Hindawi
Security and Communication Networks
Volume 2018, Article ID 7064131, 16 pages
https://doi.org/10.1155/2018/7064131

Research Article

Demadroid: Object Reference Graph-Based Malware Detection in Android

Huanran Wang, Hui He, and Weizhe Zhang

Department of Computer Science and Technology, Harbin Institute of Technology, 92 Xidazhi Street, Harbin, Heilongjiang 150001, China

Correspondence should be addressed to Hui He; hehui@hit.edu.cn and Weizhe Zhang; wzzhang@hit.edu.cn

Received 29 December 2017; Accepted 11 April 2018; Published 31 May 2018

Academic Editor: Dafang Zhang

Smartphone usage has been continuously increasing in recent years. In addition, Android devices are widely used in our daily life, becoming the most attractive target for hackers. Therefore, malware analysis of Android platform is in urgent demand. Static analysis and dynamic analysis methods are two classical approaches. However, they also have some drawbacks. Motivated by this, we present Demadroid, a framework to implement the detection of Android malware. We obtain the dynamic information to build Object Reference Graph and propose λ-VF2 algorithm for graph matching. Extensive experiments show that Demadroid can efficiently identify the malicious features of malware. Furthermore, the system can effectively resist obfuscated attacks and the variants of known malware to meet the demand for actual use.

1. Introduction

Android is a mobile operating system developed by Google, based on the Linux kernel, and designed primarily for touchscreen mobile devices such as smartphones and tablets [1]. On top of the kernel level, there are middleware, libraries, and APIs written in C programming language. And the kernel level is independent of other resources [2].

With the popularity of smartphones, the number of users of Android dramatically rises [3]. However, the popularity of Android also attracts the attention of malware, which has become an urgent threat to users [4]. According to the types of threats, malicious apps can be divided into at least six categories: abuse of value-added services software, advertising fraud software, data theft software, malicious downloading software, malicious decoding software, and spyware. Research from security company Trend Micro shows that the premium service abuse is the most common type. For example, text messages are sent from infected phones without the permission of users [5]. Android has become the hardest hit. However, Google engineers have argued that the malware and virus threat on Android is being exaggerated by security companies for commercial reasons. A survey published by F-Secure showed that only 0.5% of Android malware reported had come from the Google Play store [6].

In addition, the source of malware is very extensive. Different from the PC virus, Android malicious attack has its own features; various types of malicious codes cover almost every level. The proportion of various malware types is shown in Figure 1 [7].

Motivated by this, a great number of Android malware detecting methods are proposed which are divided into two types as follows [8].

The first kind of methods is static analysis. Static methods analyze the executable file directly instead of running it. For example, DroidDet [9] statically detects malware by utilizing the rotation forest model. However, this work cannot resist the obfuscated attack.

Another type of approaches is dynamic analysis. Different from the static methods, dynamic methods extract the malicious features at runtime, which improves the effectiveness of detection. By contrast, dynamic analysis has stronger robustness. Dynamic analysis techniques are not compatible in some cases because developing tools that allow the dynamic analysis of malware is very challenging, and such techniques require extensive resources and often do not have enough scale to be used in practice [10]. Shabtai A et al. [11] propose a new dynamic technique, sandbox, which is built by the kernel LKM (Loadable Kernel Module). They analyze the

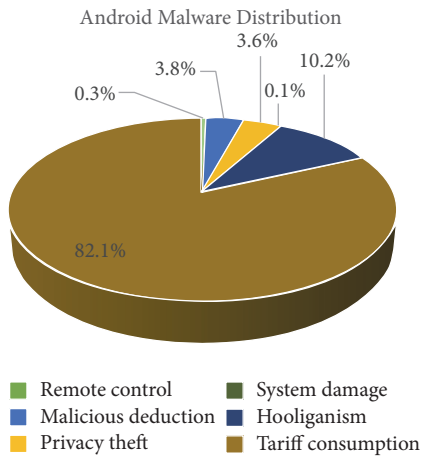

Figure 1: Android malware distribution. This figure is reproduced from 360 Internet Security Center [7] (2017) [under the Creative Commons Attribution License/public domain].

system calls from the kernel to create the log file. However, the modification to the kernel level causes the instability of operations, and the user interaction is only simulated by automatic tools, which is no real operation [12].

To address these problems, we propose a more effective Android dynamic technology to detect malware. This is a new technique of establishing dynamic birthmarks. We extract the reference relationships between objects allocated in heap memory and then establish ORG (Object Reference Graph) to build ORGB (Object Reference Graph Birthmark) as the feature. In addition, we propose λ-VF2 algorithm to match the subgraph isomorphism.

Compared with the existing dynamic birthmark methods, we utilize the information in heap, which can also be used to solve the problem of code plagiarism. In summary, the main contributions of this paper are listed as follows.

(i) We establish ORG by extracting all the referential relationships between objects allocated in heap memory.

(ii) With the analysis of the program class, we extract the feature classes to build ORGB as the birthmark of malware.

(iii) Based on VF2 algorithm, we propose λ-VF2 algorithm to improve the false negative rate and false positive rate.

(iv) We propose an Android malware detection system Demadroid which resists the obfuscated attack. To demonstrate the effectiveness of the proposed approaches, we conduct extensive experiments. Experimental results show that the proposed system and algorithm perform well.

The rest of the paper is organized as follows. In Section 2, we discuss the related work, and we give the details of our algorithm in Section 3. Section 4 presents the framework of Demadroid. The evaluation of Demadroid is depicted in Section 5. In Section 6, we summarize the whole work.

2. Related Work

Several approaches have been proposed recently to detect malware in Android. Generally, they are divided into static analysis and dynamic analysis.

Static analysis inspects app without executing it. Julia is a Java bytecode static tool for Android platform, but it cannot parse the classes generated by the XML file mapping. Payet É et al. [10] improve it to analyze the bytecode of Dalvik Virtual Machine. Kui Luo et al. [13] propose a bytecode conversion tool for privacy stolen malware and enable it to convert into DVM bytecodes and analyze Android programs. Literature [14] uses the existing tools dex2jar and FindBugs for analysis, which traversed the flowchart of Android programs and obtains the functional dependencies between Intent objects. The above works are based on existing tools, which have a great number of limitations. Batyuk L et al. [15] present disassembly method by disassembling the malicious code of Android. They get the malicious part and modify it to separate the malicious code. This method is effective for the untreated apps but cannot deal with the obfuscated code. Based on sensitive data access, Di Cerbo F et al. [16] study the privacy-stealing code. By analyzing the permissions feature of the program request, they compare with the defined features to determine whether the program is malicious. One important problem in this work is that Android does not have permission restrictions on the use of API. Therefore, it cannot identify the malicious code utilizing Android vulnerabilities. In a word, the drawbacks of static methods are obvious; their robustness is weak. And several attacks such as code obfuscation, Junk Code, and other antidetection techniques can easily avoid detection.

Dynamic analysis can resist the code obfuscation attack but is more expensive than static methods. Isohara T et al. [17] use a kernel-level monitoring method to record the system call of Android program. This method can effectively analyze the record of system calls. However, it is just used for the monitoring of stolen information. Based on this, Schmidt A D et al. [18] present further research and divide the monitoring into Android application layer, system application layer, and system kernel layer. However, there are no valid experimental tests to verify the feasibility of the work. Crowdroid [19] is a classifier based on anomaly detection. The system uses the existing Strace program to monitor system calls and create record files. After being uploaded to the server, the files are classified by the K-Means algorithm. However, in this case, the amount of data and the network traffic of the system are relatively large, and the problem of data security is brought at the same time. Attackers can easily fabricate the key information and interfere with the result. Shabtai A et al. [11] mention a dynamic analysis technique, sandboxing, which is a new direction for Android malicious code detection. However, the current sandbox technology is incomplete. Myles et al. [20] use the control flow of apps to identify malicious behaviors. Experiments show that control-flow analysis is more effective than static birthmark analysis in dealing with attacks utilizing the semantics.

3. VF2 Algorithm

3.1. Isomorphic Patterns of Graphs. In the past decades, graph matching has been one of the main research topics in computer science. In general, graph matching can be classified into two lines, exact-matching algorithms and inexact-matching algorithms. Exact-matching algorithms require strict consistency between two candidate graphs. The most stringent pattern of exact-matching algorithms is graph isomorphism, which requires the mapping of nodes and edges on both graphs to be bijections [21]. The fuzzier pattern of exact-matching is subgraph isomorphism which requires at least the strict consistency between the subgraph and the ideograph [22].

Moreover, inexact-matching algorithms, which are also called fault-tolerance matching, relax the constraints with errors and noises. Monomorphism is the inexact-matching which gets rid of the bidirectional requirement of edge-remaining bases on subgraph isomorphism. It requires that every node of the first graph can map different nodes and edges in the second graph, which allows the redundant edges and nodes. The weaker graph match pattern is the homomorphism, which is a many-to-one mapping that does not require that every node of the first graph is mapped to a different node of the second graph. Isomorphism matching is another method to match the subgraphs, of which the result is not unique. It is also used to find the largest subgraph match, which is called the maximum common subgraph (MCS).

3.2. Analysis of the Subgraph Isomorphism Matching Algorithm. All the isomorphic patterns are NP-complete problems except graph isomorphism. Whether graph isomorphism is NP-complete problem has not been proved till now [23]. At present, polynomial time algorithms are matched for special types of graphs, and there is no general polynomial time algorithm for general graphs. For this reason, the time complexity of the exactly matching algorithm is exponential in the worst case. However, in practical problems, the cost of time is basically acceptable. Because the type of graph encountered in practical problems is not the worst case and the attributes of the nodes and edges can greatly reduce the search time.

The problem of graph isomorphic matching is a very classic problem in graph theory, and the algorithms used in different scenarios are different. In practice, the data required for the establishment of a graph will inevitably be disturbed; that is why graph isomorphism is rarely used. Subgraph isomorphism and monomorphism are commonly used patterns. They are more effective in dealing with practical problems. Many algorithms have been developed for these two problems. At present, the exact match algorithm is more effective for the basic graphs and searching for MCS.

3.2.1. Ullmann Algorithm. One of the most important types of graph matching algorithm is the Ullmann algorithm [24], which was proposed in 1976. It can solve the isomorphic problems, such as isomorphism, subgraph isomorphism, and monomorphism. At the same time, the algorithm also provides a way to deal with the maximum matching, so it can also be used to solve the CMS problem.

To reduce the bad matching branches, Ullmann algorithm proposes predictive equation to control backtracking process, significantly reduce the scale of search space, and improve the performance of the algorithm.

3.2.2. Ghahraman Algorithm. Ghahraman proposed another backtracking based monomorphism algorithm in 1980 [25]. To reduce the search space, a technique like association graph is used in this paper. The matching search is carried out on the NetGraph matrix. This matrix is generated by the product of the Descartes product between the nodes of the matched two graphs. The monomorphism matching of the two graphs is related to a subgraph of the NetGraph. The author finds two necessary conditions for the partial matching to produce the result.

One of the main disadvantages is that the storage of NetGraph requires at least one matrix of $N_2 * N_2$ size, in which N represents the number of nodes. Therefore, this algorithm is more suitable for a graph with lower number of nodes.

3.2.3. Nauty Algorithm. Nauty algorithm [26] is the most famous tree search algorithm which is not based on backtracking. It only deals with the isomorphic problem and is recognized as the fastest one. By using the conclusion group theory, it creates an automorphism group for each input. And every automorphism group produces a standard label to guarantee that the only node order is introduced by each equivalent class of the automorphism group. Then, the isomorphic comparison of the two graphs is equivalent to the adjacency matrix comparison of the standard label.

The time complexity of comparison is $O(N_2)$ of the worst case. In most cases, the time performance is acceptable. Because the establishment of standard tags can be carried out independently. Therefore, it is more suitable for the graph matching in a large library.

3.2.4. VF and VF2 Algorithm. The VF algorithm proposed by Cordellac [27] is applied to both isomorphism and subgraph isomorphism. Cordellac defined a heuristic search by analyzing the adjacent nodes of matched nodes. This heuristic algorithm is significantly better than Ullman and other algorithms in many cases.

Cordella improved the algorithm in 2001, which is called the VF2 algorithm [28]. The improvement reduces the space complexity from $O(N_2)$ to $O(N)$, in which N donates the number of nodes. In this way, the algorithm can be applied to the matching of large graphs.

The VF2 algorithm is also used in many other related fields. For example, Jonathan Crussell et al. propose DNADroid [29], a tool which uses VF2 algorithm to detect cloned apps. In this work, VF2 algorithm is used to compute subgraph isomorphism. The experiment proves that VF2 algorithm is suitable for graphs containing a variety of node types.

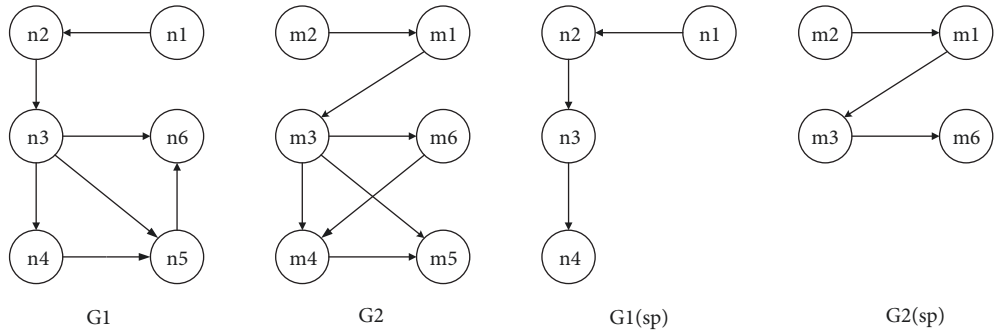

Figure 2: *SSR* instance diagram.

3.3. Comparison of Subgraph Isomorphism Matching Algorithms. In this section, we analyze several classical algorithms mentioned in Section 3.2 and select the proper algorithm as the foundation of our matching process. The main types of graph include the bounded Valence Graph, the two-dimensional grid graph (2D Mesh Graph), and the random connection graph (Randomly Connected Graph). Foggia et al. analyze the above algorithms by experiments [30]. The ORG in our work is similar to random connection graph and quite different from the other two kinds. Therefore, we only discuss the condition of random connection graphs. Foggia uses a control group with different density of nodes and edges. The experimental result shows that VF2 algorithm and Nanty algorithm are better than Ullmann algorithm in dealing with random connection graphs. VF2 performs better than VF algorithm when the density is different. Compared with Nauty algorithm, VF2 algorithm has a better effect to match spares graphs. And Nauty algorithm is more applicable to dense graphs.

In this paper, we match the subgraphs between object reference dependency graphs, in which nodes represent classes, and directed edges represent references between classes. According to the analysis of samples, the number of nodes in ORG is within 100. Therefore, the algorithm used in this paper is based on VF2 algorithm.

3.4. Review of VF2 Algorithm. VF2 algorithm is applicable to isomorphism, subgraph isomorphism, and monomorphism because it does not impose restrictions on the topology of matched graphs. The algorithm adopts the concept of state space representation (from now on SSR) in the matching process and proposes five feasible rules to prune the search space. Compared with VF algorithm, the most significant improvement is the strategy of traversing the search tree and the data structure making the algorithm applied to match the graph with thousands of nodes.

The primary idea of the VF2 algorithm is as follows. Given the digraphs $G_1(N_1, B_1)$ and $G_2(N_2, B_2)$, shown in Figure 2, we are looking for the isomorphic mapping between them. Map M is used to express (n, m), in which n donates a node of G_1 and m donates a node of G_2. The process of finding the mapping M is described by SSR. Each state s in the matching process is a partial mapping $M(s)$, which is a subset of M.$G_1(s)$ donates the subgraph of the mapping $M(s)$ associated with G_1, and $G_2(s)$ donates the subgraph of G_2 matched by $M(s)$.$V_1(s)$ and $V_2(s)$, respectively, represent the set of vertices in $G_1(s)$ and $G_2(s)$. E_1(s) and E_2(s), respectively, denote the edge set in $G_1(s)$ and $G_2(s)$. Given the middle state sp, the partial M is as follows:

$$\begin{aligned} M &= \{(n1, m2), (n2, m1), (n3, m3), (n4, m6), (n5, m4), \\ &\quad (n6, m5)\} \\ M(sp) &= \{(n1, m2), (n2, m1), (n3, m3), (n4, m6)\} \\ V1(sp) &= \{n1, n2, n3, n4\} \\ V2(sp) &= \{m2, m1, m3, m6\} \\ E1(sp) &= \{\langle n1, n2\rangle, \langle n2, n3\rangle, \langle n3, n4\rangle\} \\ E2(sp) &= \{\langle m2, m1\rangle, \langle m1, m3\rangle, \langle m3, m6\rangle\} \end{aligned} \tag{1}$$

There are multiple states in the matching process, and state s is converted to another state by adding a pair of new nodes. By adding different pairs of nodes, s is converted to various states. In this way, the new state is described using a tree structure in which parent node represents the original state and the child node represents the new state. In Figure 2, s converts to sq after adding node $(n5, m4)$. Figure 3(a) shows that the node pairs $(n5, m4)$ are just one of many possible ones. Therefore, we need to select the appropriate state by backtracking the search tree. In Figure 3(b), after joining $(n5, m4)$, G_1(sp) and G_2(sp) are successfully converted to G_1(sq) and G_2(sq).

In the matching process, M is obtained by searching the SSR. VF2 algorithm proposes five feasible rules to reduce the time complexity by pruning the search space. According to the proposed rules, the unsatisfied child nodes are removed. The remaining nodes set is called the candidate set $H(s)$, which is traversed in the depth-first order. The pseudocode of VF2 algorithm is shown in Algorithm 1.

The following definitions are given:

(1) $T_1^{out}(s)$: it denotes a vertex set of G_1, vertexes of which are descendent vertexes of G_1(s) but not contained in G_1(s).

(2) $T_2^{out}(s)$: it denotes a vertex set of G_2, vertexes of which are descendent vertexes of $G_2(s)$ but not contained in G_2(s).

```
Input: G1, G2, State s, initialized state: s0, M(s0) is set empty
Output: The isomorphic map: M
(01) PROCEDURE VF2 Match(s)
(02)     IF |M(s)| = |G2| THEN
(03)         Successful Match
(04)     ELSE
(05)         Find H(s) which is the set of possible pairs for M(s)
(06)         FOREACH h in H(s)
(07)          IF all rules are satisfied for h added to M(s) THEN
(08)                 s' = put h into M(s)
(09)                 CALL VF2Match (s')
(10)             ENDIF
(11)         ENDFOREACH
(12)     Restore data
(13)     ENDIF
(14) END PROCEDURE VF2MATCH
```

Algorithm 1: The original VF2 algorithm.

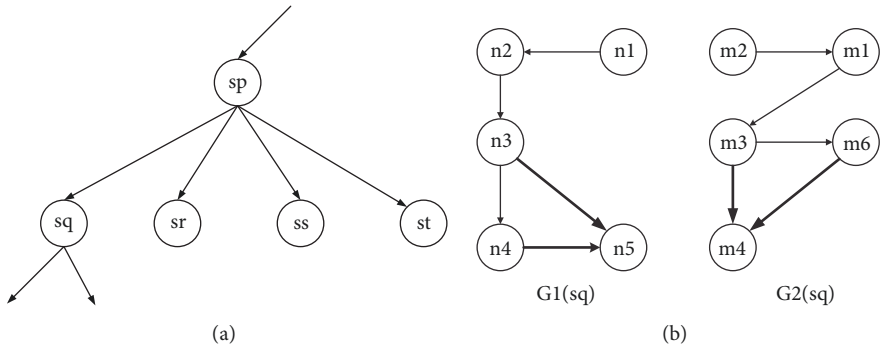

Figure 3: SSR state transition diagram.

(3) $T_1^{in}(s)$: it denotes a vertex set of G_1, vertexes of which are antecedent vertexes of $G_1(s)$ but not contained in $G_1(s)$.

(4) $T_2^{in}(s)$: it denotes a vertex set of G_2, vertexes of which are antecedent vertexes of $G_2(s)$ but not contained in $G_2(s)$.

The steps of selecting $H(s)$ are as follows:

(1) If $T_1^{out}(s)$ and $T_2^{out}(s)$ are not empty sets, then $P(s) = T_1^{out}(s) * T_2^{out}(s)$.

(2) If $T_1^{out}(s)$ and $T_2^{out}(s)$ are both empty sets and $T_1^{in}(s)$ and $T_2^{in}(s)$ are not empty sets, then $P(s) = T_1^{in}(s) * T_2^{in}(s)$.

(3) If $T_1^{out}(s)$, $T_2^{out}(s)$, $T_1^{in}(s)$, and $T_2^{in}(s)$ are empty sets, then $P(s) = (V_1 - V_1(s)) \times (V_2 - V_2(s))$.

(4) Other conditions prune the state s.

As described above, if one of $T_1^{out}(s)$ and $T_2^{out}(s)$ or one of $T_1^{in}(s)$ and $T_2^{in}(s)$ is an empty set, state s is pruned. For state s, the algorithm needs to check all the candidate nodes (m, n) by the feasibility function $F(s, n, m)$, in which s denotes the current state, n denotes a vertex of $G1$, and m represents a vertex of $G2$. The return value of $F(s, n, m)$ reflects whether the given node is feasible. If the node is not feasible, the path of it will be pruned.

The feasibility rules are divided into grammatical and semantic. The grammatical rules express the topological structure of the graph, and the semantic ones express the properties of the vertices and edges. In this work, we consider the grammar rules because there are no properties in edges and vertexes of ORG. Therefore, $Fsyn(s, n, m)$ is defined as follows:

$$Fsyn(s, n, m) = R_{pred} \wedge R_{succ} \wedge R_{in} \wedge R_{out} \wedge R_{new} \quad (2)$$

Five feasible grammar rules are defined in $Fsyn(s, n, m)$, in which R_{pred} and R_{succ} are the consistency of $M(s)$. After the candidate node (m, n) is added, R_{in}, R_{out}, and R_{new} are used to prune the search space.

$Pred(G, n)$ denotes the set of the antecedent nodes of n in figure G, and $Succ(G, n)$ denotes the set of the descendent nodes of n in figure G. The algorithm defines $T_1(s) = T_1^{in}(s) \vee T_{out}(s)$, $N_1'(s) = N_1(s) - M_1(s) - T_1(s)$. $T_2(s)$ and $N_2'(s)$ are defined as $T_2(s) = T_2^{in}(s) \vee T_{out}(s)$, $N_2'(s) = N_2(s) - M_2(s) - T_2(s)$.

```
Input: G1, G2, state s, the initial state: s0, M(s0) is empty, λ: Precision control parameters
Output: Isomorphic Mapping
(01)  PROCEDURE VF2 Match(s)
(02)        IF |M(s)| ≥ λ|G2| THEN
(03)              Successful Match
(04)        ELSE
(05)              Find H(s) which is the set of possible pairs for M(s)
(06)              FOREACH h in H(s)
(07)              IF all rules are satisfied for h added to M(s) THEN
(08)                        s' = put h into M(s)
(09)                        CALL VF2Match(s')
(10)              ENDIF
(11)        ENDFOREACH
(12)        Restore data
(13)        ENDIF
(14)  END PROCEDURE VF2MATCH
```

Algorithm 2: λ-VF2 algorithm (based on VF2 algorithm).

Rule 1 ($Rpred(s,n,m)$).

$$\begin{aligned}&\left(\left(\forall n' \in M_1(s)\right) \cap Pred(G_1,n)\ \exists m'\right.\\&\quad \in Pred(G_2,m) \mid \left(n',m'\right) \in M(s)\Big)\\&\wedge \left(\left(\forall m' \in M_2(s)\right) \cap Pred(G_2,n)\ \exists n'\right.\\&\quad \in Pred(G_1,n) \mid \left(n',m'\right) \in M(s)\Big)\end{aligned} \tag{3}$$

Rule 2 ($R_{succ}(s,n,m)$).

$$\begin{aligned}&\left(\left(\forall n' \in M_1(s)\right) \cap Pred(G_1,n)\ \exists m'\right.\\&\quad \in Succ(G_2,m) \mid \left(n',m'\right) \in M(s)\Big)\\&\wedge \left(\left(\forall m' \in M_2(s)\right) \cap Pred(G_2,n)\ \exists n'\right.\\&\quad \in Succ(G_1,n) \mid \left(n',m'\right) \in M(s)\Big)\end{aligned} \tag{4}$$

Rule 3 ($R_{in}(s,n,m)$).

$$\begin{aligned}&\left(Card\left(Succ(G_1,n) \cap T_1^{in}(s)\right)\right.\\&\quad \geq Card\left(Succ(G_2,m) \cap T_2^{in}(s)\right)\Big)\\&\wedge \left(Card\left(Pred(G_1,n) \cap T_1^{in}(s)\right)\right.\\&\quad \geq Card\left(Pred(G_2,m) \cap T_2^{in}(s)\right)\Big)\end{aligned} \tag{5}$$

Rule 4 ($R_{out}(s,n,m)$).

$$\begin{aligned}&\left(Card\left(Succ(G_1,n) \cap T_1^{out}(s)\right)\right.\\&\quad \geq Card\left(Succ(G_2,m) \cap T_2^{out}(s)\right)\Big)\\&\wedge \left(Card\left(Pred(G_1,n) \cap T_1^{out}(s)\right)\right.\\&\quad \geq Card\left(Pred(G_2,m) \cap T_2^{out}(s)\right)\Big)\end{aligned} \tag{6}$$

Rule 5 ($R_{new}(s,n,m)$).

$$\begin{aligned}&\left(Card\left(N_1'(s) \cap Pred(G_1,n)\right)\right.\\&\quad \geq Card\left(N_2'(s) \cap Pred(G_2,n)\right)\Big)\\&\wedge \left(Card\left(N_1'(s) \cap Succ(G_1,n)\right)\right.\\&\quad \geq Card\left(N_2'(s) \cap Succ(G_2,n)\right)\Big)\end{aligned} \tag{7}$$

The above five rules are applied to the subgraph isomorphism pattern. In addition, for isomorphism pattern, "$\geq$" in R_{in}, R_{out}, and R_{new} is replaced by "=". If the newly added node pair is satisfied by the five feasibility rules, the algorithm adds them and continues the searching.

3.5. The Implementation of λ-VF2 Algorithm. In this section, we propose λ-VF2 algorithm based on the environment of Android to detect subgraph isomorphism between the ORG and ORGB. According to Section 3.4, the VF2 algorithm is aimed at isomorphism and subgraph isomorphism. However, for the study of ORG, in the case of subgraph isomorphism, it is still difficult to match the subgraph with the original graph. The reason is that the running time for an app injected with malicious code is not sufficient, which causes the creation of the incomplete references. Therefore, the algorithm needs to be adjusted to relax the matching condition. To relax the matching condition, the algorithm finishes when the matching ratio of vertex reaches a proper threshold.

The threshold $\lambda \in (0,1)$ is set as the input of the algorithm, which is determined by the user. λ indicates that the algorithm is terminated only when the ratio of matched vertices is bigger than or equal to λ; the algorithm returns *success*. In this way, the pseudocode of λ-VF2 algorithm is shown in Algorithm 2.

3.6. Performance Analysis. The time and space complexity of VF algorithm is positively correlated with λ. As an input

parameter, λ is independent of the algorithm. In this section, λ is considered as 1 at the worst case.

3.6.1. Time Complexity. Our algorithm is a graph SSR-based isomorphism algorithm. The time complexity consists of two parts: the time of traversing and the processing time for each state.

(i) Traversing Time. At best, each state has only one satisfied candidate node; namely, there is no need for backtracking. The total number of states that need to traverse is the number of nodes in given graph. The worst case is that there are no unsatisfied states. In the $d+1$th level of the search tree, there are $N(N-1)(N-2)\cdots(N-d)$ nodes. And the total number of tree nodes is

$$\begin{aligned} &1 + N + N(N-1) + N(N-1)(N-2) + \cdots \\ &\quad + N(N-1)(N-2) + \cdots + 2 = 1 + \frac{N!}{(N-1)!} \\ &\quad + \frac{N!}{(N-2)!} + \frac{N!}{(N-3)!} + \cdots + \frac{N!}{1!} = 1 + N! \sum_{d=1}^{N-1} \frac{1}{d!} \end{aligned} \tag{8}$$

$\sum_{d=1}^{N-1}(1/d!)$ is less than 2. Thus, the total number of sizes is $O(N!)$.

(ii) Processing Time of Each State. The processing time for each state consists of three parts: the calculation time T_H of the candidate set $H(s)$, the calculation time T_F of the feasible function $F(s,n,m)$, and the calculation time T_{new} of the new state. The total time of every single state: $T = T_H + T_F + T_{new}$.

T_H: the processing time for each state in the candidate set is constant, and the maximum size of the set is N. Therefore, T_H is $O(B)$.

T_F: in the process of $F(s,n,m)$, each edge costs constant time and the number of edges in the worst case is the number of nodes which is connected to every remaining node. Thus, $T_F = (B)$.

T_{new}: the calculation time of the new status includes the time of $M(s')$, $V_1^{in}(s)$, $V_1^{out}(s)$, $V_2^{in}(s)$, and $V_2^{out}(s)$, in which $M(s')$ is cost constant time. And the other four sets need to iterate over the edges of the newly joined one, which is $O(B)$ at the worst case.

B is the number of edges that a node is connected to. Given a directed graph of N vertexes, the number of edges connected to one given vertex achieves the maximal number of $2 * (N-1)$. Therefore, $O(B) = O(N)$ in the worst case.

In summary, $T = T_P + T_F + T_{new} = O(N) + O(N) + O(N) = O(N)$.

Final Time Complexity. According to the above analysis, the time complexity of the VF2 algorithm is the multiplication of the two parts.

In the best case, $O(N) * O(N) = O(N^2)$.

In the worst case, $O(N) * O(N!) = O(N * (N!))$.

3.6.2. Space Complexity. The VF2 algorithm adopts the sharing data structure. Thus, the storage space number required by each state is constant. The searching process traverses the search tree in the depth-first order, and the maximum depth of the tree is less than N. Therefore, the space complexity is $O(N)$.

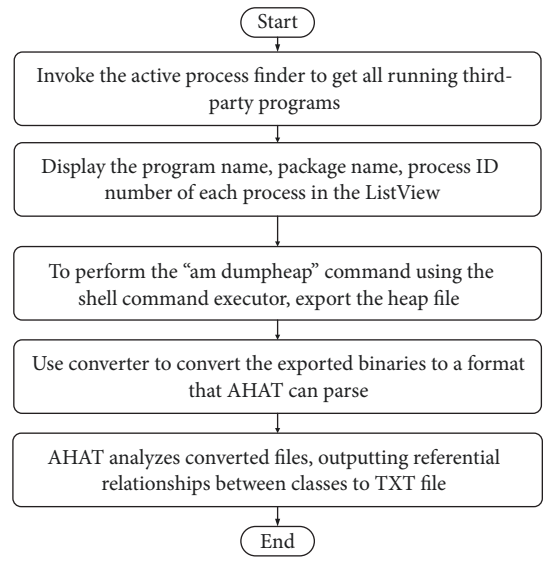

Figure 4: Malware detection inspection flowchart.

4. Framework of Demadroid

Demadroid mainly includes two parts: Android client and PC server. Android client is responsible for extracting data and passing it to the server side, and PC server is responsible for the malware detection.

4.1. Design and Implementation of Android Client. The main function of the Android module is to extract the object reference information from a process. We construct MalwareDetection to analyze the running process (except the system process) and export the dynamic information file for further analysis.

The main components of MalwareDetection include front-end interface, active process finder, shell command executor, Convertor, and AHAT. The extraction flow is shown in Figure 4.

In general, the existing malicious code is embedded in the normal apk. After installation, the malicious code starts with the host app, sharing the process resource in memory. Objects are created in the process, each of which has mutual references with each other. The information we need includes the objects created by the injected process and references between them. We extract the information above in Android client. The reason is that the size of raw memory file is too large. For example, a lightweight app "calculator" generates a memory file of 10 M. There are many processes running in the memory at the same time. Therefore, it is necessary to extract the useful information to reduce the network burden when uploading to PC server.

Table 1: The extraction environment of the Android memory data.

PC OS	Tools used	Virtual Android System Version
Windows 7	ADT (Android Developer Tools)	Version 4.0.3

```
VersionHead
Identifier(4B)
File Creaton Date(8B)
Unit 1
Unit 2
...
Unit n
```

Box 1: Dumpheap file format.

There are three steps in the extraction process. The first step is the acquisition of raw heap information. The second step is to convert the raw memory file format. The third step is to analyze the dynamic information.

4.1.1. The Acquisition of Heap Memory Information Files. The Android SDK provides feature-rich memory monitoring tools, such as dumpheap tools for heap data monitoring. And it is supported by Android 2.3 version or more. To facilitate the analysis, we use AVD to virtualize Android 4.0.3 and successfully extract the heap data of the test process by dumpheap. The data extraction environment is shown in Table 1.

The dumpheap command is in the format of "*am dumpheap PID path*". We integrate dumpheap into Android program. In the extraction process, we first use the adb tools to obtain the equipment information. After the execution of this command, the heap data of process is saved in files. In this way, a complete file of raw heap information is obtained. This file is binary and cannot be read directly from the contents. Therefore, the format of the binary file needs to be converted.

For example, we start the "calculator" application in the virtual device. With the obtained process ID number, we export the memory raw data of the *Calculator* process by dumpheap.

4.1.2. The Format Conversion of the Memory Information File. The raw memory data is binary and it cannot be analyzed directly. We develop Convertor to convert it into an available format.

The analysis tool we propose, AHAT, is based on JHAT, which is used in PC environment. The version of the binary memory file generated by dumpheap is *1.0.3*, while the version JHAT can analyze is *1.0.2*, and the file format needs to be converted from *1.0.3* to *1.0.2* on Android platform.

The function of Convertor is similar to HprofConv tools of SDT, which is used in PC environment. The first step is to analyze the two versions. The binary file format produced by dumpheap is shown in Box 1. The format of the binary file is fixed, beginning with a version string, such as "*Java PROFILE 1.0.2*", followed by the 4-byte ID information, followed by 8-byte file creation date information. After creation date information is the memory data, which is the body of the binary file.

```
Type(1B)
TimeStamp(4B)
Length(4B)
DetailInfo(Length ∗ 1B)
```

Box 2: The format of unit.

Table 2: New type of 1.0.3 unit.

Type	Hexadecimal mark
HPROF_HEAP_DUMP_INFO	0xfe,
HPROF_ROOT_INTERNED_STRING	0x89,
HPROF_ROOT_FINALIZING	0x8a,
HPROF_ROOT_DEBUGGER	0x8b,
HPROF_ROOT_REFERENCE_CLEANUP	0x8c,
HPROF_ROOT_VM_INTERNAL	0x8d,
HPROF_ROOT_JNI_MONITOR	0x8e,
HPROF_UNREACHABLE	0x90,
HPROF_PRIMITIVE_ARRAY_NODATA_DUMP	0xc3,

The memory data consists of units. Each of these units stores the information of a Java object. The format of a unit is shown in Box 2. The data structure includes a 1-byte type field, a 4-byte timestamp field, a 4-byte data length field *n*, and finally the n-byte object information field.

The main difference between the two versions is that the number of types is in Detail Info field. In the old version, there are thirteen types in the Detail Info field. In the new version, nine new types are added, which are shown in Table 2.

The types shown in Table 2 make the information unanalyzable. The solution is to remove the new types, which is irrelevant to our work.

We use the unit types as the member of *Convertor* class, which is used in the analysis process to determine whether a given type is useful. Finally, the file is reorganized in the format of the *1.0.2* version.

4.1.3. Extraction of Object and Reference Information. We develop AHAT, a tool used to analyze binary files in Android which is similar to JHAT in PC environment. AHAT mainly consists of four parts: Model, Parser, Util, and external call interface. The relationship between the four modules is shown in Figure 5.

Model. It defines the types (data structures) of all involved objects, and the objects of these data structures constitute a model. There are 29 classes corresponding to object types of Java, the most important of which is the *Snapshot*, the largest unit of the memory snapshot model.

Table 3: Classes that need to be filtered out during extraction.

boolean	long	javax.net.	javax.transaction.
char	sun.	javax.print.	javax.xml.parsers.
float	java.	javax.rmi.	javax.xml.transform.
double	javax.accessibility	javax.security.	org.ietf.jgss.
byte	javax.crypto.	javax.sound.	org.omg.
short	javax.imageio.	javax.sql.	org.w3c.dom.
int	javax.naming.	javax.swing.	org.xml.sax.

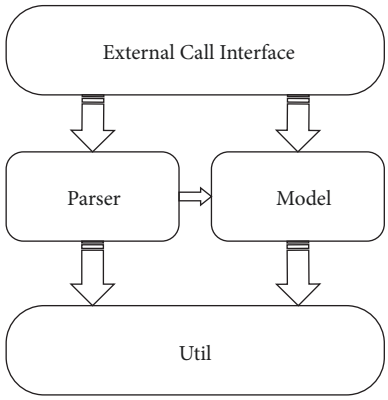

Figure 5: The structure of AHAT.

Parser. It is used for reading binary files, analyzing data, and using it with model objects to build a model. Parser consists of 7 classes; the main class is *HprofReader*, used for heap binary parsing.

Util. it is a common toolkit.

External Call Interface. AHAT is responsible for invoking each module to make it work properly. The *activity* class is interacting with the user on Android, so the *main* class is the *MainActivity* class and the *QueryClassInfo* class used to get the referential relationship between the classes.

According to the work process of JHAT, there are four steps in the implementation of AHAT:

(1) Create: AHAT first creates a snapshot for preparing to store data.

(2) Read: the *HprofReader* class parses the binary file to obtain the necessary information and builds the *Snapshot* object.

(3) Resolve: the *Snapshot* object uses the object information to initialize the data structure which includes the reference relationships between classes.

(4) Query: based on the constructed model, we query the class reference and write it in files.

4.1.4. Important Data Structures and Methods

(1) *Snapshot* class: It represents a Snapshot of a Java object in the JVM which contains the dynamic object information as well as references between them. The data structures involved are defined in the model module.

(2) *HprofReader* class: It parses the binary file to extract the memory information of each unit and uses it to build a *Snapshot* object. After this, we initialize the data structure, calculate the specific information of each object, like package name, class name, class ID, class member variable, reference relation between classes, and so on. The above process is the key to dynamic information extraction.

(3) *QueryClassInfo* class: The function of *QueryClassInfo* class is to extract the references between classes of *Snapshot* object. The variable *referrersStat* in the *process* function is a Hashmap which stores the referenced information of this class and the variable *referrersStat* is used to store the referencing information. All the classes in the memory are obtained by the function *getClasses* of *Snapshot*.

(4) *PlatformClasses* classes: In the obtaining process of object references, there are thousands of classes returned by function *getClasses*, most of which are *platform-supplied* classes, like the Java Standard API classes, the API classes provided by the Android system, and so on. These classes are irrelevant to our work. What is more, the existence of them can cover the references between the key classes. Therefore, we remove such irrelevant classes (shown in Table 3) by function *PlatformClasses*.

Table 4: The experimental operating environment of AHAT.

Device name	Galaxy Nexus 3
Android version	Android4.1.2
Mobile RAM	1 GB
CPU	Texas Instruments OMAP4460, dual-core, Frequency 1228 MHz

4.1.5. Results of AHAT. The AHAT requires Android 4.0 or more. We test it on Google's Galaxy Nexus 3, of which the environment is shown in Table 4.

The analysis process includes the reading of dumpheap files, binary data parsing, class reference relationship analysis, and the creation of result files. The result is stored in the dumpheap folder of the SD card.

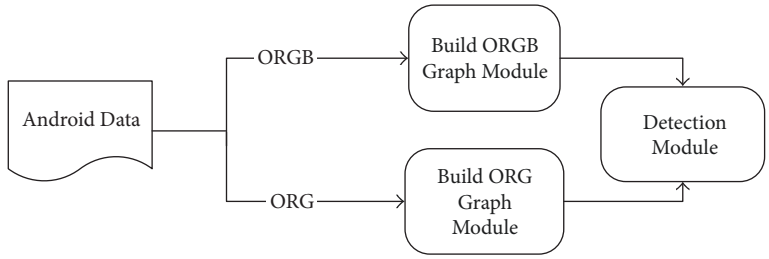

Figure 6: The overall architecture of the server.

Table 5: Android setup.

Name	Android System Version	CPU	Phone Memory
Galaxy Nexus 3	Android4.1.2	Texas instrument OMAP4460, dual core, Frequency: 1228 MHz	1 GB

4.2. Design and Implementation of Server Side. There are three parts in PC server: the establishment of ORG, the establishment of ORGB, and graph matching. The architecture of PC server is shown in Figure 6. After ORG is created, it is sent to the detection module to match with ORGB by λ-VF2 algorithm.

4.2.1. The Establishment of ORG. ORG is a digraph created by the information obtained in Android client. There is no system class in ORG, in which the nodes represent classes and the edges represent the references between classes. The flow chart of ORG establishment module is shown in Figure 7.

The node ID in the program is a number, and the class name in the file needs to be converted to ID. Thus, we create an index file to assign an ID for each class. In the parsing process, the class name is identified in the index file and added to ORG as a node. When the process identifies the string "*Referrers by type*", the referencing class is added and the directed edge is established from this node to the referenced node. When the program identifies "*Referees by type*", it reads the referenced class and adds it to ORG with the directed edge.

4.2.2. The Establishment of ORGB. ORGB is a digraph used to express the feature of malicious code. ORGB only collects the classes of malicious code as nodes, and the class list of malicious code is obtained by manual analysis. The flow chart of ORGB establishment module is shown in Figure 8.

4.2.3. Detection Module. In this part, we propose λ-VF2 algorithm. When the value of λ is 1, λ-VF2 algorithm degrades to the original VF algorithm. In the experiment, the results are different by setting λ with different values. The flow of the detection module is shown in Figure 9.

The program first inputs the value of λ and selects ORG and then matches the selected ORG with every ORGB in the malware library. The matching process will be terminated by a successful match. For the convenience of the experiment, ORG and ORGB are stored in binary file with no attribute of nodes and edges.

Table 6: PC setup.

OS	CPU	Memory
Windows 7	Xeon E3-1200 v2, Quad core, 8 threads, 3300 MHZ	1 GB

Table 7: Number of simulative code samples.

Origin	Extra Reference	Extra Class	Class Replacement
4	2	2	2

5. Experiments

5.1. Setup. In our experiments, we run the Android apps and extract original data by the tools we developed. We construct ORG and test it with the malware dataset.

(i) Android Setup. We extract memory data on a real device. Table 5 shows the experiment environment.

(ii) PC Setup. The ORG is sent to PC server. The environment of PC server is shown in Table 6.

5.2. Datasets. We use two kinds of datasets in our experiments, simulative malicious samples, and real malware samples.

5.2.1. Simulative Samples. Each simulative sample is built by manual construction, which consists of two packages. One is malicious and the other is benign. In a given category of simulative malware, the different sample contains different benign packages and the same malicious packages. The advantage of simulating samples is that we can control the scale and operation of malware. In the experiments, we construct 10 simulative samples. The malicious codes in these samples are basically the same. In order to test different effects of VF2-isomorphism, VF2 monomorphism, λ-VF2 isomorphism, and λ-VF2 monomorphism, we adjust the malicious codes to simulative the attacks. Table 7 shows the number of each type in simulative code samples.

(1) Origin samples: the malicious codes are the same, and the benign parts are different.

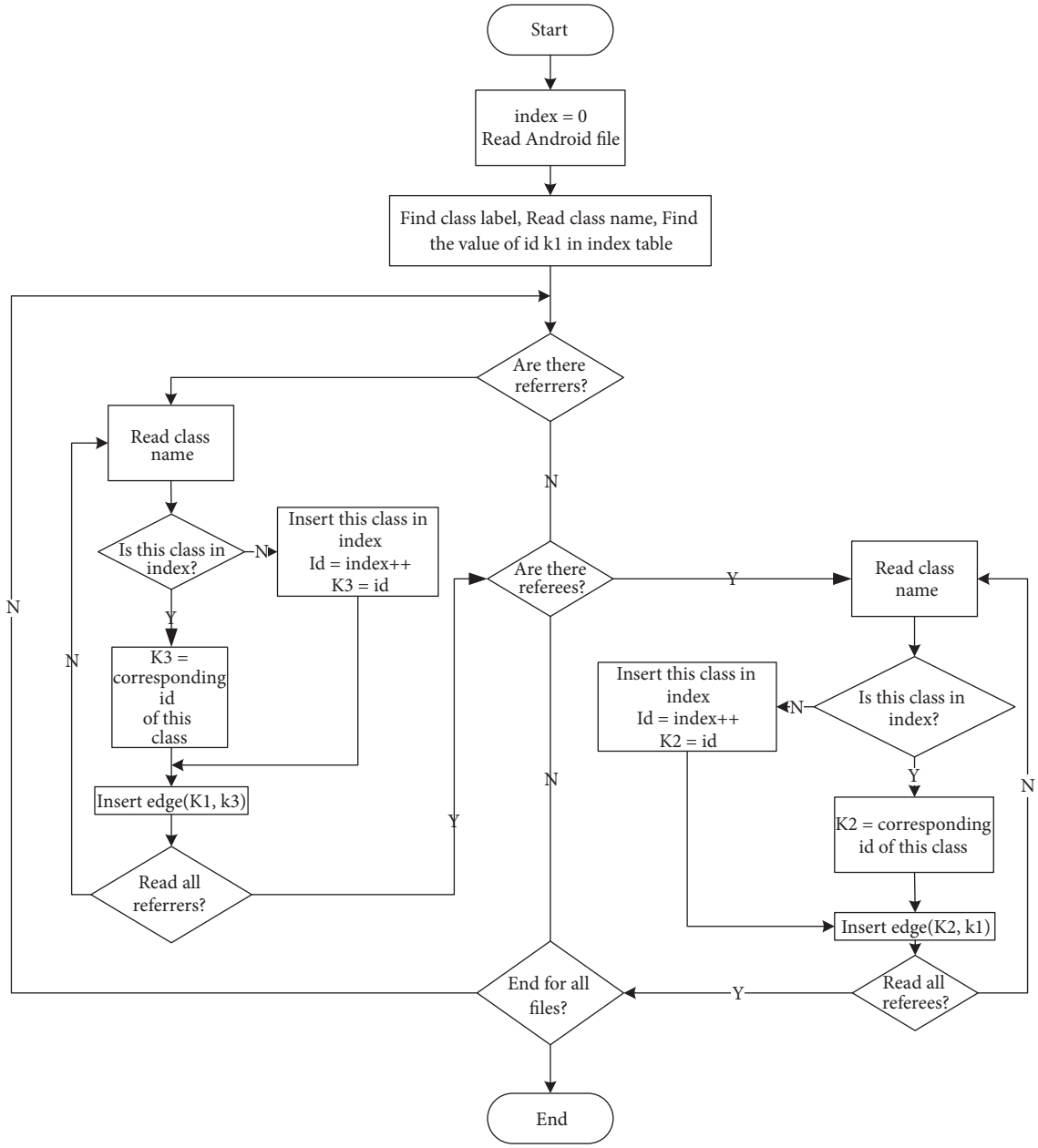

Figure 7: The flow chart of the ORG establishment module.

(2) Extra Reference samples: this kind of samples is simulating the malware which is intended to avoid detection by adding disturbance reference. The classes in malicious codes are identical. However, compared with the original malicious ones, there are several new meaningless references added between classes.

(3) Extra Class samples: new classes are added based on the original malicious codes to simulate the malicious variations.

(4) Class Replacement samples: based on the variations of simulative malicious codes, some classes are deleted and some classes are added.

Table 8: Number of real code samples.

ADRD	Bgserv
22	16

5.2.2. Real Malicious Samples. We also collect two kinds of real malware which is shown in Table 8.

To extract the ORGB of the given category of malware as the dynamic feature, we select some samples from each category randomly and then analyze them manually.

The APK file is generated from packetized dx tools. We use JD disassembler to reverse the source code to obtain

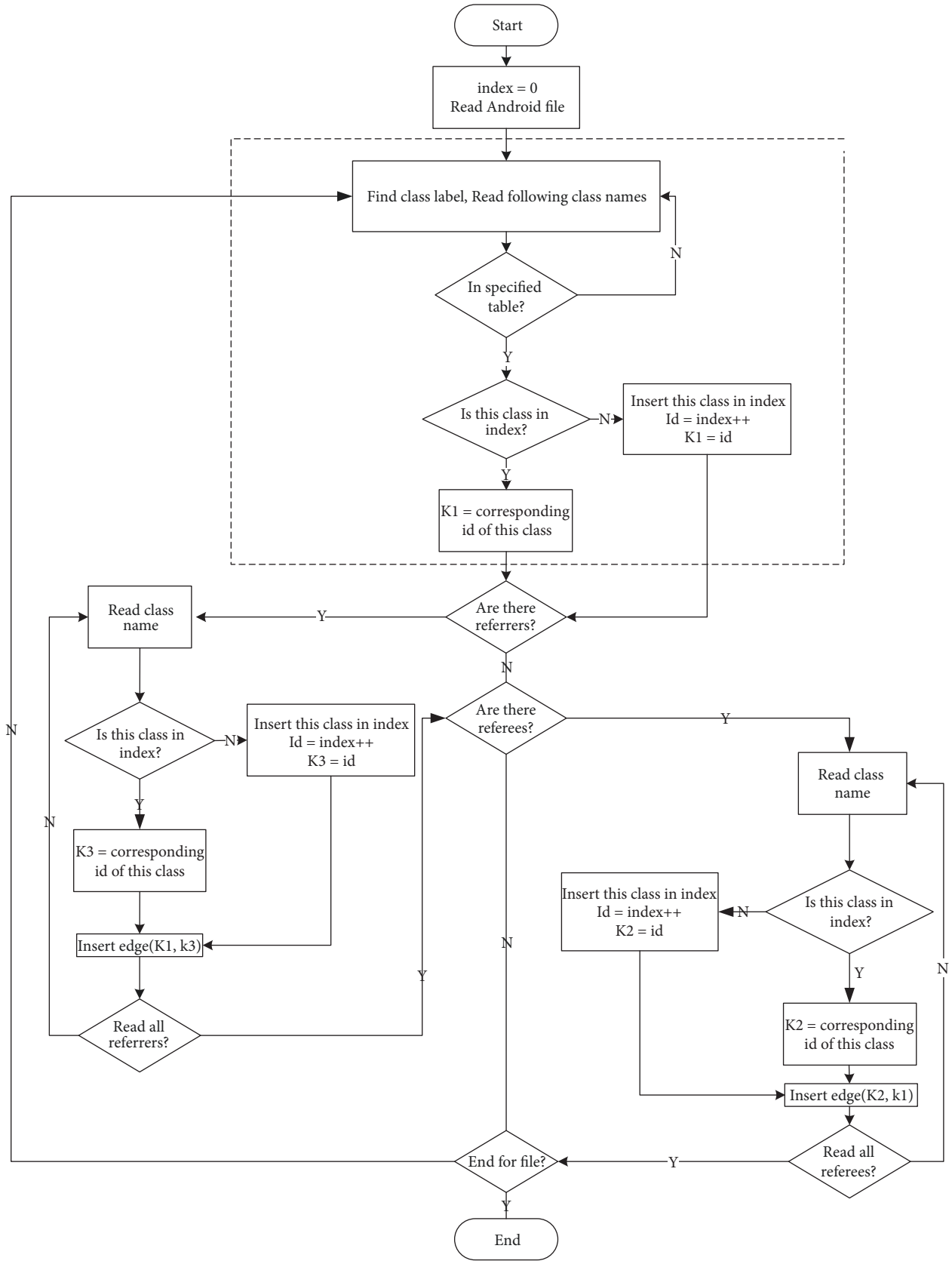

Figure 8: The flow chart of the ORGB establishment module.

Table 9: Detection results of simulative samples.

Algorithm	Origin	Extra Reference	Extra Class	Class Replacement
VF2 Subgraph Isomorphism	4/4	0/2	2/2	0/2
VF2 Monomorphism	4/4	2/2	2/2	0/2
λ-VF2 Subgraph Isomorphism	4/4	1/2	2/2	1/2
λ-VF2 Monomorphism	4/4	2/2	2/2	2/2

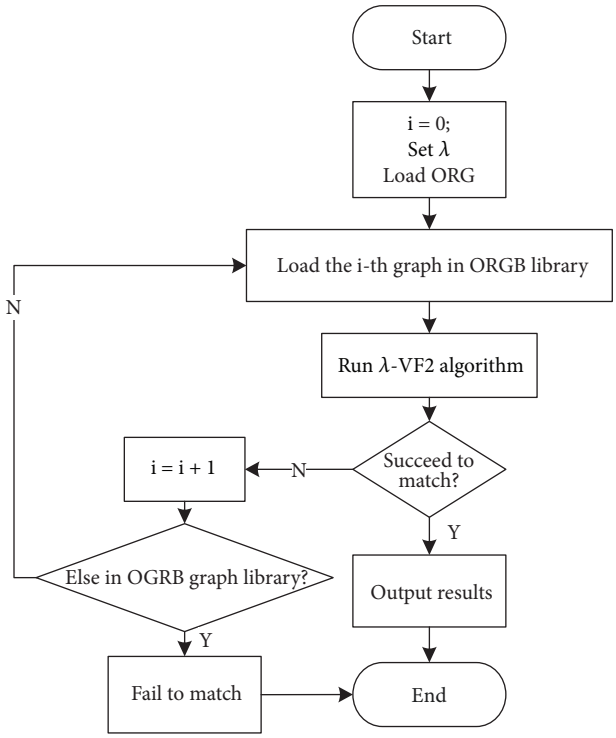

Figure 9: Detection module.

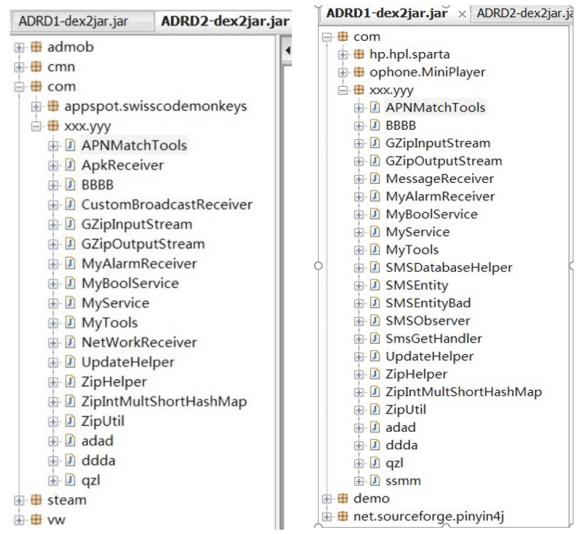

Figure 10: Comparison of malicious code classes.

the classes. By comparison, we acquire the malicious classes. Classes of malicious codes are generally stored in independent packages, which makes it possible to identify malicious categories manually. Figure 10 shows the file structures in two APKs which contains ADRD malicious codes. Obviously, both apk contains malicious package "*xxx.yyy*". In this way, we obtain the list of ADRD.

5.3. Experimental Results on Simulative Samples

5.3.1. Simulation Sample Test Results and Analysis. In our experiments, we first construct ORGB from Origin samples. Then, we construct complete ORG of the 10 samples. Finally, we, respectively, detect ORGB with four kinds of VF2 algorithm. Experimental results are shown in Table 9, where λ is 0.8.

As Table 9 shows, all algorithms can completely detect original malicious codes with new classes added for interference. The reason is that the new classes reflected the new nodes in ORG and ORGB is still a subgraph of ORG. It indicates that our method is effective in the variants added new classes.

VF2 subgraph isomorphism algorithm is unable to detect the attack of Extra Reference. The reason is that some meaningless references are added, which leads to new edges in ORG. However, subgraph isomorphism requires the complete matching of edges; namely, the new edge is required in both ORG and ORGB.

Extra Reference and Class Replacement are incompletely detected λ-VF2 subgraph isomorphism. This is because the impact of the added references is not completely eliminated and the matching condition is overqualified.

λ-VF2 monomorphism has the weakest constraint and is successful in the four kinds of detection. In practice, even the same kind of malicious codes is not totally identical. And the created objects are different in memory. In consideration of these factors, λ-VF2 monomorphism is the best choice. And the effectiveness needs to be verified on real malware samples.

5.3.2. Confused Variation Detection of Simulative Code Samples. Code confusion is the most common technique used in malware. With code confusion, malware can easily hide the malicious characteristics or generate the variations rapidly, which can avoid static detection.

ProGuard is a famous open source code obfuscation tool, which is integrated into Android. To make it usable, "*proguard.config= ${sdk.dir}/tools/proguard/ proguard-android.txt: proguard-project.txt*" needs to be added at the end of the *properties* file.

Table 10: Detection results of real samples.

Algorithm	ADRD	Bgserv
VF2 Subgraph Isomorphism	1/22	1/16
VF2 Monomorphism	2/22	2/16
λ-VF2 Subgraph Isomorphism	12/22	9/16
λ-VF2 Monomorphism	16/22	11/16

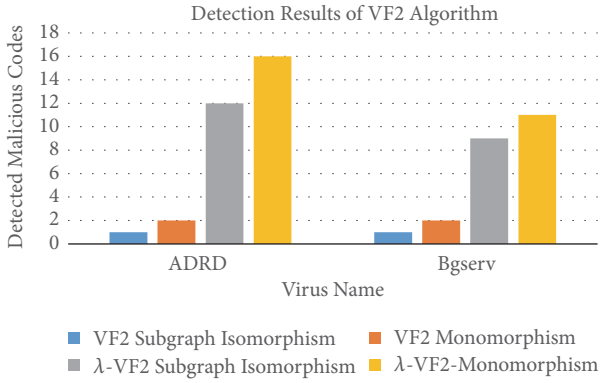

Figure 11: Detection results of VF2 algorithm.

In experiments, we utilize ProGuard to obfuscate four Origin simulative samples and regenerate their ORGs. Then, we detect them with the original ORGB by λ-VF2 monomorphism algorithm. Experimental results show that the four ORGs are all matched successfully.

5.4. Experimental Results on Real Samples

5.4.1. Effect of VF2 Algorithm on Malicious Code Detection. The VF2 algorithm is a precise graph matching algorithm, which requires the complete match of the subgraph. This algorithm achieves high accuracy with the low false positive rate. However, the effect of noise leads to the low possibility of complete matching. Thus, the practicability needs to be further tested.

We test the categories of ADRD and Bgserv by VF2 algorithms, and the value of λ is set to 0.8. The experimental results are shown in Table 10 and Figure 11.

As depicted in Table 10 and Figure 11, the success rates of VF2 subgraph isomorphism and VF2 monomorphism are low; the main reasons include the following:

(1) The feature of malicious codes is not sufficiently extracted because of the difference between samples of each category.

(2) In the process of extracting, malicious process dynamically creates and destroys classes, which leads to the deficient loading of the key feature in the memory.

(3) These two algorithms are both precisely matching. And the above two reasons can cause the failure of matching of ORGB and ORG.

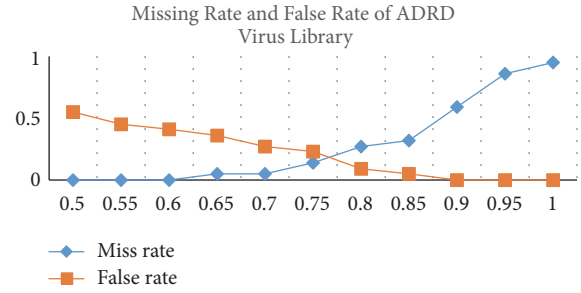

Figure 12: Results of ADRD Virus Library-Varying λ.

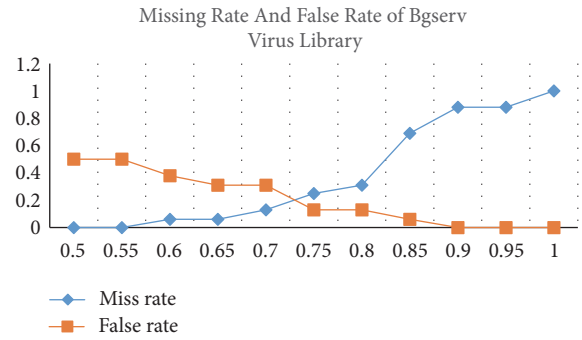

Figure 13: Results of BGSERV Virus Library-Varying λ.

It can be concluded that the reduction of matching precision can decrease the effect of noise and achieve high matching accuracy.

5.4.2. Effect of λ-VF2 Algorithm Varying Precision. λ-VF2 monomorphism algorithm is effective in real malicious codes. The value of λ affects a lot on matching results. If we decrease the value of λ, the matching precision reduces and the false positive rate increases when it tends to 0. If we increase the values of λ, the matching precision reduces and the false negative rate increases when it tends to 1. Thus, the proper value of λ needs to be tested.

To obtain the false rate when λ decreases, we use a malware group and a benign app group for each test value. And the benign group has the same number of apps with the malware group. λ starts from 0.5 and increases by 0.05 for each group. We obtain the false negative rate from the malware group and the false positive rate from the benign app group. Experimental results are shown in Table 11.

As Table 11 shows, when λ is 0.9, the miss rate achieves 0.5, which impossibly meets the practical needs. When λ is 0.75, the false rate achieves 0.23, which is unsatisfied. Thus, we select the value of λ from 0.75 to 0.85. The variation of miss rate and the false rate is illustrated in Figure 12. Experimental results are shown in Table 12.

As Table 12 shows, when λ is 0.85, the miss rate achieves 0.69, which impossibly meets the practical needs. When λ is 0.7, the false rate achieves 0.31, which is unsatisfied. Thus, we select the value of λ from 0.7 to 0.8. The variation of miss rate and the false rate are illustrated in Figure 13.

Table 11: Results of ADRD Virus Library-Varying λ.

λ	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
Total Number	22	22	22	22	22	22	22	22	22	22	22
ADRD	22	22	22	21	21	19	16	15	9	3	1
Normal	12	10	9	8	6	5	2	1	0	0	0
Missing Rate	0.00	0.00	0.00	0.05	0.05	0.14	0.27	0.32	0.59	0.86	0.95
False Rate	0.55	0.45	0.41	0.36	0.27	0.23	0.09	0.05	0.00	0.00	0.00

Table 12: Results of Bgserv Virus Library-Varying λ.

λ	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
Total Number	16	16	16	16	16	16	16	16	16	16	16
Bgserv	16	16	15	15	14	12	11	5	2	2	0
Normal	8	8	6	5	5	2	2	1	0	0	0
Missing Rate	0.00	0.00	0.06	0.06	0.13	0.25	0.31	0.69	0.88	0.88	1.00
False Rate	0.50	0.50	0.38	0.31	0.31	0.13	0.13	0.06	0.00	0.00	0.00

As observed in the two groups of experiments, as λ rises, the miss rate of malicious codes increases while the false rate decreases. These two parameters are a trade-off. In practice, to guarantee that the miss rate and false rate are satisfied, we set the value of λ according to the needs. From the experiments, it can be concluded that when λ is around 0.85, we can achieve a better performance.

6. Conclusion

In this paper, we present ORG to depict the references between objects allocated in heap memory and extract ORGB as the feature of Android malware from ORG. We propose Demadroid, a dynamic system for Android malware detection. After extracting ORG in memory, Demadroid matches ORG with the ORGB of each malware category by λ-VF2 algorithm. Experimental results demonstrate the effectiveness and efficiency of our algorithm. And Demadroid can effectively resist obfuscated attacks and detect the variants of known malware to meet the demand for actual use.

Our important future work is to take the deeper optimization of the graph match algorithm and the ORG establishment. And we can build a virus library in the cloud and combine the algorithm with cloud computing in the future. In this way, our framework can be improved from efficiency and accuracy in various scenarios.

Disclosure

Professor Hui He and Weizhe Zhang are the corresponding authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work is supported by the National Key Research and Development Program of China under Grant no. 2016YFB0800801 and the National Science Foundation of China (NSFC) under Grant nos. 61472108 and 61672186.

References

[1] W. Zhang, H. He, Q. Zhang, and T.-H. Kim, "PhoneProtector: protecting user privacy on the android-based mobile platform," *International Journal of Distributed Sensor Networks*, vol. 2014, Article ID 282417, 10 pages, 2014.

[2] A. Developers, "What is android," 2011.

[3] A. Azfar, K.-K. R. Choo, and L. Liu, "Android mobile VoIP apps: a survey and examination of their security and privacy," *Electronic Commerce Research*, vol. 16, no. 1, pp. 73–111, 2016.

[4] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, Article ID 3017427, 2017.

[5] W. Zhang, X. Li, N. Xiong, and A. V. Vasilakos, "Android platform-based individual privacy information protection system," *Personal and Ubiquitous Computing*, vol. 20, no. 6, pp. 875–884, 2016.

[6] S. Arshad, M. A. Shah, A. Khan, and M. Ahmed, "Android malware detection protection: a survey," in *Proceedings of the International Journal of Advanced Computer Science and Applications*, vol. 7, pp. 463–475, 2016.

[7] http://zt.360.cn/1101061855.php?dtid=1101061451&did=210467032.

[8] P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen, "A pragmatic android malware detection procedure," *Computers & Security*, vol. 70, pp. 689–701, 2017.

[9] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "DroidDet: effective and robust detection of android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, 2018.

[10] É. Payet and F. Spoto, "Static analysis of Android programs," *Information and Software Technology*, vol. 54, no. 11, pp. 1192–1201, 2012.

[11] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[12] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proceedings of the 5th International Conference on Malicious and Unwanted Software (Malware '10)*, pp. 55–62, Nancy, France, October 2010.

[13] K. Luo, "Using static analysis on Android applications to identify private information," Tech. Rep., Dept. of Computing and Information Sciences, Kansas State University, 2011.

[14] S. Dienst and T. Berger, "Static analysis of app dependencies in android bytecode," Technical Note, 2012.

[15] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A.-D. Schmidt, and S. Albayrak, "Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications," in *Proceedings of the 6th International Conference on Malicious and Unwanted Software, Malware 2011*, pp. 66–72, IEEE, October 2011.

[16] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, "Detection of malicious applications on android OS," *ICWF*, pp. 138–149, 2010.

[17] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the 7th International Conference on Computational Intelligence and Security (CIS '11)*, pp. 1011–1015, IEEE, December 2011.

[18] A. D. Schmidt, R. Bye, H. G. Schmidt et al., "Monitoring android for collaborative anomaly detection: a first architectural draft," TUB-DAI 8, Technische Universität Berlin-DAI-Labor, 2008.

[19] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 15–26, October 2011.

[20] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the Behavior Based Software Theft Detection*, pp. 280–290, 2009.

[21] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014.

[22] W.-S. Han, J. Lee, and J.-H. Lee, "TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD Conference on Management of Data (SIGMOD '13)*, pp. 337–348, June 2013.

[23] S. Fortin, "The graph isomorphism problem," 1996.

[24] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[25] D. E. Ghahraman, A. K. C. Wong, and T. Au, "Graph optimal monomorphism algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 10, no. 4, pp. 181–188, 1980.

[26] B. D. McKay, "Practical graph isomorphism," 1981.

[27] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "Fast graph matching for detecting CAD image components," in *Proceedings of the 15th International Conference In Pattern Recognition*, vol. 2, pp. 1034–1037, IEEE, 2000.

[28] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An improved algorithm for matching large graphs," in *Proceedings of the 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pp. 149–159, May 2001.

[29] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: detecting cloned applications on android markets," *European Symposium on Research in Computer Security*, vol. 12, pp. 37–54, 2012.

[30] P. Foggia, C. Sansone, and M. Vento, "A performance comparison of five algorithms for graph isomorphism," in *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pp. 188–199, May 2001.