feature matching algorithm that combines three matching methods with two importance-weighting methods.

In order to identify different types of code reuses, we take into account the code structures of OSS projects to recognize two types of file groups, *self-implemented groups* and *OSS imported groups* from a third-party project. By leveraging this information, we construct exact reuse relationships between a target COTS software binary and an OSS project. We have identified three types of reuse relationships, *simple reuse*, *partial reuse* and *pseudo propagated reuse*. The former two are true reuse cases while the last one is a false reuse case that should be eliminated. Note that the partial reuse is usually missed by BAT [1] and OSSPolice [2] due to its low matching score while the pseudo propagated reuse is often identified incorrectly by BAT [1] due to its high matching score.

We have implemented B2SFINDER with an optimized data structure, in terms of an inverted index and a Trie (a prefix tree). We have evaluated B2SFINDER on a custom dataset consisting of 21991 binaries from 1000 COTS software products collected from the internet and 2189 popular OSS projects crawled from Ubuntu Packages. Our experimental results show

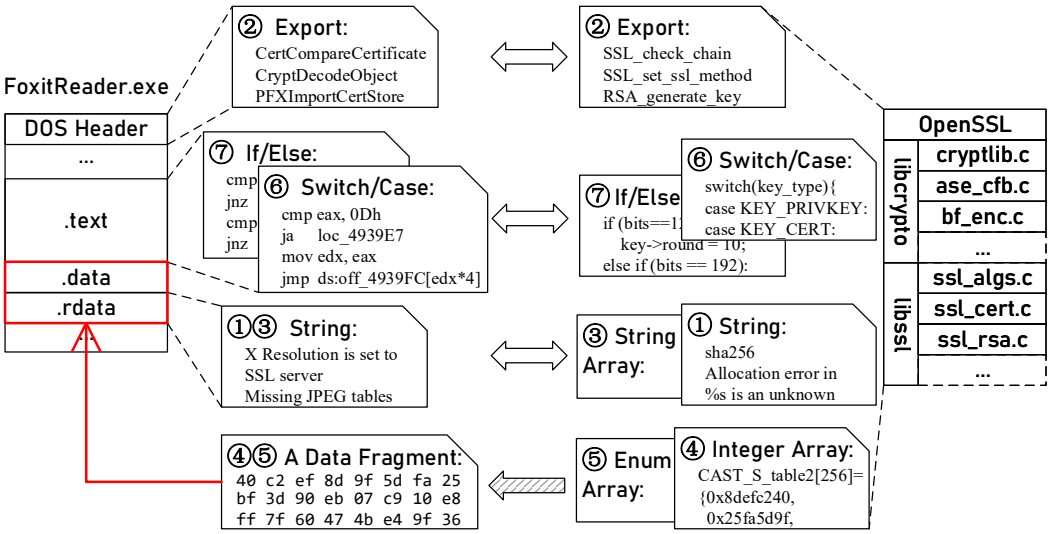

Figure 2: Binary-to-source matching between `FoxitReader.exe` and `OpenSSL` for the seven kinds of code features selected.

that B2SFinder achieves a precision of 92.3% and a recall of 88.5%, thereby significantly outperforming state-of-the-art approaches in all the configurations considered. Comparing with BAT [1], B2SFinder finds 2.15x as many reuse cases in top-1 and 1.47x as many reuse cases in top-5 in 53.85 seconds per binary file on average. In addition, B2SFinder finds that 63.4% of COTS software products reuse OSS projects, with 4.6% of them being reused more than 10 times, and moreover, 54.7 potential vulnerabilities are found to exist in the top 10 frequently reused libraries on average.

Overall, this paper makes the following contributions:

- We propose a novel binary-to-source matching approach, by employing a weighted feature matching algorithm that combines three matching methods (for dealing with different code features) with two importance-weighting methods (for computing the weight of an instance of a code feature in a given software application based on its specificity and occurrence frequency).
- We introduce a new concept of reuse type and utilize it to improve the precision of code reuse detection.
- We have developed a prototyping open-source implementation for B2SFinder and evaluated its efficiency and precision using 21991 binaries from 1000 popular COTS software products and 2189 OSS projects. Compared with the state of the art, B2SFinder has successfully found up to 2.15x as many reuse cases in 53.85 seconds per binary file on average. B2SFinder is also shown to be capable of detecting OSS reuse vulnerabilities in practice.

II. Motivation

We walk through our binary-to-source matching approach and illustrate our insights for detecting OSS reuses by using two real-world motivating examples, `Foxit Reader` and `GIMP`. `Foxit Reader` is a well-known viewing and editing tool for PDF files. The two binaries, the core executable file `FoxitReader.exe` and a dynamically-linked library `ssleay32.dll`, selected from `Foxit Reader` both reuse `OpenSSL`, a widely used OSS project implementing SSL. `GIMP` is a popular raster graphics editor used for image editing. Its dynamic library `libopenjp2-7.dll` reuses an OSS project, `OpenJPEG`. Although the two examples are both in the PE format, our approach is capable of dealing with all the other native binary files in a similar manner.

We explain how B2SFinder analyzes these two examples in its two phases as shown in Figure 1.

A. Matching Score Calculation

Since we compare directly the binary of a COTS software application with the source code of an OSS project, code features selected from both must be matchable. String literals and exported function names can be matched directly as they are not usually correlated to compiler flags. However, as discussed in Section I, such features are not applicable to many binaries in COTS software. In our approach, one key observation is to identify a range of code features that are not likely to change during the compilation (with the exception of string literals and exported function names).

As shown in Figure 2, there are no common exported function names between `FoxitReader.exe` and `OpenSSL`. In addition, both share as few as 19.7% of common string literals. Given these two facts, both BAT [1] and OSSPolice [2] are ineffective in analyzing `FoxitReader.exe`. In order to leverage more stable code features present in both binary and source code, we have examined the `FoxitReader.exe`. Except for the string literals in the `.rdata` segment and exported function names in the `DOS header` segment, we have found some other features potentially selectable from the `.rdata`, `.data` and `.text` segments.

One of these newly selected features is numerical data. These data items are usually the initial values of the global variables stored in the `.data` segment (for non-constant variables) and the `.rdata` segment (for constant variables). Therefore, we have selected two kinds of such features, global

integer arrays and global enumeration arrays. For example, in Figure 2, the global integer array `CAST_S_table2[256]={0x8defc240,0x25fa5d9f,...}` in `OpenSSL` exists in `FoxitReader.exe` as a bitstream with a prefix of `0x40c2ef8d9f5dfa25`. We have opted for array variables instead of scalar variables for performance reasons (as the number of array variables is significantly less).

We observe that we can also find some potential code features in the `.text` segment. Although control flow information can be changed by some compiler optimizations, a sequence of branches representing some complex logic is relatively stable. This is because such a sequence is usually complex and thus does not satisfy the underlying optimization criteria required. For a given binary file, we therefore search for its data or instruction sequences, with the encoded complex switch/case structures and sequences of consecutive if/else conditional branches extracted from its code sections. For example, `FoxitReader.exe` has a jump table, `[0,9,16,17,20]`, which is found to contain the same labels as a switch statement in function `aes_ccm_ctrl()` from `OpenSSL`. This shows that `FoxitReader.exe` reuses `OpenSSL` with a high possibility.

In total, we have selected seven kinds of code features as shown in Figure 2. However, how to match a feature in source code with the same feature that appears in its compiled binary remains challenging, especially for those features that may be slightly changed during the compilation.

For a given code feature, its feature instances in the binary of a COTS software application are extracted by reverse engineering while its feature instances in the source code of an OSS project are extracted by applying program analysis. A *feature instance* is a concrete feature object belonging to a certain kind of features, as illustrated in Figure 2. For different code features, we will recognize them by applying three different matching methods. For different instances of a code feature, we will recognize their relative importance by applying two different importance-weighting methods.

For strings, string arrays and exported functions, their feature instances are all in the form of strings, which can be directly recognized by using string matching algorithms. Specifically, an inverted index is used to accelerate string matching due to the existence of a large number of strings in practice. On the other hand, instances of two numeric features, global integer arrays and global enumeration arrays, are handled differently. Due to the lack of data structure information in binaries, we encode these feature instances found in binaries into bitstreams of certain lengths in certain data types. For each candidate OSS project, each bitstream is then searched in its source code to find its instances in either the big or little endian byte order. Finally, matching the feature instances of control-flow types, constants in a switch/case feature, and constants in an if/else feature, is more complex. We will compare semantically the corresponding constants in one feature instance of a control-flow type in the binary of a COTS software application and another feature instance of a control-flow type in the source code of an OSS project.

Note that the matching results of feature instances are not directly related to how the underlying matching scores are computed. For example, a simple array, `[0x1,0x10,0x100,0x1000]`, can be easily matched with irrelevant data. Thus, we will assign a small *specificity weight* to this array by computing the entropy of its bitstream. In addition, for some arrays that are commonly present in candidate OSS projects (e.g. `SQR_tb[16]={0,1,4,5,16,...}` in `OpenSSL` also exists in at least 14 other OSS projects), a TF-IDF-like *frequency weight* is also used to assign relatively small weights to such feature instances. With the weights thus determined, the degrees of contribution from different features in binary-to-source matching can be distinguished.

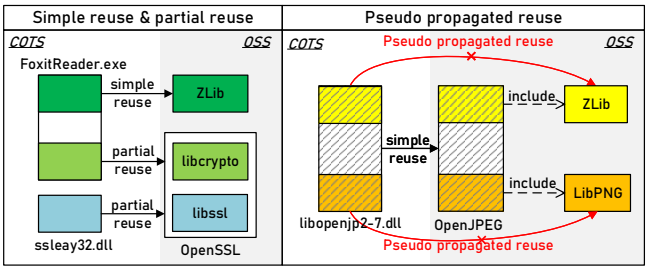

Figure 3: Three types of code reuses.

B. Reuse Type Identification

When we have a feature match between a COTS software application and a candidate OSS project, a high matching score may not always indicate a true reuse and a low matching score may not always indicate a false reuse. To improve the precision in reuse detection, we distinguish between different types of code reuses by exploiting the code structures of OSS projects.

In many cases, a high matching score indicates a true reuse. We define such reuse as *simple reuse*, which may also be discovered by existing approaches [1, 2]. However, we have found two other types of reuses that show inconsistency between their actual reuse relationships and their matching scores obtained, as shown in Figure 3. For example, `libssl` (aka `ssleay`), one of the `OpenSSL` libraries, is generated by only 7.6% of `OpenSSL` source files. Any binaries that reuse `libssl`, e.g., `ssleay32.dll` in `Foxit Reader`, reuse `OpenSSL` partially, despite their relatively low matching scores. In contrast, the matching score is quite high between `libopenjp2-7.dll` in `GIMP` and `LibPNG` because a code fragment in `libopenjp2-7.dll` is similar to that in `LibPNG`. However, this code fragment is actually compiled by the `libpng` module of `OpenJPEG`, which may be a variation of the original `LibPNG`. So the reuse between `libopenjp2-7.dll` and `LibPNG`, which is referred to as *pseudo propagated reuse*, should be eliminated.

We observe that, in order to identify partial reuse and pseudo propagated reuse, it is necessary to consider the complex code structures of OSS projects. To do this, we first decompose an OSS project into independent library modules, by analyzing its compilation process to build the mappings from source files to generated libraries. For example, how the source files in `OpenSSL` are mapped to the library `Libssl` can be determined from its linker flags. We then construct a so-

called *inclusion relationship* among the OSS projects. This is established by comparing the sets of feature instances matched in two OSS projects. As shown in the right side of Figure 3, an *include* edge from `OpenJPEG` to `LibPNG` is added since the set of matched feature instances in `OpenJPEG` contains the set of matched feature instances in `LibPNG`. These two types of code structures are helpful in identifying such reuses.

III. Design

In this section, we introduce our detailed design of B2SFinder for detecting OSS reuses in COTS software.

A. Selecting Code Features

Due to significant differences between source and binary code representations, code features selected are expected to provide a uniform way for enabling feature matching. There are two criteria for selecting a code feature. First, a code feature must be present simultaneously in both binary and source code. Second, a code feature in source code should not be changed drastically during the compilation. Table I lists a total of 10 candidate code features, covering all the features used in existing binary similarity analysis techniques [10, 14–16]. All the 10 features meet the first criterion, but only 7 of the 10 features satisfy the second criterion.

Table I: Candidate code features and their susceptibility.

Feature Class	Feature Name	Susceptible?	Selected?
Symbol	Export	Not	✓
Constant data	String	Slightly	✓
	Global string array	Not	✓
	Integer number	A lot	
	Global integer array	Not	✓
	Global enum array	Not	✓
Control flow	Control Flow Graph	A lot	
	Call Graph	A lot	
	Consts in switch/case	Slightly	✓
	Consts in if/else	Slightly	✓

String literals and exported function names are traditional code features that are also suitable for COTS software. However, some OSS projects, such as `UnRAR` and `bzip2`, may not contain any useful strings for matching purposes, and in addition, COTS software applications often end up with string literals stripped away in order to hide their software composition. In many OSS projects, exported function names are frequently hidden when the code in the OSS projects is called internally. Thus, new code features are called for.

Numerical and control-flow-related features can be invaluable. In the previous binary analysis work [3, 9, 10, 15, 17], constants are widely used, but not all of them are suitable in our setting. We find that numeric constants in source code that appear as immediate values in instructions of its compiled binary code are heavily affected during the compilation. In contrast, global integer arrays, global enumeration arrays, and global string arrays are usually not affected. Furthermore, these global arrays usually contain some key information in an OSS project. For example, an OSS project using the CAST5 cipher must have an `S` table like `CAST_S_table2[256]={0x8defc240,0x25fa5d9f,...}` in `OpenSSL`, making arrays important code features in reuse detection.

```
switch(ax){
case AXIS_NAMESPACE: //8
    {...} break;
case AXIS_CHILD: //3
case AXIS_DESCENDANT: //4
case AXIS_DESC_OR_SELF: //5
case AXIS_ANCESTOR: //0
case AXIS_PRECEDING: //10
    {...} break;
};
```

Listing 1: A switch/case structure.

```
if (a < 0x80)
    *length = 1;
else if (a < 0x800)
    *length = 2;
else if (a < 0x10000)
    *length = 3;
else if (a < 0x200000)
    *length = 4;
else
    {*length = 0; return;}
```

Listing 2: An if/else structure.

Compared with numeric features, control-flow-related features can be more easily changed by compiler optmizations applied during the compilation. Under different compiler flags, the binary codes generated from the same source function may have completely different control-flow structures. A program's call graph will also be affected by function inlining. Hence, the Control Flow Graph (CFGs) and Call Graphs (CGs) between binary and source code are not directly comparable. Fortunately, complex branch sequences, such as complex switch/-case and if/else statements structures, as in Listings 1 and 2, are relatively stable during the compilation. The constants in such structures are also selected as code features.

All of the seven features mentioned above are extracted by analyzing binary and source code. For the binary code, its switch/case features are found by parsing its jump tables, and its if/else features are found by analyzing its `cmp` and jump instructions. For the source code, all numeric features are extracted based on reasoning about the subtrees with specific patterns in its abstract syntax tree, and all the control-flow-related features are found by traversing the conditional branches in its intermediate representation.

B. Matching Code Features

We have divided seven kinds of code features into three different types, strings, integers and control flow, as shown in Colum 1 of Table II. Different methods will be used for matching different types of code features.

Table II: The weighting algorithms for seven kinds of code features.

Feature Name	Specificity Wt.	Frequency Wt.
String-Type Features		
String Export Global String Array	Number of Specific Substrings	S-IDF
Integer-Type Features		
Global Integer Array Global Enum Array	Entropy of Bitstream	S-IDF
Control-Flow-Type Features		
Consts in Switch/Case Consts in If/Else	Length of Const Sequences	S-IDF

Exact Matching for String-Typed Features. As a string always stays the same during the compilation phase, a string extracted from binary code and a string extracted from source code are considered to be equivalent when they are identical.

Search-Based Matching for Integer-Typed Features. Global integer/enum arrays are stored in the data segments of a binary file as searchable continuous bitstreams. Thus, we encode each as a bitstream according to the width of its data type, e.g., 2

bytes, 4 bytes or 8 bytes, and search directly for the bitstream in the `.data` and `.rdata` segments of a binary file.

Semantics-Based Matching for Control-Flow-Typed Features. Switch/case structures and if/else sequences are compared semantically as follows. For a switch/case feature in the source code of an OSS project, we represent it as an unordered list of case label sets with a default branch appended. The default branch can be considered to match with any set of case labels in the binary of a COTS software application. For example, the switch/case feature `[{0},{9},{16},{17},{20}]` from source code is considered to match with `[{0},{9},{16},{17},{20},{1,2,3,4,5,6,7,8,10,11,12,13,14,15,18,19}]` from binary code, since the default branch from the source code matches with the last branch from the binary code. For an if/else feature, we calculate the longest ordered common subsequence for each pair of if/else sequences, with one from source code and one from binary code, and then determine their equivalence according to the length of their common subsequence. For example, an if/else sequence `[0x1,0x80,0x800,0x10000,0x200000,0x10]` from binary code matches with `[0x80,0x800,0x10000,0x200000]` from source code, with a length of `4`.

C. Determining the Importance-Weights of Feature Instances

As OSS projects grow in size, the number of feature instances extracted from these OSS projects will increase rapidly. In fact, feature instances do not contribute equally in feature matching. Therefore, we can reduce the time spent on feature matching by considering only relatively important feature instances. To differentiate feature instances in terms of their contribution to feature matching, each feature instance is assigned a *specificity weight* and a *frequency weight*. These two weights measure the contribution of a feature instance based on the information carried and its occurrence frequency in an OSS project, respectively. Consider an `S` table in the CAST5 cipher (`{0x8defc240,0x25fa5d9f,...}`) as an example. It is unique and less likely to be similar to other data. Thus, this `S` table will have a relatively large specificity weight. However, CAST5 is a popular cipher, with at least 15 OSS projects containing this `S` table. Since such frequent appearance does not help much with reuse detection, this `S` table will be assigned a relatively small frequency weight.

As shown in Table II (Columns 2 and 3), all code features rely on the same algorithm, S-IDF (a variant of TF-IDF), for computing their frequency weights. For three different types of features, three different methods are used for computing their specificity weights. For a string-typed feature, we use the number of its substrings, including URLs and copyright information (among others). For an integer-typed feature, the entropy for its bitstream is used. For a control-flow-typed feature, the length of its constant sequence is used.

Below we explain the algorithm used for computing the entropy of a bitstream (the specificity weight for an integer-typed feature) and S-IDF for computing frequency weights.

1) Computing the Specificity Weights of Bitstreams as Entrophy: Although many global arrays (e.g., the `S` table of CAST5 cipher) are specific, there are still many others containing little useful information. For example, `[0x0001,0x0010,0x0100,0x1000]`, as a list of flag bits, has the same bitstream as many unrelated data, potentially resulting in some incorrect matches. For this reason, we propose an entropy algorithm for a bitstream in order to compute the specificity weight of an integer-typed feature.

Table III: Examples for computing the entropy of global arrays.

	Array #1	**Array #2**
Array	[0x6a09e667, ...]	[0x0001, ...]
Hex	$(67e6096a...)_{16}$	$(01001000...)_{16}$
Bit	$(01100111111100110...)_2$	$(0000000100000000...)_2$
Fragment(x_i)	0·11·00·111111·00·11·0...	0000000·1·00000000...
Length(l_i)	[1, 2, 2, 6, 2, 2, ...]	[7, 1, ...]
Entropy	$\frac{1}{2^1}+\frac{2}{2^2}+...$	$\frac{7}{2^7}+\frac{1}{2^1}+...$

During the process of feature matching, an integer array will be converted into a bitstream, as illustrated in Table III. Specifically, a bitstream is partitioned into a list of bit fragments, where each fragment represents a maximum sequence of the same bit value (0 or 1). We can then compute the entropy of a bitstream based on the information entropy in information theory [18]. Let x_i be a bit fragment. Its occurrence probability, denoted $p(x_i)$, is the probability of x_i appearing in a bitstream defined as follows:

$$p(x_i) = \frac{1}{2^{l_i}} \tag{1}$$

where l_i is the length of x_i. Then, the entropy $\omega_s(x)$ of a bitstream x can be calculated as follows:

$$\omega_s(x) = -\sum p(x_i)\log_2(p(x_i)) \tag{2}$$

$$= -\sum \frac{1}{2^{l_i}}\log_2(\frac{1}{2^{l_i}}) \tag{3}$$

$$= \sum \frac{l_i}{2^{l_i}} \tag{4}$$

2) S-IDF for Computing Frequency Weights: The occurrence frequency of a feature instance is another factor that determines its contribution in the overall feature matching process. For example, the constant table in the `UnRAR` project exists only in one OSS project while the constant table in the `CAST5` cipher exists in at least 15 OSS projects. If the former table is matched, we can easily determine that the `UnRAR` project has been reused. However, if the latter table is found, it will not be as easy to make a conclusive decision.

Therefore, we make use of an S-IDF algorithm, a TF-IDF-like approach for computing the frequency weight $\omega_f(x)$ of a feature instance x. Here, TF stands for Term Frequency and IDF stands for Inverse Document Frequency. Let $S(x)$ be the appearance frequency of x in a given candidate OSS project:

$$S(x) = \frac{n(x)}{k_n(x)} \tag{5}$$

where $n(x)$ is the number of x appearing in the candidate OSS project and $k_n(x)$ is the total number of feature instances contained in the OSS project. Following [19], we defined:

$$\omega_f(x) = S(x) \times IDF(x) = S(x) \times \log_2 \frac{N}{N(x)} \tag{6}$$

where N is the total number of OSS projects and $N(x)$ is the number of OSS projects containing x.

Algorithm 1: Matching Score Calculation

Input: a binary file b and a set S of candidate OSS projects
Output: a set R of potential reuse relationships

1 **Function** *ReuseDetect*(b, S)
2 $R \leftarrow \varnothing$;
3 **foreach** $s \in S$ **do**
4 **foreach** $f \in F$ **do**
  // match feature instances
5 $F_m \leftarrow \varnothing$;
6 $F_b \leftarrow parseBinFeature(b, f)$;
7 $F_s \leftarrow parseSrcFeature(s, f)$;
8 **foreach** $f_b \times f_s \in F_b \times F_s$ **do**
9 **if** $isMatched(f_b, f_s, f)$ **then**
10 $F_m.add(f_s)$;
  // calculate weights
11 **foreach** $f_s \in F_s$ **do**
12 $W \leftarrow \omega_s(f_s) \cdot \omega_f(f_s)$;
13 $W_s \leftarrow W_s + W$;
14 **if** $f_s \in F_m$ **then**
15 $W_m \leftarrow W_m + W$;
  // calculate matching scores
16 $score_f \leftarrow W_m / W_s$;
17 **if** $score_f \geq threshold_f$ **then**
18 $r \leftarrow tuple(b, s)$;
19 $R.add(r)$;
20 break;
21 return R;
22 **Function** *isMatched*(f_b, f_s, f)
23 **if** $f \in F_{string}$ **then**
24 **if** $f_b = f_s$ **then** return true;
25 **else if** $f \in F_{integer}$ **then**
26 $bits \leftarrow encode(f_s)$;
27 **if** $f_b.find(bits)$ **then** return true;
28 **else** // $f \in F_{control}$
29 **if** f *is switch-case* **then**
30 $f_{temp} \leftarrow addDefault(f_s)$;
31 **if** $f_b = f_{temp}$ **then** return true;
32 **if** f *is if-else* **then**
33 $c \leftarrow getCommonSub(f_b, f_s)$;
34 **if** $len(c) > threshold_{if}$ **then** return true;
35 return false;

D. Computing Matching Scores

A matching score is computed in terms of weighted feature instances matched, and consequently, increases as the number of matched feature instances increases. Therefore, we bound it from above by a threshold, which is determined empirically for each feature. If the matching score of any feature is larger than its threshold, then the target binary is considered to have a reuse relationship with the corresponding source code.

Algorithm 1 shows how to compute a matching score. Given a binary file b and a set S of candidate OSS projects, our approach identifies a set of OSS reuses, R, based on seven kinds of code features, F. For each OSS project $s \in S$ and each feature $f \in F$, we extract a set of feature instances, F_b, from b and a set of feature instances, F_s, from s (lines 6 – 7). For each feature instance from $f_b \in F_b$ and each feature instance from $f_s \in F_s$, if f_b and f_s are matched, we add f_s into the set of matched feature instances, F_m (lines 8 – 10). Meanwhile, we compute the weight W of each f_s by using our importance-weighting algorithm (line 12). Then we calculate two weighted sums, W_m for F_m and W_s for F_s, and subsequently, obtain the matching score $score_f$ for the feature f by dividing W_m with W_s (lines 11 – 16). If $score_f$ is larger than $threshold_f$, the OSS project s is considered to be potentially reused in the binary b and the reuse $r = (b, s)$ is added to R (lines 17 – 20). Finally, we identify the type of reuse in r. by exploiting the code structure of s and removing the pseudo propagated reuses from R, which will be introduced in Section III-E below.

Let us examine the matching function $isMatched$. For a string-typed feature, f_b and f_s are matched only if they are identical. For an integer-typed feature, if f_s can be found in the bitstream of f_b as discussed in Section III-B, then a match is found. For a switch-case feature, after we have introduced a default branch in f_s, which is considered to be equal with any branch (Section III-B), f_s is considered to match with f_b when all their case labels are identical. For an if/else feature, we first obtain the longest ordered common subsequence of f_b and f_s and then check whether both has a common subsequence that is longer than a pre-determined threshold.

E. Identifying Reuse Types

The existence of a reuse relationship for a given feature instance is not always positively correlated with its matching score, as illustrated in Table IV. In these three examples, the simple reuse performs consistently well in terms of code similarities. In contrast, the *partial reuse* represents a true reuse but with a low matching score, while the *pseudo propagated reuse* represents a false reuse but with a high matching score.

Table IV: Partial reuses and pseudo propagated reuses that cannot be determined accurately based on only matching scores

Score	Reuse	Example	Reuse
High	✓	FoxitReader.exe → ZLib	Simple reuse
High	×	libopenjp2-7.dll → LibPNG	Pseudo propagated reuse
Low	✓	FoxitReader.exe → OpenSSL	Partial reuse

1) Identifying Partial Reuses: Partial reuse is common, since a COTS software application often shares only part of an OSS project. In order to reduce the false negative rate caused by low matching scores for some partial reuses, we recognize independent libraries by building a *compilation dependency layered graph* (CDLG), and taking a single library instead of the whole OSS project as a code matching unit.

A CDLG captures the relationships between source files and compiler-generated files. By parsing the default compiler and linker command lines for OSS projects from their auto-build files, i.e., MAKEFILEs, the relationships between source files (with the `.c` and `.cpp` suffixes), object files (with the `.o` suffix), and dynamic libraries (with the `.so` and `.a` suffixes) are established. Then the files related to the same library are grouped together and the files that are not involved during the compilation are eliminated with the help of CDLG.

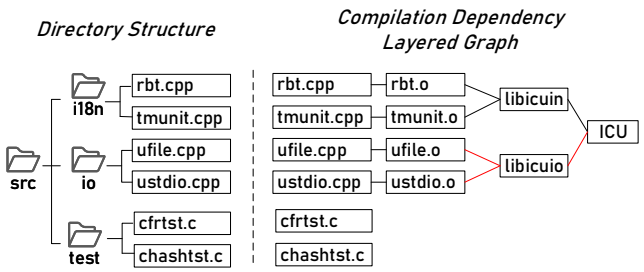

Figure 4: A compilation dependency layered graph (CDLG) for ICU.

Table V: 37.7% of the source files in ICU are separated into six libraries and 62.3% are ignored during the compilation.

Uses	# of Related Source Files	% of All Source Files
Libraries	443	37.7%
libicuin	214	18.2%
libicuuc	186	15.9%
libicutu	24	2.0%
libicuio	12	1.0%
libicutest	6	0.5%
libicudata	1	0.1%
Samples, tests and others	731	62.3%
ALL	1174	100.0%

Take the unicode supporting project ICU as an example, with its CDLG depicted in Figure 4. The source files belonging to different libraries are separated in different groups. In addition, 62.3% of the source files that are not related to any libraries are no longer analyzed, as shown in Table V.

2) Identifying Pseudo Propagated Reuses: These are false reuses that are caused by a propagation of irrelevant reuse relationships among the candidate OSS projects. For example, libopenjp2-7.dll in GIMP reuses OpenJPEG, which includes LibPNG as a third-party library. The matching score between libopenjp2-7.dll and LibPNG is high, but their relationship is a pseudo propagated reuse.

We recognize the inclusive relationships among the candidate OSS projects by comparing the matched feature instances in these candidate OSS projects. Let $F(Y)$ be the set of all feature instances in a library Y and $MF(X,Y)$ be the set of all the matched feature instances between a binary program X and the library Y. Give a pair of libraries Y_1 and Y_2 whose matching scores with X are both larger than their underlying score thresholds (Section III-D). There are two cases.

If $MF(X,Y_1)$ and $MF(X,Y_2)$ are nearly identical, i.e., $MF(X,Y_1) \approx MF(X,Y_2)$, then we have:

$$Size(F(Y_1)) \ll Size(F(Y_2)) \quad (7)$$

$$\Rightarrow Y_2 \text{ reuses } Y_1 \textbf{ and } X \text{ reuses } Y_1 \quad (8)$$

If $MF(X,Y_1) \subset MF(X,Y_2)$, then we have:

$$Size(F(Y_1)) \ll Size(F(Y_2)) \quad (9)$$

$$\Rightarrow Y_2 \text{ reuses } Y_1 \textbf{ and } X \text{ reuses } Y_2. \quad (10)$$

Based on the above two rules, we can recognize pseudo propagated reuses efficiently without having to perform another round analysis for the OSS projects as in OSSPolice [2].

IV. Implementation

We have implemented a file-level matching framework, B2SFinder, for OSS reuse detection in COTS software.

A. Architecture

Figure 5 depicts the architecture of B2SFinder. There are three modules, *Extractor*, *Matcher* and *Detector*. The Extractor extracts the code features from both binary and source code, and then stores them using an effective storage model to improve efficiency (Section IV-C). The compiler and linker command lines are also parsed to enhance the robustness of the extractor (Section IV-B). The Matcher is responsible for code feature matching between binary and source code by applying our matching methods and computing matching scores. The Detector identifies reuses based on the set of matched feature instances and the CDLG of an OSS project, and generates a list of reuse relationships for a target COTS software.

B. Feature Extraction from Large-Scale Source Projects

We extract the code features from source code by developing some static analysis tools on top of LLVM and Clang. However, care must be taken with static analysis. In many OSS projects, header files are stored in some independent directories rather than the directories where the OSS projects reside, causing some header files to be missed. Meanwhile, a particular version of macros used for conditional compilation may depend on the environments and specific compilers used, causing some macros to be missed or used incorrectly.

To avoid these problems, we proceed in three steps. First, we search for files like CMAKELISTS.txt and autogen.sh in the OSS projects to automatically detect the MAKEFILEs used. Second, we parse these MAKEFILEs to obtain the gcc or libtool commands used without actually compiling an OSS project entirely. Finally, we locate the include paths and macros used from the arguments provided to the compiler flags -I and -D. Thus, our static source code analyzer is capable of analyzing the most of the source code in an OS project.

C. Storage Model for Large-Scale Code Features

The time complexity of feature matching is closely related to the storage model used for code features. In a naive implementation, analyzing a single COTS product can take over one day. To shorten this, we make use of two data structures, an inverted index and a Trie (a prefix tree).

Inverted index. In order to speed up string searching, we build an inverted index for string features in Cassandra, a key-value database [20]. As Cassandra relies on a hash tree, the average-case time complexity for retrieving a string has been reduced effectively from $O(M)$ to $O(1)$, where M is the number of string literals in an OSS project.

Trie. We build a Trie for two integer-typed features, global integer arrays and global enum arrays. A Trie is an ordered data structure based on the prefix of a target data. When searching arrays in a binary file, we traverse the Trie and prune its subtrees if the prefix array does not exist.

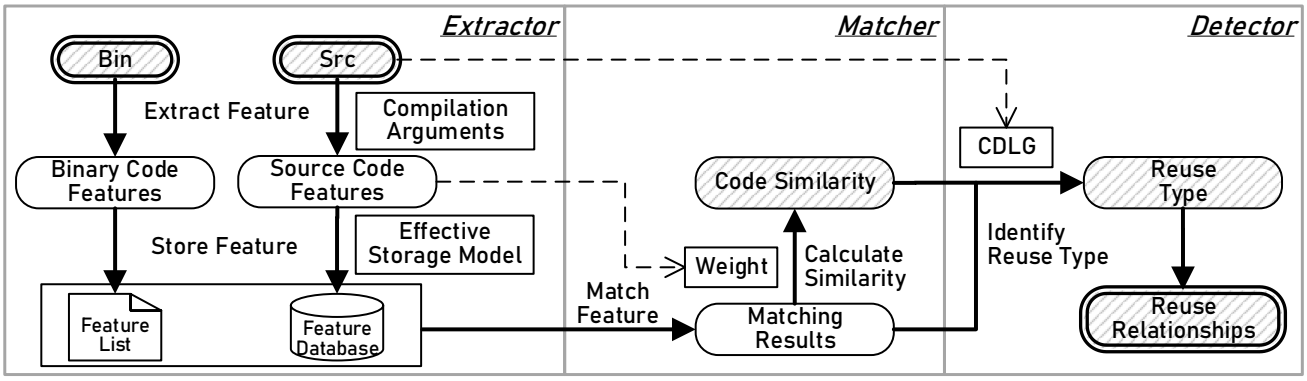

Figure 5: The architecture of B2SFINDER.

V. EVALUATION

We evaluate B2SFINDER in terms of its precision and efficiency. Then we give our findings in detecting OSS reuse vulnerabilities in a large number of real-world COTS software products. We make use of two binary-code datasets and one source-code dataset described below.

- **Dataset 1: Binaries with Known Reuses (*B1*).** This dataset contains 46 official (stripped) binaries compiled from 23 commonly used open-source projects that cover different application areas including video parsing (e.g., `VLC`), PDF rendering (e.g., `SumatraPDF`), and network protocols (e.g., `OpenSSL`).
- **Dataset 2: Real-World COTS Software (*B2*).** This dataset contains 21991 binaries from 1000 COTS software products obtained from a collection of Web sites. We give a hashed list of these products at https://github.com/1dayto0day/B2SFinder/COTS_list.txt.
- **Dataset 3: Public Open-Source Libraries (*S*).** This dataset contains 2189 open-source libraries crawled from Ubuntu Packages, the official package archive of Ubuntu.

A. Precision

To measure precision of B2SFINDER, a benchmark suite consisting of 46 official binaries (dataset *B1*) and 2189 OSS projects (dataset *S*) is used. We have manually labeled a total of 78 real reuses, covering simple reuses, partial reuses and pseudo propagated reuses, in the benchmark suite. Based on the labeled reuses, we compare B2SFINDER with BAT [1], the only binary-to-source matching tool that can handle both PE- and ELF-formatted executables and dynamic libraries. We summarize our experimental results in Table VI.

Since BAT simply ranks all the potential reuses (without concluding which are reuses and which are not), we introduce three criteria for BAT: (1) the first one as a reuse (top 1), (2) the first five containing a reuse (top 5), and (3) the first x that allows BAT to achieve the highest precision (highest P).

B2SFINDER has successfully detected 69 out of the 78 reuses with only 5 false positives, obtaining a precision of 93.2% and a recall of 88.5%, which significantly outperforms BAT in all the settings. BAT reaches the highest accuracy (F1) for top-1, but detects only 32 reuses, while B2SFINDER finds 69 reuses in top-1. Even in the case of top-5, BAT still fails to detect 31 out of 78 reuses (39.7%). In contrast, B2SFINDER can detect 26 of these 31 reuses (83.4%) missed by BAT.

Table VI: Comparing B2SFINDER with BAT [1] with different criteria.

	Reuse	TP[1]	FP[1]	FN[1]	P[2]	R[2]	F1
B2SFinder	78	**69**	**5**	**9**	**93.2%**	**88.5%**	**0.908**
BAT (Top 1)		32	14	46	69.6%	41.0%	**0.516**
BAT (Top 5)		**47**	181	**31**	20.6%	**60.2%**	0.307
BAT (Highest P)[3]		17	**2**	61	**89.5%**	21.8%	0.351

[1] TP: True Positive. FP: False Positive. FN: False Negative.
[2] P: Precision. R: Recall.
[3] Highest Pricesion: MP(Matching Percentage)>0.2, P(Possibility)>0.6.

OSSPolice [2] is another existing binary-to-source matching tool targeting libraries, by considering only string literals and exported function names as code features. To compare with it, we have configured B2SFINDER to use only these two kinds of code features. As a result, B2SFINDER fails to find 17 (24.6% of 69 reuses), demonstrating the necessity for also considering the other five code features in this research work.

Unlike BAT, OSSPolice analyzes the relationships between a given binary and a candidate OSS project independently, making it difficult to handle partial reuses and pseudo propagated reuses precisely. By relying on matching scores only, these complex reuses may lead to a lot of false negatives and false positives. By exploiting the code structures of OSS projects, B2SFINDER has substantially improved precision, by recognizing 13 partial reuses and 13 pseudo propagated reuses correctly. As a result, the false negatives have been reduced from 22 to 9 (with a reduction of 59.1%) and the false positives have been lowered from 18 to 5 (with a reduction of 72.2%).

B. Efficiency

We have applied B2SFINDER to analyze 21991 binaries from 1000 real-world COTS software products (dataset *B2*) and 2189 public open-source libraries (dataset *S*) on four virtual machines in an OpenStack cloud. Each virtual machine is equipped with two shared cores of Intel Xeon E5-2603 V4, 4GB memory and 128GB disks. On average, B2SFINDER

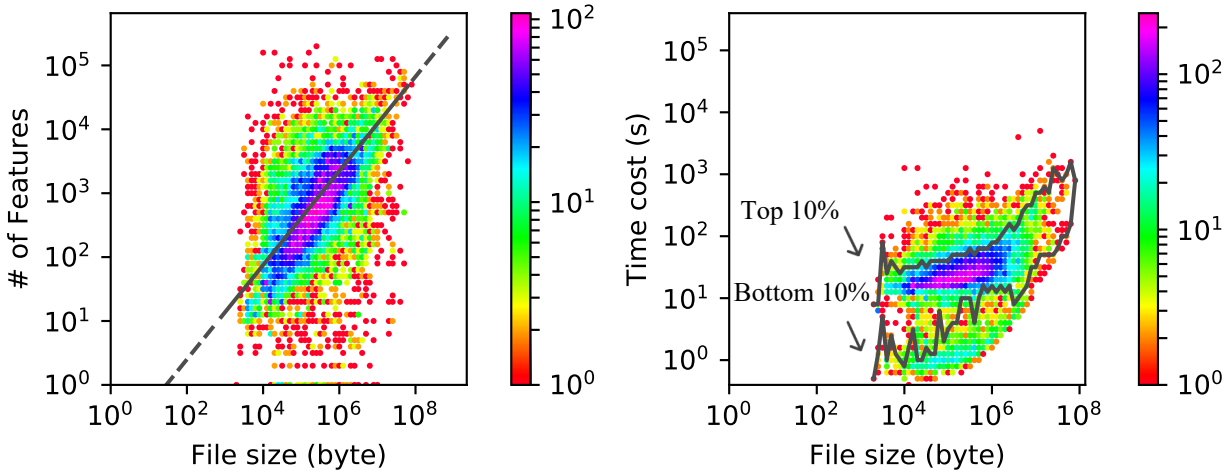

Figure 6: The correlation between the number of feature instances and the size of it containing binary file.

takes 53.85 seconds to analyze a binary file in the presence of 2189 open-source libraries. As shown in Figure 6, the number of feature instances correlates roughly linearly with the size of its containing binary file. It takes less than 100 seconds to analyze 91.4% of all the binary files, which is expected to be fast enough for an offline deployment.

To improve the efficiency of B2SFinder in detecting OSS reuses in large-scale OSS projects, we have employed two optimized data structures, inverted index and Trie. Compared with a naive matching algorithm in searching in 2189 libraries, our implementation reduces (1) the time spent on matching string-types features for a COTS software product from 20.61 hours to 6.63 minutes on average, with a speedup of 186.5x from the inverted index used, and (2) the time spent on matching integer-typed features for a COTS software product from 10.3 hours to 12.0 minutes on average, with a speedup of 51.5x from the Trie used.

C. Large-Scale Analysis

We now describe our findings in detecting OSS reuse vulnerabilities in large-scale real-world COTS software products.

1) OSS Reuses: During the large-scale analysis as discussed in Section V-B, we have found 10208 pairs of reuse relationships between the 21991 binaries from dataset *B2* and the 2189 OSS projects from dataset *S*. As shown in Table VII, 19.2% of the binaries, contained in 63.4% of all the COTS software products, are found to have reused at least one open-source library. This shows that OSS reuses are ubiquitous in COTS software. Interestingly, the reuse relationships in audio and video parsing software products are relatively more complex. Of the 21 COTS software products that reuse more than 30 different OSS projects, 12 out of them are audio or video parsing software products. In addition, 47.1% of the binaries with some reuses contain at least two reuses, indicating that multiple reuses coexist in a large number of binaries.

For the 2189 OSS projects studied, 4.6% of them have been reused more than 10 times. However, how frequent these OSS projects are reused varies from project to project. As shown in Figure 7, the top 10 frequently reused libraries are

Table VII: The number of OSS reuses in product- and file-level.

Level	All	NR[1]	CR[1]≥1	CR[1]≥5	CR[1]≥15	CR[1]≥30
Product	1000	366	634	331	27	21
File	21991	18427	3564	402	89	0

[1] NR: No Reuse. CR: Count of Reuses.

`ZLib`, `LibJPEG-turbo`, `LibPNG`, `OpenSSL`, `SQLite`, `FreeType`, `LibTIFF`, `Libsndfile`, `UnRAR` and `Expat`. These 10 libraries contain at least five previously disclosed vulnerabilities. In particular, `OpenSSL` contains up to 194 CVEs reported earlier. On average, these 10 libraries have been found to contain 54.7 vulnerabilities.

2) Potential OSS Reuse Vulnerabilities: New vulnerabilities are continuously discovered in frequently reused OSS projects. By detecting OSS reuses from a specific OSS library, we can discover potentially vulnerable COTS software products with OSS reuse vulnerabilities. To demonstrate this, we have selected three popular OSS projects, `ZLib` (a compression library), `Pango` (a font parsing library) and `Xerces-C` (an XML parsing library), against 9558 COTS software products continually collected for two months as a case study.

In total, we have found 1059 reuses of `ZLib`, 151 reuses of `Pango` and 53 reuses of `Xerces-C`, all of which are versioned. The version distributions are shown in Figure 8, in which each color refers to a unique version. To our surprise, 100% of reuses for `Xerces-C`, 98% of reuses for both `ZLib` and `Pango` include the source codes released at least 5 years ago (earlier than 2014). Almost all codes with those versions have unfixed vulnerabilities and are potentially vulnerable.

VI. Related Work

The previous work that is the most related to this work falls into two broad areas, code clone detection and reuse detection.

A. Code Clone Detection

Code clone detection aims at evaluating the similarity of two software programs and finding code clones based on a similarity score. Different techniques are applied in analyzing binary and source code due to their differences.

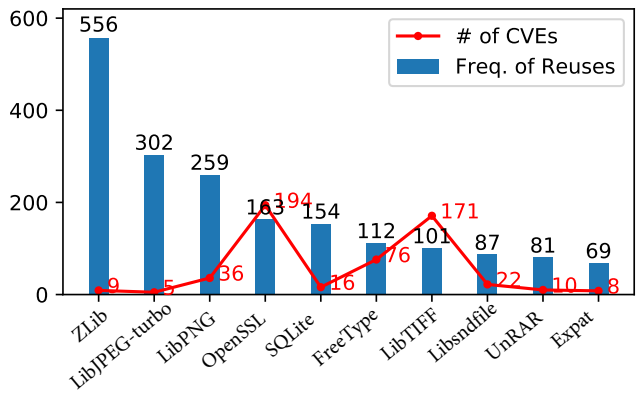

Figure 7: The top 10 frequently reused OSS projects.

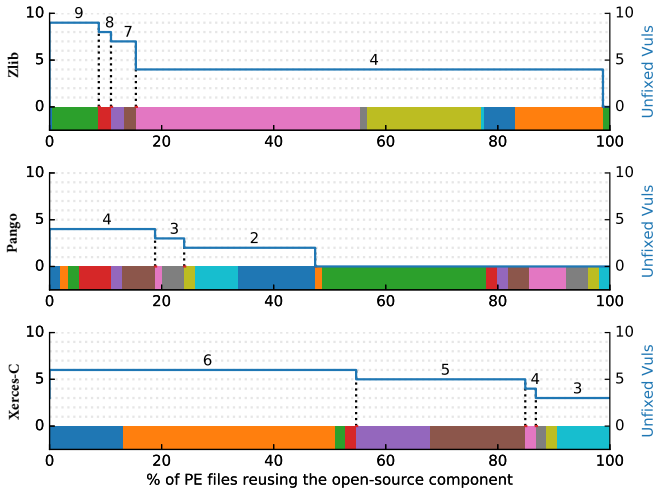

Figure 8: Version distributions of ZLib, Pango and Xerces-C

Source-to-Source Matching. Some techniques compare source code as plain text directly [21–23]. They can be efficient but not precise enough as the semantic information is completely ignored. More techniques perform syntactic analysis on source code and then carry out a similarity analysis in terms of tokens [24–28], abstract syntax trees (ASTs) [29], and internal code dependencies [30]. Some others combine machine learning and program analysis to learn the most important code features related to bugs and vulnerabilities [3, 26].

Binary-to-Binary Matching. Binary code analysis is harder than source code analysis. Disassembly is the first step for any binary analysis. Afterwards, some techniques use directly the textual features in the assembly code [17, 31] to measure code similarity, but most techniques opt to perform some semantic analysis in the assembly code. For example, control flow graphs [9, 10, 15, 17, 32–35] and call graphs [32, 36, 37] are commonly selected as code features, and are often manipulated by graph-based algorithms. In addition, binary raw data [11], the I/O behavior of basic blocks [38, 39] and execution traces [40–42] are also good choices. Constants in code are also a good supplement [17]. In order to facilitate code searching, self-defined semantic hashing is commonly used in code indexing [15, 17, 38]. Symbolic execution and theorem proving are being considered when trading efficiency for precision [32, 43].

Binary-to-Source Matching: There are relatively few efforts on binary-to-source matching, including BAT [1], OSSPolice [2] and FIBER [13]. BAT [1] takes string literals as the only code feature and produces a ranked list of potential OSS reuses. OSSPolice [2] considers not only string literals but also exported function names as code features, and introduces a hierarchical indexing scheme to index features. FIBER [13] generates semantic signatures on control flow for functions to identify the syntax and semantic changes introduced by a patch under the assumption that the targeted functions are known.

B. Reuse Detection

The objective here is to detect reuse relationships in software precisely. Although there has been a lot of research on detecting code reuses, little has been done on detecting different types of reuse. Code matching scores can not always accurately reflect the complex reuse relationships, including partial reuses and pseudo propagated reuses.

Some earlier attempts [2, 36] are concerned with partial reuses. OSSPolice [2] does not handle partial reuse and discuss the false negatives thus caused. In [36], a claim is made on detecting code theft where only partial code is reused but without providing any experimental evaluation. In the case of pseudo propagated reuses, OSSPolice [2] builds a hierarchical index to record the inclusive relationships among the OSS projects in order to eliminate pseudo propagated reuses.

VII. Conclusion

In this paper, we propose a novel binary-to-source matching approach, B2SFinder, for detecting OSS reuses in COTS software. B2SFinder proceeds by first computing matching scores and detecting different types of reuses. To generate a precise matching score, seven kinds of code features are selected and three matching methods are introduced for handling three different types of features. Then we assign a specificity weight and a frequency weight to each feature instance based on our importance-weighting methods. This enables us to compute a weighted sum of matched feature instances as the overall matching score. In addition, we present a precise reuse type identification method by exploiting the code structures of OSS projects. With B2SFinder, we can discover OSS reuses and potential OSS reuse vulnerabilities in COTS software.

Acknowledgement

We would like to thank all the anonymous reviewers for their comments on an earlier version of this paper and our shepherd, Dr Junaid Haroon Siddiqui, for helping us improve the presentation of the final version. This work is supported in part by Chinese National Natural Science Foundation (61602470, 61802394, U1836209), National Key Research and Development Program of China (2016QY071405), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200, XDC02010000), and Australian Research Council Grants (DP170103956 and DP180104069).

REFERENCES

[1] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 63–72.

[2] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "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, 2017, pp. 2169–2185.

[3] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 201–213.

[4] (2018) Owasp top 10 application security risks. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[5] (2019) Cwe/sans top 25 most dangerous software errors. [Online]. Available: https://www.sans.org/top25-software-errors

[6] *Transforming Open Source to Open Access in Closed Applications: Finding Vulnerabilities in Adobe Reader's XSLT Engine*, Zero Day Initiative (ZDI) Std., May 2017. [Online]. Available: https://static1.squarespace.com/static/5894c269e4fcb5e65a1ed623/t/592493f140261d8c41ae30c1/1495569406556/ZDI-Adobe_XSLT_Report.pdf

[7] (2018, Mar.) mpengine contains unrar code forked from unrar prior to 5.0, introduces new bug while fixing others. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1543&desc=2#maincol

[8] (2019) Synopsys 2018 open source security and risk analysis report. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2018-ossra.pdf

[9] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[10] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.

[11] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 667–678.

[12] (2019) Ubuntu packages. [Online]. Available: https://packages.ubuntu.com/

[13] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 887–902.

[14] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE Transactions on Software Engineering*, 2018.

[15] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 480–491.

[16] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 678–689.

[17] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 329–338.

[18] E. T. Jaynes, "Information theory and statistical mechanics," *Physical review*, vol. 106, no. 4, p. 620, 1957.

[19] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242. Piscataway, NJ, 2003, pp. 133–142.

[20] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[21] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[22] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 37–58, 2006.

[23] B. S. Baker, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1343–1362, 1997.

[24] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[25] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 106–115.

[26] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[27] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 48–62.

[28] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: a scalable approach for vulnerable code clone discovery," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 595–614.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[30] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "Libd: scalable and precise third-party library detection in android markets," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 335–346.

[31] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*. ACM, 2005, pp. 314–318.

[32] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[33] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.

[34] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.

[35] Bindiff. [Online]. Available: https://www.zynamics.com/bindiff.html

[36] X. Wang, Y.-C. Jhi, and S. Zhu etc., "Detecting software theft via system call based birthmarks," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 149–158.

[37] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 280–290.

[38] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.

[39] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 2017, pp. 88–98.

[40] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 678–689.

[41] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-os binary code search with emulation," *IEEE Transactions on Software Engineering*, 2018.

[42] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *International Conference on Information Security*. Springer, 2004, pp. 404–415.

[43] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 389–400.