Software Bertillonage: Finding the provenance of an entity

Julius Davies†, Daniel M. German†, Michael W. Godfrey‡,
† Department of Computer Science, University of Victoria, Canada
‡ David R. Cheriton School of Computer Science, University of Waterloo, Canada
juliusd@uvic.ca, dmg@uvic.ca, migod@uwaterloo.ca

ABSTRACT

Deployed software systems are typically composed of many pieces, not all of which may have been created by the main development team. Often, the provenance of included components — which may include external libraries or cloned source code — is not clearly stated. This raises a number of both technical and ethical/legal concerns. Technically, it is often hard to maintain such a system if its external dependencies are not well documented. Ethically, code fragments that have been copied from other sources, such as open source software, may not have licences that are compatible with the released system. In this work, we motivate the need for recovery of the provenance of software entities by a broad set of techniques that include source code fact extraction, software clone detection, call flow graph matching, string matching, and historical analyses. We liken our goals to that of Bertillonage, a simple and approximate forensic analysis technique based on bio-metrics that was developed in France before the advent of fingerprints.

As a motivating example of this kind of work, we consider the PCI DSS security standard for e-commerce, which requires that an application should provide precise version information about any libraries that are packaged with it. In practice, this information is often not provided and so we have sought ways to infer it from available evidence.

We used a single Bertillonage metric of own invention, anchored signature matching, to analyze Java libraries from a proprietary e-commerce Java application. The application of this single metric allowed us to automatically provide exact version information for over 57% of our sample set, and to narrow the search space significantly for another 39%, providing actionable information on 96% of the libraries within the e-commerce application.

Keywords: Code search, mining software repositories, open source systems.

ICSE '2011 Waikiki, Honolulu, Hawaii

1. INTRODUCTION

Most deployed software systems are composed of many pieces drawn from a variety of disparate sources. While the bulk of a given software system's source code may have been developed by a relatively stable set of known developers, often components of the shipped product may have come from external sources. For example, software systems commonly require the use of externally developed libraries, which evolve independently from the target system. To ensure library compatibility — and avoid what is often called "DLL-hell" — a target system may be packaged together with specific versions of libraries that are known to work with it. In this way, developers can ensure that their system will run on any supported platform regardless of the particular versions of library components that clients might or might not have already installed.

Many North American financial instutions implement the Payment Card Industry Data Security Standard (PCI DSS) [1]. Requirement 6 of this standard states "All critical systems must have the most recently released, appropriate software patches to protect against exploitation and compromise of cardholder data." Suppose a java application running inside a finanical institution is found to contain a dependency on a java archive named foo.jar. Satisfying the PCI DSS requirement in this example is difficult:

- Which version of foo.jar is the application currently running?
- How hard will it be to upgrade to the latest version of foo.jar?
- Has the copyright or patent license of foo.jar changed in the newest version in a way that prevents upgrading?

We can use a variety of techniques for this: If we have source code we can do software clone detection. If we have binaries, we can use a variety of techniques including clone analysis of assembler token streams, call flow graph matching, string matching, mining software repositories, and historical analyses.

This kind of investigation can be performed at various levels of granularity, from code chunks to function and class definitions to files and subsystems up to compilation units and libraries. But the fundamental question we are concerned with is this: Given a software entity, can we determine where it came from? That is, how can we establish its *provenance*?

1.1 Contributions

We summarize the contributions of this paper as follows:

- We define the concept of software Bertillonage: a method to reduce the search space when trying to find an entity in a corpus where simple comparisons fail and no fingerprinting exist for the type of entity.
- We exemplify the use of software Bertillonage by presenting our method: anchored signature matching. This method finds matches between a supplied Java artifact (binary jar or Java source archive) and original Java artifacts within a large corpus.
- We demonstrate the effectiveness of our method with an empirical study that involves finding exact version information of binary jars used in an e-commerce application of a financial institution.

2. FORENSIC BERTILLONAGE AND SOFTWARE PROVENANCE

In the mid to late 18th century, police forces in Europe and elsewhere were beginning to take advantage of emerging technologies. For example, suspected criminals in Paris were routinely photographed upon arrest, and the photos were organized by name in a filing system. Of course, criminals soon found out that if they gave a false name upon being arrested that their chances of being identified from the huge pool of photos was very small unless the police were particularly patient or happened to recognize them from a previous encounter. Alphonse Bertillon, the son of a statistician, had the idea that if suspects could be routinely subjected to a series of simple physical measurements — such as height, length of right ear, length of left foot, etc. — then the photos could be organized hierarchically using the bio-metrics data, and the set of photographs that had to be examined for a given suspect could be reduced to a small handful. This approach, which was subsequently termed *Bertillonage* in his honour, proved to be very effective and was a huge step forward in criminology [2].

As a forensic approach, Bertillonage also had its drawbacks. Using the specialized measuring equipment required extensive training and practice to be reliable, and it was time-consuming to perform. Each of 10 measurements was performed three times, because if even one measurement was off then the system did not work. Also, the measurements taken did not have a high degree of independence; tall people tended to have long arms too.[1] In time, the emerging science of fingerprinting proved to be a much more effective and accurate identification mechanism and Bertillonage was forgotten. Nevertheless, Bertillon and his other inventions — including the modern mugshot and crime scene photograpy — showed how simple ideas combined intelligently could greatly reduce the amount of manual effort required in forensic investigations. Despite its limitations, Bertillonage was considered the best method of identification for two decades [3].

Our goal in this work is to devise a series of techniques to aid determining the *provenance* of software entities. That is, given a software entity such as a function definition or an included library, we would like to be able answer the question: *Where did this come from?* Of course, most often the answer will be that the entity in question was designed to fit exactly where it is within the design of the system, but sometimes entities are moved around, designs are refactored, new is copied from old and then tweaked. We would like be able to answer this question authoritatively: this is version 1.3.7 of the X library; this SCSI driver is a tweaked clone of a driver of a similar card; most of this function f was split off from function g during a refactoring effort in the last development cycle. Sometimes, however, our answers will be a best effort guess, especially if we do not have authoritative access to the original developers.

We therefore use the metaphor of software Bertillonage, rather than say software fingerprinting, as we often lack sufficient evidence to make a conclusive indentfication. Instead, we use a set of simple and sometimes ad hoc techniques to narrow the search space down to a level where a manual determination may be feasible.

[1]When Francis Galton realized this, he devised statistical correlation.

3. RELATED WORK

In software engineering research, similar questions have been addressed in various guises. For example, there is a large body of work in software clone detection that asks the question: Which software entities have been copied (and possibly tweaked) from other software entities. Our own work [4] on the problem of "origin analysis" asked: If function f is in the new version of the system but not the old, is it really new or was it moved / renamed / merged / split from another entity in the old version? The emphasis here is to broaden the question even more: given the recent advances in the field of mining software repositories, can we take advantage of the vast array of different kinds of software development artifacts to draw conclusions about the provenance of software entities?

There exist many studies on the origin, maintenance and evolution of clones [5, 6, 7, 8, 9, 10]. Others have concentrated on their lifespan and genealogy [11]. Our main difference with those studies is that we study provenance across applications, and are not only interested in finding similar (or identical) entities, but where they come from. We are also interested in matching similar (or identical) entities to when one of them is in compiled form.

Clone detection methods (such as [12, 13]), as well as the tracking of clones between application [14] provided a starting point for our investigation. Similar to Holmes et al. [15] we build our own code-search index.

Di Penta et al. [16] used code search engines to find the source code that corresponds to a Java archive (they used the fully qualified name of the class). They found that their main limitation was the inability to match a binary jar to the precise version it came from. We, however, consider this a simple method of Bertillonage.

4. A FRAMEWORK FOR SOFTWARE BERTILLONAGE

The main goal of Software Bertillonage is to address the problem of finding the source of a software artifact, even when such artifact is in binary form (compiled). Software Bertillonage (Bertillonage from now on) will help to significantly reduce the large search space for potential candidates. This is different from software fingerprinting, where the ob-

jective is to define a method to uniquely identify a subject– Bertillonage is not accurate enough to consistently provide a unique identifier. In fact, accurate fingerprinting is still an open problem. To our knowledge, there are currently no published fingerprinting techniques that will always match a binary to its source code.

First we define a 'subject' as the entity for which we are interested in tracking its provenance. We define 'candidates' as a set of entities that are likely matches to the subject. A desirable property of Bertillonage is to provide, for any subject, a relatively small set of candidates.

Once a smaller set of candidates has been identified, other methods (usually significantly more expensive to apply) can be used to verify the identity of the subject (such as clone detection, or differencing tools, if the subject and the candidates are all in source code form) or assembly code to source comparison (if the subject is binary and the candidates are source), or assembly to assembly comparison (if candidates and subjects are all binary). Some of these methods might require significant manual labour.

4.1 Bertillonage Metrics

Like the traditional Bertillonage, it is necessary to define a set of metrics that can be measured in a potential subject, that will be relatively unique to it. This is particularly difficult when trying to match binary to source code, because many of the original features of the source code might be lost during the compilation; for example, identifiers might be lost, some portions might not be compiled, source code entities are translated into binary form (which might include optimizations), etc.

Given the variety of programming languages, we presume that each will require different Bertillonage metrics. For instance, compilation to Java is easier to analyze than compilation to C++, and contains richer information. In turn, C++ binaries maintain more information than compiled C (C++ maintains parameters types –to support overloading– while C does not).

Another important consideration is: what is the level of granularity of the Bertillonage? To match an entire software system it might not be necessary to look inside each function/method. But if the objective is to match a function/method, then the only information available to measure are method bodies and type signatures.

Bertillonage is concerned with measuring the intrinsic properties of a subject. These measurements can be of different types, for example:

count-based: how many of these "objects" the entity contains, such as number of calls to external libraries, or uses of an obscure feature (how many times is *setjmp*, *longjmp* used);

set-based: compute a set of "objects" the entity contain, such as a set of string literals defined by this entity (for example, The *GPL Compliance Engineering Guide*, developed under the auspices of the Linux Foundations, recommends the extraction of literal strings to determine potential violations [17]), or set of classes defined in a package;

sequence-based: compute a sequence of "objects" in the entity (i.e. preserve the order of the objects), such as the sequence of methods signatures of a class, or sequence of calls in a method, sequences of tokens types (commonly used in clone detection), etc;

relationships-based: use other entities or "objects" that the "object" under measurement is related to; for example, what are the dynamic libraries used by this program, what are the C standard library functions used by this function, what is the internal callgraph of this program.

Table 1 summarizes these observations. A good Bertillonage metric should be computationally inexpensive, applicable to the desired level of granularity and programming language, and when applied, it should significantly reduce the search space (i.e match few candidates from a large number of potential ones).

Granularity	Code Snippet Function/Method Class Package Program/Library etc.
Type of subject	Source Binary
Type of Metric	Count-based Set-based Sequence-based Relationship-based
Applicable Language	C Java etc.

Table 1: Characteristics of Bertillonage Metrics

5. MATCHING THE EXACT VERSION OF A JAVA BINARY ARCHIVE TO ITS SOURCE CODE

To exemplify the concept of Bertillonage, we propose a Bertillonage metric that addresses the problem: if we are given a java binary archive, can we determine its original source code?

As described in Di Penta et al. [16], Java applications are frequently bundles of binary artifacts. To avoid dependency problems such applications include in their distributable archive copies of the dependencies. These copies are of two types. The first is embedding binary archives (which in turn might include other binary archives); and the second making copies of the source code of these dependencies, such that they are compiled and included as part of the application.

The most obvious source of information is the name of the archive itself (i.e. `commons-codec-1.1.jar` would come from `commons-codec`, an Apache project, version 1.1)[2]. But not all projects adhere to consistent naming and numbering policies, and mechanisms for auto-numbering a binary's version

[2]This is similar to a policeman asking a subject for her/his name and expecting a correct answer.

can be difficult to implement. This approach breaks when the source of the project is copied into the tree of the given application, as there is no longer a specific archive for the embedded dependency.

Another approach, implemented by Di Penta et al. [16], is to use fully qualified name of each class, and a code search engine. While effective to find the source of a class, the main drawback of this approach is that it cannot match the binary to a precise version of the source.

On the opposite end of the spectrum, we could build a database of exact source-to-byte matches by compiling all known sources and indexing the results. False positives are impossible under such a scheme, and thus matches would provide a direct and unquestionable link back to source code. But false negatives could arise in several ways, among these: variation of compilers (e.g. Oracle's javac7 vs IBM's jikes1.22), debugging symbols (on or off), and different optimization levels. To address these we could try to compile our sources under all known compiler variations. Unfortunately, additional avenues for false negatives remain. For example, the build scripts themselves might inject information at build-time directly into the class files (mechanisms for auto-numbering binary versions sometimes do this). Theoretical limitations aside, many practical concerns prevent us from attempting to compile all known versions of all known sources in the open source java universe: engineering such an index would be computationally and organizationally challenging, and library dependencies can be difficult to satisfy (especially for older artifacts) making full compilation a problem.

Applying Bertillonage to binary archives requires us to find characteristics that we can use to match the binary archive components (the class files) to their source code, and that are easy to measure and compare such that, even if they do not guarantee a perfect match, they will significantly reduce the search space.

Class and package names very rarely change over a library's lifetime since such changes break drop-in compatibility for integrators, but other features of the library are free to change as a library evolves (besides the source code inside each method body). We are particularly interested in features that survive the compilation process. For Java we considered the following list of attributes that are present in both source and binary forms:

- inheritance tree,
- implemented interfaces,
- constructors,
- annotations,
- method names, their return types, and their parameters (names and types),
- class, method, and constructor visibility,
- some class and method modifiers (i.e. abstract), and
- relative position of methods in the class.

Many other features are lost during compilation. This includes: comments, import statements (the original import statement is "resolved" and replaced with the actual one that depends on the environment in which it is compiled—such as the value of the CLASSPATH variable), parameter modifiers (such as `final`), and absolute position of methods, since line numbers are preserved only when the class is compiled with debug info.

```
1  package a.b;
2
3  import g.h.*;
4
5  /**
6   * @author Jane Doe
7   * @since January 1, 2001
8   */
9  public class D implements I<Number> {
10
11   synchronized static int a(
12     String s
13   ) throws E {
14     // }}}}} A comment!!!!
15     String b = "// {";
16     return b.hashCode() - s.hashCode();
17   }
18 }
```

Figure 1: Hypothetical source code of a class D.

In a nutshell, our proposal for applying Bertillonage to binary archives is to define a *metric* that can be used to match a binary class file to its likely source file (we will formally define it below). Not all source code classes are included into a binary; for example, test classes are usually not included, and sometimes a source archive is split into two or more binary archives. To match a binary archive, we try to find the source archives with the largest overlap of classes between the binary archive and a source archive.

5.1 Anchored Class Signature

We characterize a class C, with methods $M_1, ..., M_n$ (in either source or binary form) to possess an *anchored class signature*, denoted as $\vartheta(c)$, and defined as a tuple:

$$\vartheta(c) = \langle \sigma(c), \langle \sigma(M_1), ...\sigma(M_n) \rangle \rangle$$

where $\sigma(a)$ is the type signature of the class or methods a. That is, the anchored signature of a class is the type signature of the class itself, and the ordered sequence of the type signatures of each of its methods. We say the signature is anchored because it includes the fully qualified name, including the namespace, of the Java file. Similarily we could define an un-anchored signature to be the same, but without the fully qualifed name (missing the "package" part).

When building the surface signature, all fully qualified names in the decompiled bytecode are stripped of their package prefixes (i.e. `g.h.I` becomes `I` and `java.lang.String` becomes `String`) since identifying the fully qualified names from source depends on Java's import mechanism, which is indeterminate, as mentioned above. Fully qualified names referenced directly in source, though rare, are also stripped of their package prefixes, since we have no way of knowing in the bytecode if the name came from an import or from an inline declaration.

Consider a class file `D.java` (depicted in figure 1) and its corresponding decompiled bytecode (shown in figure 2). The java compiler will insert an empty constructor if no other constructors are defined, and for that reason the bytecode version contains an empty one. Class D's surface signature is depicted in figure 3, and it is composed of the type signature of the class, the type signature of the default constructor D, and the type signature of its method `a`.

```
1  package a.b;
2
3  public class D extends java.lang.Object implements
       g.h.I {
4
5    public D() {
6      // An empty default constructor inserted by
         javac,
7      // since all classes must have constructors.
8    }
9
10   synchronized static int a(
11     java.lang.String s
12   ) throws a.b.E {
13
14     /* [compiled byte code] */
15
16   }
17 }
```

Figure 2: Hypothetical decompiled version of a class D.

$\sigma(D)$ = `public class a.b.D extends Object implements I`
$\sigma(M_1)$ = `public D()`
$\sigma(M_2)$ = `default synchronized static int a(String) throws E`

$$\vartheta(D) = \langle \sigma(D), \langle \sigma(M_1), \sigma(M_2) \rangle \rangle$$

Figure 3: Normalized class signature for both D.java and D.class.

5.2 Similarity Index of Archives

To compare two archives we define a metric called the *similarity index* of archives, which is intended to measure how similar are the two archives with respect to the query surfaces of the classes that compose them. Formally, given an archive A composed of n classes c_i $A = c_1, ...c_n$, we extend the definition of surface signature to an archive as the

$$\vartheta(A) = \{\vartheta(c_1), ..., \vartheta(c_n)\}$$

We define the *Similarity Index* of two archives A, B, denoted as $sim(A, B)$, as the Jaccard coefficient of their surface signatures:

$$sim(A, B) = \frac{|\vartheta(A) \bigcap \vartheta(B)|}{|\vartheta(A) \bigcup \vartheta(B)|}$$

Ideally, a binary archive B would have been originated in source archive S if $sim(B, S) = 1$. In practice, however, this is not the case, as many classes in the source archive are not included in the binary archive (such as test cases). A source archive with a very large number of test classes might have a low similarity coefficient with its binary archive. Similarly, a large archive that includes a copy of source code of the original system (shade linking) will have a low similarity index with its binary archive, even though it might contain an exact match within its source code. To address these issues we define the concept of inclusion similarity.

5.3 Inclusion Index of Archives

The inclusion index of archive A in B, denoted as $inclusion(A, B)$ is the proportion of class signatures found in both archives with respect to the size of A.

$$inclusion(A, B) = \frac{|\vartheta(A) \bigcap \vartheta(B)|}{|\vartheta(A)|}$$

We interpret that, when the inclusion index between a binary archive A and source archive B is close to 1, the classes in A are present in the source code of B.

5.4 Finding candidate matches of a binary archive

Given a binary archive b, we can use the similarity and inclusion indexes to rank the likelihood that any archive in a corpus might contain the same code found in the binary archive. The higher the similarity index, the more likely both are instances of the same source code; and the higher the inclusion index, the more likely the candidate matched archive will contain a copy of the source code that created the subject binary archive. A candidate archive that has low similarity index, but high inclusion index is likely to be a bundle of several java applications, one of them the one that corresponds to the subject binary archive.

If the similarity index is zero, then no archive in the corpus contains a single class signature in common with the binary archive. A very low inclusion index might point towards a match to an archive that implements a common class signature.

We can formalize finding the best match(es) for a binary archive in an archive corpus as follows: given a set of archives $S = \{s_1, ..., s_n\}$ (the corpus), we find the *best candidate matches* a of binary archive b as the subset of $L \subseteq S$ such that:

$$\forall s_i \in L \quad sim(b, s_i) > 0 \wedge sim(b, s_i) = max_{sim}[S, b]$$

where $max_{sim}[S, b]$ is the maximum similarity index of b and the elements in S.

Ideally, the size of L is 1: only one source (or one binary) archive matches the binary archive. In general, the corpus could have several candidate matches of the archives (identical, or non-identical, such as changes in documentation). Furthermore, the same archive signature might be matched by more than one version of the same system (such as when an upgrade might not make any changes to the signature of any of the methods or classes in the archive, nor has added a new class–this is typical in minor release updates).

6. IMPLEMENTATION

6.1 Building a corpus

To be effective, any system that implements Bertillonage requires a corpus that is as comprehensive as possible. For Java the Maven 2 central repository fulfils this requirement. This repository acts as the Java community's de facto library archive. The repository was originally developed as a place from where the Maven build system could download required libraries to build and compile an application. Thanks of its broad coverage and depth, many competing java build systems and dependency resolvers currently make use of it.

6.2 Extracting the class signatures

We developed two distinct tools to extract the signatures of compiled class files and source files. Each tool addresses one type of artifact: one for bytecode and one for source. Similar to Di Penta et al. [16] we chose the bcel5 library for bytecode. We wrote our own parser for analyzing source files. As described in the previous sections, our goal is to create a normalized class signature that can be used to match bytecode to source.

6.2.1 Extracting a class signature from source

When analyzing a source file we need to first discard four things: annotations, import statements, generics and parameter names and modifiers. After discarding, we then must extract the class signature as we have defined it.

- *Extract the package and class line.* We must re-introduce the default 'extends Object' declaration as part of our normalization if the source in question does not subclass anything. For our example in Figure 1, the result is:

  ```
  public class a.b.D extends Object implements I
  ```

  For classes that implement more than one interface, we sort the interfaces in lexicographical order.

- *If necessary, re-introduce the default constructor.* The java compiler will insert an empty constructor if no other constructors are defined; we must also do the same:

  ```
  public D()
  ```

- *Extract methods, careful to preserve order.* If there are exception types listed in the throws clause, we sort these lexicographically. As for visibility, should a method declare itself neither private, protected, nor public, we then store it as 'default' in our signature, i.e:

  ```
  default synchronized static int a(String) throws E
  ```

  Due to limitations of our prototype java parser, we are currently ignoring inner-classes, abstract methods, and native methods.

6.2.2 Extracting a class signature from bytecode

The approach on the bytecode side is somewhat inverted. Consider the hypothetical 'decompiled' Figure 2 for D.class, from before.

Normalization here involves three activities:

- *Extract the package and class line.* As before we assemble a package and class line; this time we must shorten the fully-qualified names that bcel5 extracts.

- *Extract methods, careful to preserve order.* We shorten the fully-qualified names among the method paramater types and exception types. A visibility of 'default' is stored if necessary. Unlike the current version of our source parser, bcel5 has no problem extracting interfaces, abstract methods, inner classes, and native methods. We must note such and ignore them.

- *Remove methods introduced to implement non-generic interfaces.* Classes sometimes contain additional methods added by javac at the end of their definitions to satisfy non-generic implementations of genericized interfaces (for backwards compatibility). We tried our best to remove such methods, since they cause otherwise perfectly matching signatures to diverge. We suspect these continue to cause a number of match failures in our index despite our best efforts. Future work is needed here.

When normalization completes for either of our two examples, D.java, and D.class, we should possess a class signature identical to Figure 3.

6.3 Matching a subject artifact to artifacts within the corpus

The source and bytecode tools we developed to extract the signatures are employed both in the construction of a corpus database, as well as the generation of queries for the database. Matching subjects against the corpus involves 4 steps:

1. We create a database with all the class signatures from the corpus. To improve performance we index the database using a hash (SHA1) of the class signature.

2. For each subject artifact, we first extract its set of signatures using the same logic for building the database. We then query the database for any intersections with this subject set.

3. Intersections are grouped by the associated artifacts' absolute path within our Maven 2 mirror. Grouping only by jar name is inadequete because of the chance of duplicate jar names existing on separate paths.

4. The cardinalities of the intersections and unions are calculated. From these the index of similarity is calculated as shown in the formula above for $sim(A, B)$.

Note that, even in a perfect match, the archive signature similarity index might not be equal to 1. This is because the source package might contain some source java files that are not included in the binary jar, such as unit tests. However, every class in the binary archive should be present in the source archive.

7. EMPIRICAL STUDY

To evaluate the usefulness of our method for identifying correct original artifacts (binary and source) using a subject artifact of unknown provenance, we performed an empirical study. Using a mirrored version of the Maven 2 repository as our corpus, and 84 jars from a proprietary e-commerce java application as our subject set, our objective was to answer the following research questions:

RQ1: How useful is the archive signature similarity index at finding the original binary archive for a given binary archive?

RQ2: How useful is the archive signature similarity index at finding the original sources for a given binary archive?

RQ2: How reliable is the version information stored in a jar file's name?

We downloaded the complete Maven 2 central repository (between June 12th and 15th, 2010) using the following command:

```
rsync -v -t -l -r mirrors.ibiblio.org::maven2 .
```

Thus we obtained over 150G of jars, zips, tarballs, and other files. First we decompressed all tar-related archives to disk (.tgz, .tar.gz, .tar.bz2, etc.), including tars inside tars. Zip-related archives, including jar files, were processed in memory, including zips inside zips. We were surprised by the number of times an archive is included in another one; for example over 75,000 class signatures came from archives

Archives within archives	Level
jasmine-assemblies-deployment-1.1.2-bin.zip	1
bundles	
org.ow2.jasmine.jade.legacy.jonas4-4.8.6.jar	2
jonas4.8.4-tomcat5.5.17.zip	3
lib	
client.jar	4
org	
jacorb	
ir	
gui	
typesystem	
remote	
IRSequence.class	

Figure 4: Example of how archives are included in other archives. In this case, it results in 4 different levels of inclusion.

nested 4-levels deep. Figure 4 shows an example of this deep nesting.

There were a total of 130,000 binary jars[3]. Of them 75,000 were unique. We processed a total of 27 million binary class files and 4 million source Java files (including many duplicates). We were surprised by the disproportion between the number of binary and source files (6 times more). We will revisit this issue in the next section.

For RQ1, for each of the 84 e-commerce jars, we computed their similarity index against every binary archive in the corpus, and selected the set of matches with the highest similarity as the binary archive match.

For RQ2, the same procedure is performed as in RQ1, but instead the similarity index is computed against every source archive in the corpus.

For RQ3, we manually extracted the version information from the 84 e-commerce jars, and we determined whether the version information was correct or not by performing binary comparisons against jar files downloaded from project websites.

For RQ1 and RQ2 we classified a match into one of three categories:

Perfect. The set of matches included a version identical to the e-commerce subject jar.

Correct product. The set of matches included versions either precedent or subsequent of the same library as the e-commerce subject jar, but an identical version was not matched.

Incorrect. The set of matches was either the empty set (no matches found), or the matches included only libraries that were different than the e-commerce subject jar. Any matches in this case indicated a degree of cloning between the matched jar and the subject jar.

8. RESULTS

This section reports results of analyzing jar libraries from a proprietary e-commerce java application to answer the research questions formulated in Section 5. Data for replication is available on-line[4].

8.1 RQ1: How useful is the archive signature similarity index at finding the original binary archive for a given binary archive?

Similarity index	Type of match	Perfect	Correct product	Incorrect
1	Single	48	3	
	Multiple	19	1	
	Subtotal	67	4	0
> 0 & < 1	Single	1	9	2
	Multiple			
	Subtotal	1	9	2
0	No match			1
Total		68	13	3

Table 3: Using a binary-to-binary bertillonage technique to determine the provenance of 84 open source binary archives in a proprietary e-commerce application.

8.1.1 Single match, Similarity = 1

For 51 of the 84 binary jars (60.7%), our method correctly found a single candidate from the corpus with a similarity index of 1.0. This represents the best possible case for our anchored signature approach: the search space was narrowed such that additional metrics to further narrow the results were unnecessary. Of these 51 jars, 48 were perfect matches, and 3 were correct-product matches.

Subsequent analysis for each of these 3 correct product-matches revealed the e-commerce application was using library versions missing from the corpus's collection. A better corpus would improve our results here, giving us perfect matches instead of merely correct-product matches. Unfortunately, two scenarios show that some jar versions will probably never be found in any corpus:

1. The application developers may choose to use an experimental or "pre-released" version of a library that is unlikely to appear in any formal corpus. We observed one example of this in our study (stax-ex-1.2-SNAPSHOT.jar).

2. Developers may download libraries directly from an open source project's version control system, for example, should they require a bleeding edge feature or a particularly urgent fix. In these cases the jar is built directly from the VCS instead of from an official released version.

Fortunately, the matches were close in version to the correct (missing) candidates. As shown in Table 4, the three matched jars were close to the actual versions.

8.1.2 Multiple match, Similarity = 1

For 20 of the 84 binary jars (23.8%), our method found several candidates in the corpus with similarity of 1.0. In all cases the candidate set covered a contiguous sequence of versions, as shown in Table 5, save for holes in the corpus's

[3]Our definition of binary archive is a jar file that contains at least .class file.

[4]http://juliusdavies.ca/uvic/jarchive/

Classification	Library in question	Top match	Similarity	Inclusion
Perfect match	commons-digester-1.5.jar	commons-digester-1.5.jar	0.100	0.182
Correct product matches	commons-http.jar	commons-http-1.1.jar	0.333	0.500
	commons-ssl.jar	not-yet-commons-ssl-0.3.11.jar	0.090	0.333
	jax-qname.jar	j2ee-1.4.jar	0.002	1.000
	jaxws-rt.jar	jaxws-rt-2.1.3.jar	0.898	0.952
	jsse.jar	j2ee-1.3.1.jar	0.026	0.265
	namespace.jar	stax-api-1.0.1.jar	0.143	1.000
	parser.jar	xml-apis-2.4.D1.jar	0.060	0.104
	sjsxp-1.0-04.jar	sjsxp-1.0.jar	0.915	0.956
	soap-2.1.jar	soap-2.2.jar	0.625	0.800
Incorrect match	jcert.jar	secureftp.jar	0.015	1.000
	vreports.jar	itext-0.99.jar	0.253	0.341

Table 2: 12 of the 85 jars (14.1%) resulted in low-confidence matches, with similarity scores between 0.000 and 0.999. The inclusion score represents the percentage of the library in question found inside the top match. A low similarity index coupled with a high inclusion ratio can indicate cloning.

Correct jar (not in corpus)	Sim.	Close match
jaxws-api-2.1.3.jar	1.0	jaxws-api-2.1.jar
stax-ex-1.2-SNAPSHOT.jar	1.0	stax-ex-1.2.jar
streambuffer-0.5.jar	1.0	streambuffer-0.7.jar

Table 4: Three matches with similarity=1 were close in version to the correct (missing) jars.

collection. Of these 20 multiple matches, the perfect match was present for 19 cases. The remaining case, xsdlib.jar, we classified it as a correct-product match, (since the matched jars, xsdlib-1.5.jar and xsdlib-20050614.jar, came from the correct open source project), but as an incorrect version. The correct (missing) version, xsdlib-20040524.jar, was not present in the corpus.

Similarity to asm-attrs-2.2.3.jar	Artifacts from corpus
1.0	asm-attrs-2.1.jar
1.0	asm-attrs-2.2.jar
1.0	asm-attrs-2.2.1.jar
1.0	asm-attrs-2.2.3.jar

Table 5: An example of multiple matches with similarity=1. The perfect match, asm-attrs-2.2.3.jar, is present.

For some of the jars the resulting candidate set was small (2 or 3 candidates) such that a little manual work would likely produce the correct version from the corpus. But in other cases over 30 candidates were returned; in these cases additional bertillonage metrics would be advantageous.

8.1.3 Single match, similarity between 0 and 1

For 12 of the 84 binary jars (14.3%), our method found matches, but none had perfect 1.0 similarity. The three categories of non-perfect similarity matches are listed in Table 2.

8.1.4 No information

One of the 84 jars was not present in our corpus, and so no information could be found. We verified that the jar was an open source library by locating its project website (in sourceforge.net), but for reasons unknown to us the Maven2 repository has never republished this particular library.

To answer RQ1, the archive signature similarity index is highly useful for finding original binary archives. We found correct-product or perfect binary matches for 81 of the 84 binary jars in our sample set (96.4%).

8.2 RQ2: How useful is the archive signature similarity index at finding the original sources for a given binary archive?

Similarity index	Type of match	Perfect	Correct product	Incorrect
1	Single	13	2	
	Multiple	6	1	
	Subtotal	19	3	0
> 0 & < 1	Single	21	18	1
	Multiple	4	2	
	Subtotal	25	20	1
0	No match			16
Total		34	23	17

Table 6: Using a binary-to-source bertillonage technique to determine the provenance of 84 open source binary archives in a proprietary e-commerce application.

Our results for binary-to-source matching were similar in character to RQ1's binary-to-binary results, as shown in Table 6, although generally inferior across the board.

1. Similarity=1 occurred for only 22 cases (26.2%) as opposed to 71 cases (84.5%) for RQ1.
2. Binary-to-source matching found half as many perfect matches (34 compared to 68), and 75% more of the less desirable correct-product matches (23 compared to 13).
3. In addition, 16 jars could not be matched with any sources. This compares with only 1 jar finding no binary matches for RQ1.

We suspect two factors are contributing to the inferior performance. First, our corpus contains only 4 million java

source files compared to almost 27 million compiled class files. This results in many fewer source archives available for matching. For example, batik-util-1.6.jar matched no source archives, and yet for RQ1 the same jar file matched 15 distinct binary archives, ranging from similarity 1.000 down to 0.006, with zip timestamps between December, 2001 and June, 2008.

Second, binary-to-binary matching made use of identical programming logic both in the construction of the index and the construction of the similarity query. Source-to-binary matching required a separate logic in the construction of the index. While we intended that our source indexing would result in identical signatures, we observed this was not always the case.

To answer RQ2, the archive signature similarity index is useful the majority of the time at finding original source archives. We found correct-product or perfect source matches for 57 of the 84 binary jars in our sample set (67.9%).

8.3 RQ3: How reliable is the version information stored in a jar file's name?

We observed 9 cases, listed in Table 7, where the version information stored in the jar name was either missing or incorrect. Through manual analysis we verified the correct name, by downloading various versions from the library's original project website, and performing binary comparisons.

Observed jar name	Correct jar name
Incorrect in application	
jta.jar	jta-1.0.1B.jar
jtidy.jar	jtidy-4aug2000r7-dev.jar
soap.jar	soap-2.1.jar
stax-ex.jar	stax-ex-1.2.jar
streambuffer.jar	streambuffer-0.5.jar
xml-resolver-2.6.2.jar	xml-resolver-1.1.jar
xsdlib.jar	xsdlib-20040524.jar
Incorrect in corpus	
jnet-3.2.1.jar	jnet-1.0.3-03.jar
sjsxp-1.0.jar	sjsxp-1.0-04.jar

Table 7: Incorrect version information in jar names.

To answer RQ3, the version information stored in the jar name was sometimes unreliable. In the sample set of 84 jars only 77 were correct (91.7%), a suprisingly low result considering the importance of this information. We also observed incorrect version information in our corpus.

9. DISCUSSION

In our study we observed several interesting facts regarding the Maven 2 central repository:

- The Maven 2 repository contains significantly more binary archives than source code. Maven 2 contained 6 times more binaries than source code. It will be interesting to understand the reasons why this is the case.
- Similarly, Maven 2 contains some binaries but no corresponding source code.
- In some cases, some versions of specific packages are not present in Maven 2.
- Some versioned binaries in the Maven repository are incorrectly labelled. We do not know the reason, but this could pose a potential problem for Maven's goal of dependency management.

10. THREATS TO VALIDITY

This section discusses the main threats to validity that can affect the study we performed.

In particular, threats to *construct validity* may concern imprecision in the measurements we performed. Our logic for detecting java and class files in the Maven 2 repository relied on accurate detection of .java and .class files, as well as .jar, .zip, .tar.gz, .tar.bz2, and .tgz archives. No other search patterns were employed, and thus some archives may have been missed. This threat is diminished thanks to the very large amount of data we managed to extract from just those seven search patterns.

Our subsequent logic for extracting the class signatures could be faulty, in particular our java source parser. We are less concerned about faults in our bytecode analysis, since the bcel-5.2.jar tool we used is 4 years old, very popular, and very well tested. Bearing in mind that our java source parser is potentially a problem, we believe our results nonetheless resemble exactly the shape one would expect for a class-signature-index approach, with matches resembling a bell curve that drops off as version-numbers diverge from the exact match. In addition, queries involving only bytecode (e.g. queries for bytecode using bytecode) resulted in a similar bell curve, alleviating concerns over our source parser.

Threats to *internal validity* arise primarily from our technique for verifying a correct match: we visually check the version number in the names of jars and zip files. RQ1 in particular makes the threat clear: are version numbers in jar files at all reliable? According to current tradition in software development the ultimate authority on version is the tag in the version control system (VCS). We did not try to address this weakness empirically by comparing tagged VCS code against published jars and zips, and such a study would be highly valued here. Instead we hypothesized that developers and software engineers involved in the creation and/or packaging of open source libraries for the Maven 2 repository strive to publish correct version information, since the Maven 2 dependency management system (as well as others) relies on such for its operation.

Threats to *external validity* concern the generalization of our results. The small sample size for RQ3, 35 jars, of which 11 were invalid (contained no bytecode), is a serious threat to our study. We hope to alleviate this in future work, and our sample of 250 random jars in RQ2 is a first step in this direction. Another threat to our external validity comes from Maven's own composition: is Maven's repository a good sample of open source software in the java ecosystem? Given its critical position in industry with respect to java dependency resolution (even unrelated dependency resolvers such as Ivy use the Maven 2 repository), we believe it is representative. We have one complaint about its composition: it contains too many alpha, beta, milestone, and release-candidate artifacts that are likely of little interest to integrators. In future studies we may consider filtering these out. We control some of the over-reprentation by enforcing a

unique index in our database for each java/class file. In this way many popular jars such as ow2-util-scan-impl-1.0.18.jar appear only once in our database, despite 113 occurences scattered throughout the Maven 2 repository.

11. CONCLUSION

To conclude, we have introduced the notion of software Bertillonage to refer to a method that uses software metrics in order to narrow the search space when one is looking for a match of a software entity within a corpus. As an example of software Bertillonage, we built a method to identify the source code of binary Java archives, and successfully demonstrated with an empirical study in which we searched for the correct product and version of a set of Java binary archives that compose an proprietary e-commerce application against a corpus created from Maven 2, a repository of tens of thousands of binary and source Java archives.

Among our results we found that the name of a binary jar will likely point to the project from where it comes from, but that version numbers are sometimes incorrect (both in the Maven 2 and in the subject of our study). We also discovered that Maven 2 contains many more binaries than sources (and in some cases, entire projects have sources absent from Maven 2).

We believe that until accurate software methods are developed to match exactly one binary to its source, software Bertillonage can be a useful method to reduce the search space, the way its human counterpart was effective until fingerprinting was developed.

12. REFERENCES

[1] "Payment Card Industry (PCI) Data Security Standard, Version 1.2.1," https://www.pcisecuritystandards.org/security_standards/pci_dss_download.html, July 2009.

[2] J. A. Siegel, P. J. Saukko, and G. C. Knupfer, *Encyclopedia of Forensic Sciences*. Academic Press, 2000.

[3] M. M. Houck and J. A. Siegel, *Fundamentals of Forensic Science*. Academic Press, 2006.

[4] M. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, Feb. 2005.

[5] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 170–178.

[6] ——, "Is cloned code more stable than non-cloned code?" in *SCAM 08: Proceedings of the Working Conference on Source Code Analysis and Manipulation*, 2008, pp. 57–66.

[7] A. Lozano, "A methodology to assess the impact of source code flaws in changeability and its application to clones," in *ICSM 08: Proceedings of the International Conference of Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 424–427.

[8] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007, p. 18.

[9] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "How clones are maintained: an empirical study," *Emp. Soft. Engineering*, 2009 (to appear).

[10] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *ESEC/FSE*, vol. 30, no. 5, pp. 187–196, 2005.

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

[13] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *ICSE*. IEEE Computer Society, 2007, pp. 106–115.

[14] D. M. Germán, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *MSR*, M. W. Godfrey and J. Whitehead, Eds. IEE, 2009, pp. 81–90.

[15] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Trans. Software Eng.*, vol. 32, no. 12, pp. 952–970, 2006.

[16] M. Di Penta, D. M. Germán, and G. Antoniol, "Identifying licensing of jar archives using a code-search approach," in *Proceedings of the 7th Intl. Working Conference on Mining Software Repositories (MSR-2010)*, 2010, pp. 151–160.

[17] A. Hemel, "The GPL Compliance Engineering Guide version 3.5," http://www.loohuis-consulting.nl/downloads/compliance-manual.pdf.