International Journal of
Scientific Engineering and Technology Research

ISSN 2319-8885
Vol.05,Issue.36
October-2016,
Pages:7426-7432

www.ijsetr.com

Package Description by Runtime Standards Also Its Presentation Towards Software Piracy Finding

M. KIRAN[1], Y. SRINIVAS[2]

[1]PG Scholar, Dept of CSE, Siddhartha Institute of Engineering and Technology, Ibrahimpatnam, Hyderabad, TS, India.
[2]Associate Professor, Dept of CSE, Siddhartha Institute of Engineering and Technology, Ibrahimpatnam, Hyderabad, TS, India.

Abstract: Misusing same code in same or similar code fragments has become a major issue to software community. Simarility in misuse of code raises where plagiarizers can use different obfuscation techniques to hide stolen code from being detected. Several researches has been done but can't handle obfuscation techniques. The source code analysis cannot be implemented. Depending on the observation some critical values replaced with semantics-preserving techniques. so, a novel approach of dynamic characterization of executable programs is flexible to control the data obfuscation techniques. Using this technique, how the values can be extracted and dynamically changing during runtime helps to resolve issues in detecting software plagarism. a prototype with a dynamic taint analyzer atop a generic processor emulator has been implemented. Value-based plagiarism detection method evaluates to verify whether two code fragments belong to the same lineage. Most of the experimental results have been proved that proposed technique like value –based method is best suitable to detect software plagiarisms.

Keywords: Software Plagiarism Detection, Dynamic Code Identification.

I. INTRODUCTION

Detecting duplicate codes among various programs is very important issue in applications .it can degrade the performance in development and execution phase. in such situations, code identification techniques such as clone detection can be used to identify and re factor the duplicate code fragments to improve the program. Various programmers has been developing these kind of programs individually they do not embed any public domain code. Hence duplicate code leads to 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 [9], [10], [11]. In order to handle such cases, code characterization and identification techniques must be able to detect semantically equivalent code (i.e., two code fragments belonging to the same lineage) without being easily circumvented by code transformation techniques researching are highly insufficient in meeting the two highly desired requirements: (r1) resiliency to the automated semantics-preserving obfuscation tools that can easily transform most of the syntactic features such as strings [9], [12], [13], [14], [15]; and (r2) ability to directly work on binary executables of suspected programs since, in some applications such as code theft cases, the source code of suspect software products often cannot be obtained until some strong evidences have been collected the existing schemes can be broken down into four classes to see their limitations with respect to the a forementioned three requirements: (c1) static source code comparison methods [16], [17], [18], [19], [20], [21], [22], [23]; (c2) static executable code comparison methods [24]; (c3) dynamic control flow based methods [25]; (c4) dynamic api based methods [26], [27], [28].

We may briefly summarize their limitations as follows. First, class c1, c2 and c3 do not satisfy requirement r1 Because they are vulnerable to semantics-preserving obfuscation techniques such as outlining and ordering transformation. Second, c1 does not meet r2 because it has to access source code. Hence, deal with various above issues invented a novel approach for dynamic characterization of executable programs. after we examined various runtime properties of executable programs, we found an interesting observation that some runtime values (or computation results of some machine instructions) of a program are hard to be replaced or eliminatedby semantics-preserving transformation techniques such as optimization techniques, obfuscation techniques, different compilers, etc. we call such values core values. Note core values are values computed at runtime from program execution, not the static constants embedded in the executables such as strings, which can be easily obfuscated. to investigate the resilience of core values (to semantics-preserving code transformation), we generated e1::5, five different versions of executable files of test program p written in c, by compiling p with each of the Five optimization switches of gcc (-o0, -o1, -o2, -o3, and -os). from each of e1::5 given the same test input, we extracted a value sequence, a sequence of values (4- bit, 8-bit, 16-bit, or 32-bit)

Copyright @ 2016 IJSETR. All rights reserved.

written as computation results of arithmetic instructions and bit-wise instructions in the execution path.

As a way of retaining (in the value sequence) only the values derived from input, we implemented a dynamic taint analyzer.1 when we analyzed the value sequences of e1::5, we found that some values survived all of the five optimization switches. moreover, the sequence of the values surviving all of the five optimization switches was enclosed almost perfectly by the value sequences of executables generated by compiling p with different compilers (we tested tiny c compiler [29] and open watcom c compiler [30]). this indicates that core-values do exist and we can use them to check whether two code fragments belong to the same lineage in this paper, we show (1) how we extract the values revealing core-values; and (2) how we apply this runtime property to solve problems in software plagiarism detection. we have implemented a value extractor with a specific dynamic taint analyzer and value refinement techniques atop a generic processor emulator, as part of our value-based program characterization method. as a machine code analyzer which directly works on binary executables, our technique satisfies r2. Regarding the requirement r1, we have implemented a value-based software plagiarism detection method (vapd) that uses similarity measuring algorithms based on sequences and dependence graphs constructed from the extracted values. we have evaluated it through a set of real world obfuscators including two commercial products, zelix pty ltd.'s klassmaster [15] and semantic designs inc.'s thicket [14]. our experimental results indicate that the vapd successfully discriminated 34 plagiarisms obfuscated by sandmark [12] (totally 39 obfuscators, but 5 of them failed to obfuscate our test programs); plagiarisms eavily obfuscated by klassmaster,2 programs obfuscated by the thicket c obfuscator, and executables obfuscated by control flow flattening implemented in the loco/diablo link-time optimizer [13].

CONTRIBUTIONS: in summary, we make the following contributions:

1. We present a novel code characterization method based on runtime values. To our best knowledge, our work is the first one exploring the existence of the core-values.
2. By exploiting runtime values that can hardly be changed or replaced, our code characterization technique is resilient to various control and data obfuscation techniques.
3. Our plagiarism detection method (vapd) 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. We evaluate vapd through a set of real world programs.

This paper is organized as follows. In the next section, we briefly discuss related works. In section 3, we discuss the existence of core-values implied by our experimental results. in section IV and V, we evaluate our value based code characterization method by applying it to the problems of software plagiarism detection. In sectionVI, we address reordering attacks and evaluate our dependence graph based method.Finally, the limitations, some potential counterattacks, and future work are discussed in sectionVII.

II. OVERVIEW OF EXISTING SYSTEMS

Methods for identifying the similarity in source code was created with string –based algorithms used in finding plagiarism of original code. These common sections referred as code replicas [5]. Finding replicas in software analysis has become a major issue. 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 runtime 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.

III. PROPOSED APPROACH

We evaluate our proposed method through a set of real-world automated obfuscators. Our experimental results show that the value-based method successfully discriminates 34 plagiarisms obfuscated by sand mark, plagiarisms heavily obfuscated by k lass master, programs obfuscated by thicket, and executables obfuscated by loco/diablo. Thus it could greatly reduce plaintiff's risks through providing strong evidences before filing a lawsuit related to intellectual property.

IV. 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.

V. 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 corevalues. 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.

VI. 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.).

VII. 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.

VIII. RESULTS

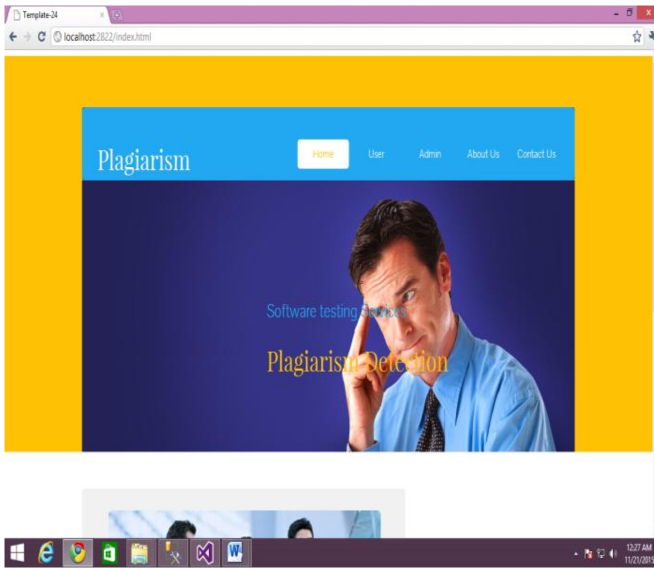

Fig1. Home Login.

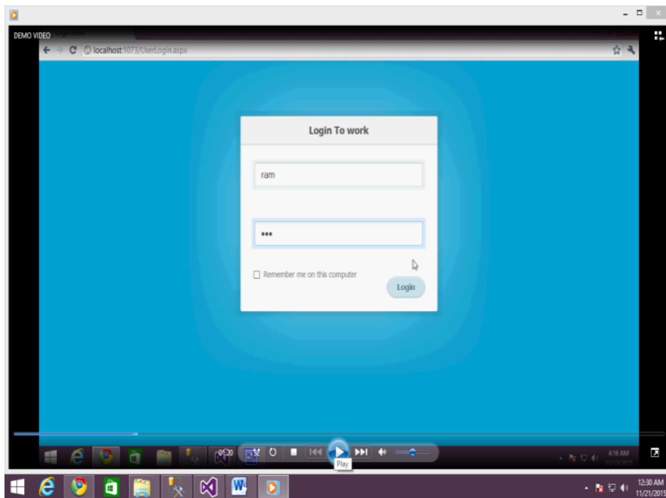

Fig2. Login Page.

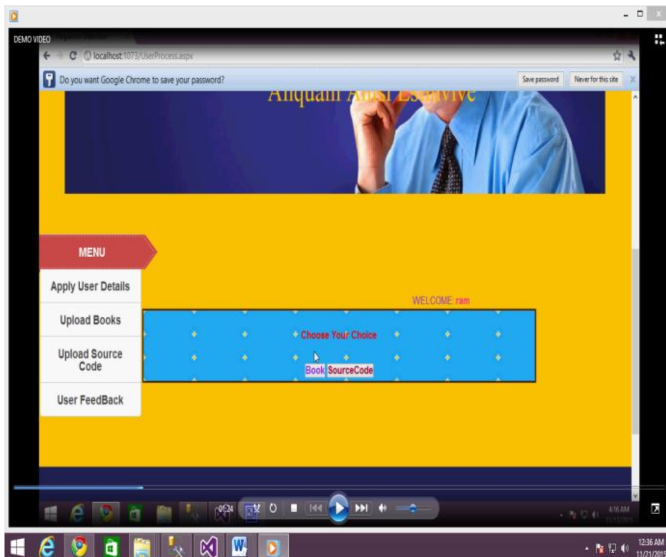

Fig3. User booking details.

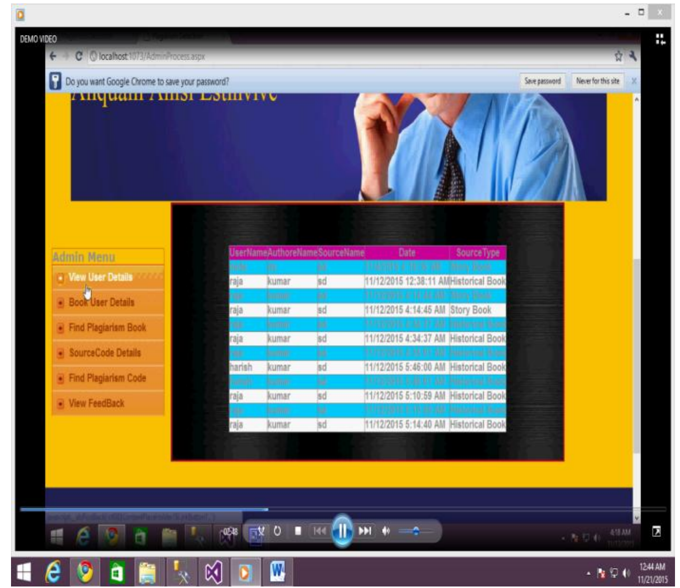

Fig4. View book details.

Fig5. Plagiarism detection.

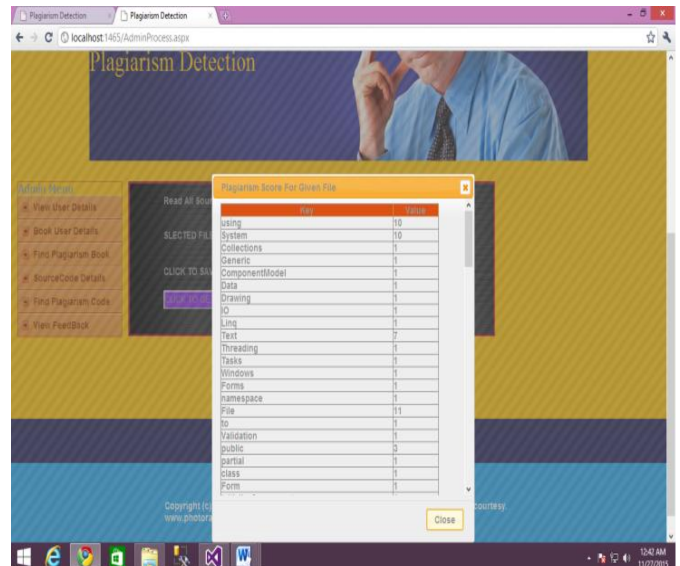

Fig6. finding the original.

IX. CONCLUSION

The Code Analysis Applications with Code theft Detection is prior to the Obfuscation Resilient Code Characterization in order to detecting code theft. In future work, along with an efficient runtime to support this approach. Results show that it can greatly improve the performance of Identifying Software Plagiarism Our Technique Is Resilient to Various Control and Data Obfuscation Techniques. Hence Proposed Approach i.e. Value-Based Method is highly efficient for detecting the duplicate codes.

X. REFERENCES

[1] B. S. Baker, “On Finding Duplication And Near-Duplication In Large Software Systems,” In Proceedings Of 2nd Working Conference On Reverse Engineering (Wcre ’95), 1995, Pp. 86–95.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant’anna, And L. Bier, “Clone Detection Using Abstract Syntax Trees.” In Int. Conf. On Software Maintenance, 1998.

[3] K. Kontogiannis, M. Galler, And R. Demori, “Detecting Code Similarity Using Patterns.” In Working Notes Of 3rd Workshop On Ai And Software Engineering, 1995.

[4]J.Krinke, “Identifying Similar Code With Program Dependence Graphs.” In Proceedings Of Eighth Working Conference On Reverse Engineering (Wcre ’01), 2001, Pp. 301–309.

[5] T. Kamiya, S. Kusumoto, And K. Inoue., “Ccfinder: A Multilinguistic Token-Based Code Clone Detection System For Large Scale Source Code.” Ieee Transactions On Software Engineering, Vol. 28,No. 7, Pp. 654–670, 2002.

[6] M. Gabel, L. Jiang, And Z. Su, “Scalable Detection Of Semantic Clones,” In Proceedings Of The 30th International Conference On Software Engineering (Icse ’08), 2008, Pp. 321–330.

[7] L. Jiang, Z. Su, And E. Chiu, “Context-Based Detection Of Clonerelated Bugs,” In Proceedings Of The The 6th Joint Meeting Of The European Software Engineering Conference And The Acm Sigsoft Symposium On The Foundations Of Software Engineering, Ser. Esecfse
’07, 2007, Pp. 55–64.

[8] L. Jiang, G. Misherghi, Z. Su, And S. Glondu, “Deckard: Scalable And 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 Univeristy 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 Sigplan Symposium On Partial Evaluation And Semantics-Based Program Manipulation (Pepm ’06), 2006, Pp. 140–144.

[14] Semantic Designs, Inc., “Thickettm,” Http:// Www. Semantic designs.Com. [15] Zelix Pty Lt, “Java Obfuscator - ZelixKlassmaster,”Online,Http://Www.Zelix.Com/Klassmaster/Features.Html.

[16] C. Liu, C. Chen, J. Han, And P. S. Yu, “Gplag: Detection Of Software Plagiarism By Program Dependence Graph Analysis,” In Proceedings Of The 12th Acm Sigkdd International Conference On Knowledge Discovery And Data Mining (Kdd ’06), 2006, Pp. 872–881.

[17] H. Tamada, M. Nakamura, A. Monden, And K. Ichi Matsumoto, “Design And Evaluation Of Birthmarks For Detecting Theft Of Java Programs,” In Iasted Conference On Software Engineering (Iasted Se ’04), February 2004, Pp. 569–574, Innsbruck, Austria.

[18] W. Yang, “Identifying Syntactic Differences Between Two Programs,”Software: Practice And Experience, Vol. 21, No. 7, Pp. 739–755, 1991.

[19] Y.-C. Kim And J. Choi, “A Program Plagiarism Evaluation System,”In Information And Communication Technology Education Workshop,2005.

[20] N. Truong, P. Roe, And P. Bancroft, “Static Analysis Of Students’Java Programs,” In Ace ’04: Proc. Of The 6th Conf. On Australasian Computing Education, 2004.

[21] L. Prechelt, G. Malpohl, And M. Philippsen, “Finding Plagiarisms Among A Set Of Programs With Jplag,” Universal Computer Science,2000.

[22] S. Schleimer, D. S. Wilkerson, And A. Aiken, “Winnowing: Local Algorithms For Document Fingerprinting.” In Acm Sigmod Int. Conf. On Management Of Data, 2003.

[23] J.-H. Ji, G. Woo, And H.-G. Cho, “A Source Code Linearization Technique For Detecting Plagiarized Programs,” In Proceedings Of The 12th Annual Sigcse Conference On Innovation And Technology In Computer Science Education (Iticse ’07), 2007, Pp. 73–77.

[24] G. Myles And C. Collberg, “K-Gram Based Software Birthmarks,” In Proceedings Of The 2005 Acm Symposium On Applied Computing (Sac ’05), 2005, Pp. 314–318.

[25] G. Myles And C. S. Collberg, “Detecting Software Theft Via Whole Program Path Birthmarks,” In Proceedings Of 7th International Conference On Information Security (Isc ’04), 2004.

[26] D. Schuler, V. Dallmeier, And C. Lindig, “A Dynamic Birthmark For Java,” In Proceedings Of The Twenty-Second Ieee/Acm International Conference On Automated Software Engineering (Ase ’07), 2007, Pp.274–283.

[27] H. Tamada, K. Okamoto, M. Nakamura, And A. Monden, “Dynamic Software Birthmarks To Detect The Theft Of Windows Applications,”In Int’l Symp. On Future Software Technology (Isfst),October 2004.

[28] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, And K. Ichi Matsumoto, “Design And Evaluation Of Dynamic Software Birthmarks Based On Api Calls,” Nara

Institute Of Science And Technology,Info. Science Technical Report Naist-Is-Tr2007011, Issn 0919-9527, May 2007.
[29] F. Bellard, “Tiny C Compiler,” Http://Bellard.Org/Tcc/.
[30] Open Watcom Contributors, “Open Watcom,” Http://Www.Openwatcom.Org.
[31] A. Sebjornsen, J. Willcock, T. Panas, D. Quinlan, And Z. Su,“Detecting Code Clones In Binary Executables,” In Proceedings Of The Eighteenth International Symposium On Software Testing And Analysis (Issta ’09), 2009, Pp. 117–128.
[32] L. Luo, J. Ming, D. Wu, P. Liu, And S. Zhu, “Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison With Applications To Software Plagiarism Detection,” In Proceedings Of The 22nd Acm Sigsoft International Symposium On The Foundations Of Software Engineering (Fse 2014), November 2014.
[33] X. Wang, Y.-C. Jhi, S. Zhu, And P. Liu, “Behavior Based Software Theft Detection,” In Proceedings Of The 16th Acm Conference On Computer And Communications Security (Ccs ’09), 2009, Pp. 280–290.
[34] F. Zhang, Y. Jhi, D. Wu, P. Liu, And S. Zhu, “A First Step Towards Algorithm Plagiarism Detection,” In Proceedings Of The 2012 International Symposium On Software Testing And Analysis (Issta ’12).Acm, 2012, Pp. 111–121.
[35] F. Zhang, D. Wu, P. Liu, And S. Zhu, “Program Logic Based Software Plagiarism Detection,” In Proceedings Of The 25th Annual International Symposium On Software Reliability Engineering (Issre 2014), November 2014.
[36] W. Zhou, Y. Zhou, X. Jiang, And P. Ning, “Detecting Repackaged Smartphone Applications In Third-Party Android Marketplaces,” In Proceedings Of The Second Acm Conference On Data And Application Security And Privacy (Codaspy ’12), 2012.
[37] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, And D. Song, “Juxtapp: A Scalable System For Detecting Code Reuse Among Android Applications,” In Proceedings Of The 9th Conference On Detection Of Intrusions And Malware & Vulnerability Assessment, 2012.
[38] R. Potharaju, A. Newell, C. Nita-Rotaru, And X. Zhang, “Plagiarizing Smartphone Applications: Attack Strategies And Defense Techniques,” In Engineering Secure Software And Systems, Lecture Notes In Computer Science, 2012, Pp. 106–120.
[39] J. Crussell, C. Gibler, And H. Chen, “Attack Of The Clones: Detecting Cloned Applications On Android Markets,” Computer Security–Esorics 2012, Pp. 37–54, 2012.
[40] H. Huang, S. Zhu, P. Liu, And D.Wu, “A Framework For Evaluating Mobile App Repackaging Detection Algorithms,” In Proceedings Of The 6th International Conference On Trust And Trustworthy Computing (Trust ’13), 2013.
[41] F. Zhang, S. Zhu, D. Wu, And P. Liu, “Viewdroid: Towards Obfuscation-Resilient Mobile Application Repackaging Detection,” In Proceedings Of The 7th Acm Conference On Security And Privacy In Wireless And Mobile Networks (Wisec 2014), July 2014.
[42] J. Newsome And D. Song, “Dynamic Taint Analysis For Automatic Detection, Analysis, And Signature Generation Of Exploits On Commodity Software,” In Proceedings Of The Network And Distributed System Security Symposium (Ndss ’05), 2005.0098-5589 (C) 2015 Ieee. Personal Use Is Permitted, But Republication/Redistribution Requires Ieee Permission.SeeHttp://Www.Ieee.Org/Publications_Standards/ Publications/ Rights/Index.Html For More Information. This Article Has Been Accepted For Publication In A Future Issue Of This Journal, But Has Not Been Fully Edited. Content May Change Prior To Final Publication. Citation Information: Doi 10.1109/Tse.2015.2418777, Ieee Transactions On Software Engineering Ieee Transactions On Software Engineering, Vol. Vv, No. Nn, Mm Yyyy 20
[43] F. Bellard, “Qemu, A Fast And Portable Dynamic Translator,” In Atec ’05: Proc. Of The Annual Conference On Usenix Annual Technical Conference. Berkeley, Ca, Usa: Usenix Association, 2005, Pp.41–41.
[44] I. D. Baxter, C. Pidgeon, And M. Mehlich, “Dms: Program Transformations For Practical Scalable Software Evolution,” In Proceedings Of The 26th International Conference On Software Engineering (Icse ’04), 2004, Pp. 625–634.
[45] E. J. Berk And C. S. Ananian, “Jlex: A Lexical Analyzer Generator For Java,” Online, Http://Www.Cs.Princeton.Edu/ ∼Appel/Modern/Java/Jlex/.
[46] S. Mccamant, “Large Single Compilation-Unit C Programs,”Jan2006,Http://People.Csail.Mit.Edu/Smcc/Projects/Single-File-Programs/.
[47] S. Drape, A. Majumdar, And C. Thomborson, “Slicing Aided Design Of Obfuscating Transforms,” In 6th Ieee/Acis International Conference On Computer And Information Science (Icis ’07). Los Alamitos, Ca, Usa: Ieee Computer Society, 2007, Pp. 1019–1024.
[48] H. Tamada, M. Nakamura, A. Monden, And K. Ichi Matsumoto,“Introducing Dynamic Name Resolution Mechanism For Obfuscating System-Defined Names In Programs,” In Proc. Iasted International Conference On Software Engineering (Iasted Se 2008, 598-074), February 2008, Pp. 125–130.
[49] S. T. Chow, Y. Gu, And H. J. Johnson, “Tamper Resistant Software Encoding,” Jan. 11 2005, Us Patent 6,842,862.
[50] D. Knuth, The Art Of Computer Programming, Volume Two, Seminumerical Algorithms. Addison-Wesley, 1998.
[51] S. Chow, P. Eisen, H. Johnson, And P. C. Van Oorschot, “A Whitebox Des Implementation For Drm Applications,” In Digital Rights Management. Springer, 2003, Pp. 1–15.
[52] C. Linn And S. Debray, “Obfuscation Of Executable Code To Improve Resistance To Static Disassembly,” In Proceedings Of The 10th Acm Conference On Computer And Communications Security (Ccs 2003). Acm, 2003, Pp. 290–299.
[53] Y. Kanzaki, A. Monden, M. Nakamura, And K.-I. Matsumoto, “Exploiting Self-Modification Mechanism For Program Protection,” In Proceedings Of The 27th Annual International Computer Software And Applications Conference (Compsac 2003). Ieee, 2003, Pp. 170–179.
[54] K. M. A. Alzarooni, “Malware Variant Detection,” Ph.D. Dissertation,Unversity College London, 2012.
[55] H. Il Lim, H. Park, S. Choi, And T. Han, “A Method For Detecting The Theft Of Java Programs Through Analysis Of

The Control Flow Information," Information And Software Technology, Vol. 51,No. 9, Pp. 1338–1350, 2009. [Online]. Available: Http://Www.Sciencedirect.Com/Science/Article/Pii/S0950584909000469

[56] Y. Mahmood, Z. Pervez, S. Sarwar, And H. Ahmed, "Similarity Level Method Based Static Software Birthmarks," In High Capacity Optical Networks And Enabling Technologies, 2008. Honet 2008.International Symposium On, Nov 2008, Pp. 205–210.

[57] M. Egele, C. Kruegel, E. Kirda, H. Yin, And D. Song, "Dynamic Spyware Analysis," In Usenix Annual Technical Conference. Usenix, 2007, Pp. 233–246.

[58] L. Cavallaro, P. Saxena, And R. Sekar, "On The Limits Of Information Flow Techniques For Malware Analysis And Containment," In Proceedings Of The 5th International Conference On Detection Of Intrusions And Malware, And Vulnerability Assessment (Dimva '08), 2008, Pp.143–163.

[59]Oreans Technologies, "Code Virtualizer," Http://Www.Oreans.Com/Codevirtualizer.Php.