Determining the Provenance of Software Artifacts

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

ABSTRACT

Software clone detection has made substantial progress in the last 15 years, and software clone analysis is starting to provide real insight into how and why code clones are born, evolve, and sometimes die. In this position paper, we make the case that there is a more general problem lurking in the background: software artifact *provenance* analysis. We argue that determining the origin of software artifacts is an increasingly important problem with many dimensions. We call for simple and lightweight techniques that can be used to help narrow the search space, so that more expensive techniques — including manual examination — can be used effectively on a smaller candidate set. We predict the problem of software provenance will lead towards new avenues of research for the software clones community.

Categories and Subject Descriptors

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

General Terms

Management, Measurement

Keywords

Bertillonage, provenance, code evolution, code fingerprints

1. A VERY BRIEF HISTORY

Much of the work on software code clone detection over the last 15 years has concentrated on the problem of computing "similarity" based on lexical, syntactic, and (sometimes) semantic properties of the software artifacts under consideration. If two artifacts are within an acceptable similarity threshold, they are then considered to be clones of each other.[1] That is, we measure similarity, and if we are impressed with the results, we infer that an act of cloning must have taken place at some point in the past. Of course, the reason we measure similarity instead of tracking history is that we typically do not have access to edit logs of the developers[2]. If we did, then we could settle these debates of origin quickly and authoritatively.

Initially, most authors of clone detection papers cited maintenance problems as the rationale behind performing clone detection: if your codebase had clones, then your developers were lazy, and your code was bloated and inconsistently maintained. As Kent Beck and Martin Fowler say in their chapter, "Bad Smells in Code," from *Refactoring* [4]:

> Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

Recently, a more nuanced view of cloning as a principled design practice is emerging [2, 7]; that is, code cloning can be seen as a kind of engineering tool, with advantages and disadvantages depending on the situation and use. Clone detection is typically an expensive and tedious process requiring manual intervention and expertise [6]. Consequently, the rationale behind performing clone detection is now commonly cited as *program comprehension*. We no longer go after clones as part of a search-and-destroy mission; rather, we try to understand the context for performing cloning in the first place to better comprehend the original design rationale of the developers [5].

2. THE PROBLEM OF PROVENANCE

The term *provenance* traditionally refers to the documented history of a work of art, which can be used as a guide toward the work's authenticity. In the previous decade the term has been used in a digital context to refer to data objects (e.g., files and database records), and the research questions have revolved around software security, data traceability, and IT management [1]. We see a natural application of this term within the realm of software development. That is, we sometimes look at a software artifact and wonder, where did this come from? Why is it here? What is its real history and origin?

Apart from a general desire to improve program comprehension, developers and IT managers have very concrete reasons to be concerned about the provenance of artifacts that

[1] Ira Baxter's put it best in 2002: "Software clones are segments of code that are similar ... according to some definition of similarity."

[2] To rule out coincidental clone originations, one would need to log every cut-and-paste operation inside the editing environment. Even then, a clone could occur from transcription instead of pasting, for example, from a developer typing in code from a printout.

IWSC 2011 May 23, 2011, Waikiki, Hawaii, USA

comprise their systems. Copying of source code across FOSS is well documented. Sojer and Henkel interviewed several hundred developers and discovered that copying FOSS code by commercial enterprises is now common, but in many cases software developers lack understanding of the legal risks associated with this activity, and their organizations lack policies to guide them [9]. If open source code is improperly copied into a commercial product, the company could be sued or suffer significant negative publicity as a result[3]. Or, if a software component is found to be acting in a suspicious manner, an IT manager might wish to know who designed the component and what it is supposed to be doing.

Finally, we note that it is common practice among Java developers to package external libraries within their applications. This is done to avoid "DLL-hell" when a developer cannot be certain what versions of what libraries may be present within a client's deployment environment. However it is not always the case that precise version information is available for these external libraries. Consequently, application users may be unaware that, for example, their system uses an out-of-date library with known security holes. This problem is exacerbated in industries where IT managers are required to certify their systems meet specific industry standards. Knowing provenance of included libraries can be an important prerequisite for achieving compliance with respect to these standards.

3. BERTILLONAGE

The real problem of software provenance (and digital provenance in general) is that it is wide ranging and ill defined. There is no single technique that will work across many of these problems, let alone all of them. And the specific questions we have will depend heavily on the particular concerns we are trying to address. What *is* common across this problem space is that we seek an authoritative answer to the question: where did this come from? We typically also have a large candidate set to compare against, but brute force comparison is generally infeasible. Thus, we seek techniques that are fast, approximate, and narrow the search space, ideally by rejecting as many low-likelihood matches as possible.

In a recent paper, we have used the term *software Bertillonage* to describe the kind of approach we seek [3]. Bertillonage was a 19^{th} century French forensic technique aimed at reducing a large search space — thousands of criminal mugshots — into a much smaller set of candidates by the use of simple biometrics. That is, when a criminal was originally arrested, she was photographed and measured along 11 dimensions. The thousands of photographs in the police files were then organized hierarchically using the biometrics data. Then when a suspect was arrested later on, her measurements were taken and her photograph was sought in the small pile that matched her measurements. As a forensic technique, Bertillonage was a huge step forward; however, it was also imprecise and error prone. When the science of fingerprint identification emerged soon after, Bertillonage was soon forgotten.

What our desired approach has in common with Bertillonage is that it is fast, approximate, and can narrow a large pool of possibilities down to a tractable set of likely suspects that can be examined using expensive techniques such as manual analysis. In addition, we need techniques that can provide useful answers even when the exact answers may be missing: the forensic equivalent of matching the suspect's sister (and knowing it's her sister) when the suspect herself is not yet in the database. We have not used a more precise metaphor — such as fingerprinting or DNA analysis — because often the answer is not precise or certain.

[3]For example, in 2009 Microsoft had to recall the *Windows 7 USB/DVD Download Tool* because it found it improperly contained open source code [8].

4. BEYOND BERTILLONAGE

The software development practice is changing. Copying code from one application (and from one organization) to another is here to stay. Research is needed to provide ways to identify provenance, but that is only the tip of the problem. Once provenance is determined, how are clones going to be managed? What happens when the origin starts to diverge from the clone? Should the clone be updated? The latest version might be buggy, hence finding a stable one is paramount. What if the clone has been locally modified? How should the changes be propagated to the new version? Provenance adds a new dimension to the original $n \times n$ clone detection problem: $n \times n \times chronology$. This in turn greatly magnifies the performance required of techniques, and also alters the shape of the resulting answers. These are similar problems to those that software clones research has addressed to date, but with orthogonally different motivation, requirements and scope. Nonetheless, we believe the problem of software provenance provides fertile ground for future research in the software clone community.

5. REFERENCES

[1] J. Cheney, S. Chong, N. Foster, M. Seltzer, and S. Vansummeren. Provenance: A future history. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications: Onward! Session*, pages 957–964, Oct. 2009.

[2] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proc. of the 2003 IEEE Intl. Workshop of Program Comprehension*, IWPC-03, pages 196–206, 2003.

[3] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage: Finding the provenance of an entity. In *Proc. of the 2011 Working Conference of Mining Software Repositories*, 2011.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[5] D. M. Germán, M. D. Penta, G. Antoniol, and Y.-G. Guéhéneuc. Code siblings: Phenotype evolution. In *Proc. of the 3rd Intl. Workshop on Detection of Software Clones*, 2009.

[6] J. Harder and N. Göde. Quo vadis, clone management? In *Proc. of the 4th Intl. Workshop on Software Clones*, 2010.

[7] C. Kapser and M. Godfrey. *Cloning considered harmful* considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, 2008. 10.1007/s10664-008-9076-6.

[8] E. Protalinski. Microsoft pulls Windows 7 tool after GPL violation claims. Ars Technica, `http://arstechnica.com/microsoft/news/2009/11/microsoft-pulls-windows-7-tool-after-gpl-violation-claims.ars`, Nov 2009.

[9] M. Sojer and J. Henkel. License Risks from Ad-Hoc Reuse of Code from the Internet: An Empirical Investigation. Preprint, available at `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1594641`, 2010.