Revisiting the Challenges and Opportunities in Software Plagiarism Detection

Xi Xu * †, Ming Fan *†, Ang Jia *†, Yin Wang *†, Zheng Yan ‡§, Qinghua Zheng *†, and Ting liu *†
*Key Laboratory of Intelligent Networks and Network Security, Ministry of Education, China
†Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, China
‡ State Key Lab on Integrated Services Networks, School of Cyber Engineering, Xidian University, China
§ Department of Communications and Networking, Aalto University, Finland
xx19960325@stu.xjtu.edu.cn; mingfan@mail.xjtu.edu.cn; jiaang@stu.xjtu.edu.cn;
wy0724@stu.xjtu.edu.cn; zyan@xidian.edu.cn; qhzheng@xjtu.edu.cn; tingliu@mail.xjtu.edu.cn

***Abstract*—Software plagiarism seriously impedes the healthy development of open source software. To fight against code obfuscation and inherent non-determinism of thread scheduling applied against software plagiarism detection, we proposed a new dynamic birthmark called *DYnamic Key Instruction Sequence (DYKIS)* and a framework called *Thread-oblivious dynamic Birthmark (TOB)* for the purpose of reviving the existing birthmarks and a thread-aware dynamic birthmark called *Thread-related System call Birthmark (TreSB)*. Though many approaches have been proposed for software plagiarism detection, they are still limited to satisfy the following highly desired requirements: the applicability to handle binary, the capability to detect partial plagiarism, the resiliency to code obfuscation, the interpretability on detection results, and the scalability to process large-scale software. In this position paper, we discuss and outline the research opportunities and challenges in the field of software plagiarism detection in order to stimulate brilliant innovations and direct our future research efforts.**

***Index Terms*—software plagiarism detection, software birthmark, source code similarity, binary code similarity**

I. Published Work

Software plagiarism plagues software industry. The problem is exacerbated by the growing popularity of open source software, whose licenses typically grant the recipient of a piece of software extensive rights to modify and redistribute that software, but not without restriction.

The goal of software plagiarism detection is to determine whether a defendant program contains plagiarism code by comparing it to a plaintiff program. In this paper, we use the plaintiff and the defendant to represent the original program and the program suspected of plagiarism, respectively. In the past few years, we focused on binary based software plagiarism detection and published three works. The descriptions of the works and main contributions are summarized as follows:

A. Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences

The burst of mature automated code obfuscation techniques and tools make the plagiarism detection daunting. Obfuscation is intended initially to protect software intellectual property but is now applied to hidden stolen code by plagiarists to avoid detection. To make the matter worse, such plagiarism is difficult to detect and prove because different compilation processes of the same source code naturally produce different binary code representations forming a kind of obfuscation. The dynamic approaches make it possible to focus more on the program semantics, rather than its syntax, for the execution sequences can clearly reflect how the inputs are processed.

Thus, to fight against code obfuscation, we propose a new dynamic birthmark extracted from the execution sequences, called *DYnamic Key Instruction Sequence (DYKIS)* [1]. Analyzing the complete execution sequences is neither effective (e.g., there are many instructions which are irrelevant to the program logic) nor efficient (e.g., a complete execution sequence of a program is usually extensive). Thus, we only keep key instructions that are inherent to program logic, and any change to such instructions leads to malfunction of copied code. To achieve the desired properties of key instructions, the instructions that both generate new values (value-updating instructions) and propagate taints from the input (input-correlated instructions) [2] are considered as key instructions. For the operands in the key instructions are sensitive to code obfuscation, an operand stripper is added to remove them. Since a particular sequence is an abstraction of the whole program, we use multiple executions and the *k-gram* algorithm [3] to compute the similarity of birthmarks.

Moreover, we implement a plagiarism detection tool *DYKIS-PD*, which is available at [4]. Our experiments on 342 versions of 28 different programs show that our approach is not only resilient to most semantics-preserving obfuscation techniques but also able to detect cross-platform plagiarism. Furthermore, the comparison with *SCSSB* [5] indicates that our approach achieves higher accuracy and superior performance with respect to any of the three performance metrics including *URC (Union of Resilience and Credibility)*, *F-Measure* and *MCC (Matthews Correlation Coefficient)*.

B. Reviving Sequential Program Birthmarking for Multithreaded Software Plagiarism Detection

With the popularity of multithreaded programs, a gap is created between the current software development practice and the software plagiarism detection technology, as the existing dynamic approaches are applicable only to the plagiarism on sequential programs.

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

To address the challenge, a systematic solution [6] is proposed. First, a new concept called thread-aware birthmarks is introduced to shield the influence of thread schedules on executions. Then, we propose a framework called *Thread-oblivious dynamic Birthmark (TOB)*, that revives existing techniques so they can be applied to detect plagiarism of multithreaded programs.

Given a multithreaded execution trace, the trace is first projected to multiple thread slices, each of which belongs to an individual thread. Then, the thread slice birthmarks are extracted from the thread slices applying the existing birthmark generate techniques. Finally, individual thread slice birthmarks are combined into a thread-oblivious birthmark for the multithreaded program by using *Slice Aggregation (SA)* or *Slice Set (SS)*.

Furthermore, we have implemented a set of tools collectively called *TOB based Plagiarism Detection tool (TOB-PD)* by applying *TOB* to three existing representative dynamic birthmarks, including *SCSSB* [5], *DYKIS* [1], and *JB* [7]. The source code of *TOB-PD* are publicly available at [8].

We have conducted extensive experiments on 418 versions of 35 different multithreaded programs. Our empirical study shows that *TOB-PD* is highly effective in detecting multithreaded plagiarism and is resilient to most state-of-the-art semantics-preserving obfuscation techniques.

C. Exploiting Thread-Related System Calls for Plagiarism Detection of Multithreaded Programs

The inherent non-determinism of thread scheduling severely impacts plagiarism based on dynamic analysis. Despite thread interleavings are complex, there exist characteristics or rules that ensure the correct execution under the chaos. We found some system calls that govern thread synchronization, priority setting, thread initiating, and disposing. They can enforce thread scheduling rather than being affected. They are also essential to the semantics and correct executions of a multithreaded program. We call them thread-related system calls and believe they form a favorable basis for generating thread-aware birthmarks.

Based on analyzing Linux system calls, we treat 65 system calls as thread-related. They accomplish tasks, including thread and process management (such as creation, join and termination, capability setting and getting), thread synchronization, signal manipulating, as well as thread and process priority setting.

Based on the above discussions, to address the challenge of plagiarism detection of multithreaded programs, we propose a thread-aware dynamic birthmark called *Thread-related System call Birthmark (TreSB)* [9]. For it is difficult to compare the thread-related system call sequences across multiple runs directly, the *k-gram* algorithm [3] is adopted to bound the sequences with a length k window. Moreover, a birthmark tool based on *TreSB* is implemented.

Our extensive experiments on a publicly available benchmark [10] consisting of 234 versions of 35 different multithreaded programs show that *TreSB* is resilient to most state-of-the-art semantics-preserving obfuscation techniques implemented in the best commercial and academic tools. In addition, a comparison of our method against two recently proposed thread-aware birthmarks $SCSSB_{SA}$ and $SCSSB_{SS}$ [11] shows that TreSB outperforms both of them in terms of *URC*, *F-Measure* and *MCC*.

D. Contribution

1. We propose a new dynamic birthmark called *DYKIS*, which is obfuscation-resilient and platform-independent, to enrich the birthmark-based plagiarism detection family.

2. We propose a framework called *TOB* that revives existing techniques to address the challenge of nondeterministic thread interleavings so they can be applied to detect plagiarism of multithreaded programs.

3. We propose a thread-aware dynamic birthmark called *TreSB* that can effectively detect plagiarism of multithreaded programs.

4. In the domain of software plagiarism detection, our three works were widely cited by top conferences and top journals [12], [13], and many favorable comments were made upon the good resilience to code obfuscation techniques and the capability to work on multithreaded programs.

II. Recent State of the Art and Practice

Although our proposed approaches demonstrate fairly good performance on constructed datasets, unfortunately, their practical feasibility in real-world scenarios has not been well evaluated due to several realistic requirements. For the application of existing approaches in practical software plagiarism detection, it is difficult for the plaintiff to file a software plagiarism lawsuit since it needs to provide clear evidence to prove that its software is illegally used by the defendant, which violates the current copyright laws.

Specifically, the analysis objects are always in the form of binaries since the source code of the defendant program is typically unavailable in real world. Therefore, the evidence should be extracted from the binaries that are hard for humans to understand. Note that most existing approaches only provide a similarity score between the plaintiff program and defendant program, which cannot be used as evidence. In our opinion, we should explain "where" and "why" we declare plagiarism between the plaintiff program and defendant program.

To better understand the actual need of software plagiarism detection, we identify the following five requirements:

- **R1.** *The applicability to handle binary.* The approach should have the ability to analyze binaries since the source code of the defendant program is generally difficult to obtain [14].
- **R2.** *The capability to detect partial plagiarism.* The defendant program may only steal the core code that constitutes a small portion of the plaintiff program, which is problematic to perform a comparison at a coarser granularity [15]. For example, the approaches that only calculate a similarity score for two whole programs would be unsound to detect the partial plagiarism.

- **R3.** *The resiliency to advanced code obfuscation.* The mature obfuscation techniques would make the detection harder since they can easily change the structure of the code while preserving the code semantics [16]. For example, different compilation options are always used as a type of obfuscation techniques to change the assembly code, resulting in a low similarity score.
- **R4.** *The interpretability of detection results.* The detection results of existing approaches are usually reported in the form of similarity scores (e.g., [17], [12]), which cannot be used as evidence in law. In other words, the results only weigh the degree of similarity between two programs rather than pinpointing on the exact cause.
- **R5.** *The scalability to process large-scale software.* Software plagiarism is always detected in a limit number of suspicious programs in existing approaches. However, in reality, the number of suspicious programs is much bigger than those in the experiments of existing approaches, thus making the scalability to process large-scale software becomes a critical need [15].

With respect to the above requirements, we summarize and compare the existing plagiarism detection approaches in Table I by applying the following criteria:

- ●: The approach satisfies the requirement.
- ◐: The approach satisfies the part of the requirement.
- ○: The approach does not satisfy the requirement.

TABLE I: The Evaluation of Existing Software Plagiarism Detection Approaches

Approach	R1	R2	R3	R4	R5
Moss [18]	○	●	○	●	●
JPlag [19]	○	●	○	●	●
AST [20]	○	●	◐	○	◐
GPLAG [21]	○	●	●	○	○
SKB [3]	●	●	○	○	●
SWKB [22]	●	●	○	○	●
SFB [23]	●	●	◐	○	◐
WSPB [17]	●	●	◐	○	◐
Cop [16]	●	●	●	○	○
DYKIS [1]	●	○	●	○	○
TOB [6]	●	○	●	○	○
TreSB [9]	●	○	●	○	○
LoPD [12]	●	○	●	○	○

According to the results presented in Table I, we observe that none existing approach can meet all the five requirements, which motivates us to enhance the software plagiarism techniques in the future.

For **R1**, the source code based approaches (e.g., *Moss* [18], *JPlag* [19], *AST* [20], and *GPLAG* [21]) cannot satisfy since they require the source code of the defendant program.

For **R2**, most dynamic analysis binary based approaches, (e.g., *DYKIS* [1], *TOB* [6], *TreSB* [9], and *LoPD* [12]) would fail because they detect whole program plagiarism and only generate a similarity score. The approaches based on static analysis (e.g., *SKB* [3], *SWKB* [22], *SFB* [23] and *WSPB* [17]) can satisfy **R2**, because static analysis make it possible to be applied at different granularities, such as basic blocks, functions, which can be detected locally.

For **R3**, *Moss* [18] and *JPlag* [19] fail to satisfy because they rely on token matching, which would be unsound for the code obfuscation such as junk code insertion and statement reordering. **R3** is not satisfied in *SKB* [3] and *SWKB* [22], because they ignore the program syntax and semantic information. *AST* [20] based on abstract syntax tree only partially satisfy **R3**, as a result of failing to handle advanced code obfuscation, for example, instruction rearrangement, statement splitting. *SFB* [23] and *WSPB* [17] also only partially satisfy **R3**, because they are vulnerable to control flow obfuscation. *GPLAG* [21] which relies the analysis on *Program Dependency Graphs (PDGs)* can perform better robustness against advanced semantics-preserving code obfuscation techniques and satisfies **R3**. By combining rigorous program semantics with longest common subsequence based fuzzy matching, *Cop* [16] is obfuscation-resilient and can satisfy **R3**. Since the dynamic approaches can concentrate more on the program semantics, *DYKIS* [1], *TOB* [6], *TreSB* [9], and *LoPD* [12] meet **R3** quite well.

For **R4**, *Moss* [18] and *JPlag* [19] satisfy well. In addition to providing the similarity scores, *Moss* [18] and *JPlag* [19] show the code fragments in defendant program that are matched to the plaintiff program to support the detection results. Thus, the detection results of *Moss* [18] and *JPlag* [19] have good interpretability. Unfortunately, there is no binary based approach considering the interpretability of detection results.

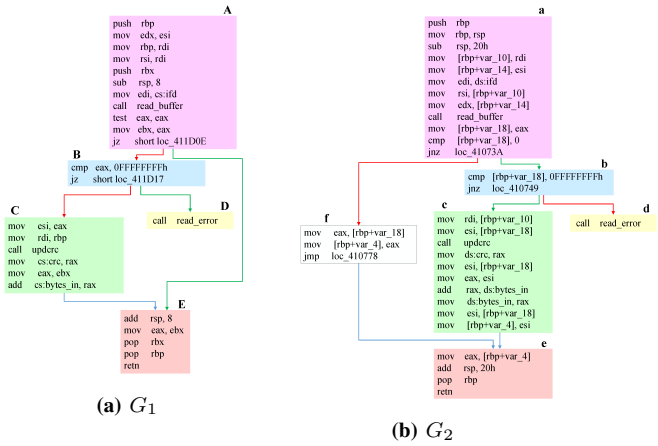
Finally, **R5** is a real challenge for most graph-based approaches. For example, *Cop* [16] is time consuming since the graph isomorphism is a NP-hard problem. In addition, symbolic execution combined with theorem proving is not scalable due to the explosion of state space.

Except for the above approaches, there are many related works in other domains [24], [25], [26], [27], such as clone detection and bug detection. However, clone detection (e.g., [28], [29]) assumes the availability of source code and minimal code obfuscation. Bug search (e.g., [30], [31]) is based on binary code, and it does not consider obfuscation in general. Meanwhile, the results of these approaches are ill-suited as plagiarism evidence because of the interpretability problem. In a sense, the results of plagiarism detection are much more elaborate than conventional these approaches.

III. Discussion

Capitalizing on the performed literature review, we find although the researches of software plagiarism detection have made significant progress, we still suffer from great limitations in terms of the resiliency to code obfuscation, the interpretability of detection results, and the scalability to process large-scale software, in the practice of software plagiarism detection.

In this section, we revisit the challenges by reviewing related work and pointing their weakness on satisfying the expected requirements. Moreover, we explore new research opportunities, and identify possible research directions that can be followed to address the challenges effectively in future.



(a) G_1 **(b)** G_2

Fig. 1: The CFG of *file_read*

A. The Resiliency to Advanced Code Obfuscation

Although many efforts are being made to fight against code obfuscation, we are still facing many challenges. Specifically, the compilation processes of a program is a "grey box", the input is source code, and the output is the corresponding binary program. The property provides the cradle for the hiding of plagiarism since the source code of the defendant program is generally not public. The plagiarist can modify any of the compilation parameters (i.e., compiler vendor, compiler version, optimization level, target architecture, and platform) in the "grey box" to produce a different binary for the plagiarized code. Especially if the changed target architecture uses a different instruction set, the produced binary can be totally different. Plagiarism could also be hidden with deliberate use of existing mature obfuscation tools.

Taking the function *file_read* in the single-file lossless data compression utility *gzip* for example, Fig. 2(a) and Fig. 2(b) present the control flow graphs of the same source code compiled with *gcc 4.8* [32] and *clang 3.9* [33], respectively. For presentation purposes, the basic blocks in G_1 and G_2 are displayed in colours and marked with letters. Matching blocks are in the same colour, and blocks in white only exist in G_2. We use capital letters for basic blocks in G_1 and corresponding lowercase letters for matching blocks in G_2.

The two graphs demonstrate that with different compilation, the control flow graphs are quite different, even for the same function. The differences between the two graphs include the structural differences (e.g., basic block adding or deleting) and syntactical differences (e.g., operand substitution, instruction adding or deleting). The best way to tackle this problem is to identify the similar program based on the program semantics abstracted from the obfuscated code. However, it is challenging to extract program semantics from binary. Thanks to the availability of the source code of the plaintiff program, the problem gets much easier when the binary-to-binary comparison can be turned into the source-to-binary comparison. The more program semantics can be easily extracted from source code and potentially improve accuracy compared with binary. We can first select the unique and stable part from the source code of the plaintiff program. Then we generate its corresponding binary signature, and extract features from the signature to compare with the defendant program. Predictably, fighting against code obfuscation should be paid much attention to in the future study.

B. The Interpretability of Detection Results

As mentioned above, there is a blindingly obvious and yet long-ignored problem in software plagiarism detection, the interpretability of detection results.

The detection results of the most existing plagiarism detection approaches are usually reported in the form of similarity scores between the plaintiff and defendant program. Furthermore, there is no evidence presented to explain and describe the similarity between the plaintiff and defendant program. To this extent, these approaches are only a program for idea demonstration, but not a practical tool.

Therefore, when proposing a software plagiarism detection approach, the consideration of the interpretability of detection result as an additional design driver can improve its practicality. Interpretability can act as insurance that guarantees the accuracy and reliability of the detection results.

Moss [18] and *JPlag* [19] provide the matching code to support the decision. Although these approaches are limited in practice for the requirement of source code and weakness in the presence of obfuscation techniques, we can scale that idea into other detection approaches to improve significantly the interpretability of detection results.

Based on providing the matching code, identifying the core codes, checking their semantics equivalence, and reproducing the transformation process from the plaintiff program to the defendant program can make the evidence more compelling. Although this problem is full of challenges and difficulties, it must be solved before applying in practical detection.

C. The Scalability to Process Large-Scale Software

Although many approaches are proposed to improve the detection accuracy, the scalability to process large-scale software plagiarism remains insufficiently addressed to date. For example, the semantics-based approach *COP* [16] took an hour in the comparison *thttpd* and *sthttpd*, and half a day in the comparison between *Gecko* and *Firefox*.

There are many existing software plagiarism detection approaches based on the graph feature (e.g., *Control Flow Graph (CFG)*) which can depict the program semantics. However, the similarity calculation between the graph-based features is limited by the efficiency of existing graph matching approaches, which is known as an NP-hard problem. These approaches are not effective to process large-scale software plagiarism. Thus, the further development of scalability is needed to improve the detection approaches.

Qian et al. [34] propose a real-time bug search engine, *Genius*, by converting the CFGs into high-level numeric vectors. Eschweiler et al. [35] implement a bug search phototype *discovRE*, which allows for a fast similarity

comparison between binary functions based on a set of robust numeric features.

In addition to these, deep learning is well combined with binary analysis because of high accuracy and high efficiency. Xu et al. [36] propose a neural network-based vulnerability detection method, *Gemini*, to generate the embedding, based on the *Attributed Control Flow Graph (ACFG)* of each binary function. Even though these approaches have achieved a lot in scalability, they cannot be used in plagiarism detection directly due to some limitations. *Genius* [34] and *discovRE* [35] rely on graphs which would cause the loss of concrete instruction-level semantics. *Gemini* [36] labels each basic block with a set of manually selected attributes, which would ignore important semantic information. Moreover, it overlooks the importance of the order of the nodes.

Many challenges still remain, but we believe these approaches can be combined with software plagiarism detection to improve scalability.

IV. Conclusion

In this paper, we first summarize our three previous publications on software plagiarism detection. Then, we identify five highly desired requirements by seriously considering the demand of practical application. Employing the specified requirements, we further evaluate the existing approaches and discuss their advantages and limitations. Finally, we point out the main research challenges and propose a number of potential research directions to motivate our future work.

Acknowledgment

This work was supported by National Key R&D Program of China (2016YFB1000903), National Natural Science Foundation of China (61902306, 61632015, 61772408, U1766215, 61721002, 61532015, 61833015), Ministry of Education Innovation Research Team (IRT_17R86), China postdoctoral science special foundation (No. 2019TQ0251), and Project of China Knowledge Centre for Engineering Science and Technology.

References

[1] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE TSE*, vol. 41, no. 12, pp. 1217–1235, 2015.

[2] X. Zhang, Z. Yang, Q. Zheng, Y. Hao, P. Liu, and T. Liu, "Tell you a definite answer: Whether your data is tainted during thread scheduling," *IEEE TSE*, 2018.

[3] G. Myles and C. S. Collberg, "K-gram based software birthmarks," in *Proc.SAC*, 2005.

[4] "DYKIS-PD," http://labs.xjtudlc.com/labs/wlaq/dbpd/site/, 2015.

[5] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Proc.ACSAC*. IEEE, 2009, pp. 149–158.

[6] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE TSE*, vol. 44, no. 5, pp. 491–511, 2017.

[7] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in *Proc.ASE*. ACM, 2007, pp. 274–283.

[8] "TOB-PD," http://labs.xjtudlc.com/labs/wlaq/TAB-PD/site/, 2017.

[9] Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang, and Z. Yang, "Exploiting thread-related system calls for plagiarism detection of multithreaded programs," *JSS*, vol. 119, pp. 136–148.

[10] "Benchmark," http://labs.xjtudlc.com/labs/wlaq/TAB-PD/site/download.html, 2017.

[11] Z. Tian, Q. Zheng, T. Liu, M. Fan, X. Zhang, and Z. Yang, "Plagiarism detection for multithreaded software based on thread-aware software birthmarks," in *Proc.ICPC*. ACM, 2014, pp. 304–313.

[12] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *TRELIAB*, vol. 65, no. 4, pp. 1647–1664, 2016.

[13] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE TIFS*, vol. 12, no. 8, pp. 1772–1785, 2017.

[14] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proc.ICSE*. IEEE, 2011, pp. 756–765.

[15] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proc.CCS*. ACM, 2009, pp. 280–290.

[16] 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 *Proc.FSE*. ACM, 2014, pp. 389–400.

[17] H.-i. Lim, H. Park, S. Choi, and T. Han, "Detecting theft of java applications via a static birthmark based on weighted stack patterns," *IEICE TRANSACTIONS on Information and Systems*, vol. 91, no. 9, pp. 2323–2332, 2008.

[18] "Moss: A system for detecting software plagiarism," http://theory.stanford.edu/~aiken/moss/, 2013.

[19] L. Prechelt, G. Malpohl, and M. Philippsen, *JPlag: Finding plagiarisms among a set of programs*. Citeseer, 2000.

[20] L. Zhang, D. Liu, Y. Li, and M. Zhong, "Ast-based plagiarism detection method," in *IOT*. Springer, 2012, pp. 611–618.

[21] C. Liu, C. Chen, J. Han, and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis," in *Proc.KDD*. ACM, 2006, pp. 872–881.

[22] X. Xie, F. Liu, B. Lu, and L. Chen, "A software birthmark based on weighted k-gram," in *Proc.ICIS"*, vol. 1. IEEE, 2010, pp. 400–405.

[23] H.-i. Lim and T. Han, "Analyzing stack flows to compare java programs," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 2, pp. 565–576, 2012.

[24] M. Fan, X. Luo, J. Liu, M. Wang, C. Nong, Q. Zheng, and T. Liu, "Graph embedding based familial analysis of android malware using unsupervised learning," in *Proc.ICSE*. IEEE, 2019, pp. 771–782.

[25] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *Proc. ISSRE*, 2016.

[26] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE TIFS*, vol. 13, no. 8, pp. 1890–1905, 2018.

[27] M. Fan, X. Luo, J. Liu, C. Nong, Q. Zheng, and T. Liu, "Ctdroid: leveraging a corpus of technical blogs for android malware analysis," *IEEE Transactions on Reliability*, 2019.

[28] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *Proc.ICSE*. IEEE, 2011, pp. 301–310.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc.ICSE*. IEEE, 2007, pp. 96–105.

[30] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 392–404.

[31] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc.ACSAC*. ACM, 2014, pp. 406–415.

[32] "gcc," https://gcc.gnu.org, 2019.

[33] "clang," https://clang.llvm.org, 2019.

[34] F. Qian, R. Zhou, C. Xu, C. Yao, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," 2016.

[35] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *Proc.NDSS*, 2016.

[36] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc.CCS*. ACM, 2017, pp. 363–376.