JSidentify: A Hybrid Framework for Detecting Plagiarism Among JavaScript Code in Online Mini Games

Qun Xia
kelvinhhxia@tencent.com
Tencent Inc., China

Zhongzhu Zhou*
zhouzhzh8@mail2.sysu.edu.cn
Sun Yat-sen University, China

Zhihao Li
edgarlli@tencent.com
Tencent Inc., China

Bin Xu, Wei Zou, Zishun Chen, Huafeng Ma, Gangqiang Liang
addyxu,nofreezou,zishunchen@tencent.com
Tencent Inc., China

Haochuan Lu
luhc17@fudan.edu.cn
Fudan University, China

Shiyu Guo
whiteguo@tencent.com
Tencent Inc., China

Ting Xiong, Yuetang Deng
candyxiong,yuetangdeng@tencent.com
Tencent Inc., China

Tao Xie†
taoxie@pku.edu.cn
Peking University, China

ABSTRACT

Online mini games are lightweight game apps, typically implemented in JavaScript (JS), that run inside another host mobile app (such as WeChat, Baidu, and Alipay). These mini games do not need to be downloaded or upgraded through an app store, making it possible for one host mobile app to perform the aggregated services of many apps. Hundreds of millions of users play tens of thousands of mini games, which make a great profit, and consequently are popular targets of plagiarism. In cases of plagiarism, deeply obfuscated code cloned from the original code often embodies malicious code segments and copyright infringements, posing great challenges for existing plagiarism detection tools. To address these challenges, in this paper, we design and implement JSidentify, a hybrid framework to detect plagiarism among online mini games. JSidentify includes three techniques based on different levels of code abstraction. JSidentify applies the included techniques in the constructed priority list one by one to reduce overall detection time. Our evaluation results show that JSidentify outperforms other existing related state-of-the-art approaches and achieves the best precision and recall with affordable detection time when detecting plagiarism among online mini games and clones among general JS programs. Our deployment experience of JSidentify also shows that JSidentify is indispensable in the daily operations of online mini games in WeChat.

KEYWORDS

Plagiarism Detection, Online Mini Games, JavaScript, Clone Detection

ACM Reference Format:
Qun Xia, Zhongzhu Zhou, Zhihao Li, Bin Xu, Wei Zou, Zishun Chen, Huafeng Ma, Gangqiang Liang, Haochuan Lu, Shiyu Guo, Ting Xiong, Yuetang Deng, and Tao Xie. 2020. JSidentify: A Hybrid Framework for Detecting Plagiarism Among JavaScript Code in Online Mini Games. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381352

1 INTRODUCTION

Online mini games are lightweight game apps, typically implemented in JavaScript (JS), that run inside another host mobile app (such as WeChat, Baidu, and Alipay). These mini games do not need to be downloaded or upgraded through an app store, making it possible for one host mobile app to perform the aggregated services of many apps. These online mini games are typically implemented in JS, which has been a very popular programming language [45] in recent years thanks to its high expressiveness and portability. Hundreds of millions of users play tens of thousands of mini games, which make a great profit, and consequently are popular targets of plagiarism. For example, almost tens of code plagiarism cases happen just in a day from the developers' submission of the original game program. Such plagiarism poses great threats to mobile security (given that the plagiarized code can embody malicious code segments) and intellectual property. In practice, it is highly critical for the platform of the host mobile app to compare the developers' submitted game program with the game programs from the existing repository of online mini games to detect plagiarism effectively and efficiently.

In practice, plagiarists (i.e., developers who conduct plagiarism) deeply obfuscate their plagiarism code cloned from the original code, resulting in their plagiarism code being a sophisticated type of code clones. In general, clones are broadly classified into two types: syntactic and semantic clones [42]. Syntactic clones can be further divided into Identical Clones (Type I), Renamed Clones (Type II), and Gapped Clones (Type III) [42]. In cases of plagiarism in

*The research done by this author was during his internship at Tencent Inc. His work is supported in part by the National Natural Science Foundation of China under Grant U1911201, Guangdong Special Support Program under Grant 2017TX04X148.
†The author is affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. His work is supported in part by NSF under grant no. CNS-1564274, CCF-1816615.

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7123-0/20/05…$15.00
https://doi.org/10.1145/3377813.3381352

online mini games, plagiarists often conduct the code modifications corresponding to the Type II and Type III clones (i.e., renaming and gapping) to obfuscate the plagiarism code. Furthermore, the plagiarists apply advanced obfuscation operations such as control flow flattening, nested function, and string array encoding [4]. We name the finally resulting plagiarism code as Type IV clones, which are similar to semantic clones. Type IV clones refactor standard code constructs such as control flow or function calls. Thus, it is often impossible even for human inspectors to reach a high-confidence verdict of plagiarism in code unless they comprehensively play all versions of online mini games, whose version quantity reaches more than 200,000 in WeChat as focused in our work, and compare their behavior similarity.

In cases of plagiarism among online mini games, deeply obfuscated code cloned from the original code often poses great challenges for existing plagiarism detection tools. The existing approaches of code clone detection typically resolve the problem of detecting duplicated code without intentional obfuscation [20]. Despite being applied for code plagiarism detection, these approaches cannot effectively handle obfuscation. The existing approaches for detecting code plagiarism or clone can be classified into six types [14, 41]: textual [47, 49, 50], lexical token based [5, 19, 32], Abstract Syntax Tree (AST) based [6, 34], Program Dependency Graph (PDG) based [13, 17, 20, 36, 39], metric based [24, 30, 40], and hybrid [48]. However, majority of textual and lexical-token-based approaches cannot detect Type III or IV clones. AST, PDG, metric-based, and hybrid approaches do not support JS, because of dynamic compilation features and restriction imposed by the online mini game engine. These approaches cannot detect Type IV clones effectively [16, 44] and suffer from scalability issues, being unable to handle the huge number of online mini games in WeChat's repository.

In the past two years of attempting to apply and adapt the existing approaches for being adopted in WeChat, we have first-hand observed their significant limitations for plagiarism detection among online mini games hosted by WeChat. In particular, we attempted to apply MOSS [47], PMD [8], and Simian [9], which support JS clone detection. However, their recall on JS code is lower than 5% in online mini games hosted by WeChat. In addition, previous studies [16, 30] suggest that MOSS is very coarse-grained and is not suitable for clone detection. We also attempted other approaches designed for JS (e.g., Jsinspect [6], jscpd [5]). They cannot detect Type IV or mixture of multiple clone types. Their recall, precision, and F1-Score are very poor. JSCD(safe) [20] and other PDG-based approaches can detect a few Type IV clones except those undergoing obfuscations of control flow flattening and nested function. Other metric-based (e.g., Heap-Based Software Theft Detection [17]), hybrid [48], and malware-detection approaches [19, 33] need to set very low values of similarity threshold, which can greatly compromise precision.

To address these limitations, in this paper, we propose a novel hybrid framework, named JSidentify. JSidentify includes both static and dynamic analyses for plagiarism detection integrated through a constructed priority list. To evaluate JSidentify and compare it against other related approaches [5, 6, 20, 47], we collect 400 mini games (including both plagiarism and non-plagiarism ones) from WeChat's repository along with general JS programs being obfuscated [43] to synthesize clones.

This paper makes the following main contributions:

- **Formulation of Plagiarism Detection in Online Mini Games.** According to the literature [20] and our investigation of numerous plagiarism cases among WeChat's online mini games, we define code clones undergoing advanced obfuscation operations as Type IV clones in the setting of online mini games, and analyze the limitations of the existing clone detection approaches in this setting.
- **Novel Framework.** We propose a novel framework named JSidentify, including (1) the Winnowing plus technique to improve the existing Winnowing technique [47], (2) the Sewing technique applied upon the dynamically compiled JS bytecode, and (3) the Scene Tree technique based on Scene Tree, an abstract representation of online mini games.
- **Evaluation Results.** Our evaluation results show that JSidentify outperforms other existing related approaches and achieves the best precision and recall within affordable detection time when detecting plagiarism among online mini games and clones among general JS programs.

In 2018, plagiarism games accounted for **21%** of online mini games in WeChat, a popular messenger app with over 1 billion monthly active users. Currently there are already more than 200,000 game versions for tens of thousands of online mini games in WeChat's repository. After we deploy JSidentify, JSidentify has conducted 1.5 billion comparisons (between a submitted game and a repository game version) during its deployment period of about 11 months. JSidentify can scale to comparing a mini game (averagely 10,000 LOC) against more than 200,000 game versions in WeChat's repository, to effectively conduct plagiarism detection in averagely 6 minutes. Thanks to JSidentify, the proportion of plagiarism games has dropped to **4%** so far, indicating that JSidentify is indispensable in the daily operations of online mini games in WeChat.

2 BACKGROUND

WeChat's online mini games are hosted through JS on a messaging app, named WeChat. More and more people enjoy online mini games with friends. In WeChat's online mini games, plagiarism games account for as high as 21% in 2018. Without proper mechanisms to guard against plagiarism games, it has an abominable impact on fair, profitable, and healthy game ecological environment in WeChat; therefore, plagiarism detection is an urgent challenge [23].

However, based on our experience of technology adoption, the existing plagiarism detection approaches and supporting tools [20] being applied on WeChat's mini games demonstrate poor effectiveness and efficiency. We first apply multiple existing open-source tools to attempt to find plagiarism code based on comparing a submission program and each program from the repository of online mini games. Most of tools that support JS plagiarism detection check on the text, lexicon, and abstract syntax tree (AST) [27, 38, 44]. These tools cannot find those plagiarism games cloned from the original games as clones. We then attempt to adapt existing approaches [20, 22]. For example, we adapt approaches based on the program dependency graph (PDG) or birthmark [20, 25, 48] to compute similarity. The recall improves but the efficiency is so poor, because these approaches dynamically run a game, costing a lot of detection time, and these approaches achieve low precision.

We next introduce WeChat's online mini games and clone types in WeChat's repository of online mini games. We then discuss the related work on plagiarism detection.

2.1 WeChat's Online Mini Games

Mini games are lightweight game apps that run inside another host mobile app (such as WeChat, Baidu, and Alipay). A user just needs to open an online mini game and play it without downloading or installing it. Hundreds of millions of users play tens of thousands of mini games, which make a great profit. During our recent efforts of detecting source code with copyright infringement, we find that there exists substantial plagiarism in WeChat's online mini games submitted by third-party developers.

WeChat's online mini games are lightweight games that run on WeChat. They are driven by the JS engine within WeChat. This JS engine is adapted from the JS V8 engine [11], a dynamic JS compiler. The JS engine generates bytecode based on the Ignition interpreter in V8. The JS engine achieves high speed via just-in-time (JIT) [26] compilation so that bytecode is generated at runtime. Because of this "lazy" mode, existing dynamic analysis approaches that require to access all the bytecode before runtime are not applicable here. Upon the JS engine, there is a game engine (e.g., Cocos2d [1], LayaAir [7]).

2.2 Code Obfuscator Types in WeChat's Repository of Online Mini Games

JS plagiarism code in WeChat's repository of online mini games is usually obtained by performing the following types of changes/obfuscations, corresponding to four common types of code clones [35, 44, 46]:

- (Type I) Identical code except change in whitespace or comments.
- (Type II) Change in name, cases, replacement of identifiers with expressions.
- (Type III) Addition or deletion of redundant code fragments, gapping.
- (Type IV) Advanced obfuscation operations such as control flow flattening, nested function, and string array encoding [4].

The first type is the same code fragment, except for whitespace changes (which may also be layout changes) and comments (Type I). The second type is the structurally/syntactically identical code fragments except the variations in identifiers, literals, types, layout, and comments (Type II). In the third type, statements can be changed, added, or removed in addition to variations in identifiers, literals, types, layout, and comments (Type III). We summarize from WeChat's cases of online mini games and classify into Type IV clones those cases resulted from advanced obfuscation operations such as control flow flattening, nested function, and string array encoding [4]. The existing approaches that support JS cannot be adapted to detect Type IV clones here.

A plagiarist may simultaneously apply multiple obfuscation operations and may apply further changes to a key part of the program. All these factors contribute to the challenge of detecting plagiarism code in WeChat's repository of online mini games.

2.3 Related Work

Related approaches [6, 8, 9, 47] based on textual or lexical information consider the source code as a text and try to find equal substrings or compare similar characteristics such as a sequence of tokens. Among tools that can detect duplication of JS code, MOSS [47], Simian [9], and PMD [8] are widely studied in the literature [44]. MOSS summarizes a program as its n-gram token distribution. But previous studies [16] have shown that the recall of MOSS on Java is only 10%. In addition, an extensive comparison of clone detection tools on 17 Java and C systems has shown that both Simian [9] and PMD [8] are good at detecting identical clones but not renamed and gapped clones; both text-based and token-based approaches are not suitable in detecting Type III or IV clones unless they set a low threshold value that can cause poor precision.

Abstract Syntax Trees (ASTs) capture structural aspects of a program. Some detection tools [6] compute similarity through ASTs. Jsinspect [6], a tool for detecting JS code clones, uses the ASTs of the parsed code. Jsinspect compares the similarity of nodes and depths of ASTs. But it cannot detect Type III/IV clones. A previous study [43] shows that AST-based tools work well for Type I/II clones and detect Type III clones with a low threshold value. The precision of AST-based tools is higher than textual/lexical-based tools [20].

Metric-based and PDG-based approaches are widely used for clone detection. Based on metrics computed from ASTs and PDGs [39], metric-based approaches can achieve great efficiency and yet low precision [20]. However, while we apply a metric-based approach on WeChat's repository of online mini games, the approach still spends 1.3 hours to compare a program with more than 200,000 game program versions. A heap-graph-based approach [17] measures similarity by heap graphs' isomorphism. JSCD(safe) [20], a JS Clone Detector integrated in SAFE, uses Deckard and the LSH algorithm (a metric-based algorithm). JSCD(safe) and the heap-graph-based approach can detect Type III or IV clones but cannot detect plagiarism involving multiple types of clones. In addition, the detection problem is an NP-hard problem. The execution time is too long even with approximate algorithms.

3 JSIDENTIFY FRAMEWORK

Our JSidentify framework includes three components (Filter, Scheduler, and Judger) and takes as input a mini-game program submitted by developers. The Filter component in the framework preprocesses the submitted program to produce its simplified code by simplifying variable names and identifiers, compressing literals, removing whitespaces and comments, and removing dead code through static analysis. Then the Scheduler component applies the included detection techniques in the priority list one by one on the submitted program, by comparing its simplified code with the simplified code of each repository program (from the given repository of online mini games). The Scheduler stops until a technique detects the submitted program to be plagiarism or until all techniques are applied to compare the submitted program against each repository program from the repository and no plagiarism is detected by any technique. After being applied, a detection technique produces similarity metric values between the submitted program and each repository program. The Judger component then uses the similarity threshold value specified for this detection technique

to decide whether the submitted program plagiarizes an existing repository program.

JSidentify includes multiple techniques in the priority list constructed via the mechanism described in Section 3.4. Sections 3.1-3.3 describe three techniques already included in JSidentify: the Winnowing Plus, Sewing, and Scene Tree techniques, respectively.

3.1 Winnowing Plus Technique

The Winnowing technique [47] adopted by MOSS achieves great efficiency but low precision for plagiarism detection. For example, MOSS on Java achieves only 10% precision [16], and our preliminary study shows that MOSS on JS cannot detect Type II, III, IV, or hybrid clones. To improve precision while enjoying great efficiency, we improve the Winnowing technique to produce our Winnowing Plus technique to pre-detect plagiarism (being placed on top of the priority list).

In particular, we design multiple pre-processing techniques to address some limitations of the Winnowing technique as incorporated in MOSS. The use of MOSS requires the provision of values for size parameters k (the length of character subsequence to form a word) and w (the number of words to form a sliding window of words) as follows. MOSS first treats each program under comparison as a string s by removing spaces in the program. From s, MOSS produces L as a list of words each of which consists of s's consecutive character subsequence of length k and starts from s's ith character where $1 <= i <= (|s| - k + 1)$. From the word list L, MOSS produces sliding windows (of words) each of which is of w words and starts from L's ith word where $1 <= i <= (|L| - w + 1)$. MOSS then samples a representative word from each sliding window of words to form a representative-word vector. Based on the features consisting of the representative-word vector, MOSS computes the similarity across the programs under comparison to detect plagiarism. However, the Winnowing technique performs poorly with code undergoing either dead-code injection or advanced obfuscation operations such as control flow flattening, nested function, and string array encoding [4]. Thus, we design multiple pre-processing techniques to make the technique more robust against obfuscations:

- Deobfuscate the detected obfuscated code.
- Normalize variable names to $v0$, $v1$, ... according to the occurrence order in the code.
- Normalize string constants and names of classes that are instantiated at least once to $sc0$, $sc1$, ... and $cn0$, $cn1$, ..., respectively, according to the occurrence order in the code.
- Eliminate semantics-lacking characters (blank, comments) and remove dead functions (determined by analyzing function call relationship using the AST).

These four pre-processing techniques in our Winnowing Plus technique improve the recall of the Winnowing technique by improving its robustness to Type I, II clones, and a part of Type IV clones. In general, the Winnowing Plus technique incurs the shortest detection time among all the techniques currently within JSidentify and is placed on top of the priority list.

3.2 Sewing Technique

Type III and IV clones are challenging to detect. Dead-code injection, control flow flattening, nested function, and string array encoding cannot be detected by text-based or other basic approaches [8, 9, 47]. Thus, we design our Sewing technique to compute similarity of bytecode inspired by Needle [31]. For JS code, general approaches explore detection in a high level of program representation (e.g., ASTs and PDGs) [6, 20]. Multiple approaches attempt to detect clones in the heap [17] at runtime or in the machine code [29]. But a JS game cannot be easily characterized by its heap data, and collecting heap data at runtime can incur high cost for a JS game. Approaches based on machine code do not support JS. Thus, we utilize the JS engine within WeChat and design our Sewing technique for JS bytecode. The Ignition interpreter in the JS engine dynamically generates bytecode, and can collect runtime calling relationship from the bytecode.

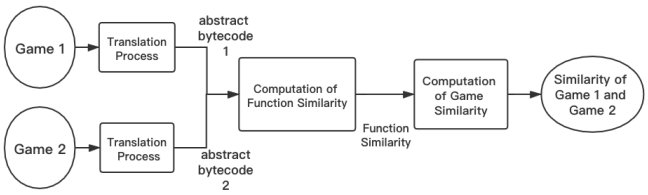

Figure 1: The overview of the Sewing technique

To compare two games Games 1 and 2, the Sewing technique works in three steps. First, it translates each function of Games 1 and 2 in JS code into abstract bytecode. Second, it computes the similarity across two abstract-bytecode functions pairwise across Games 1 and 2. Third, it computes the similarity between Games 1 and 2 based on the Bytecode Function Graph (BFG) constructed based on similarity of abstract-bytecode functions pairwise across Games 1 and 2. Figure 1 shows the process of the Sewing technique.

3.2.1 Translation Process. As discussed in Section 2.1, generating bytecode is a lazy process. We leverage a testing tool[1] adapted from Google Monkey [2] to automatically run an online mini game and generate its bytecode, with an example bytecode as follows:

```
1        StackCheck
2        CreateClosure [0], [0], 0
3        Star r1
4        LdaGlobal [1], [1]
5        Star r2
6        Call UndefinedReceiver1 r1, r2, [3]
7        Star r0
8        Return
```

According to IR [3] for JS bytecode from Ignition, the abstraction module in the Sewing technique converts the preceding bytecode to "0000000100001 10100100101001000". Figure 2 shows the translation process for resulting in the abstract bytecode.

3.2.2 Computation of Function Similarity. To compute similarity between two functions, the Sewing technique borrows ideas from the Needle [31] and Winnowing [47] techniques. In particular,

[1] In WeChat's application setting, we set the running time of the testing tool as 5 minutes. Note that the testing tool needs to be applied on each repository game or each submitted game only once, even when each repository game will be compared against many submitted games over time.

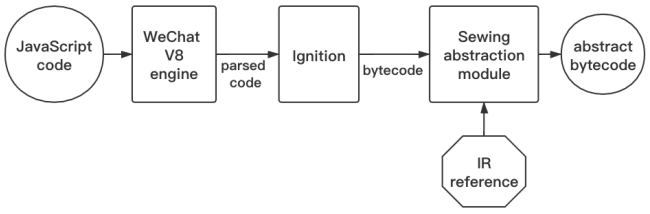

Figure 2: Translation process in the Sewing technique

the Sewing technique uses the Longest Common Subsequence (LCS) [15] and k-gram [47] algorithms.

Consider that the length of a JS function f, denoted as $|f|$, is n, indicating that f has n lines of bytecode $t_1, ..., t_n$. A window within f is its code portion of the specified size s including consecutive lines of bytecode. From f, we extract a sequence of sliding windows $s_1, ..., s_{n-k+1}$, each of which, s_i, contains k consecutive lines of bytecode $t_i, t_{i+1}, ..., t_{i+k-1}$. Based on this notion, let us consider two functions f_i and f_j where $|f_i| <= |f_j|$. We define the similarity between the pair of functions f_i and f_j as

$\delta(f_i, f_j) = max_{l \in 1,2,...,|f_j|-k+1} |LCS(f_i, s_l)|$

Intuitively, $\delta(f_i, f_j)$ denotes the length of f_i's longest common subsequence (LCS) that is contained in the sliding windows extracted from f_j. A larger similarity value indicates that more equivalent instructions are similar.

Algorithm 1 shows the steps for computing similarity across two functions f_i and f_j where $|f_i| <= |f_j|$. Note that in WeChat's application setting, we set k (the window size as the third parameter of Algorithm 1) as $|f_i|$ (which is $min(|f_i|, |f_j|)$) to find which part of f_j is cloned with f_i.

Algorithm 1 Computation of function similarity

```
1:  function COMPUTATION_OF_FUNCTION_SIMILARITY(f_i, f_j, k)
2:      set S = [[t_1, ..., t_k], ..., [t_{|f_j|-k+1}, ...t_{|f_j|}]]
3:      set similarity = MIN_INT
4:      set q = 0
5:      repeat
6:          set q = q + 1
7:          LCS_result = |LCS(S[q], f_i)|
8:          similarity = max(similarity, LCS_result)
9:      until q = |f_j| - k + 1
10:     return similarity
11: end function
```

Time complexity of calculating LCS (with binary search [28]) in two strings whose maximum length is z is $O(z * log(z))$. We need to compare f_i against $(n-k+1)$ sliding windows (i.e., k-grams) where n is the number of lines in f_j. So time complexity of computing function similarity is $O((n-k) * n * log(n))$. In Section 2.1, we know that bytecode is collected for only those executed functions. At runtime of a typical mini game, almost 1,000 functions can be executed. After we remove duplicate functions and library functions among the executed functions during the data pre-processing, there are about 100 to 500 unique functions. We define the number of unique functions as p. Time complexity of computing similarity between all pairs of functions across two mini games is $O(p^2 * (n-k) * n * log(n))$ where $p, n, k <= 500$. In WeChat's parallel computing setting, our algorithm typically has the runtime cost of minutes when being applied on all functions from two mini games.

3.2.3 Computation of Game Similarity. Based on $\delta(f_i, f_j)$ for each pair of f_i and f_j from two games G1 and G2, respectively, we compute the similarity between G1 and G2 via a weighted flow network named Bytecode Function Graph (BFG) where each edge (a,b) is labelled with its capacity value (denoted as $capacity(a, b)$) and weight value (denoted as $weight(a, b)$. To construct the BFG, we first construct two virtual nodes s and t to represent G1 and G2, respectively. For each function f_i in G1 and each function f_j in G2, we construct nodes in the BFG to represent these functions. To simplify the illustration, we next use f_i and f_j to refer to their corresponding nodes in the BFG, respectively.

For each function node f_i in G1, we construct an edge (s, f_i) with $capacity(s, f_i) = |f_i|$ and $weight(s, f_i) = 0$. For each function node f_j in G2, we construct an edge (f_j, t) with $capacity(f_j, t) = |f_j|$ and $weight(f_j, t) = 0$. For each node f_i in G1 and each node f_j in G2, we construct an edge (f_i, f_j) with $capacity(f_i, f_j) = \delta(f_i, f_j)$ and $weight(f_i, f_j) = \frac{1}{1+e^{-\alpha * \delta(f_i, f_j) + \beta}}$ as the sensitivity of embedding derived with a logistic function. In WeChat's application setting, we configure $\alpha = 2$ and $\beta = 0.5$. The higher possibility that a function can be embedded into another (i.e., similar to each other), the weight is closer to 1.

The similarity between G1 and G2 is defined as

$\delta(G1, G2) = \frac{MaximumWeightFlow(BFG)}{\sum_{i \in 1,2,...,N} |f_i|}$

where BFG represents the BFG constructed from G1 and G2 as described earlier, N represents the number of functions in G1, f_i represents each function in G1, and the algorithm for the $MaximumWeightFlow$ function can be found elsewhere [12].

We can know that a large $\delta(G1, G2)$ indicates that more of G1's lines of bytecode can be embedded into G2. We set an empirical threshold ϵ to decide whether $\delta(G1, G2)$ represents plagiarism. The time complexity of $MaximumWeightFlow$ in our setting is $O(max(capacity) * N * M * log(N + M))$, where N represents the number of unique executed functions in G1, and M represents the number of unique executed functions in G2. As discussed earlier, we can know that $max(capacity) <= 500$ and mostly $N, M <= 500$ for a mini game. In WeChat's parallel computing setting, the total time of computing both function similarity and game similarity typically has the runtime cost of minutes when being applied on two mini games.

3.3 Scene Tree Technique

In a game-engine-based implementation, a scene is defined as a User Interface (UI) in a period of time of running a mini game, and game code is designed in units of scenes. Plagiarism code may undergo multi-type obfuscations (as described in Section 2.2), but these obfuscations do not change the scenes substantially. Thus, we design the Scene Tree technique to detect plagiarism. First, we define features of a scene as a Scene Tree. The tree describes the runtime data in the scene. Each node in the tree represents runtime data (e.g., positions and invoked methods) of a component such as a UI controller, sprite, and action. We next describe the two

steps in the Scene Tree technique: tree construction and similarity computation.

3.3.1 Tree Construction. The step of tree construction is to use the mini-game engine (described in Section 2.1) to gather runtime data for a mini game, and use the data to construct Scene Trees. In particular, when running the mini game automatically with a testing tool[2] adapted from Google Monkey [2], we gather the runtime data from the game engine to construct Scene Trees. The Scene Tree constructed for each scene contains multiple nodes. The root node represents the entrance of the scene. At runtime, the engine generates runtime data for components in the scene. In the Scene Tree, we construct a node for each component in the scene to store its data, e.g., positions and invoked methods. Then we set these nodes as the children of the root node. When there is a derivative component c based on a component p, the node for c becomes the child node of the node for p in the Scene Tree.

3.3.2 Similarity Computation. Various tree comparison algorithms have been proposed [21, 30]. In consideration of particularity of the scene data, we propose a customized comparison technique for Scene Trees as follows.

We first define the edit distance between two nodes n_1, n_2 as $\delta(n_1, n_2) = max(a_1, a_2) - c$ where c is the number of the same data across two nodes, a_1 is the number of the data in n_1, and a_2 is the number of the data in n_2.

Then we compute the edit distance between two Scene Trees T_1 and T_2 as $\delta(T1, T2) = \sum_{i,j} \delta(n_i, n_j)$ where i and j are pairs of the corresponding nodes in the same tree depth and positions across T_1 and T_2. Basically, to prevent precision loss, we just compute the edit distance of $T1$ and $T2$ by summing all the edit distances between the corresponding nodes in the same tree depth and position between T_1 and T_2.

Finally, we compute the similarity of G1 and G2 as follows. We define e and f as the number of the Scene Trees in Games G1 and G2, respectively. We consider each Scene Tree as a node and construct a bipartite graph [37] of G1 and G2. We set an empirical threshold ϵ such that we construct an edge between two nodes pairwise across G1 and G2 only if the two nodes' edit distance is $<= \epsilon$. We use the Hungarian algorithm [37] to find the matching number of the bipartite graph of G1 and G2, denoted as m. We then compute the similarity of G1 and G2 as $similarity(G1, G2) = m/min(e, f)$. Figure 3 shows the process of constructing Scene Trees and computing similarity in JSidentify.

The time complexity of comparing two games G1 and G2 is $O(q * w * e * f + (e + f) * h)$, where q is the maximum number of nodes in a Scene Tree from G1 and G2, w is the maximum number of data in a Scene Tree node, e and f are the number of the Scene Trees in two games G1 and G2, respectively, and h is the number of edges in the bipartite graph. For a typical mini game, $q <= 500$, $w <= 20$, and there are only tens of scenes (i.e., Scene Trees). In WeChat's parallel computing setting, our algorithm for computing similarity of two games typically has the runtime cost of seconds.

[2]In WeChat's application setting, we set the running time of the testing tool as 5 minutes. Note that the testing tool needs to be applied on each repository game or each submitted game only once, even when each repository game will be compared against many submitted games over time.

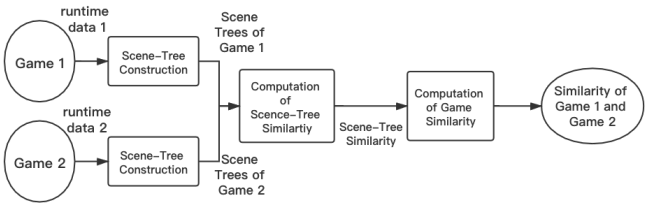

Figure 3: Overview of Scene Tree Construction and Comparison in JSidentify

3.4 Constructing Priority List of Applying Techniques

A key consideration in our hybrid framework is to deal with a huge number of programs including substantially obfuscated code. In addition, along the way of defending against plagiarism, more and more techniques are proposed and added in our framework in addition to the preceding three techniques (Sections 3.1, 3.2, and 3.3). If we apply all the techniques (even in a parallel computing setting), the running time can be long while consuming expensive computing resources.

To improve efficiency while ensuring high effectiveness, we detect plagiarism by applying the included techniques one at a time in a priority list, and each applied technique compares the submitted program with repository programs in a parallel computing setting. In particular, the Scheduler (a component of JSidentify) applies a technique X to attain the similarity between the submitted program Y and repository programs. Based on the threshold value specified for this technique, the Judger (a component of JSidentify) determines whether the submitted program plagiarizes any existing repository program or not. If not, the Scheduler then applies X's subsequent technique in the priority list. The Scheduler stops until the Judger considers the submitted program to be plagiarism or until all techniques are applied to compare the submitted program against each repository program from the repository and no plagiarism is determined by the Judger.

We construct the order in the priority list based on each included technique's detection gain, defined similar to gain computed by information entropy in a general priority list. To compute the detection gain for each included technique, we apply the technique on the same large number of pairs of affirmatory plagiarism code denoted as CC below, sampled from online mini games in WeChat's repository. Then we compute average similarity and average running time (of applying the technique for detection). We define the technique's detection gain as follows:

$$DetectionGain = \frac{AvgSimilarity}{AvgRunningTime} \text{ where}$$

$$AvgSimilarity = \frac{\sum_{i=1}^{M*(M-1)} Similarity(CC_i)}{M*(M-1)}$$

$$AvgRunningTime = \frac{\sum_{i=1}^{M*(M-1)} RunningTime(CC_i)}{M*(M-1)}$$

where $CC =< (C1, C2), (C1, C3), ..., (C_i, C_j)..., (C_M, C_M - 1) >$ $i = 1, ..., M, j = 1, ..M - 1, i \neq j$

A larger detection gain of a technique indicates that this technique tends to achieve higher detection recall and/or higher detection efficiency, and this technique should be placed closer to the top of the priority list. Note that the gain-computation time is

just one-time effort, not needed when applying a technique on a submitted program.

4 EVALUATIONS

We evaluate our JSidentify framework (we refer to JSidentify as an approach in the rest of this section for ease of presentation) on both plagiarism detection for JS online mini games and clone detection for general JS code. In this section, we first introduce evaluation setup and then discuss evaluation results.

4.1 Evaluation Setup

4.1.1 Evaluation Metrics. In our evaluations, we use the metrics of recall, precision, and F1-score to assess JSidentify and other related approaches under comparison. The recall metric is commonly used in Information Retrieval (IR) research; in our setting, the recall is the number of detected real plagiarism-case pairs divided by the total number of real plagiarism-case pairs. In particular, we use T to denote the number of plagiarism-case pairs detected by an approach, F to denote the number of detected plagiarism-case pairs that are actually not real ones, TP to denote the number of real plagiarism-case pairs. The recall and precision metrics are defined as below:

$Recall = \frac{T-F}{TP}$

$Precision = \frac{T-F}{T}$

We also use the F1-score metric to measure the overall effectiveness of an approach:

$F1\text{-}score = \frac{2*Precision*Recall}{Precision+Recall}$

4.1.2 Related Approaches Under Comparison. Due to the high evaluation cost, we first collect an initial set of related state-of-the-art approaches and compare their effectiveness on clone detection, and then select the most effective ones for being used as related approaches under further comparison in our evaluations.

In particular, our initial set of five related state-of-the-art approaches include MOSS [47], JSCD(safe) [20], Jsinspect [6], jscpd [5], and the JS version for an approach of detecting Android malware [18, 19] based on PDGs and User Interfaces (UIs); we refer to the last approach as JSMalware in the rest of this paper. To set a desirable threshold value for each approach, we apply each approach on samples from WeChat's repository of online mini games by varying similarity threshold values to observe how the precision, recall, and F1-score vary. For each approach, we set the threshold value using which the approach can achieve the highest F1-score overall (if the highest F1-score can be achieved using multiple threshold values, among these threshold values we set the threshold value using which the approach can achieve the highest precision).

For the initial screening of the related approaches, we choose an open-source JS project named math.js (https://mathjs.org/), which includes 103,334 lines of code with most of its functional code included in a JS file. We obfuscate it to synthesize its clones with each of the six obfuscation operations (as listed in Columns 2-7 of Table 1) provided by two obfuscation tools: obfuscator [4] and UglifyJS [10]. To synthesize a clone, we apply twice each of the six obfuscation operations on randomly chosen places in math.js. In particular, for each of the six obfuscation operations, we first apply obfuscator with this operation on a randomly chosen place in math.js, and upon the resulting math.js, we further apply UglifyJS with the same operation on one more randomly chosen place to produce the final clone.

Table 1 shows the similarity levels between the original math.js and its synthesized clones, as reported by the five related approaches along with JSidentify. In general, the similarity levels reported by the five related approaches are undesirably much lower than the ones reported by JSidentify. JSidentify reports high similarity levels (4 greater than 90% and 2 around 80%) for all the 6 synthesized clones, and JSCD(safe) and Jsinspect report >= 80% similarity levels for 4 and 3 out of the 6 synthesized clones, respectively, while the remaining MOSS, jscpd, and JSMalware perform undesirably, which report >= 80% similarity levels for 0, 1, and 1 clones out of the 6 synthesized clones, respectively.

According to the results, we select three approaches: JSCD(safe), Jsinspect, and MOSS, as our final set of related approaches under further comparison in our evaluations. JSCD(safe) and Jsinspect report relatively high similarity levels, much higher than the ones reported by the remaining related approaches. Jsinspect uses an algorithm based on ASTs, and JSCD(safe) uses one based on PDGs. To compare against a representative related approach using a text-based algorithm, we also include MOSS as our additional related approach under comparison, even given that it demonstrates low effectiveness.

4.1.3 Evaluation Datasets and Setting. For the evaluation on detecting plagiarism among WeChat online mini games, we randomly select 100 pairs of plagiarism games in WeChat's repository of plagiarism games, and then select 200 non-plagiarism games[3], each of which has been manually confirmed not to be involved in a plagiarism case. Finally, we conduct all pair combinations among the preceding 400 games (i.e., pairing each of the 400 games with each of the remaining 399 games) to form the final $400 * 399$ pairs as our evaluation dataset.

For the evaluation on detecting clones among general JS code, we choose 10 well-known, classic JS projects with different sizes and different functionalities in GitHub such as JS Math and JS JSON. These projects range from approximately 1K to 10M LOC. For each project, we randomly obfuscate multiple functions in the project with hybrid plagiarism types (i.e., applying all six obfuscation operations) discussed in Section 2.2 to synthesize the project's clone. The 10 pairs of the original project and its synthesized project clone constitute our evaluation dataset.

We conduct our evaluations in WeChat's detection environment with Intel(R) Xeon(R) Gold 61xx CPU and 16GB RAM. We repeat our evaluations at least three times and the variation of the observed evaluation results is less than 5%. For each approach's basic parameter setting, we set default values for general parameters (e.g., $\alpha = 2$ and $\beta = 0.5$ in the Sewing technique).

4.2 Plagiarism Detection for Online Mini Games

We compare JSidentify with the three related approaches MOSS, Jsinspect, and JSCD(safe) by applying them on the evaluation dataset described in Section 4.1.3. Because the similarity threshold values

[3]Some of these 200 games also show relatively high similarity between each other.

Table 1: Similarity levels (between math.js and its clones synthesized with each of six obfuscation operations) reported by JSidentify and five related approaches

Detectors	Identifier Modifications	Dead Code Injection	Control Flow Flattening	String Splitting	Nested Function	String Array Encoding
JSidentify	99.1%	99.7%	83.5%	96.5%	79.6%	93.2%
MOSS	77.5%	25.4%	6.0%	0.0%	23.3%	15.1%
jscpd	94.4%	41.2%	9.2%	0.0%	5.1%	0.0%
Jsinspect	95.7%	93.2%	30.5%	87.1%	5.1%	25.1%
JSCD(safe)	96.8%	99.9%	64.7%	95.3%	17.4%	96.4%
JSMalware	89.5%	45.3%	55.1%	7.0%	3.5%	1.3%

Table 2: The best F1-score result (across all the threshold values) and average detection time of JSidentify and three related approaches in plagiarism detection for online mini games

Detectors	Recall	Precision	F1-score	Avg Detection time
JSidentify	100.0%	99.1%	99.54 %	13.6s/pair
Jsinspect	68.0%	78.0%	72.65 %	5.7s/pair
JSCD(safe)	61.0%	89.0%	72.18 %	81.3s/pair
MOSS	100.0%	50.0%	66.77 %	1.2s/pair

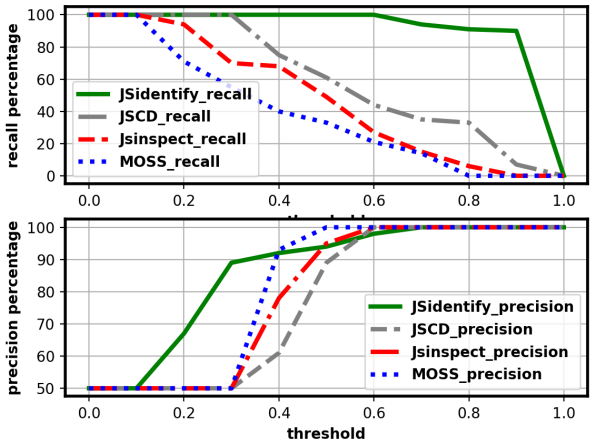

Figure 4: Recall and precision with different threshold values in plagiarism detection by JSidentify and three related approaches for online mini games

used by different approaches to determine plagiarism are different, we compute the metric values of precision, recall, and F-1 score with respect to the similarity threshold value ranging from 0 to 1 with interval of 0.1. Figure 4 shows the evaluation results. The results show that JSidentify achieves the best effectiveness in precision and recall no matter what threshold value is set. MOSS also achieves great precision but suffers from the lowest recall. When the threshold value is set as low, each approach achieves high recall. However, low precision achieved by an approach makes it not applicable in practice. When the threshold value is increased, the recall achieved by an approach tends to become lower; however, being able to handle Type IV clones, JSidentify still achieves relatively high recall when the threshold value is relatively high. Column 4 of Table 2 shows the best F-1 score results (and their corresponding recall and precision in Columns 2 and 3) across all the threshold values. In summary, JSidentify achieves the best evaluation results.

We also measure average detection time for each approach, as listed in the last column of Table 2. The results show that JSCD(safe) spends on average 81.3 seconds per game pair to detect plagiarism, whereas JSidentify spends on average 13.6 seconds per game pair. Despite being higher than the average detection time of MOSS and Jsinspect, JSidentify's average detection time is still reasonable in WeChat's application setting.

4.3 Clone Detection for General JS Code

Because there are no game engines in general JS code, this evaluation assesses JSidentify without including the Scene Tree technique. Considering the influence of threshold values, for an approach assessed in this evaluation, we adopt the threshold value used to achieve the best F1-score result (as shown in Table 2).

Because we set the resulting clone project (synthesized via obfuscation operations) as the plagiarism case in the 10 project pairs, we can ensure the ground-truth cases. Table 3 shows the detection results of the 10 project pairs. The results show that JSidentify achieves the best effectiveness results. Moreover, JSCD(safe) reaches 80.0% recall and precision in clone detection for general JS code. JSCD(safe) achieves great effectiveness for online mini games too as shown in Table 2. However, Jsinspect achieves only 40.0% recall and 57.1% F1-score in clone detection for general JS code, in contrast to its high effectiveness in plagiarism detection for online mini games.

Figure 5 shows the detection time of JSidentify and the three related approaches. The results show that JSidentify spends affordable detection time with scalability up to 10 million LOC. But when dealing with 10 million LOC, the detection time of JSidentify is not affordable in practice. Still, this result demonstrates JSidentify's very good scalability already. In contrast, JSCD(safe) cannot produce results even for 10 million LOC, because it adopts a graph isomorphism algorithm with high time complexity. Compared to other approaches, JSidentify does not spend the shortest detection

Table 3: Recall, precision, and F1-score of JSidentify and three related approaches in clone detection for general JS code

Detector	Recall	Precision	F1-score
JSidentify	100.0%	100.0%	100.0%
JSCD(safe)	80.0%	80.0%	80.0%
Jsinspect	40.0%	100.0%	57.1%
MOSSM	20.0%	100.0%	33.3%

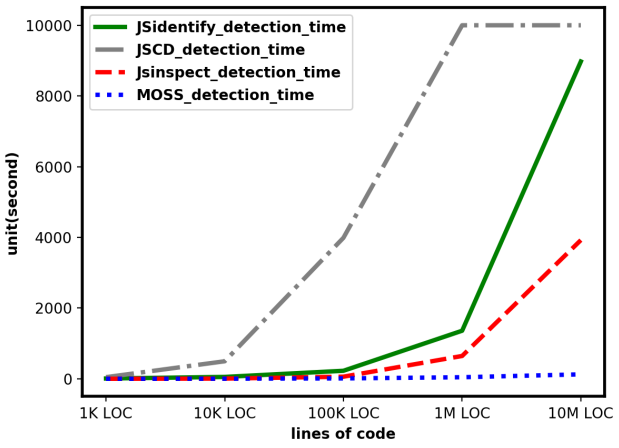

Figure 5: Detection time of JSidentify and three related approaches in projects with different LOCs (the detection time's reaching 10000 seconds indicates that the approach is unable to finish the detection within 10000 seconds)

time, but it can detect various plagiarism cases. While MOSS spends the shortest detection time, it focuses on only limited textual clones.

4.4 Summary of Evaluation Results

JSidentify achieves the best recall, precision, and F1-score for general or mini-game JS plagiarism. The detection time of JSidentify is not the shortest, longer than the detection time of Jsinspect and MOSS, because of substantial time required by some included techniques such as the Sewing technique in JSidentify. In summary, JSidentify achieves the best effectiveness (the best recall, precision, and F1-score) for detecting JS plagiarism, with affordable detection time.

5 DISCUSSION

In 2018, plagiarism games accounted for **21%** of online mini games in WeChat, a popular messenger app with over 1 billion monthly active users. Currently there are already more than 200,000 game versions for tens of thousands of online mini games in WeChat's repository. After we deploy JSidentify, JSidentify has conducted 1.5 billion comparisons (between a submitted game and a repository game version) during its deployment period of about 11 months. JSidentify can scale to comparing a mini game (averagely 10,000 LOC) against more than 200,000 game versions in WeChat's repository, to effectively conduct plagiarism detection in averagely 6 minutes. Thanks to JSidentify, the proportion of plagiarism games has dropped to **4%** so far, indicating that JSidentify is indispensable in the daily operations of online mini games in WeChat.

However, given that plagiarists can continue to obfuscate code for further attempting to escape the detection by JSidentify, detecting plagiarism code in JS is still a long-standing challenge for our ongoing and future work, with three example aspects listed below.

First, JSidentify integrates various techniques only loosely by prioritizing the applications of these techniques in order to reduce the detection time. To achieve higher effectiveness and efficiency, we plan to design techniques to tightly integrate various techniques [48] by getting the best of these techniques without suffering from their respective weaknesses.

Second, JSidentify achieves low precision when an online mini game under detection includes code from third-party libraries such as code from the game engine. In such situations, the similarity between the online mini game and a repository game can be high. Existing approaches cannot handle these situations effectively. To address this limitation, we maintain a collection of code from third-party libraries (typically used by online mini games) collected from various sources. Based on code matching against this collection during pre-processing, we can remove or tag common code from third-party libraries and ignore it during plagiarism detection.

Third, JSidentify incurs higher detection time when the number of online mini games in WeChat's repository increases over time. With more and more online mini games being uploaded to WeChat, massive data pose a major system challenge. Over time, the existing storage system has suffered from a rapid decline of file I/O write/read speed, and if the number of online mini games continues to rise, the system may spend a lot of time to fetch a program from the storage system. To address scalability issues faced by the storage system, we plan to adopt a new file system with high scalability.

In addition, for an upcoming program under detection, before going through JSidentify's detection with relatively high cost, we plan to first apply a much faster but less robust approach to efficiently match the program against our collection of programs with plagiarism, e.g., based on simple string hashing of the program code. If the program under detection is a duplicate of an already detected program with plagiarism, this faster approach can succeed and we can skip the application of JSidentify.

6 CONCLUSION

In this paper, we have presented JSidentify, a novel framework for detecting plagiarism cases in WeChat's online mini games. We have illustrated three JSidentify techniques proposed based on different levels of code abstraction. JSidentify applies the included techniques in the constructed priority list one by one to reduce overall detection time. Our evaluation results show that JSidentify outperforms other existing related approaches and achieves the best precision and recall within affordable detection time when detecting plagiarism among online mini games and clones among general JS programs. Our deployment experience of JSidentify has

also shown that JSidentify is indispensable in the daily operations of online mini games in WeChat.

REFERENCES

[1] 2020. Cocos2d game engine. http://www.cocos2d.org/.
[2] 2020. Google Monkey. https://developer.android.com/studio/test/monkey.
[3] 2020. Ignition compiler. https://v8.dev/blog/ignition-interpreter.
[4] 2020. JavaScript Obfuscator Tool. https://obfuscator.io/.
[5] 2020. jscpd. https://github.com/kucherenko/jscpd.
[6] 2020. Jsinspect. https://github.com/danielstjules/jsinspect.
[7] 2020. LayaAir game engine. https://github.com/layabox/LayaAir.
[8] 2020. PMD's copy/paste detector. http://pmd.sourceforge.net/pmd-5.0.5/cpd-usage.html.
[9] 2020. Simian–similarity analyser. http://www.harukizaemon.com/simian/index.html.
[10] 2020. UglifyJS. https://github.com/mishoo/UglifyJS.
[11] 2020. V8 compiler. https://v8.dev/.
[12] RK Ahujia, Thomas L Magnanti, and James B Orlin. 1993. Network Flows: Theory, Algorithms, and Applications. *New Jersey: Pentice-Hall* (1993).
[13] Arutyun Avetisyan, Shamil Kurmangaleev, Sevak Sargsyan, Mariam Arutunian, and Andrey Belevantsev. 2015. LLVM-based code clone detection framework. In *Proc. Computer Science and Information Technologies (CSIT)*. 100–104.
[14] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering (TSE)* 33, 9 (2007), 577–591.
[15] Lasse Bergroth, Harri Hakonen, and Timo Raita. 2000. A survey of longest common subsequence algorithms. In *Proc. International Symposium on String Processing and Information Retrieval (SPIRE)*. 39–48.
[16] Elizabeth Burd and John Bailey. 2002. Evaluating clone detection tools for use during preventative maintenance. In *Proc. IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. 36–43.
[17] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. 2012. Heap graph based software theft detection. *IEEE Transactions on Information Forensics and Security (TIFS)* 8, 1 (2012), 101–110.
[18] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proc. International Conference on Software Engineering (ICSE)*. 175–186.
[19] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale. In *Proc. USENIX Security Symposium (USENIX Security)*. 659–674.
[20] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. 2016. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering (ESE)* 21, 2 (2016), 517–564.
[21] Hung Chim and Xiaotie Deng. 2007. A new suffix tree similarity measure for document clustering. In *In Proc. International Conference on World Wide Web (WWW)*. 121–130.
[22] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-aware malware detection. In *Proc. IEEE Symposium on Security and Privacy (S&P)*. 32–46.
[23] Georgina Cosma and Mike Joy. 2008. Towards a definition of source-code plagiarism. *IEEE Transactions on Education (TOE)* 51, 2 (2008), 195–200.
[24] Neil Davey, Paul Barson, Simon Field, Ray Frank, and D Tansley. 1995. The development of a software clone detector. *International Journal of Applied Software Technology (IJAST)* (1995).
[25] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*. 109–118.
[26] Arnaldo Hernandez and Arnoldo Hernandez. 1989. *Just-In-Time manufacturing: A practical approach.* Prentice Hall Englewood Cliffs, NJ.
[27] Rosco Hill and Joe Rideout. 2004. Automatic method completion. In *Proc. IEEE International Conference on Automated Software Engineering (ASE)*. 228–235.
[28] Costas S Iliopoulos and M Sohel Rahman. 2008. New efficient algorithms for the LCS and constrained LCS problems. *Information Processing Letters (IRL)* 106, 1 (2008), 13–18.
[29] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. 2011. Value-based program characterization and its application to software plagiarism detection. In *Proc. International Conference on Software Engineering(ICSE)*. 756–765.
[30] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. International Conference on Software Engineering (ICSE)*. 96–105.
[31] Yanyan Jiang and Chang Xu. 2018. Needle: Detecting code plagiarism on student submissions. In *In Proc. ACM Turing Celebration Conference - China (TURC)*. 27–32.
[32] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering (TSE)* 28, 7 (2002), 654–670.
[33] Saruhan Karademir, Thomas Dean, and Sylvain Leblanc. 2013. Using clone detection to find malware in Acrobat files. In *Proc. Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*. 70–80.
[34] Raminder Kaur and Satwinder Singh. 2014. Clone detection in software source code using operational similarity of statements. *SIGSOFT Softw. Eng. Notes* 39, 3 (June 2014), 1–5.
[35] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proc. European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 187–196.
[36] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proc. Working Conference on Reverse Engineering (WCRE)*. 301–309.
[37] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval Research Logistics (NRL)* 2, 1-2 (1955), 83–97.
[38] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. In *Proc. International Conference on Software Engineering (ICSE)*. 310–320.
[39] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proc. ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 872–881.
[40] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*. 244–253.
[41] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology (IST)* 55, 7 (2013), 1165–1199.
[42] Chanchal K Roy and James R Cordy. 2007. *A survey on software clone detection research.* Technical Report 2007-541. School of Computing Queen's University at Kingston, Ontario, Canada.
[43] Chanchal K Roy and James R Cordy. 2009. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proc. IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICST)*. 157–166.
[44] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming (SCP)* 74, 7 (2009), 470–495.
[45] Sukyoung Ryu and Sukyoung Ryu. 2017. Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–34.
[46] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling code clone detection to big code. In *Proc. International Conference on Software Engineering (ICSE)*. 1157–1168.
[47] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: Local algorithms for document fingerprinting. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 76–85.
[48] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. 2018. A detection framework for semantic code clones and obfuscated code. *Expert Systems with Applications (ESA)* 97 (2018), 405–420.
[49] Leonardo Humberto Silva, Daniel Hovadick, Marco Tulio Valente, Alexandre Bergel, Nicolas Anquetil, and Anne Etien. 2016. JSClassFinder: A tool to detect class-like structures in JavaScript. *arXiv preprint arXiv:1602.05891* (2016).
[50] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: A token based large-gap clone detector. In *Proc. International Conference on Software Engineering (ICSE)*. 1066–1077.