Kamino: Dynamic Approach to Semantic Code Clone Detection

Lindsay Neubauer
Columbia University
neubauer@cs.columbia.edu

Abstract

Discovering code clones in a runtime environment helps software engineers identify hard to find logic-based bugs. Yet most research in the area of code clone discovery deals with source code due to the complexity of finding clones in a dynamic environment. KAMINO manipulates Java bytecode to track control and data flow dependencies at the method-level of Java programs during runtime. It then matches similar flows to find semantic code clones. With positive preliminary results indicating code clones using KAMINO , future tests will compare the its robustness compared to existing code clones detection tools.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords code clones, clone detection

1. Background

Discovering duplicate code, or code clones, in large software systems is useful for bug detection, plagiarism detection, and code refactoring. Syntactic code clones are sections of code that are indistinguishable from each other at the source code level except for minor differences between identifiers, literals, types, whitespace, layout, and comments [13]. Semantic code clones are sections of code that are functionally similar but syntactically different [6]. Their discovery is used to fix redundancy issues including program correctness and maintenance [7].

Most of the research in the area of code clones focuses on static approaches to detection. This simplifies both discovery and validation, and there are numerous findings that indicate the importance of clone detection using this method [8]. Selim et al. transform source code into an intermediate representation before using existing source-based clone detection to discover harder to find syntactic clones [16]. Davis and Godfrey find clones at the assembly language level, analyzing code used in the build process [3]. There is some research that analyzes Java bytecode, using static analysis [9], process algebras [15] and pattern and content matching [10]. Sæbjørnsen et al. detect clones using binary executables to cluster normalized assembly instructions [14].

Dynamically discovering code clones is more difficult but finds clones that are inaccessible using a static approach. Khan et al. combine source code clones created using static analysis techniques with execution trace information to find the extent clones are active during runtime [11]. Some determine semantic code clones based on input and output behavior, using random testing to generate input values [5, 6]. KAMINO aims to dynamically discover semantic code clones to help software engineers identify harder to find logic-based bugs.

2. Kamino

This research extends work done by Sethumadhavan and Demme using λSignatures generated for C programs [4]. λSignatures are used to determine semantic similarity between basic blocks of code, where a basic block is defined as a section of code with one entry and one exit point. KAMINO manipulates Java bytecode using ASM [12] to track control and data flow between basic blocks at the method level of a program during runtime. In KAMINO, data flow between basic blocks is defined as a variable read or write. There are four types of control flow: Conditional jumps, Unconditional jumps, Fall throughs, and Exceptions. Currently KAMINO does not record exceptions, but because exceptions are a standard workflow in Java they will be added in the future. KAMINO keeps track of the starting and ending basic blocks when a control or data flow occurs dynamically, and the distance between these blocks. In the case of data flows, it also keeps track of the variable for which the data flow belongs.

KAMINO generates output for Java applications ranging from a single file to programs that rely on a directory structure. There are two different output options. It can generate program dependency graphs similar to those created by [4], where each basic block is represented as a node and control and data flows between basic blocks are represented as weighted edges. This output will be used to approxi-

mately cluster graphs to find program similarity. Alternatively, KAMINO can keep track of control and/or data flows between basic blocks in a string format. This output is used with a longest common substring algorithm to compare identical sequences within a linear execution.

To generate large dynamic program graphs for this project, as well as for future code clone analysis, I used Apache Tomcat [1] versions 7 and 8. Tomcat is a large open source application that implements the Java Servlet and JaveServer Pages technology [1]. Comparing multiple versions of the same application should result in a significant number code clones, working as a ground truth for the validity of our approach. Previous studies have found that between versions of Tomcat there are large amounts of duplicate code, which present as static code clones when compared [17–19].

Preliminary tests were run using the longest common substring approach. I ran a small number of tests from both versions 7 and 8 of tomcat and compared them against themselves by doing a manual check of each method within the tests. These tests generated the same strings for each of the methods tested. I checked the same number of tests from the tomcat versions and compared them against each other and did a manual check of each method within the tests. These tests generated similar strings for each of the methods, including those that were exactly the same between versions. Further tests will be conducted on the longest common substring to determine the slight differences between versions. Because these test were not conclusive, tests using the approximate graph clustering approach will be conducted.

3. Research Plan

Once a ground truth is established, I will compare clones discovered with KAMINO in one version of Tomcat to those of the duplicate code finder provided in the open source development tool Checkstyle [2]. Checkstyle, which finds static syntactic clones, takes a strict approach to code clones by guaranteeing no false positives [2]. This is ideal for a comparison with dynamic code clones because any of the clones that are found by both approaches are guaranteed to be static and syntactic clones. Any additional clones that are discovered using the dynamic approach will be inspected to establish that they are true clones and to determine whether they are semantic or syntactic. Any clones found by Checkstyle [2] that are not found by the dynamic approach will be considered false negatives. Pending positive results from this study, I will continue to compare KAMINO to other dynamic and semantic code clone detectors.

4. Acknowledgments

The author is a member of the Programming Systems Laboratory, which is funded in part by NSF CCF-1302269, CCF-1161079, NSF CNS-0905246, and NIH U54 CA121852.

References

[1] Apache tomcat. `http://tomcat.apache.org`, May 2014.

[2] Checkstyle. Duplicate code. `http://checkstyle.sourceforge.net/config_duplicates.html`.

[3] I. Davis and M. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *Working Conference on Reverse Engineering*, 2010.

[4] J. Demme and S. Sethumadhavan. Approximate graph clustering for program characterization. *ACM Trans. Archit. Code Optim.*, 2012.

[5] R. Elva and G. Leavens. Semantic clone detection using method ioe-behavior. In *International Workshop on Software Clones*, 2012.

[6] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA '09*.

[7] E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy and paste. *Software Maintenance and Reengineering*, 2010.

[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE '09*.

[9] T. Kamiya. Agec: An execution-semantic clone detection tool. In *ICPC '13*.

[10] I. Keivanloo, C. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *International Workshop on Software Clones*, 2012.

[11] M. A. A. Khan, C. K. Roy, and K. A. Schneider. Active clones: Source code clones at runtime. *International Workshop on Software Clones*, 2014.

[12] OW2Consortium. Asm. `http://asm.ow2.org`, May 2014.

[13] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009.

[14] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA '09*.

[15] A. Santone. Clone detection through process algebras and java bytecode. In *International Workshop on Software Clones*, 2011.

[16] G. Selim, K. Foo, and Y. Zou. Enhancing source-based clone detection using intermediate representation. In *Working Conference on Reverse Engineering*, 2010.

[17] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *International Workshop on Principles on Software Evolution*, 2003.

[18] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06*.

[19] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *Software Metrics*, 2005.