Software Bertillonage: Finding the Provenance of an Entity

Julius Davies†, Daniel M. German†, Michael W. Godfrey‡, Abram Hindle*
† Department of Computer Science, University of Victoria, Canada
‡ David R. Cheriton School of Computer Science, University of Waterloo, Canada
* Department of Computer Science, University of California, Davis, USA
juliusd@uvic.ca, dmg@uvic.ca, migod@uwaterloo.ca, abram@softwareprocess.es

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 — such as external libraries or cloned source code — is not clearly stated, and this uncertainty can introduce technical and ethical concerns that make it difficult for system owners and other stakeholders to manage their software assets. In this work, we motivate the need for the recovery of the provenance of software entities by a broad set of techniques that could include signature matching, source code fact extraction, software clone detection, call flow graph matching, string matching, historical analyses, and other techniques. We liken our provenance goals to that of Bertillonage, a simple and approximate forensic analysis technique based on bio-metrics that was developed in 19^{th} century France before the advent of fingerprints. As an example, we have developed a fast, simple, and approximate technique called *anchored signature matching* for identifying library version information within a given Java application. This technique involves a type of structured signature matching performed against a database of candidates drawn from the Maven2 repository, a 150GB collection of open source Java libraries. An exploratory case study using a proprietary e-commerce Java application illustrates that the approach is both feasible and effective.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Management, Measurement

Keywords

Bertillonage, provenance, code evolution, code fingerprints

MSR'11, May 21–22, 2011, Waikiki, Honolulu, HI, USA

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.

However, if software components are included without clearly identifying their origin then a number of technical and ethical concerns may arise. Technically, it is hard to maintain such a system if its dependencies are not well documented; for example, if a new version of a library is released that contain security fixes, system administrators will want to know if their existing applications are vulnerable. 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; when open source code is discovered within a proprietary system, it can be costly and embarrassing to the company.

Many North American financial instutions implement the Payment Card Industry Data Security Standard [1] (PCI DSS). 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 financial institution is found to contain a dependency on a Java archive named `foo.jar`. Ensuring that the PCI DSS requirement is satisfied entails addressing some difficult questions:

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

We can use a variety of techniques for this. For example, if we have access to the source code we can do software clone detection. If we have access to binaries, we can perform 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

1. We introduce the general concept of software Bertillonage: a method to reduce the search space when trying to locate a software entity within a large corpus of possibilities.
2. We present an example technique of software Bertillonage: anchored signature matching. This method aids in reducing the search space when trying to determine the identity and version of a given Java archive within a large authoritative corpus of archives, such as the Maven repository.
3. We demonstrate the effectiveness of our method with a case study that involves finding exact version information of binary jars used in a real world e-commerce application.

1.2 Bertillonage and Software Provenance

In the mid to late 19th 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 who worked as a clerk for the Paris police, 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, later 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 photography — 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].

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

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 entity come from?* Of course, most often the answer will be that the entity in question was created to fit exactly where it is within the greater 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, etc. Sometimes, however, our answers will be best effort guesses, 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 identification. 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.

1.3 Replication

Data for replication is available at:
`http://juliusdavies.ca/2011/msr/bertillonage/`

2. 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 in our work 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]. Others have concentrated on their lifespan and genealogy [10]. The distinction between these studies and our own is that we study provenance across applications, and are not only interested in finding similar entities, but where they come from. We are also interested in matching similar entities when one of them is in compiled (binary) form.

Clone detection methods (such as [11, 12]), as well as the tracking of clones between applications [13] provided a starting point for our investigation. Similar to Holmes et al. [14] we build our own code-search index.

Di Penta et al. [15] 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 consider their work a simple method of Bertillonage.

3. A FRAMEWORK FOR SOFTWARE

The goal of software Bertillonage is to provide computationally inexpensive techniques to narrow the search space when trying to determine the provenance of a software entity. More formally, we define a 'subject' as the entity whose provenance we are investigating. We define 'candidates' as a set of entities from a given corpus that are likely matches to the subject. A desirable property of Bertillonage is thus to provide, for any subject, a relatively small set of high-likelihood candidates.

We use the metaphor of Bertillonage — an approximate approach fraught with errors — rather than a more precise forensic metaphor of fingerprinting or DNA analysis to emphasize that while we may have a lot of evidence, often we do not have authoritative answers. For example, one of the problems we examine involves trying to match a compiled binary against a large set of candidate source files. If we know the exact details of the creation of the binary — the version of the compiler, the compilation options used, the exact set of libraries used for linking, etc. — then we can compile our source candidates accordingly and use simple byte-to-byte comparison. But in reality the candidate binaries are often compiled under varying conditions, and this can result in two binary artifacts that have the same provenance yet are not byte-for-byte equivalent in their binary representations.

It may also be the case that "the suspect is not on file", i.e., that there may be no correct match for the subject within the corpus. In our example of anchored signature matching, we compare Java archives from subject software systems against the Maven2 repository. However, Maven2 is not a comprehensive list of all possible versions of all possible Java libraries; in consists only of those library versions that someone has explicitly contributed. So our subject archive may not be present within the corpus in any form (which is likely to be easy to determine), or the archive may be present but not the particular version that we seek. Consequently, we must always be willing to consider the possibility that what we are looking for is not actually there.

Thus, instead of precision we take as our goal of software Bertillonage the narrowing of a large search space. We seek to prune away the low probability candidates leaving a relatively small set of likely suspects, against which we may choose to apply more expensive techniques, such as clone detection, compilation, or manual examination. We realize that establishing provenance may take some effort, and that it may not even be possible in a given situation.

3.1 Bertillonage Metrics

As with forensic Bertillonage, it is necessary to define a set of metrics that can be measured in a potential subject and 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, as 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, usually by considering different kinds of its sub-parts, which we will call "objects of interest" (OOIs). These measurements can be performed in various ways:

Count-based: count the *number* of OOIs that the subject contains, such as number of calls to external libraries, or uses of an obscure feature (e.g., How many times is *setjmp*, *longjmp* used);

Set-based: compute a *set* of OOIs that the entity contains, such as the string literals defined by this entity[2] or set of classes defined in a package;

Sequence-based: compute a *sequence* of OOIs in the entity (i.e., preserving the order), such as the sequence of methods signatures of a class, the sequence of calls within a method, the sequence of tokens types, etc.;

Relationship-based: consider external OOIs that the subject is *related to* in some way; for example, what are the dynamic libraries used by this program?

The dimensionality of possible software Bertillonage metrics also includes the *granularity* (code snippet, function / method, class / file, package / namespace), *artifact kinds* (source code, binary, structured text, natural language), and the *programming language* (C, C++, Java). 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.

4. METHOD

To exemplify the concept of software Bertillonage, we propose a metric that addresses the problem: if we are given a Java binary archive, can we determine its original source code? The most obvious source of information is the name of the archive itself, i.e., one would expect that `commons-codec-1.1.jar` comes from `commons-codec`, an Apache project, release 1.1.[3] However, in practice this does not always work: some projects do not adhere to consistent naming and numbering policies, sometimes beta tags are removed from version identifiers, and sometimes version identifiers are removed altogether when the library sources are copied into the source tree of the application.

Alternatively, 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

[2] *The GPL Compliance Engineering Guide* recommends the extraction of literal strings to determine potential violations [16].

[3] This is analogous to a policeman asking a suspect for her/his name and expecting a correct answer.

```java
package a.b;
import g.h.*;

/**
 * @author Jane Doe
 * @since January 1, 2001
 */
public class D
implements I<Number> {

  synchronized static int a(
    String s
  ) throws E {
    return "abc".hashCode() - s.hashCode();
  }
}
```

Figure 1 – Source code of a class D.

```java
package a.b;

public class D extends java.lang.Object
implements g.h.I {

  public D() {
    // Empty constructor added by javac;
    // all classes need constructors.
  }

  synchronized static int a(
    java.lang.String s
  ) throws a.b.E {
    /* [compiled byte code] */
  }
}
```

Figure 2 – Decompiled version of a class D.

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 jikes 1.22), debugging symbols (on or off), and different optimization levels. Furthermore, library dependencies can be difficult to satisfy (especially for older artifacts) making full compilation a problem. Even without compiler variation, avenues for false negatives remain; for example, the build scripts themselves might inject information at build-time directly into class files.

The philosophy we propose, software Bertillonage, requires us to seek characteristics that are easy to measure and compare such that, even if they do not guarantee an exact match, they will significantly reduce the search space. 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: 1. inheritance tree, 2. implemented interfaces, 3. constructors, 4. annotations, 5. method names, their return types, and their parameters (names and types), 6. class, method, and constructor visibility, 7. some class and method modifiers (i.e., abstract), and 8. relative position of methods in the class. Many other features are lost during compilation, including comments, import statements, parameter modifiers (such as `final`), and absolute position of methods, since line numbers are preserved only when the class is compiled with debug info.

In a nutshell, we propose a Bertillonage metric for binary Java archives that can be used to match a binary class file to its likely source file. Not all of the source code classes may be included in the ultimate binary; for example, test classes are often excluded, and sometimes a source archive may be 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.

4.1 Anchored Class Signatures

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* since it includes the fully qualified name of the Java file, and in this way our signature preserves attributes used by Java's own built-in name resolution mechanism (i.e., the `CLASSPATH`). We note, however, that developers copy and pasting (cloning) complete classes into their own application sometimes alter the namespace declaration of the original class, in essence relocating the copied logic into a new namespace. Our *anchored* approach will be unable to find matches in these cases, but our results should also possess less noise; for example, very small single-constructor exception-handling classes that happen to be coincidentally named will not pollute our results.

When building the signature, all fully qualified *parameter* types (including `throws` clauses) in the decompiled bytecode are stripped of their package prefixes; for example, `g.h.I` becomes `I` and `java.lang.String` becomes `String`. This is done because identifying the fully qualified names from source depends on Java's `import` mechanism, which is indeterminate. Fully qualified names that are referenced directly in source — although 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` (Fig. 1) and its corresponding decompiled bytecode (Fig. 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 constructor. Class D's signature (Fig. 3) 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()`.

$\sigma(D)$ = `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 – Anchored class signature for `D.java` & `D.class`.

4.2 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 signatures of the classes within them. Formally, given an archive A composed of n classes $A = \{c_1, ..., c_n\}$, we define the signature of an archive as the set of signatures of its contained classes.

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

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

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

Ideally, a binary archive b would have 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 often excluded from 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 complete copy of source code from the original system will, due to the increased size, also have a low similarity index. To address these issues we define the concept of inclusion index.

4.3 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.

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

The intuition here is that when the inclusion index of a binary archive A in archive B is close to 1, then the classes in A are present in B.

4.4 Finding Candidate Matches

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 which 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. In the ideal case, L has only one member. In practice, however, the corpus often has several candidate matches with equal maximum similarity scores. We have found several reasons for multiple archives having the same maximal score: there may be identical redundant archive copies in Maven2; some archives differ only in documentation or other non-code attributes; some non-identical archives may simply achieve equal scores; and the signature of an archive may remain constant across multiple versions if there are implementation changes but no interface changes. This last case is typical in minor release updates.

5. IMPLEMENTATION

5.1 Building a Corpus

To be effective, any approach that implements the Bertillonage philosophy requires a corpus that is as comprehensive as possible. For Java, the Maven2 Central Repository[4] fulfills this requirement. Maven2 provides a large public repository of reusable Java components and libraries under various open source licenses, often including multiple versions of each component; it serves as the Java development community's de facto library archive. Originally, the repository was developed as a place from where the Maven build system could download required libraries to build and compile an application. Because of the repository's broad coverage and depth, even competing dependency resolvers make use of it (i.e., http://ant.apache.org/ivy/).

Maven2, as a whole, is unversioned: today's Maven2 collection will be different from tomorrow's, as there is a continual accumulation of artifacts. This is unlike the major GNU/Linux compilations of free and open source software such as Debian, where Debian 5.0 is a fixed collection after its official release date.

5.2 Extracting the Class Signatures

We developed two tools to extract anchored class signatures from Java archives: a handcrafted parser for analyzing source code, and a byte code analyzer based on the `bcel5` library. We now briefly describe them.

5.2.1 Extracting a Class Signature From Source

When analyzing a source file we first discard comments, import statements, parameter names, and parameter modifiers, since such attributes of source files are not reliably preserved during compilation. We also discard generics, annotations, and inner classes, since our Java parser cannot yet process these attributes correctly. Once this is done, we then extract the class signature in this way:

1. *Extract the package and class line.* We must re-introduce the default 'extends Object' declaration as part of our signature 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.
2. *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.
3. *Extract methods preserving 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 currently ignore inner-classes, abstract methods, and native methods.

5.2.2 Extracting a Class Signature From Bytecode

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

[4]http://repo1.maven.org/maven2/

cewolf-1.0.jar $\|A\|$	$\|B\|$	$\|A \bigcap B\|$	$\|A \bigcup B\|$	$sim(A,B)$	$inclusion(A,B)$	path for each B
77	77	77	77	1.000	1.000	maven2/cewolf/cewolf/1.0/cewolf-1.0.jar
77	77	68	86	0.791	0.883	maven2/cewolf/cewolf/0.12.0/cewolf-0.12.0.jar
77	77	64	90	0.711	0.831	maven2/cewolf/cewolf/0.10.3/cewolf-0.10.3.jar

Table 1 – Three candidate jars are found after we query the corpus using 'cewolf-1.0.jar'.

from before. Creating the anchored class signature here involves three activities:

1. *Extract the package and class line.* As before we assemble a package and class line; here, we must shorten the fully-qualified names that `bcel5` extracts.
2. *Extract methods, preserving order.* We shorten the fully-qualified names among the method parameter 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 ignore them.
3. *Remove methods introduced to implement non-generic interfaces.* Classes sometimes contain additional methods added by the Java compiler to satisfy non-generic implementations of genericized interfaces (for backwards compatibility). We removed such methods, since they cause otherwise perfectly matching signatures to diverge.

When this process completes for both of our two examples, `D.java`, and `D.class`, we should possess a class signature identical to Figure 3.

5.3 Matching a Subject Artifact to Candidates

The source and bytecode tools we developed to extract the signatures are employed both in the construction of a corpus index, as well as the generation of queries to find matching candidates. The two phases are described below.

Building the Corpus Index: we scan every source and binary archive within the Maven2 repository, including archives within archives. For each source and compiled class file we compute its signature using the steps described in section 5.2. To improve response time for finding matches, we index each signature using its SHA1 hash.

Finding Matches: we are interested in finding what archives have matching classes with the subject, and what these classes are. To perform this step efficiently we use the following algorithm:

1. For each class present in the subject, find its matching classes (with identical class signature) in the corpus.
2. Group the union of all matching classes (for all the classes in the subject) by their corresponding archive. This will result in a list of all archives that have at least one matching class with the subject, and for each archive, the list of matching classes with the subject.

At this point we can now compute the similarity and inclusion indexes of the subject archive, with each of the archives that have at least one matching class. Table 1 shows an example where a subject artifact (`cewolf-1.0.jar`) is matched to candidate artifacts within the corpus.

Note that, even in an exact 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.

6. EVALUATION

In a related research project we had to perform a license and security audit of a real world e-commerce application comprised of 84 open source libraries. The audits had to be performed against both the binary and source code forms of these included libraries. Before we could conduct the audits, we needed to determine the provenance of all included libraries.

Accurate and precise provenance information forms an important foundation for many types of higher-level analyses. Such analyses include, among others, license audits, security vulnerability scans, and patch-level assessments (as required by the PCI DSS security standard). A license audit of software dependencies must reflect the reality that software licenses sometimes evolve (change between releases). Similarily, known security holes in libraries will affect specific releases or version ranges. The PCI DSS requirement #6, "All critical systems must have the most recently released, appropriate software patches," cannot be satisfied without knowledge of the existing patch versions. In this vein we believed that conducting a license audit and a security audit would provide real value to the developers of the e-commerce application, while also providing us with a chance to test our Bertillonage approach in the field. We applied our technique in two different modes, binary-to-binary, and source-to-binary, and we define a research question for each of these modes:

RQ1: **How useful is the similarity index for narrowing the search space to find an original *binary* archive when provided a subject binary archive?**

RQ2: **How useful is the similarity index for narrowing the search space to find an original *source* archive when provided a subject binary archive?**

6.1 Setting

We downloaded the complete Maven2 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 nested 4-levels deep.

There were a total of 130,000 binary jars[5]. 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).

For RQ1, for each of the 84 e-commerce jars, we computed

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

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

Exact. 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), or the matches included only libraries that were different than the e-commerce subject jar.

6.2 Results

This section reports results of analyzing jar libraries from a proprietary e-commerce Java application to answer the research questions formulated in Section 4. The analysis was performed on a single Athlon 4850e 2.5ghz PC with 4GB or RAM running Debian 5.0.5 and PostgreSQL 8.4. The SQL queries we ran (2 queries per jar) required approximately 15 minutes in total to complete. Decompressing and extracting signatures to build our index of the 150GB of archives comprising the Maven2 repository required 48 hours. Downloading Maven2 over `rsync` was by far the longest phase of our experiment, requiring four days.

6.2.1 RQ1: Binary-to-Binary Matches

Similarity index	Type of match	Exact	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 2 – Using a binary-to-binary bertillonage technique to determine the provenance of 84 open source binary archives in a proprietary e-commerce application.

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 exact 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. 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.

The matches were close in version to the correct (missing) candidates, as shown in Table 3.

Correct jar (not in corpus)	Sim	Close match (from corpus)
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 3 – Three matches with similarity=1 were close in version to the correct (missing) jars.

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 4, save for holes in the corpus's collection. Of these 20 multiple matches, the exact 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 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 4 – Example of multiple matches with similarity=1. The exact match is asm-attrs-2.2.3.jar.

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. The median number of candidates was 4 and the average was 6.25.

Single Match, Similarity Between 0 and 1. For 12 of the 84 binary jars (14.3%), our method found matches, but none had 1.0 similarity. Among these 12 the median similarity score was 0.122 and the average was 0.288. In two of these cases, the e-commerce application was using development snapshots (not official releases). In another two cases the versions in Maven2 were mislabeled. Five cases were very old XML and Crypto libraries that predate Maven2. One case was due to binary cloning in a proprietary jar: `vreports.jar` contained several classes from other popular open source libraries. The final two matches in this category were packaged in ways that confused our technique. For example, `jaxb-xjc-2.1.6.jar` included several identical class files in separate locations inside its archive. The

other example, `commons-digester-1.5.jar`, had the order of its methods permuted, but was otherwise identical to the copies of `commons-digester-1.5.jar` in the corpus.

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 does not include this particular library.

To answer RQ1, the similarity index is highly useful at narrowing the search space to find original *binary* archives. We found correct-product or exact binary matches for 81 of the 84 binary jars in our sample set (96.4%).

6.2.2 RQ2: Binary-to-Source Matches

Similarity index	Type of match	Exact	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 5 – Using a binary-to-source bertillonage technique to determine the provenance of 84 OSS 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 5, although slightly 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 exact matches (34 compared to 68), and 75% more of the 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 1.0 down to 0.006, with zip timestamps between Dec.'01 and June '08.

Second, binary-to-binary matching made use of a single tool both in the construction of the index and the construction of the similarity query. Source-to-binary matching required a separate tool for the construction of the source index. While we saw that our source tool constructed identical signatures over 99% of the time, there were instances where the binary tool constructed a different signature, despite the binary being known to originate from the source.

To answer RQ2, the similarity index is useful the majority of the time to narrow the search space to find original *source* archives. We found correct-product or exact source matches for 57 of the 84 binary jars (67.9%).

6.3 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 Maven2 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. We only conducted a thorough byte-by-byte comparison in cases where the initial visual check failed.

Threats to *external validity* concern the generalization of our results. We provide a single case study involving a proprietary enterprise application, and us such, our study shows feasibility rather than generalizability. 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 eco-system? Given its critical position in industry with respect to Java dependency resolution (even unrelated dependency resolvers such as Ivy use the Maven2 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.

7. DISCUSSION

What is provenance? Is *name* and *release number* alone a suitable representation of provenance for our purposes? Suppose a given jar is authoritatively known to be named *foo* and to be release *x.y.z*. Our method assigned the highest similarity score to this single candidate, *foo-x.y.z.jar*, for over 60% of the subject jars in our case study. But can provenance really be boiled down to a small sequence of characters, hyphens, digits and dots. Does *foo-1.2.3* constitute provenance? This question is important, since our technique assumes it.

Fortunately, for the majority of the jars in this study, and perhaps for the majority in "current circulation" among Java developers, this notion of provenance is sufficient. As a thought experiement, imagine asking random undergraduate students enrolled in Introduction to Computer Programming at any university to download the `oro-2.0.8.jar` Java library. In all likelihood the vast majority would download

the same artifact, even those completely unfamiliar with Java. The universe of Java developers manages to avoid name and version collisions among their reusable libraries.

However, for some jars, this notion of provenance is insufficient. The underlying assumption with respect to *name* and *release number* is that the combination of these two attributes will always result in a distinct set of software code, an *authoritative* snapshot, frozen in time. Among the 84 jars studied, we observed three challenges to a *foo-1.2.3* notion of provenance:

1. Jars that, during their build process, copy classes from other jars. For example, `vreports.jar` contains copies of classes from `itext.jar`.

2. Jars with historically unstable provenance, perhaps due to corporate acquisitions, or even internal restructurings within a company. The Sun/Oracle jar named `xsdlib.jar` is an example of this. Various project websites provide conflicting testimony regarding the jar's origins. Each of these projects appears to have taken control of, or at least contributed to, `xsdlib.jar`'s development at some point in its history. The answer may very well be a combination of the projects we observed, which each project contributing to different phases of `xsdlib.jar`'s evolution. In cases such as these, our Bertillonage results can resemble a hall of mirrors. More expensive analysis methods, such as sending questions to project mailing lists, or analyzing version control repositories are required.

3. Altered Jars, e.g., a particular `foo-1.2.3.jar`, may contain 10 classes, whereas another jar with the same name and release information may contain only 9 classes. In some cases these 9 are a proper subset of the 10. Perhaps a user of the library has customized it by adding or removing a class. Which archive is authoritative in this case? We have examples of this in our data.

In the face of these challenges our Bertillonage approach was surprisingly fruitful. Our simple Bertillonage metric could readily accommodate #1 (emcompassed jars). Challenges #2 (unstable provenance) and #3 (altered jars) always required additional narrowing work, and yet our approach nonetheless still revealed when these particular challenges were occurring. Rather than reinforce our initially narrow notions of provenance, thanks to the simplicity of our metric, and particularly thanks to an immense (and messy) data source such as Maven2, our study outlined what future provenance research must tackle.

7.1 A Foundation for Higher Analyses

Developing, deploying and maintaining software systems can involve many diverse groups within (and external to) an organization. Each of these groups may require different knowledge about the software systems they are involved with. For example, testers, developers, system administrators, salespeople, managers, executives, auditors, owners, and other stakeholders may have specific questions they need answered about an organization's software assets. A salesperson may have a technically demanding client that insists on a specific release of a particular library. The security auditor wants to make sure no libraries or copy-pasted code fragments contain known security holes. The license auditor wants to know if her license requirements are being fulfilled. The manager wants to know how risky an upgrade to the latest release of a popular object-relational database mapping library might be. As noted in section 6, provenance forms a critical foundation upon which these higher level analyses rely. Without reliable provenance information in place these stakeholders cannot even begin to find answers to their questions.

Provenance information is also important to the software developers responsible for importing and integrating libraries and code fragments into their software systems. Therefore name and release information is often encoded directly into an artifact's file name (e.g., `oro-2.0.7.jar`). But sometimes developers may omit the release numbering, or they may mistype it. Also, as we noted earlier, in some cases an artifact internally encompasses additional artifacts, rendering the file name inadequate for communicating the versions of the encompassed releases. For these reasons, higher level analyses cannot depend on filename alone.

The specific metric we introduced here, anchored signature matching, will by no means be the final word in software Bertillonage. But we found our simple metric to be effective: it was able to supply useful provenance information for over 95% of the subject archives, including complex cases where an archive encompassed other archives. Of course some manual effort was required in our case study to narrow all matched candidates to single exact matches, but for the majority of these the original filename was correct, and so the manual effort was minimal. Our result minimizes the risk of relying on filenames exclusively when performing higher level analyses that depend on provenance. We also note the excellent binary-to-binary results we obtained can serve as a bridge to improved binary-to-source results: with a single binary match, manually locating the corresponding source archive (especially in the open source world) is trivial. This "bridging" idea mitigates the downside of our inferior binary-to-source results.

Our technique also performed well in a separate informal exercise to determine the moment of a copy-paste of class files. We noticed the developers of `httpclient.jar`, an open source Java library, had posed a question on their mailing list: when did Google Android developers copy-paste `httpclient.jar` classes into `android.jar`?[6] They wanted to know this to evaluate how hard it would be for Google to import a more recent version of their jar. We employed our technique to answer the original question on the mailing list, and the main developer confirmed our result. We initially identified *4.0-beta1* as the moment of the copy-paste. The developer asked if we could also test against *4_0_API_FREEZE*, an uncommon version he suspected Google had actually imported. We loaded the *FREEZE* release into our index and re-ran our analysis. This resulted in both *4.0-beta1* and *4_0_API_FREEZE* being returned as equally likely matches for `android.jar`.

We were successful in narrowing the search space for the moment of copy-paste two just two versions. In addition, the `httpclient.jar` exercise motivated future work. Precedent and subsequent releases diverge with respect to the cardinality of their intersecting signatures. Our anchored signature match is not just useful for finding exact matches. It could also prove useful at measuring the distance between

[6]See email from Bob Lee to dev@hc.apache.org on 18 Mar 2010 23:47:14 GMT, subject "Re: HttpClient in Android."

versions, which in turn could be useful for performing risk assessment of releases.

As stated earlier, we performed a license audit and security audit using the provenance information unearthed from the case study. The results of these higher analyses proved useful: the license audit pinpointed a jar where some versions used the GNU Affero license, while other versions used LGPL; similarily, the security audit located a jar with a known security hole. The organization found the results from both of these audits valuable, and steps were taken to address both issues in their application.

8. CONCLUSION AND FUTURE WORK

In this paper, we have discussed the problem of determining the provenance of a software entity. That is, given a library, file, function, or even snippet of code, we would like to be able to determine its origin: Was the entity designed to fit into the design of the system where it sits, or has it been borrowed or adapted from another entity elsewhere? We argued that determining software entity provenance can be both difficult and expensive, given that the candidate set may be large, there may be multiple or even no true matches, and that the entities may have evolved in the mean time. Consequently, we introduced the general idea of software Bertillonage: fast, approximate techniques for narrowing a large search space down to a tractable set of likely suspects.

As an example of software Bertillonage, we introduced *anchored signature matching*, a method to determine the provenance of source code contained within Java archives. We demonstrated the effectiveness of this simple and approximate technique by means of an exploratory case study performed on a proprietary e-commerce application using a corpus drawn from the Maven2 Java library repository. We found that we were able to reliably identify the correct product information of contained binary Java archives if the product was present in Maven, and such cases we were also usually able to identify the correct version. If a sought product was not present in Maven, this was usually quickly obvious. However, if a product was present we found that identifying the correct version was sometimes tricky, requiring detailed manual examination. The use of anchored signature matching proved to be very effective in eliminating superficially similar non-matches, providing a small result set of candidates that could be evaluated in detail.

Being able to determine the provenance of software entities is becoming increasingly important to software developers, IT managers, and the companies they work for. Given the wide ranging nature of the problem, the large candidate sets that must be examined, and the detailed amount of analysis required to verify matches, we feel that this is only the beginning of software Bertillonage. We need to design a wide array of techniques to narrow the search space quickly and accurately, so that we can then perform more expensive analyses on candidate sets of tractable size.

Acknowledgements

We thank Dr. Anton Chuvakin of Security Warrior Consulting (www.chuvakin.org) for his advice on PCI DSS.

9. REFERENCES

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

[2] J. Siegel, P. Saukko, and G. 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, "Is cloned code more stable than non-cloned code?" in *SCAM'08*, 2008, pp. 57–66.

[6] A. Lozano, "A methodology to assess the impact of source code flaws in changeability and its application to clones," in *ICSM 08: Proc. of the Int. Conf. of Software Maintenance*, 2008, pp. 424–427.

[7] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *MSR '07: Proc. of the 4th Int. Workshop on Mining Soft. Repositories*, 2007, p. 18.

[8] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Emp. Soft. Engineering*, vol. 15, no. 1, pp. 1–34, 2009.

[9] 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.

[10] 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.

[11] 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.

[12] 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*, 2007, pp. 106–115.

[13] 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 '09: Proc. of the Working Conf. on Mining Software Repositories*, 2009, pp. 81–90.

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

[15] M. Di Penta, D. M. Germán, and G. Antoniol, "Identifying licensing of jar archives using a code-search approach," in *MSR'10 Proc. of the Intl. Working Conf. on Mining Software Repositories*, 2010, pp. 151–160.

[16] A. Hemel, "The GPL Compliance Engineering Guide version 3.5," http://www.loohuis-consulting.nl/downloads/compliance-manual.pdf, 2010.