Platform Categorization with Runtime Ideals besides Its Presentation to Software Thieving Finding

Suresh C
Department of computer Science, Ganadipathy Tulsi's Jain Engineering College, Vellore, India.

Muthukumar S
Associate professor, Department of computer Science, Ganadipathy Tulsi's Jain Engineering College, Vellore, India.

Appandairaj C
Assistant professor, Department of computer Science, Ganadipathy Tulsi's Jain Engineering College, Vellore, India.

Abstract – Identifying similar or identical code fragments becomes much more challenging in code theft cases where plagiarizers can use various automated code transformation or obfuscation techniques to hide stolen code from being detected. Source code plagiarism has become a serious problem for the industry. Although there exist many software solutions for comparing source codes, they are often not practical. This paper presents a novel dynamic analysis approach to software plagiarism detection. Previous works in this field are largely limited in that (i) most of them cannot handle advanced obfuscation techniques, and (ii) the methods based on source code analysis are not practical since the source code of suspicious programs typically cannot be obtained until strong evidences have been collected. Based on the observation that some critical runtime values of a program are hard to be replaced or eliminated by semantics-preserving transformation techniques, we introduce a novel approach to dynamic characterization of executable programs. Our value-based plagiarism detection method (VaPD) uses the longest common subsequence based similarity measuring algorithms to check whether two code fragments belong to the same lineage. We evaluate our proposed method through a set of real-world automated obfuscators.

Index Terms – Software plagiarism, dynamic code analysis.

1. INTRODUCTION

Software plagiarism and piracy is a serious problem which is estimated to cost the software industry billions of dollars per year [6]. Software piracy for desktop computers has gained most of the attention in the past. However, software plagiarism and software piracy is also a huge problem for companies. Software theft means the unauthorized or illegal copying, sharing or usage of copyright-protected software programs. Software theft may be carried out by individuals, groups or, in some cases, organizations who then distribute the unauthorized software copies to users. Software theft is committed when someone performs any of the following: (i) Steals software media, (ii) Deliberately erases programs, (iii) Illegally copies or distributes a program, (iv) Registers or activates a software program illegally. Software plagiarism, or code theft, is the copying of computer programs without attribution, a phenomenon that has become widespread with the advent of the internet and easy access to and transmission of software.

Identifying same or similar code fragments among different programs or in the same program is very important in some applications. For example, duplicated codes found in the same program may degrade efficiency in both development phase (e.g., they can confuse programmers and lead to potential errors) and execution phase (e.g., duplicated code can degrade cache performance). In this case, code identification techniques such as clone detection can be used to discover and refactor the identical code fragments to improve the program. For another example, same or similar code found in different programs may lead us to even more serious issues. If those programs have been individually developed by different programmers, and if they do not embed any public domain code in common, duplicated code can be an indication of software plagiarism or code theft. In code theft cases, determining the sameness of two code fragments becomes much more difficult since plagiarizers can use various code transformation techniques including code obfuscation techniques to hide stolen code from detection. In order to handle such cases, code characterization and identification techniques must be able to detect the identical code (i.e., two code fragments belonging to the same lineage) without being easily circumvented by code transformation techniques.

2. OVERVIEW OF EXISTING SYSTEMS

The techniques for source code comparison originated with the string-based algorithms that were used for detecting plagiarism of ordinary English prose. Software systems often contain portions of code that are similar to other systems, and these common portions are referred to as code clones [5]. Detecting clones in source code has been recognized as an important issue in software analysis. Most of the existing approaches to detect plagiarism employ counting heuristics or string matching techniques to measure similarity in source code [1]. Source

code can be represented as graphs. Existing graph theory algorithms can then be applied to measure the similarity between source code graphs [2].

There are methods based on Program Dependency Graph (PDG) which cannot detect similarities if semantics preserving transformation is applied on the source code. Birthmarks based on dynamic analysis can also be used to detect plagiarism. Whole Program Path (WPP) birthmarks represent the dynamic control flow of a program are robust to some control flow obfuscation, but vulnerable to semantics-preserving transformations. There are variety of dynamic birthmarks based on system call, sequence of API function call and frequency of API function call. They are also vulnerable to real obfuscation techniques [14]. Chanet al [15] proposed a birthmark system for JavaScript programs based on the run-time heap. The heap profiler takes multiple snapshots of the JavaScript program during execution. The graph generator generates heap graphs containing objects created during execution as nodes. Plagiarism is detected from the heap graphs of genuine and suspected programs.

3. PROPOSED APPROACH

We are analyzing the dynamic behaviour of source codes to capture the similarities among them. Our approach uses method calling structure and the values of key variables in order to carry out different analyses. To our best knowledge, our work is the first one exploring the existence of the core-values. By exploiting runtime values that can hardly be changed or replaced, our code characterization technique is resilient to various control and data obfuscation techniques. It does not require access to source code of suspicious programs, thus it could greatly reduce plaintiff's risks through providing strong evidences before filing a lawsuit related to intellectual property.

4. DESIGN

Software theft has become a very serious concern to software companies and open source communities. In the presence of automated semantics-preserving code transformation tools, the existing code characterization techniques may face an impediment to finding sameness of plagiarized code and the original. In this section, we discuss how we apply our technique to software plagiarism detection. Later, we evaluate our method against such code obfuscation tools in the context of software plagiarism detection. Scope of Our Work: We consider the following types of software plagiarisms in the presence of automated obfuscators: whole-program plagiarism, where the plagiarizer copies the whole or majority of the plaintiff program and wraps it in a modified interface, and core-part plagiarism, where the plagiarizer copies only a part such as a module or an engine of the plaintiff program. Our main purpose of VaPD is to develop a practical solution to real-world problems of the whole-program software plagiarism detection, in which no source code of the suspect program is available. VaPD can also be a useful tool to solve many partial plagiarism cases where the plaintiff can provide the information about which part of his program is likely to be plagiarized. We present applicability of our technique to core-part plagiarism detection in the discussion section. We note that if the plagiarized code is very small or functionally trivial, VaPD would not be an appropriate tool.

5. RUNTIME VALUES

The runtime values of a program are defined as values from the output operands of the machine instructions executed programs; we observed that some runtime values of a program could not be changed through automated semantics preserving transformation techniques such as optimization, obfuscation, different compilers, etc. We call such invariant values core-values.

Core-values of a program are constructed from runtime values that are pivotal for the program to transform its input to desired output. We can practically eliminate noncore values from the runtime values to retain core-values. To identify non-core values, we leverage taint analysis and easily accessible semantics-preserving transformation techniques such as optimization techniques implemented in compilers. Let vp be a runtime value of program P taking I as input, and f be a semantics-preserving transformation. Then, the non-core values have the following properties: (1) If vp is not derived from I, vp is not a core-value of P; (2) If vp is not in the set of runtime values of f (P), vp is not a core-value of P.

6. EXTRACTION OF RUNTIME VALUES

Since not all values associated with the execution of a program are core-values, we establish the following requirements for a value to be added into a value sequence: The value should be output of a value-updating instruction and be closely related to the program's semantics.

Informally, a computer is a state machine that makes state transition based on input and a sequence of machine instructions. After every single execution of a machine instruction, the state is updated with the outcome of the instruction. Because the sequence of state updates reflects how the program computes, the sequence of state-updating values is closely related to the program's semantics. As such, in value based characterization, we are interested only in the state transitions made by value-updating instructions. More formally, we can conceptualize the state-update as the change of data stored in devices such as RAM and registers after each instruction is performed, and we call the changed data a state-updating value. We further define a value-updating instruction as a machine instruction that does not always preserve input in its output. Being an output of a value updating instruction is a sufficient condition to be a state updating value. Therefore, we exclude output values of non-value- updating instructions from

a value sequence. In our x86 implementation, the value-updating instructions are the standard mathematical operations (add, sub, etc.), the logical operators (and, or, etc.), bit shift arithmetic and logical (shl, shr, etc.), and rotate operations (ror, rcl, etc.).

7. CORE PART PLAGIARISM

Core-part plagiarism is a harder problem. In such case, only some part of a program is plagiarized. For example, a less ethical developer may steal code from some open source projects and fit the essential module into his project with obfuscation. Let IPM and ISM be the input to the plagiarized module and suspect module respectively, and V(x) be a value based characteristic such as a value sequence extracted from x, a program or a module.

Memory addresses or pointer values stored in registers or memory locations are transient. For example, some binary transformation techniques such as word alignment and local variable reordering can change pointers to local variables or offsets in stack; and heap pointers may not be the same next time the program is executed even with the same input. Therefore, we do not include pointer values in a refined value sequence.

In our VaPD prototype, we implement a range checking based heuristic to detect addresses. Our test bed dynamically monitors the changes of memory pages allocated to the program being analyzed, and it maintains a list of ranges of all the allocated pages with write permission enabled. If a runtime value is found to be within the ranges in the list, VaPD discards the value, regarding the value as an address. Although this heuristic may also delete some non-pointer values, it can remove pointers to stack and to heap with no exception. Address removal heuristic is applicable to both plaintiff and suspect programs.

Our technique bears the following limitations. First, besides the ability of extracting value sequences from the entire scope of the plaintiff program, VaPD provides the partial extraction mode in which it can extract value sequences from only a small part of the program. Based on this, we discuss about the feasibility of applying VaPD to the partial plagiarism detection problems. However, we have not yet comprehensively evaluated this issue with real world test subjects. In such case, a more efficient and scalable program emulator or logger other than QEMU may be needed. Second, VaPD may not apply if the program implements a very simple algorithm. In such cases, the value sequences can be too short, which increases sensitivity to noises. Our metric is more likely to cause false positives when a very short value sequence is compared to a much longer one. Third, as a detection system, there exists a trade-off between false positives and false negatives. The detection result of our tool depends on the similarity score threshold. Unfortunately, without many real-world plagiarism samples which are often not available, we are unable to show concrete results on such false rates. As such, rather than applying our tool to "prove" software plagiarisms, in practice one may use it to collect initial evidences before taking further investigations, which often involve nontechnical actions.

8. CONCLUSION

Results show that it can greatly improve the performance of social network analysis against state-of-the-art approaches we will ready how to employ our approach in a hierarchical way to reduce the memory overhead and evaluate its performance gain graphically. Obfuscation resilient code characterization is important for many code analysis applications, including code theft detection. Motivated by an observation that some outcome values computed by machine instructions survive various semantics-preserving code transformations, we have proposed a technique that directly examines executable files and does not need to access the source code of suspicious programs. Our results show that the value-based method is effective in identifying software plagiarism.

9. FUTURE WORK

App repackaging, a form of software plagiarism, has become a common phenomenon in the mobile app markets like Google Play and Apple iTunes. Dishonest users may repackage others' apps under their own names or embed different advertisements, and then republish it to the app market to earn monetary profit. Furthermore, to leverage the popularity of mobile apps to increase the propagation of their malware, malware writers may modify popular apps to insert malicious payloads into the original apps. A common drawback is that most of them are not obfuscation-resilient. Our research is obfuscation-resilient and can be potentially applied to the smart phone app repackaging detection. More recently, Huang et al developed a repackaging detection evaluation framework so that different methods can be systematically evaluated and compared, with obfuscations applied. View Droid applied a interface based birthmark, which is designed for user interaction intensive and event dominated programs, to detect smart phone application plagiarism.

In this section, we discuss heuristics to refine value sequences. An initial value sequence constructed through the dynamic taint analysis may still contain a number of non-identical is to compile the same source code with the same compiler with different optimization switches enabled. Motivated by this idea, we use several optimized executables of the same program to sift non-core values out. With GCC and its five selected optimization flags (-O0, -O1, -O2, -O3, and -Os), we can extract five optimized value sequences from the plaintiff program. Each optimized value sequence has been processed with the sequential refinement while it is extracted. Then, we compute a longest common subsequence of all the optimized value sequences to retain only the common values in the resulting value sequence. As we do not assume we have access

to the source code of suspect programs, this refinement heuristic is only applicable to plaintiff programs.

REFERENCES

[1] Ali AMET, Abdulla HMD, Snasel V. Survey of plagiarism detection methods. Asia Modelling and Symposium 2011. p.39-42.
[2] Graves JA. Source code plagiarism detection using a graph-based approach. Master's thesis. TennesseeTech. Univ., 2011.
[3] Jhi YC, Wang X, Jia X, Zhu S, Liu P, Wu D. Value-based program characterization and its application to software plagiarism detection.
[4] Intl. Conf. on Soft. Engg. 2011. p.756-765. Jhi YC, Wang X, Jia X, Zhu S, Liu P, Wu D. Value-based program characterization and its application to software plagiarism detection.
[5] Koschke R. Frontiers of software clone management. Frontiers of Software Maintenance 2008. p.119-128.
[6] Plagiarism Detection for Java Programs without Source Codes Anjali V.a,*, Swapna T.R.a, Bharat Jayaramanb,2014
[7] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In ACM CCS, 2009.
[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalableand accurate tree-based detection of code clones," in Proceedings of the 29th International Conference on Software Engineering (ICSE '07), 2007, pp. 96–105.
[9] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The University of Auckland, Tech. Rep.148, Jul. 1997.
[10] C. S. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98), 1998, pp. 184–196.
[11] C. Wang, "A security architecture for survivability mechanisms,"Ph.D. dissertation, University of Virginia, Charlottesville, VA,USA, 2001, adviser-John Knight.
[12] C. Collberg, G. Myles, and A. Huntwork, "Sandmark–a tool for software protection research," IEEE Security and Privacy, vol. 1, no. 4, pp. 40–49, 2003.
[13] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de)obfuscation tool," in Proceedings of the 2006 ACM SIGPLANsymposium on Partial evaluation and semantics-based program manipulation (PEPM '06), 2006, pp. 140–144.
[14] Semantic Designs, Inc., "ThicketTM," http://www.semanticdesigns.com
[15] Zelix Pty Lt, "Java obfuscator-ZelixKlassMaster," online, http://www.zelix.com/klassmaster/features.html.

Authors

Suresh C was born in Vellore, India, in 1990. He received the B.E. degree in computer science and engineering from Kings Engineering College, Chennai. He currently pursuing M.E degree in computer science and engineering from Ganadipathy Tulsi's Jain Engineering College, Vellore. His current research interests include cloud computing, enterprise application, big data, and software engineering.

Muthu Kumar S was born on 1977. He received the M.C.A. degree from Bharathidasan University. He completed M.E degree in TCET; Vandavasi.He currently works as associate professor in GTEC, Vellore. He has total 15 years of experience. His current research interests include Digital image processing, Software quality assurance etc. He has guided many UG and PG projects.

Appandairaj A was born in Arni, India, in 1982. He received the M.Tech. degree in Computer Science and Engineering from Dr.M.G.R University, Chennai. He currently works as assistant professor in GTEC, Vellore. His current research interests include cloud computing, big data. He has guided many UG and PG projects.