2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Software Ingredients: Detection of Third-party Component Reuse in Java Software Release

Takashi Ishio
Osaka University
Osaka, Japan
ishio@ist.osaka-u.ac.jp

Raula Gaikovina Kula
Osaka University
Osaka, Japan
raula-k@ist.osaka-u.ac.jp

Tetsuya Kanda
Osaka University
Osaka, Japan
t-kanda@ist.osaka-u.ac.jp

Daniel M. German
University of Victoria
Victoria, BC, Canada
dmg@uvic.ca

Katsuro Inoue
Osaka University
Osaka, Japan
inoue@ist.osaka-u.ac.jp

ABSTRACT

A software product is often dependent on a large number of third-party components. To assess potential risks, such as security vulnerabilities and license violations, a list of components and their versions in a product is important for release engineers and security analysts. Since such a list is not always available, a code comparison technique named Software Bertillonage has been proposed to test whether a product likely includes a copy of a particular component or not. Although the technique can extract candidates of reused components, a user still has to manually identify the original components among the candidates. In this paper, we propose a method to automatically select the most likely origin of components reused in a product, based on an assumption that a product tends to include an entire copy of a component rather than a partial copy. More concretely, given a Java product and a repository of jar files of existing components, our method selects jar files that can provide Java classes to the product in a greedy manner. To compare the method with the existing technique, we have conducted an evaluation using randomly created jar files including up to 1,000 components. The Software Bertillonage technique reports many candidates; the precision and recall are 0.357 and 0.993, respectively. Our method reports a list of original components whose precision and recall are 0.998 and 0.997.

Keywords

Software reuse, reverse engineering, origin analysis

1. INTRODUCTION

Software reuse is crucial for efficient software development. A software project uses components developed outside of the project, subsequently produces software that is a potentially reusable component in other projects. Software reuse activity is now commonplace in both open source software community and industry. Heinemann et al. [19] reported that black-box reuse is common among open source Java projects. Bavota et al. [5] reported that although the trend of growing projects in the Apache ecosystem is linear, the growth of component dependencies are exponential. Rubin et al. [37] reported that industrial developers extract reusable components from existing software products to develop core assets for their new products. Mohagheghi et al. [29] reported that reused components are more reliable than non-reused code.

Reusing a component may increase a potential risk of vulnerabilities and license violations caused by the component and its transitively dependent components. For example, the Google Web Toolkit (GWT) project [18] includes various components in its release so that it can be used without installation of other components. A single jar file `gwt-dev.jar` in GWT 2.7.0 includes class files reused from three vulnerable components: Apache Xalan 2.7.1, HttpClient 4.3.1, and Commons-Collections 3.2.1. The vulnerabilities have a risk to allow remote attackers to access external resources [32], perform man-in-the-middle attack [33], and execute arbitrary code [2]. It is hard for users to identify such a potential risk, because internal components are not directly visible to them. Sonatype reported that many applications include severe or critical flaws inherited from their components [7].

In order to assess a potential risk of components, release engineers and user-side security analysts need a complete list of components that are actually included in a product release. However, such a list is not always available to them outside of a development team. In the case of `gwt-dev.jar`, its build file (`build.xml`) must be analyzed to identify its internal components. The dependencies are unavailable on the Maven Central Repository, because the jar file can be reused without the dependent components.

To address the problem, an existing technique named Software Bertillonage [8, 9] has been proposed to analyze a jar file. A simple MD5 file hash comparison is unable to identify a reused class file, because it cannot compare class files generated by different compilers. Davies et al. [9] reported that 48% of jar files in Debian GNU/Linux packages have no class files that were identical to any class files in the Maven Central Repository. Instead of a file hash, Software Bertillonage

MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901773

introduced a class signature to compare two classes ignoring their details. The approach enables to identify common classes between two jar files. By comparing a product jar file with each of existing components, the technique can identify components whose classes are entirely/partially included in a product. However, common classes between jar files does not imply code reuse between the files. For example, if a product includes an entire copy of Apache Commons components, then the Software Bertillonage tool reports that the product includes the components and also includes a partial copy of GWT, because GWT includes the same classes. A user still has to manually identify the original components among the reported candidates.

In this paper, we propose a method to automatically select a set of jar files that is the most likely origin of components reused in a product. We identify existing components that cover the contents of a target jar file as much as possible in a greedy manner, assuming a product tends to include an entire copy of a component rather than a partial copy. The assumption is based on an observation that most of developers copy a jar file as a black box [16, 19]. All the classes in the components can be automatically repackaged into a single runnable jar file using existing tools such as Eclipse Runnable Jar File Exporter [13] and Apache Maven Assembly Plugin [3].

We conducted an experiment to evaluate our approach against Software Bertillionage, evaluating the ability to accurately identify original components reused in a target jar file. The experiment used artificially created jar files using a dataset named `sourcerer-maven-aug12`, which is a snapshot of the Maven Central Repository. We randomly selected up to 1,000 jar files in the dataset, put the contents into a single jar file, and then detected the original jar files. The precision and recall of Software Bertillonage are 0.357 and 0.993, respectively. Our method achieved much better results, with an improved precision and recall of 0.998 and 0.997 for the dataset. The result shows that our method successfully identifies the original components included in a product.

We use two cases to highlight practical usefulness and implications. We applied our method to actual jar files of Facebook LinkBench and muCommander 0.9.0. The first case represents a release engineering situation that compares the dependencies managed by developers with components actually included in a product. The second case assesses a risk of a product whose components are unrecorded by developers.

The contributions of the paper are summarized as follows.

- We define a method to detect multiple third-party components in a jar file. Our method is efficient and detects many components.

- Our experiment shows the accuracy of our approach compared with Software Bertillonage. The accuracy of the previous work has not been evaluated in [8, 9]. The evaluation is conducted using actual components available in a public dataset.

- We reported the result of analysis on actual software binaries. It shows the usefulness of our method.

Section 2 shows related work of our approach. The approach itself is detailed in Section 3. Section 4 presents the evaluation of our approach using a dataset constructed from the Maven repository. Section 5 presents the analysis of actual binaries. Section 6 describes the threats to validity. Section 7 describes the conclusion and future work.

2. RELATED WORK

We have categorized related work into three groups: Analysis of reuse activity, detection of software reuse, and support of software reuse. Our research is included in detection of software reuse.

2.1 Analysis of Reuse Activity

Heinemann et al. [19] analyzed code reuse in open source Java projects. They identified black-box reuse using a corpus of 20 popular libraries; they extracted references to library classes in a target program as an instance of reuse. Our method also uses a corpus of existing components, but our method identifies classes copied to a target program.

Bavota et al. [5] reported that the number of inter-project dependencies grows exponentially. The dataset is extracted from configuration files for build tools such as Maven. While the approach is likely precise in most cases, dependencies declared in a configuration file may be different from actual dependencies, as developers can keep a copy of source and binary files in their project.

Teyton et al. [41] analyzed library migrations using a list of packages for each library. The list associates a package name to a library name. Although the list-based detection is reasonable, it is not applicable to identify a version number.

Code clone detection has been used to analyze source code reuse between projects. Kamiya et al. [23] proposed CCFinder to detect similar code fragments between files. German et al. [17] used CCFinder to detect code siblings reused across projects. They identify the original project of a code sibling by investigating the source code repositories of the projects. Hemel et al. [21] analyzed vendor-specific versions of Linux kernel using their own clone detection tool. The analysis shows that each vendor created a variant of Linux kernel and customized many files in the variant.

2.2 Detection of Software Reuse

Davies et al. [8, 9] proposed Software Bertillonage to identify the origin of a jar file using classes and their methods in the file. The method compares the contents of a jar file with each of jar files in a repository, and then reports the most similar one as the original. It enables to recover a full name of a jar file (e.g. `junit-3.8.1.jar`) from a shortened name (e.g. `junit.jar`). The method also defines metrics representing the degree of inclusion relationship between two jar files so that a user can test whether a jar file includes a part of another jar file or not. German et al. [16] demonstrated the approach can detect OSS jar files included in proprietary applications. Mojica et al. [30] used the same approach to analyze code reuse among Android applications.

Di Penta et al. [10] used class names to identify software license for a jar file, instead of a component name. Since a jar file may be distributed without documents of software license, the method extracts class names as keywords for search engines. The method can recover software license if the classes are included in a well-known library available on the Internet.

Sæbjørnsen et al. [38] proposed a clone detection for binary code. Qiu et al. [35] proposed a code comparison method for a binary form to identify library functions included in an executable file. These techniques are applicable to detect functions copied from library code in a target program, by comparing all the versions of components with the target program. Our method detects reuse in a component-level rather than a function level.

Hemel et al. [20] proposed a binary code clone detection to identify code reuse violating software license of a component. The method compares the contents of binary files between a target program and each of existing components. The framework is the same as Software Bertillonage; it enables a user to test whether a component is likely included in a target program or not. While reused code fragments are detected as code clones, code clones are not always reuse. For example, if a target program A uses a library B that is also used in another library C, the method reports two code clones between A-B and between A-C. A user has to manually investigate clones to determine whether they are actual reuse or not.

Kawamitsu et al. [25] proposed a technique to identify an original version of source code in a library's source code repository. To analyze a program, a user of the technique must know what library is included in the program. Our technique automatically extracts the information from a target program.

Steidl et al. [40] proposed to detect source code move, copy, and merge in a source code repository. The approach compares the full contents of files, while our approach intentionally ignore details of binary files.

Kanda et al. [24] proposed a method to recover an evolution history of a product and its variants from their source code archives without a version control. The approach also compares the full contents of source files, using a heuristic that developers tend to enhance a derived version and do not often remove code from the derived version.

Chen et al. [6] proposed a technique to detect clones of Android applications. The analysis computes similarity between control-flow graphs of methods, and recognize an Android application as a clone of another application if the two applications have a large number of similar methods. To compare application-specific code separately from existing libraries, the technique uses a white-list including package names of popular libraries.

Luo et al. [28] proposed a code plagiarism detection applicable to obfuscated code. The detection method identifies semantically equivalent basic blocks in two functions. Our approach assumes that developers own their code and the code of a target program is not obfuscated.

Sojer et al. [39] pointed out that ad-hoc reuse from the Internet has a risk of a license violation. Inoue et al. [22] proposed a tool named Ichi-tracker to identify the origin of ad-hoc reuse. It searches clones of a source file across various repositories on the Internet and visualizes the similarities. Our method focuses on clone-and-own reuse of components rather than ad-hoc reuse.

2.3 Support of Software Reuse

Detected instances of source code reuse can be seen as clues to extract the common functionalities in software products. Bauer et al. [4] proposed to extract code clones across products as a candidate of a new library. Duszynski [12] proposed a code comparison tool to analyze source code commonalities from multiple similar product variants. Fischer et al. [15] proposed to extract common components from existing product variants and compose a new product.

Thung et al. [42] proposed a method to recommend libraries that are likely useful to developers, using item-set mining for libraries. The approach also uses the Maven repository as a source of libraries. Components detected by our method can be used as input for the recommendation system.

Dietrich et al. [11] analyzed binary compatibility between versions of a library. As similar to [16], the analysis used a binary-level content comparison to identify the origin of a library jar file. The analysis focused on libraries used by a program via method calls, rather than libraries whose copies are included in the program.

Raemaeker et al. [36] analyzed the relationship between version numbers and binary compatibility of a library. The analysis compares identifiers between versions and shows that identifiers in an old version are often unavailable in a new version. The observation implies that comparison of classes and methods is effective to identify a version of a component.

Kula et al. [26] proposed a visualization to investigate a history of component update in a project. Developers can identify outdated components in their project, using the history and statistics of dependencies extracted from the Maven repository. While the visualization extracts dependencies of a target project from a configuration file for Maven (`pom.xml`), our method can provide supplementary dependencies unrecorded in the file.

Yano et al. [43] proposed a visualization to investigate a popular combination of library versions. For example, given a pair Apache Commons HttpClient and Collections, the visualization shows the most popular combination is HttpClient 3.1 and Collections 3.2.1. Since such a popular combination works properly, developers can reduce a risk of version incompatibility. This visualization is also dependent on dependencies recorded in `pom.xml` files. Our method can provide dependencies in actual software releases for the visualization.

3. COMPONENT DETECTION

Our method detects components that are the most likely reused in a target product jar file. The method assumes all existing versions of components are accessible from a hosting repository. The hosting repository is a collection of jar files. A jar file should at least contain one class to be considered a component. The Maven Central Repository is a well-known component hosting repository; it has been used in [8, 9]. Similarily, we employ a snapshot of the Maven Central Repository named `sourcerer-maven-aug12`, that is a part of UCI Source Code Data Sets [27].

Our method takes as input a target jar file t and a set of jar files R in the repository. Our method selects jar files from the repository such that each jar file provides class files to t. Since a simple list of jar file names is not informative for users to analyze actual reuse, we represent reused jar files in t as a list of pairs (J_i, C_i); J_i is a set of jar files including a set of classes C_i in the target t. Multiple jar files are detected as the origin of reused code if they have the same classes C_i.

Our method comprises two steps: Signature extraction and signature-based comparison. To identify class files common to t and jar files in R, we use a class signature representing the contents of a single class. Hence, the first step constructs a database of signatures for each jar file in the

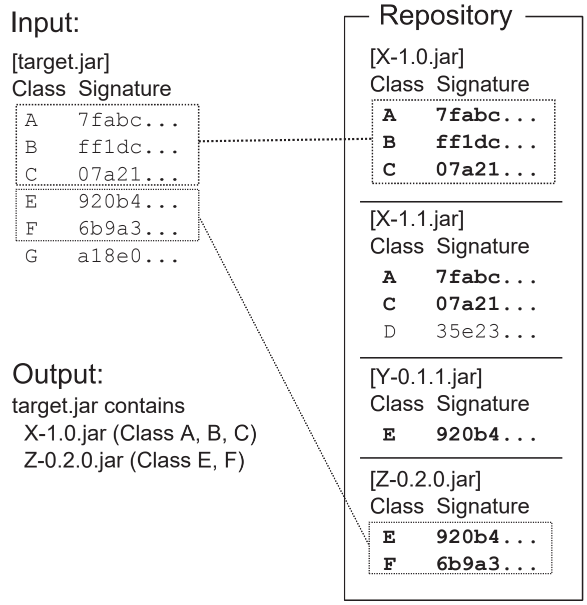

Figure 1: An example input and output of our method

repository R. The second step is a greedy algorithm that repeatedly identifies the largest jar file that covers the contents of t as much as possible using the database. Compared with Software Bertillonage, our method newly introduces an extension of class signature and the greedy algorithm.

Figure 1 illustrates an example input and output of our method. Given the `target.jar` file, our method compares the contents with jar files in the repository. Our method reports that `X-1.0.jar` is the origin of three classes A, B, and C, and `Z-0.2.0.jar` is the origin of two classes E and F, because the target jar includes all the classes in them. Although the class `E` can be seen as a copy from `Y-0.1.jar`, our method selects only `Z-0.2.0.jar` because it covers more classes.

Software Bertillonage does not provide such an automatic analysis. That method defines a similarity metric between jar files t and r, using a class signature to compare classes, as follows.

$$sim(t, r) = \frac{|Classes(t) \cap Classes(r)|}{|Classes(t) \cup Classes(r)|}$$

In the case of Figure 1, we can get the similarity metric values as follows: $sim(t, \texttt{X-1.0}) = 0.500$, $sim(t, \texttt{X-1.1}) = 0.429$, $sim(t, \texttt{Y-0.1.1}) = 0.167$, and $sim(t, \texttt{Z-1.0}) = 0.333$. Since `X-1.0` is more similar to t than `X-1.1`, `X-1.0` is more likely a code origin of t than `X-1.1`. While `X-1.1` can be removed from the candidates, the similarity metric does not show the relationship among the remaining candidates. Hence, a user has to manually analyze the contents of three jar files to identify the origin of classes.

3.1 Signature Extraction

The first step of our method translates a class file into a signature for comparison. If a signature of a class c_1 is the same as one of another class c_2, we regard the classes as the same element when comparing a set of classes between jar files.

A simple file hash such as MD5 and SHA-1 is inapplicable, because generated class content can vary and is dependent on compiler and configuration settings such as JDK version number and inclusion of debug information.

A signature of a class c is a set of the following attributes.

- Class name of c. It is an empty string if c is an anonymous class.
- Name of the parent class of c.
- Name of the outer (enclosing) class of c if c is an inner class.
- Interfaces implemented by c.
- Fields defined in c. Each field is represented by its name and type. A declaration order is ignored.
- Methods defined in c. Each method is represented by the following attributes. A declaration order is ignored.
  - Method name. Constructors and static initializers are represented by their internal names `<init>` and `<clinit>`, respectively.
  - Modifiers, e.g. `public`, `private`, `protected`, and `synchronized`.
  - The types of arguments and return value.
  - A set of method call instructions in the method. We use only method names, receiver types, and argument types, ignoring their locations in the method.
  - A set of field access instructions in the method. We use only field names, field types, and owner classes of the fields, ignoring their locations in the method.

A signature equals to another signature if and only if all their attributes are the same. The definition of a class signature is an extended version of [8, 9]; method calls and field access in methods are added. We included the additional attributes so that we can distinguish different versions of a component as much as possible.

While a class signature is defined as a set of attributes, our implementation extracts all the above attributes from a class and concatenates them into a single string, and then translates it into a single SHA-1 hash value. Hence, comparison of two classes is implemented by comparison of two hash values.

It should be noted that class names in a class signature are fully qualified names including their package names. Although developers may clone source files from a library and modify the package names, both our method and Software Bertillonage do not analyze such a case.

A signature excludes subclasses and synthetic methods and fields generated by compilers that are not included in source code. They are recognized by the `synthetic` access modifier in bytecode and a character "`$`" in their names.

All the jar files in a repository R are translated into a signature database. A signature database is represented by a simple hierarchical structure. The database contains a list of jar file names. Each jar file name is associated with a list

Algorithm 1 Component Detection

```
1: Initialize $A = Classes(t)$
2: Classify jar files in $R$ to groups and sort them by a descending order of $overlap_N$: $R_s = \langle R_1, R_2, \cdots \rangle$.
3: for $i = 1$ to $|R_s|$ do
4:     while $R_i \neq \phi$ do
5:         Select $(J, C)$ such that
               $J \subseteq R_i \ \wedge \ \forall r \in J.\ A \cap Classes(r) = C$
               $\wedge \forall r' \in R_i \setminus J.\ \ A \cap Classes(r') \neq C$
               $\wedge \ \max_{r \in R_i} |A \cap Classes(r)| = |C|$
6:         print $(J, C)$ as a result if $C \neq \phi$.
7:         $R_i \leftarrow R_i \setminus J, A \leftarrow A \setminus C$
8:     end while
9: end for
```

of classes. Each class is represented by a pair of its name and signature hash value.

A signature extraction process can analyze jar files and classes in parallel, since jar files are independent of one another and a class signature is computed analyzing only one class file at a time. We can incrementally analyze new jar files in a component repository that grows day by day.

3.2 Signature-based Comparison

The second step of our method uses a database extracted from a repository R to identify a subset of jar files in R that cover classes in t. Algorithm 1 represents the entire process of this step. In the algorithm, $Classes(x)$ denotes a set of classes in a jar file x.

The algorithm firstly extracts A, which is a set of classes to be analyzed, from t. The process is the same as the signature extraction step.

The line 2 in the algorithm determines an order of comparison of jar files in R. We use the overlap coefficient between class names in t and each r in R, defined as follows.

$$overlap_N(t, r) \quad = \quad \frac{|Names(t) \cap Names(r)|}{|Names(r)|}$$

where $Names(x)$ is a set of class names in a jar file x. We classify jar files in the repository into groups $R_s = \langle R_1, R_2, \cdots \rangle$, so that the groups satisfy a descending order of $overlap_N$ values as follows.

$$\begin{array}{ll} \forall r, s \in R_i. & overlap_N(t, r) = overlap_N(t, s) \\ \forall r \in R_i, s \in R_j. \quad i < j & \Longleftrightarrow \ overlap_N(t, r) > overlap_N(t, s) \end{array}$$

Each R_i is a subset of jar files in R. We exclude jar files whose $overlap_N$ value is zero from the analysis. On the other hand, we use no threshold for filtering. We analyze all the jar files that share at least one class name with t.

The descending order of the $overlap_N$ metric enables us to check an entirely copied jar file earlier than a partially copied jar file. If all the classes in r are copied to t, t must include the class names (i.e. $overlap_N(t, r) = 1$). We did not use the Jaccard index $\frac{|Names(t) \cap Names(r)|}{|Names(t) \cup Names(r)|}$, because a partially copied jar file may have a higher value.

The $overlap_N$ uses class names instead of class signatures. When two jar files r_1 and r_2 are copied to t, the jar files have $overlap_N(t, r_1) = overlap_N(t, r_2) = 1$ even if they include different versions of the same classes.

The while loop in the lines 4 through 8 is the main part of the algorithm. The line 5 compares a set of classes A with jar files in R_i using class signature, and then select jar files J providing the largest intersection C. The selected pair (J, C) is reported to a user at the line 6. The line 7 removes the selected classes C from A to ensure that the classes have the only one code origin. The loop continues until all the jar files in R_i are analyzed. If no jar files in R_i have a non-empty intersection with A, a pair $(J = R_i, C = \phi)$ is extracted. The pair makes R_i empty and terminates the while loop. And then, the analysis moves to the next jar file set R_{i+1}. The algorithm terminates after all the jar file sets have been analyzed.

Let us illustrate how the algorithm works on the example repository shown in Figure 1. The repository includes four jar files as follows.

$$\begin{array}{rcl} Classes(\texttt{X-1.0}) & = & \{\texttt{A}, \texttt{B}, \texttt{C}\} \\ Classes(\texttt{X-1.1}) & = & \{\texttt{A}, \texttt{C}, \texttt{D}\} \\ Classes(\texttt{Y-0.1.1}) & = & \{\texttt{E}\} \\ Classes(\texttt{Z-0.2.0}) & = & \{\texttt{E}, \texttt{F}\} \end{array}$$

Our algorithm firstly computes $overlap_N$ values. Since the target jar file includes three classes `A`, `B`, and `C`, the overlap value $overlap_N(t, \texttt{X-1.0}) = 1$. The target jar file also includes all the class names in `Y-0.1.1` and `Z-0.2.0`. Hence, the repository is split into two sets: $R_1 = \{\texttt{X-1.0}, \texttt{Y-0.1.1}, \texttt{Z-0.2.0}\}$ and $R_2 = \{\texttt{X-1.1}\}$. If `Z-0.2.0` included another version of class `C`, R_1 still includes the jar file because the $overlap_N$ value does not change. The $overlap_N$ value represents a situation that the jar file could be an entire copy but a class is accidentally overwritten by another version of the class. If `Z-0.2.0` included another class `H` instead of `C`, the jar file moves to R_2 because it is regarded as a partial copy.

The while loop in the algorithm analyzes R_1 as follows. At the beginning, a set of analyzed classes A contains all the classes in the target jar file: $A = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{E}, \texttt{F}, \texttt{G}\}$. The largest intersection with R_1 is $C = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ in `X-1.0`. Hence, a pair $(\{\texttt{X-1.0}\}, \{\texttt{A}, \texttt{B}, \texttt{C}\})$ is reported as a result. At the line 7, the jar file and classes are removed from R_1 and A respectively; i.e. $R_1 = \{\texttt{Y-0.1.1}, \texttt{Z-0.2.0}\}$, $A = \{\texttt{E}, \texttt{F}, \texttt{G}\}$. The second iteration of the loop outputs $(\{\texttt{Z-0.2.0}\}, \{\texttt{E}, \texttt{F}\})$ and results in $R_1 = \{\texttt{Y-0.1.1}\}$, $A = \{\texttt{G}\}$. In the third iteration, A no longer has an intersection with `Y-0.1.1`. The resultant pair $(\{\texttt{Y-0.1.1}\}, \phi)$ is ignored at the line 6. R_1 becomes empty, and then the algorithm proceeds to R_2. Since `X-1.1` has no intersection with A, it is also ignored. As a result, two pairs of jar files and classes are output by our method as follows.

J	C
{ X-1.0.jar }	{ A, B, C }
{ Z-0.2.0.jar }	{ E, F }

The result shows that three classes are copied from `X-1.0` and two classes are copied from `Z-0.2.0`. The result also implies the remaining class `G` is likely unique to the target jar file, because it is not found in any jar files in R.

The computational cost of the comparison step is dependent on both the size of a target jar file t and a repository R. The worst case is $O(nk)$, where n is the total number of class files in R, k is the number of jar files reused in t, respectively. Computation of $overlap_N$ values takes $O(n)$ time, because it tests whether each class name in $r \in R$ is included in t or not, using a hash set of class names for t ($O(1)$ time for each test). Similarly, computation of intersection at the line 5 compares class signatures in $O(n)$ time for each iteration, using a hash set for class signatures. The while loop is repeated k times until the k jar files in t are identified. To reduce the cost, our implementation reuses the already computed intersections in the loop; when A is updated to A' ($A' \leftarrow A \setminus C$), updated intersections for A' is obtained by $A' \cap Classes(r) = (A \cap Classes(r)) \setminus C$.

4. EVALUATION

To evaluate our method, we try to answer the following research questions:

RQ1. *How accurate is the result reported by our method?*

RQ2. *Does our method finish the analysis in a practical time?*

Research Design. For RQ1, we compare our method with Software Bertillionage. We use a golden dataset to evaluate precision and recall of the methods. To cover various combinations of existing components, we create our own dataset. More concretely, we randomly select jar files in the repository and copy the contents to a new jar file. Then we apply the methods under comparison to the created jar file and compare the reported results with the original files.

For RQ2, we evaluate practicality since our method is more computationally expensive. We use the performance data collected from RQ1 in relation to time and hardware considerations.

4.1 Comparison with Software Bartillonage

As the baseline of accuracy, we use the similarity metric of Software Bertillonage to list candidates of jar files included in a target jar file. If a jar file $r \in R$ satisfies $sim(t, r) > 0$, r is selected as a candidate. If multiple versions of the same component are involved in the list, we select only the most similar version (if tied, select all of them) and exclude less similar versions. In the case of Figure 1, the resultant list of candidates is $D = \langle\{\texttt{X-1.0}\}, \{\texttt{Y-0.1.1}\}, \{\texttt{Z-0.2.0}\}\rangle$, as illustrated in Section 3. We refer to this similarity-based list as a "Bertillonage" list. We do not use a threshold for filtering jar files so not to miss original files.

Our method added two elements to Software Bertillonage approach: an extended class signature and a greedy algorithm. To analyze how the elements contribute to the result, we introduce two variants of our method.

Bert+Sig is a list of jar files created in the same way as the Bertillonage list but using our class signature.

Bert+Greedy is a result of our greedy algorithm but uses a class signature defined in [8, 9].

We refer to our method as "Full" to distinguish it from the two variants.

4.2 Dataset Construction

We create a jar file from jar files in a repository R. We use a snapshot of the Maven Central Repository named `sourcerer-maven-aug12` as our repository R. The repository includes 172,232 jar files in the dataset, excluding source code archives and corrupted files. The files include 23,336 artifact IDs in Maven.

To analyze how accuracy is affected by the number of jar files included in a target jar file, we introduce a parameter N representing the number of original jar files. Given N, we create a jar file t through the steps in Algorithm 2. It starts with an empty jar file. The line 3 randomly selects r_i that provides at least one new class of a new component to t. Two jar files are regarded as different versions of the same component if they have the same file path except for their version number parts. The line 4 copies class files without overwriting a class file already copied to t.

Algorithm 2 Goldset Construction

```
1: Initialize $t$ as an empty file.
2: for $i = 1$ to $N$ do
3:     Randomly select a jar file $r_i \in R$ such that
       $Names(r_i) \not\subseteq \bigcup_{1 \le k \le i-1} Names(r_k)$ and $r_i$ is a different component from $r_k$ ($1 \le k \le i-1$).
4:     Copy class files in $r_i$ to $t$. Skip a class file if the class is already included in $t$.
5: end for
6: Add $t$ and $G(t) = \{r_1, \cdots, r_N\}$ to the golden dataset.
```

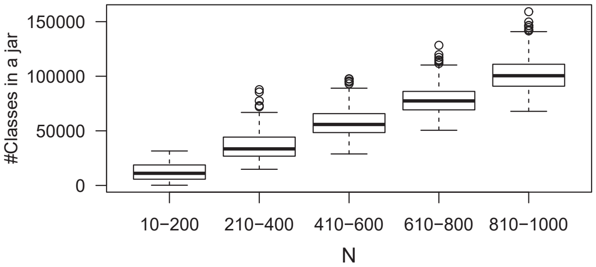

Figure 2: Distribution of the size of created jar files

The selected jar files for t form a goldset for t: $G(t) = \{r_1, \cdots, r_N\}$. In addition to the jar files, we construct a set of files $V(t)$ such that each jar file $v \in V(t)$ includes the same set of files copied at Step 4. In other words, jar files in $V(t)$ are possible code origins that could provide their classes to t. Since a jar file in $G(t)$ also satisfies the condition, $V(t) \supseteq G(t)$ holds. For example, suppose `commons-httpclient-3.1` is selected as r_i and its 167 class files are copied to t, Then, $V(t)$ includes the file and another jar file named `org.apache.servicemix.bundles.commons-httpclient-3.1_4`, because the archive also contains the same 167 class files. We identify such jar files by searching file hash values in all the jar files in R.

We vary the size of a jar file N from 10 to 1,000 by 10. For each N, we create 10 jar files. Hence, our dataset T includes 1,000 target jar files.

As an overview of the created jar files, the numbers of classes in the files are shown in Figure 2. The smallest jar file contains 196 classes, the maximum one contains 159,104 classes. The median is 56,304. We believe $N = 1,000$ and the number of classes are sufficiently large to include the size of a practical application; for example, Eclipse IDE for Java Developers 4.4.1 comprises 519 jar files. For each selected file, at least one class has been copied. The median and the maximum number of copied classes per selected file are 15 and 36,189, respectively. 58.1% of selected jar files provided classes that are unique to the jar files. On average, 4.6 jar files include the same file set. 3.4% of selected files are

Table 1: Accuracy of the methods under comparison

Method	*Precision*	$Precision_{jar}$	*Recall*
Our method (Full)	0.998	0.803	0.997
Bert+Greedy	0.997	0.678	0.997
Bert+Sig	0.357	0.231	0.994
Bertillonage	0.357	0.207	0.993

partially copied, because their contents overlap with already copied files. In those cases, 42.4% of class files in a selected file are not copied on average.

4.3 Accuracy

We evaluate accuracy of our method by precision and recall. Given a jar file t, each of the methods under comparison reports a list of detected jar file sets $D = \langle J_1, \cdots, J_{|D|}\rangle$. Each file set J_i indicates that some classes are reused from one of the jar files in it. For example, $J = \{r_1, r_2\}$ represents that "files are copied from either r_1 or r_2." We classify a set J_i to true positive if $G(t) \cap J_i \neq \phi$. The set J_i is classified to false positive if $G(t) \cap J_i = \phi$. Based on the classification, we define the precision of a list D as follows.

$$Precision(t, D) = \frac{|\{J_i \in D | J_i \cap G(t) \neq \phi\}|}{|D|}$$

A high precision indicates that a user can trust the reported set.

Even though jar files in J_i have the same class signatures, the classes may have different contents. For example, a change of a boolean operator in a conditional predicate does not affect a class signature but changes the binary. In such a case, even if J_i includes a correct answer, a user might have to identify one of them in the set. To analyze this effort, we define another precision $Precision_{jar}$ to represent how many jar files in J_i have a copy of the actually copied files.

$$Precision_{jar}(t, D) = \frac{|\bigcup_{J_i \in D} J_i \cap V(t)|}{|\bigcup_{J_i \in D} J_i|}$$

If this precision is high, J_i includes less variants. A user can easily obtain the original files.

Recall represents how many original jar files are detected, while precision indicates how many reported files are correct. If an element in the goldset is not involved in any $J_i \in D$, it is a false negative. Hence, recall of a list D is defined as follows.

$$Recall(t, D) = \frac{|\bigcup_{J_i \in D} J_i \cap G(t)|}{|G(t)|}$$

A high recall is important for users to investigate the result, because the number of existing components is too large for manual investigation.

4.4 Result

Table 1 summarizes the precisions and recall evaluated on the whole detection result for the created dataset. While we show the details later, all the metrics of our method are higher than Software Bertillonage ones, and the differences are statistically significant.

Figure 3 shows box plots indicating precision of the methods. The Full and Bert+Greedy versions achieved significantly higher precision than Bertillonage. The mean precision of our method is 0.998, while one of Bertillonage is 0.357. Kolmogorov-Smirnov test indicated that the four distributions are normal (all p values are less than 10^{-6}). A paired t-test shows that the mean difference 0.641 between our method and Bertillonage is significant ($p < 10^{-10}$).

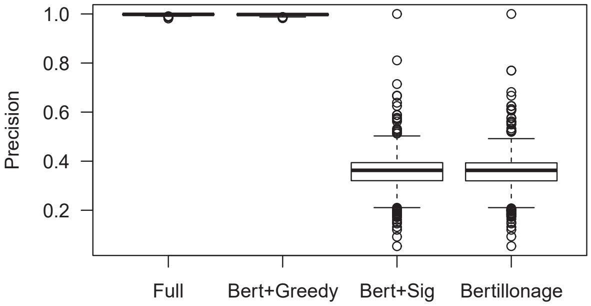

Figure 3: Distribution of ***Precision***

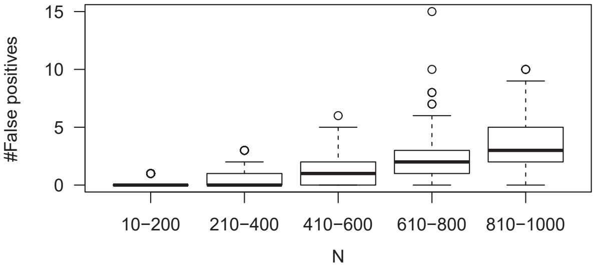

Figure 4: The number of false positives of the Full version of our method

The difference is caused by our greedy algorithm. Since a component sometimes includes an internal copy of its dependent library, a target file has a certain degree of similarity to other components including the same library and the library itself. Our algorithm links a class to at most one set of jar files, while a simple similarity-based list includes all the candidates as-is.

Our class signature does not contribute to this precision. Although a paired t-test shows the difference between the Full and Bert+Greedy versions is statistically significant ($p < 10^{-10}$), the actual difference is very small (less than 0.1%).

The number of false positives of our method is very small but correlated to the number of jar files in a target jar file (N). The correlation coefficient between them is 0.670. Figure 4 shows the distribution of the numbers. Our method generated the precise result ($Precision(t) = 1$) for 406 jar files. False positives are often reported when a combination of copied jar files is recognized as another jar file. For example, classes included in `grizzly-comet` 2.1.6 and `grizzly-http-server-core` 2.1.11 are detected as a copy of `grizzly-comet-server` 2.1.11. The falsely recognized jar file actually contains the exact copy of the class files. Similarly, a version of `org.apache.aries.jmx.core` is recognized as the same version of `org.apache.aries.jmx`. These are common cases because a project often releases a small set of classes

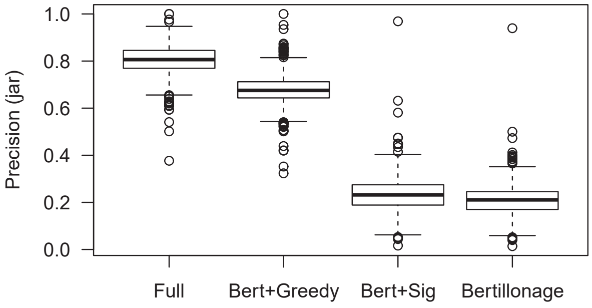

Figure 5: $Precision_{jar}$ **computed on all the reported jar files**

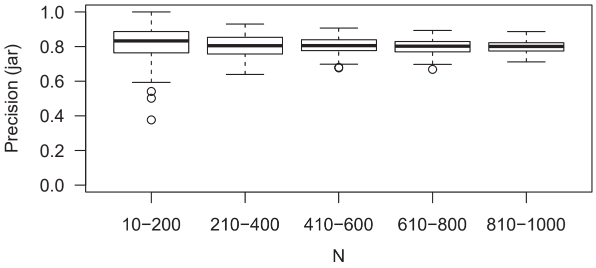

Figure 6: $Precision_{jar}$ **of the Full version of our method for different N**

as a core component and its superset as another component.

Figure 5 shows the distributions of $Precision_{jar}$. As indicated in the figure, the Full version of our method is more precise than the Bert+Greedy version. Kolmogorov-Smirnov test indicated that the four distributions are normal ($p = 0.019, 0.004, 0.001,$ and 0.003, respectively). A paired t-test shows that the precision of the Full method is significantly higher than Bert+Greedy ($p < 10^{-14}$), and the mean difference is 0.126. Hence, our class signature contributes to identify original jar files more accurately, by reducing a chance of an accidental match.

The boxplot also shows that our method may report a large number of false positive jars. The false positives are caused by a particular jar file. An example jar file causing the worst $Precision_{jar}$ in the figure is `org.glassfish.hk2.test-harness` 1.6.10. The jar file and its consecutive versions (1.6.11 and 1.6.12) include the same six files. Given a target file t includes a copy of the file, our method reports that the same classes are included in 196 versions of `com.sun.enterprise.test-harness` (e.g. 0.3.19–0.3.101 and 1.0.0-1.0.75) in addition to the original files. We regarded them as false positives, because they have different file hash values from the copied class files. Our signature-based comparison cannot distinguish them.

The ratio of false positive jars are not affected by the number of original jar files in a target file (N). Figure 6 shows the distributions of the $Precision_{jar}$ of our Full method for different N ranges. The correlation coefficient between the number of false positive jars and the number of jar files N is -0.097. Hence, the false positive jars are not caused by a combination of jar files.

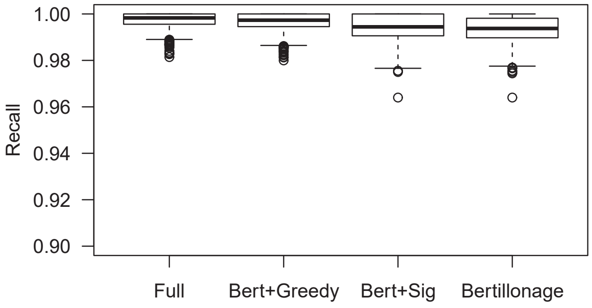

Figure 7: Distribution of Recall

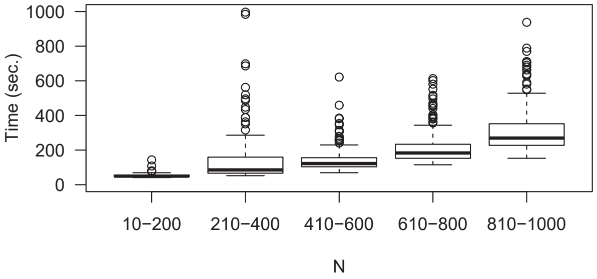

Figure 8: Time to analyze a jar file for different N

Figure 7 shows *Recall* of the methods. The mean recall of our Full method is 0.997, while the Bertillonage is 0.993. All the methods result in high recall, because we do not use a threshold to excluding less similar jar files from the analysis. The mean difference between our method and Bertillonage is only 0.004, though a paired t-test shows that the difference is significant. Our method can be seen as an automatic selection of jar files to improve precision, while slightly improving recall.

The original jar files are missing in our result because multiple original jar files are falsely recognized as a different jar file. Hence, the number of false negatives is correlated to the number of false positives. The correlation coefficient is 0.635.

4.5 Performance

We used a workstation equipped with two six core processors (Intel Xeon E5-2620 2.0GHz) and RAM 64GB. We implemented the signature extraction step as a concurrent process. It took 28.1 minutes to analyze 172,232 jar files in the repository using 20 threads. The extracted class signatures are compressed and stored in binary files. Their total size is 1.0GiB. The size is much smaller than the original repository data that occupies 77.8GiB.

Figure 8 plots the time cost of our method to analyze a target jar file using the signature database. The time includes an entire execution of a tool: Reading the signature data from files, extracting signatures from a target jar file, comparing the signatures, and generating a report file. The longest time spent for a target jar file is 16.6 minutes using a single thread. The time cost is practical to verify compo-

Table 2: A part of the ouptut of our method applied to FacebookLinkBench.jar

Jar file (#Classes)	#Copied
jruby-complete-1.6.5 (7673), jruby-1.6.5 (7673)	7673
hbase-0.94.3 (2269)	2269
guava-16.0.1 (1678)	1678
jmxutils-1.16 (1415)	1415
hadoop-core-0.20.2 (1376)	1376
org.mortbay.jetty.jsp-2.1.6.1.14 (585)	490
commons-collections-3.2.1 (458)	458
org.jbundle.util.osgi.wrapped. org.apache.http.client-4.1.2 (436)	436
jackson-mapper-asl-1.8.8 (432), jackson-mapper-lgpl-1.8.8 (432)	432
log4j-1.2.17 (314)	314
org.apache.bval.bundle-0.4 (220)	215
commons-httpclient-3.0.1 (148)	148
slf4j-log4j12-1.4.3 (6)	6
bval-bval-jsr303-0.5 (173)	5
slf4j-simple-1.7.0 (6)	3

nents actually included in every release candidate. We also expect that the time is much shorter than manual analysis to exclude false positives (60% on average) from Software Bertillonage results.

5. APPLICATION OF COMPONENT DETECTION

To demonstrate a usefulness of our method, we have applied our method to analyze two existing jar files. The first case represents a release engineering situation that compares the dependencies managed by developers with components actually included in a product. The second case assesses a risk of a product whose components are unrecorded by developers.

5.1 Facebook LinkBench

Facebook Linkbench [14] is a benchmark program. It uses Maven to download dependent components and create an executable jar file. We analyzed `FacebookLinkBench.jar` that was built using our workstation on December 2015. Since the code has been released on April 2013, we have added newer jar files in the Maven Central Repository to our repository in prior to the analysis.

Our method reported 89 jar file sets as code origins for `FacebookLinkBench.jar`. Table 2 shows a snippet of the output. In the table, file names are shortened for readability. Each file name is followed by a number indicating the number of classes in the file. The second column #Copied indicates the number of classes found in the target jar file. For example, our method reported that 7673 classes in the target jar files are copied from either jruby-complete-1.6.5 or jruby-1.6.5. The copy is likely an entire jar file, because both jar files include 7673 classes. Similarly, 78 of 89 are reported as entirely copied jar files.

The jar files jackson-mapper-asl and jackson-mapper-lgpl include the same classes under difference license (ASL and LGPL as indicated in their names). Our method cannot distinguish one from the other.

Our method reported org.apache.bval.bundle-0.4 and bval-bval-jsr303-0.5. They are different versions of relevant components. Since the project website [1] explicitly declares the bundle component includes bval-core and bval-jsr303, bundle-0.4 is likely a false positive of either bundle-0.5 or bval-core-0.5.

The `pom.xml` file of the project declares 10 dependent components such as hadoop, hbase, log4j, and slf4j-simple. All of them are identified by our method. An interesting observation is found in `org/slf4j/impl` package. The package includes nine classes copied from slf4j-log4j12 and slf4j-simple. Each jar file includes six classes. However, they include different versions of the same classes: `StaticLoggerBinder`, `StaticMarkerBinder`, and `StaticMDCBinder`. These classes in slf4j-simple are not copied, because another definition of the three classes are copied from slf4j-log4j12. The `StaticLoggerBinder` class is used to define which logger classes are used in a program. Since the target jar includes a copy from slf4j-log4j12, the program always uses log4j. Another logger class in slf4j-simple is never used, even though it is explicitly declared as a dependency.

In addition to the declared dependencies, the output of our method includes transitively dependent components. In Table 2, a variant of org.apache.http.client 4.1.2 and commons-httpclient 3.0.1 are included. The former one is a newer version of the latter one, but already outdated. A security vulnerability allows a man-in-the-middle attack [33]. Another component commons-collections-3.2.1 is also marked as vulnerable today [2].

While our method identifies components directly included in the target jar file, we can analyze the inside of the detected components by temporarily removing the components from the database. For example, our method identified a copy of Joda-Time 1.6.2 in jruby-1.6.5.

The analysis of the LinkBench jar could be hard if we used Software Bertillonage alone, because the similarity metric reported that 610 components are either entirely or partially included in the jar file. Nevertheless, Software Bertillonage is also still useful, if a user is interested in a particular component. For example, suppose a user would like to know if a target jar file includes vulnerable versions of Apache Standard Taglib that allow remote attackers to execute arbitrary code [34]. In such a case, the user can quickly scan the target jar file using the similarity metric. Indeed, the similarity metric reported that Apache Standard Taglib 1.2.0, a vulnerable version, is likely included in the application jar file. While the component name does not appear in the dependencies declared by developers, our method automatically identified jsp-2.1-6.1.14 as the origin of the class files in the package `org/apache/taglibs`. The information gives a hint for developers to update their dependencies.

5.2 muCommander 0.9.0

muCommander 0.9.0 is a file manager that can manipulate both local and remote files. In the source code repository [31], `lib` directory is unavailable in the revision tagged as `release_0_9_0`. Since third-party components for 0.8.5 are preserved in the revision `release_0_8_5`, we added the files to the repository to analyze components in 0.9.0.

Our method reported 34 jar file sets for 0.9.0. Table 3 shows a part of the result. In the table, libraries in the `lib`

Table 3: A part of the ouptut of our method applied to muCommander 0.9.0

Jar file (#Classes)	#Copied
0.8.5/jna (148)	148
0.8.5/icu4j (54)	18
commons-httpclient-3.1-beta1 (166)	61
commons-httpclient-3.1 (167), 0.8.5/commons-httpclient (167)	2
0.8.5/jcifs (247)	29
0.8.5/j2ssh (294)	67
logback-classic-1.0.2 (155)	30
logback-core-1.0.2 (261)	48

directory of 0.8.5 are indicated by a prefix "`0.8.5/`". According to the result, libraries used in 0.8.5 including JNA, ICU4J, jCIFS, and J2SSH are not updated for 0.9.0. Our method reported that commons-httpclient in the release is likely 3.1-beta1, rather than 3.1. This is probably because the copied files are modified from the original files in 3.1, although the difference is unrecorded.

The analysis result does not include some libraries such as commons-logging and J7Zip that are included in 0.8.5. Hence, they are likely removed from the version. Logback 1.0.2 is a logging library that is likely introduced in this version. However, the library names in README are inconsistent to the analysis result. The README still contains J7Zip but nothing about Logback.

In the case of muCommander 0.9.0, there is no official record of library versions. The project uses JDK 1.4; all the classes in the jar file have different MD5 file hash values from classes in the repository. Our signature-based comparison successfully identified candidates of original jar files in such a situation. The result indicates that the product uses several old lbiraries but the actual state is not provided to users of the product.

6. THREATS TO VALIDITY

Our method uses a signature-based comparison. The definition of a class signature is an extended version of [8, 9]; method calls and field access in methods are added. Our signature definition assumed that compilers generate the same numbers of method calls and field access from the same source code. Although we believe existing compilers such as Oracle JDK and Eclipse JDT satisfy the condition, it is not ensured by Java language specification. For example, a compiler may generate optimized code (e.g. loop unrolling) including additional method calls. Hence, the method still has a risk to miss existing components.

We have used a snapshot of the Maven repository as a repository of existing components. Although the repository covers various components, precision and recall are affected by the jar files stored in the repository. It should be noted that our method is not dependent on Maven. We have used artifact names in the repository only for creating the goldset for the experiment.

The experiment evaluates accuracy on an ideal repository that includes all existing components. The effect of an unknown component is dependent on its content. If all the classes are unique to the component, it is simply unrecognized. If a copy of a class is included in a jar file in the repository, the jar file is reported as a false positive.

The goldset is artificially created by a random selection of components. It does not take the popularity of components into account. In addition, it may include an unrealistic combination, e.g. incompatible components.

When creating the goldset for the experiment, we used SHA-1 file hash to identify jar files containing a copy of classes. A file hash value is dependent on a compiler and its configuration in addition to source code. Hence, we could miss class files compiled for a different JDK version. Since a component having a different file hash is regarded as a false positive, we might underestimate $Precision_{jar}$ reported by the methods under comparison.

The goldset files are created by copying entire files. A jar file is partially copied only when the same classes are already copied from another file. The process emulates a simple black-box reuse style; Maven Assembly Plugin also creates an all-in-one jar file in a similar way. While 3.4% of selected files are partially copied in the experiment, the ratio might be different from actual reuse activity.

7. CONCLUSION

Reuse of third party components is an important activity in software development. On the other hand, developers and users have to be aware of components and their versions in products to analyze security issues, update the components, and check a license. In this paper, we proposed a method to automatically detect jar files whose class files are included in a target program. Using a greedy algorithm, our method reports a list of jar files. While our method may report multiple jar files containing the same classes, 99.8% of the reported sets include a correct file. In addition, 80.3% of the jar files in the sets include the copied files.

We have applied our method to two applications: Facebook LinkBench and muCommander. The method identified vulnerable components included in the products in both files. It also identified a component that was improperly copied to the LinkBench jar file and a component that is not documented in muCommander. The reported lists of components enable developers and users to assess potential risks caused by the components in the applications.

Our implementation of the method is available online.[1]

In future work, we would like to extend our method to identify renamed and/or customized components in a product. Since our method uses identifier names, analysis of the impact of code obfuscation is also important. Another direction is an improvement of empirical studies on library migrations and code reuse activities in practice. For example, we can count the number of dependent components reused in a binary, ignoring their internal components. This kind of statistics might enable developers and managers to identify popular and/or valuable components in their organization.

Acknowledgments

This work is supported by JSPS KAKENHI Grant Numbers 25220003, 26280021, and 15H02683. This work is also supported by Osaka University Program for Promoting International Joint Research, "Software License Evolution Analysis."

[1] https://github.com/takashi-ishio/JIngredients

8. REFERENCES

[1] Apache BVal Project. Project modules. http://bval.apache.org/mvnsite/modules.html (Accessed January 29, 2016).

[2] Apache Commons Collections. Arbitrary remote code execution with InvokerTransformer, 2015. https://issues.apache.org/jira/browse/COLLECTIONS-580.

[3] Apache Maven Project. Apache maven assembly plugin. http://maven.apache.org/plugins/maven-assembly-plugin/.

[4] V. Bauer and B. Hauptmann. Assessing Cross-Project Clones for Reuse Optimization. In *Proceedings of the International Workshop on Software Clones*, pages 60–61, 2013.

[5] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 280–289, 2013.

[6] K. Chen, P. Liu, and Y. Zhang. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th IEEE/ACM International Conference on Software Engineering*, 2014.

[7] L. Constantin. Developers often unwittingly use components that contain flaws. ITWorld.com, http://www.itworld.com/article/2936575/security/software-applications-have-on-average-24-vulnerabilities-inherited-from-buggy-components.html [Posted June 16, 2015].

[8] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 183–192, 2011.

[9] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: Determining the provenance of software development artifacts. *Empirical Software Engineering*, 18:1195–1237, 2013.

[10] M. Di Penta, D. M. German, and G. Antoniol. Identifying licensing of jar archives using a code-search approach. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 151–160, 2010.

[11] J. Dietrich, K. Jezek, and P. Brada. Broken Promises: An Empirical Study into Evolution Problems in Java Programs Caused by Library Upgrades. In *Proceedings of the IEEE CSMR-WCRE 2014 Software Evolution Week*, pages 64–73, 2014.

[12] S. Duszynski, J. Knodel, and M. Becker. Analyzing the source code of multiple software variants for reuse potential. In *Proceedings of the 18th IEEE Working Conference on Reverse Engineering*, pages 303–307, 2011.

[13] Eclipse Documentation. Runnable jar file exporter. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftasks-37.htm.

[14] Facebook Engineering. Linkbench: A database benchmark for the social graph, 2013. https://www.facebook.com/notes/facebook-engineering/linkbench-a-database-benchmark-for-the-social-graph/10151391496443920/.

[15] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, pages 391–400, 2014.

[16] D. M. German and M. D. Penta. A Method for Open Source License Compliance of Java Applications. *IEEE Software*, 29(3):58–63, 2012.

[17] D. M. German, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 81–90, 2009.

[18] GWT Project. Getting started. http://www.gwtproject.org/gettingstarted.html.

[19] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck. On the extent and nature of software reuse in open source java projects. In *Proceedings of the 12th International Conference on Software Reuse*, volume 6727 of *Lecture Notes in Computer Science*, pages 207–222, 2011.

[20] 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*, pages 63–72, 2011.

[21] A. Hemel and R. Koschke. Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices. In *Proceedings of the 19th IEEE Working Conference on Reverse Engineering*, pages 357–366, 2012.

[22] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe. Where does this code come from and where does it go? – integrated code history tracker for open source systems –. In *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*, pages 331–341, 2012.

[23] 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*, 28(7):654–670, 2002.

[24] T. Kanda, T. Ishio, and K. Inoue. Extraction of product evolution tree from source code of product variants. In *Proceedings of the 17th International Software Product Line Conference*, pages 141–150, Tokyo, Japan, 2013. ACM.

[25] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue. Identifying source code reuse across repositories using LCS-based source code similarity. In *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, pages 305–314, 2014.

[26] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue. Visualizing the evolution of systems and their library dependencies. In *Proceedings of the 2nd IEEE Working Conference on Software Visualization*, pages 127–136, 2014.

[27] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010.

http://www.ics.uci.edu/~lopes/datasets/.

[28] 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*, pages 389–400, 2014.

[29] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, pages 282–291, May 2004.

[30] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. Hassan. A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2):78–86, 2014.

[31] muCommander Project. Subversion Repository of muCommander. https://svn.mucommander.com/mucommander/ (Accessed January 29, 2016).

[32] National Vulnerability Database. CVE-2014-0107, 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0107.

[33] National Vulnerability Database. CVE-2014-3577, 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3577.

[34] National Vulnerability Database. CVE-2015-0254, 2015. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0254.

[35] J. Qiu, X. Su, and P. Ma. Library functions identification in binary code by using graph isomorphism testings. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 261–270, 2015.

[36] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 215–224, 2014.

[37] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, pages 101–110, August 2013.

[38] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the 18th ACM International Symposium on Software Testing and Analysis*, pages 117–128. ACM, 2009.

[39] M. Sojer and J. Henkel. License risks from ad hoc reuse of code from the internet. *Communications of the ACM*, 54(12):74–81, 2011.

[40] D. Steidl, B. Hummel, and E. Juergens. Incremental Origin Analysis of Source Code Files. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 42–51, 2014.

[41] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 26(11):1030–1052, 2014.

[42] F. Thung, D. Lo, and J. L. Lawall. Automated library recommendation. In *Proceedings of the 20th IEEE Working Conference on Reverse Engineering*, pages 182–191, 2013.

[43] Y. Yano, R. G. Kula, T. Ishio, and K. Inoue. Verxcombo: An interactive data visualization of popular library version combinations. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension*, pages 291–294, 2015.