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

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

Zhongzhu Zhou*  
zhouzhzhi@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

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 JSDentify, a hybrid framework to detect plagiarism among online mini games. JSDentify includes three techniques based on different levels of code abstraction. JSDentify applies the included techniques in the constructed priority list one by one to reduce overall detection time. Our evaluation results show that JSDentify outperforms other related state-of-the-art approaches and achieves the best precision and recall with affordable detection time. Our evaluation results show that JSDentify is indispensable in the daily operations of online mini games in WeChat.

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE-SEIP ’20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7123-0/20/05...
https://doi.org/10.1145/3377813.3381352

**KEYWORDS**

Plagiarism Detection, Online Mini Games, JavaScript, Clone Detection

**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...
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., Jinspect [6], jsopd [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 messaging 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 program 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 JSSidentify, 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 hirnmark [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 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 sub-strings 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 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.

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]).

Our JSidentify framework includes three components (Filter, Scheduler, and Judge) 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 Judge 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 characteristic sequence 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 $i$th character where $1 \leq i \leq (|s| - k + 1)$. From the word list $L$, MOSS produces sliding windows (of words) each of which is of $w$ words and starts from $L$’s $i$th word where $1 \leq i \leq (|L| - w + 1)$. 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 $v0$, $v1$, ..., 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.

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:

```
StackCheck
CreateClosure [0], [0], 0
Star r1
LdaGlobal [1], [1]
Star r2
Call UndefinedReceiver1 r1, r2, [3]
Star r0
Return
```

According to IR [3] for JS bytecode from Ignition, the abstraction module in the Sewing technique converts the preceding bytecode to “00000001000001 1010010010010000”. 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,
Algorithm 1

1: function COMPUTATION_OF_FUNCTION_SIMILARITY($f_i, f_j, k$)
2: set $S = \{[t_1, ... , t_k], ... , [t_{|f_i|} - k + 1 , ... , t_{|f_j|}]\}$
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 $f_i$. So time complexity of computing function similarity is $O(n(n - k) * n * \log(n))$. 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 $p, n, k \ll 500$. 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 $weight(f_i, f_j) = \frac{1}{1 + e^{-\beta s(f_i, f_j)}}$ 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 $\delta(G1, G2) = \frac{\sum_{f_i \in \text{G1}, f_j \in \text{G2}} weight(f_i, f_j) \cdot similarity(f_i, f_j)}{\text{Size of G1} \cdot \text{Size of G2}}$ 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 $\epsilon$ to decide whether $\delta(G1, G2)$ represents plagiarism. The time complexity of MaximumWeightFlow in our setting is $O(max(capacity) \cdot N \cdot M \cdot \log(N + M))$, 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 $max(capacity) \ll 500$ and mostly $N, M \ll 500$ 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
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\(^3\) 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
\[
\delta(T_1, T_2) = \sum_{i,j} \delta(n_i, n_j)
\]
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 \(T_1\) and \(T_2\) 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 \(G_1\) and \(G_2\) as follows. We define \(e\) and \(f\) as the number of the Scene Trees in Games \(G_1\) and \(G_2\), respectively. We consider each Scene Tree as a node and construct a bipartite graph [37] of \(G_1\) and \(G_2\). We set an empirical threshold \(\epsilon\) such that we construct an edge between two nodes pairwise across \(G_1\) and \(G_2\) 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 \(G_1\) and \(G_2\), denoted as \(m\). We then compute the similarity of \(G_1\) and \(G_2\) as
\[
similarity(G_1, G_2) = m / \min(e, f).
\]

Figure 3 shows the process of constructing Scene Trees and computing similarity in JSidentify.

The time complexity of comparing two games \(G_1\) and \(G_2\) is \(O(q * w * e * f + (e + f) * h)\), where \(q\) is the maximum number of nodes in a Scene Tree from \(G_1\) and \(G_2\), \(w\) is the maximum number of data in a Scene Tree node, \(e\) and \(f\) are the number of the Scene Trees in two games \(G_1\) and \(G_2\), 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.

\[^{3}\text{In WeChat’s application setting, we set the running time of the testing tool as 5 minutes. Note that the testing tool needs to be applied on each repository game or each submitted game only once, even when each repository game will be compared against many submitted games over time.}\]
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:

\[
\text{Recall} = \frac{TP}{TP + F} \\
\text{Precision} = \frac{TP}{T}
\]

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

\[
F1\text{-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

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 \( \geq 80\% \) similarity levels for 4 and 3 out of the 6 synthesized clones, respectively, while the remaining MOSS, jscpd, and JSMalware perform undesirably, which report \( \geq 80\% \) 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\(^5\), 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

\(^5\) 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

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

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

<table>
<thead>
<tr>
<th>Detectors</th>
<th>Recall</th>
<th>Precision</th>
<th>F1-score</th>
<th>Avg Detection time</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSidentify</td>
<td>100.0%</td>
<td>99.1%</td>
<td>99.54%</td>
<td>13.6s/pair</td>
</tr>
<tr>
<td>Jssinspect</td>
<td>68.0%</td>
<td>78.0%</td>
<td>72.65%</td>
<td>5.7s/pair</td>
</tr>
<tr>
<td>JSCD(safe)</td>
<td>61.0%</td>
<td>89.0%</td>
<td>72.18%</td>
<td>81.3s/pair</td>
</tr>
<tr>
<td>MOSS</td>
<td>100.0%</td>
<td>50.0%</td>
<td>66.77%</td>
<td>1.2s/pair</td>
</tr>
</tbody>
</table>

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

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 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 Jssinspect, JSidentify’s average detection time is still reasonable in WeChat’s application setting.

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, Jssinspect 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

<table>
<thead>
<tr>
<th>Detector</th>
<th>Recall</th>
<th>Precision</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSidentify</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>JSCD(safe)</td>
<td>80.0%</td>
<td>80.0%</td>
<td>80.0%</td>
</tr>
<tr>
<td>Jsinspect</td>
<td>40.0%</td>
<td>100.0%</td>
<td>57.1%</td>
</tr>
<tr>
<td>MOSSM</td>
<td>20.0%</td>
<td>100.0%</td>
<td>33.3%</td>
</tr>
</tbody>
</table>

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)

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