# Software Bertillonage
## Determining the Provenance of Software Development Artifacts

by

Julius William Musseau
B.F.A., University of Victoria, 2001
B.Sc., University of Victoria, 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Software Bertillonage
Determining the Provenance of Software Development Artifacts

by

Julius William Musseau
B.F.A., University of Victoria, 2001
B.Sc., University of Victoria, 2011

Supervisor

---

Dr. Daniel M. German, Professor
(Department of Computer Science)

**Supervisory Committee**

---

Dr. Daniel M. German, Professor
(Department of Computer Science)

## 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 the source origin of binary libraries 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 275GB collection of open source Java libraries. To show the approach is both valid and effective, we conduct an empirical study on 945 jars from the Debian GNU/Linux distribution, as well as an industrial case study on 81 jars from an e-commerce application.

# Contents

# List of Tables

# List of Figures

# ACKNOWLEDGEMENTS

I would like to thank:

**Everyone** for putting up with me.

*You need to finish your feces.*
Unknown

# DEDICATION

This work is dedicated to DMG because if you take all people on Earth (minus myself and DMG), and you count all the times they will collectively read this document, counting from now until the moment the Sun goes dark, DMG will still have read the document more times.

# Chapter 1

# Introduction

Deployed software systems often include code drawn from a variety of sources. While the bulk of a given software systems source code may have been developed by a relatively stable set of known developers, a portion 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 system regardless of the particular versions of library components that clients might or might not have already installed.

However, there currently exists no bullet-proof way to precisely and accurately re-establish the identity of software components, despite the importance of such information. Once a complete system is deployed, the literal filenames are all that typically remains to identify cobundled components. If were lucky we might also find some package.json dependency files lying around or some Maven pom.xml files tucked inside a few of the Jar files. But developers do weird things, hackers do nefarious things, and not everyone remembers to generate these files. Techniques are desired that allow developers and other stakeholder to establish provenance of artifacts regardless of any meta-data stored alongside the artifacts themselves, since such is not always trustworthy or reliable.

A naive solution might involve building an index of artifacts of known provenance. The index builder could download the component from its upstream source location, build the component in a trusted environment, and then store a SHA1 or MD5 fingerprint of the final component. This technique would immediately prove futile, since

components (such as Jar files) typically embed the timestamps of all compiled files based on the moment of compilation. In other words, the fingerprint would depend on the exact moment of compilation, and thus it would be useless as an index. A slightly less naive approach would index the compiled files themselves instead of the encompassing archive file. But here we still encounter significant challenges: there are many variations in environment that can result in slightly different compiled results. Choice of compiler, compiler options, optimization levels, even minor operating system differences (e.g., line endings) can all contribute to changes in the final compiled binary file, which in turn perturb any index. Since the range of these possible variations is always evolving with new compiler releases, new optimization techniques and so-on, pre-calculating a stable index of compiled artifacts is literally impossible.

## 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's origin within a 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 corpus of archives, such as the Maven 2 central repository.

3. We establish the validity of our method with an empirical study of 945 binary jars from the Debian 6.0 GNU/Linux distribution. We demonstrate the significance of our method by replicating a case study of a real world e-commerce application containing 81 binary jars.

# Appendix A

# Additional Information

This is a good place to put tables, lots of results, perhaps all the data compiled in the experiments. By avoiding putting all the results inside the chapters themselves, the whole thing may become much more readable and the various tables can be linked to appropriately.

The main purpose of an Appendix however should be to take care of the future readers and researchers. This implies listing all the housekeeping facts needed to continue the research. For example: where is the raw data stored? where is the software used? which version of which operating system or library or experimental equipment was used and where can it be accessed again?

Ask yourself: if you were given this thesis to read with the goal that you will be expanding the research presented here, what would you like to have as housekeeping information and what do you need? Be kind to the future graduate students and to your supervisor who will be the one stuck in the middle trying to find where all the stuff was left!

# Bibliography

[1] Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors. *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*. IEEE, 2011.