Hindawi
Mobile Information Systems
Volume 2018, Article ID 8094314, 15 pages
https://doi.org/10.1155/2018/8094314

Research Article

Runtime Detection Framework for Android Malware

TaeGuen Kim,[1] BooJoong Kang,[2] and Eul Gyu Im[1]

[1]*Hanyang University, Haengdang-dong, Seongdong-gu, Seoul, Republic of Korea*
[2]*Center for Secure Information Technologies, ECIT, Belfast, UK*

Correspondence should be addressed to Eul Gyu Im; imeg@hanyang.ac.kr

Received 11 December 2017; Accepted 13 February 2018; Published 29 March 2018

Academic Editor: Elio Masciari

As the number of Android malware has been increased rapidly over the years, various malware detection methods have been proposed so far. Existing methods can be classified into two categories: static analysis-based methods and dynamic analysis-based methods. Both approaches have some limitations: static analysis-based methods are relatively easy to be avoided through transformation techniques such as junk instruction insertions, code reordering, and so on. However, dynamic analysis-based methods also have some limitations that analysis overheads are relatively high and kernel modification might be required to extract dynamic features. In this paper, we propose a dynamic analysis framework for Android malware detection that overcomes the aforementioned shortcomings. The framework uses a suffix tree that contains API (Application Programming Interface) subtraces and their probabilistic confidence values that are generated using HMMs (Hidden Markov Model) to reduce the malware detection overhead, and we designed the framework with the client-server architecture since the suffix tree is infeasible to be deployed in mobile devices. In addition, an application rewriting technique is used to trace API invocations without any modifications in the Android kernel. In our experiments, we measured the detection accuracy and the computational overheads to evaluate its effectiveness and efficiency of the proposed framework.

1. Introduction

With the growing popularity of mobile devices, many malware authors have been targeting the mobile devices [1]. According to the mobile malware report in G DATA [2], security experts discovered 750,000 new Android malware during the first quarter of 2017. Android malware is expected to be developed continuously and to be spread to attack Android devices. As a result, a lot of studies on Android malware detection have been conducted in order to defend against Android malware.

Static analysis can be used to analyze malware with low computational overheads without executing malware itself. However, static analysis-based detection methods have a problem that malware authors can apply various transformation techniques to avoid the detection. Rastogi et al. [3] developed a system that applies obfuscation techniques to generate various Android malware variants. They tested ten commercial antivirus scanners with the newly generated malware variants, and they showed that no scanners could detect any variants.

In contrast, dynamic analysis is useful to detect the transformed malware variants, because most dynamic features are preserved while the static features are changed by the transformation. In addition, many malware variants share common behaviors that can be extracted through dynamic analysis.

Despite this advantage, existing malware detection methods based on dynamic analysis have some shortcomings. First, most of the previously proposed methods require kernel modifications to analyze malware [4–10]. To monitor application behaviors, the existing methods usually insert monitoring code or instrumentation code to the Android kernel. The kernel modification has a drawback that the Android kernel needs to be rebuilt and the new kernel needs to be deployed to mobile devices. In addition, there is a concern that the kernel modification makes the process tedious for a mobile device user to update the new kernel and new security mechanisms. Secondly, many malware detection methods do not consider the limited resources of the devices. These existing methods are designed to be deployed and performed on users' devices, and these

methods would degrade the performance of the user device because of their expensive detection processes. In addition, most methods use the algorithms that are not suitable to be applied in real time because they focus on dynamic analysis for malware detection when the sufficient time for the analysis is given. Therefore, it is necessary to analyze applications using the efficient algorithms while reducing the analysis overhead in the user devices.

In this paper, we propose an Android malware detection framework which utilizes the application rewriting technique to monitor application behaviors instead of injecting the monitoring code into the kernel. In addition, our framework is designed as the client-server model in order to alleviate the overhead of malware detection processes in user devices. The user devices are only responsible for monitoring application behaviors and any other detection processes performed on the external server. To reduce the computational overheads of detection, we also designed the malware detection model based on the suffix tree and developed a probabilistic model using the suffix tree so that the model can distinguish between malware and benign applications. In addition to the efficiency of the model, the accuracy of the detection mechanism is an essential factor for the malware detection. Our proposed detection method uses a scoring method to measure and to capture the maliciousness of invoked APIs. We developed this runtime detection method and selected the suitable parameters that produce the best performance in terms of the malware detection accuracy.

2. Related Work

This section explains the previous studies that aim to secure Android devices, such as malware detection or malicious behavior analysis. For malware detection, various approaches have been proposed so far, and these approaches could be classified into two categories: static analysis-based approaches and dynamic analysis-based approaches.

2.1. Static Analysis-Based Approaches. There are various static analysis-based methods [11–19] proposed so far. Droidlegacy [11] is a tool to extract malware family signatures by decomposing malware into loosely coupled modules and extracting the API invocation frequencies of each module. Droidmat [12] and Drebin [13] use various features, including API invocation traces, intents, hardware/software components, and permissions, to distinguish between malware and benign applications. Yerima et al. [14] proposed a system that uses API frequency and permission frequency of an application as features in the Bayesian classifier to detect malware. DroidSIFT [15] classifies malware and benign applications by analyzing weighted API dependency graphs of applications. In Android, APIs are invoked in two ways: direct invocations and indirect invocations. Android provides a JAVA reflection mechanism to invoke APIs indirectly, and invoked APIs can be determined at the runtime. These aforementioned approaches may miss the APIs that are invoked at the runtime. In addition, these approaches would analyze the garbage APIs that are not even invoked at runtime because it is impossible to predict the application behaviors through static analysis. This may cause the false alarms as well as the analysis overheads.

There are also some studies to protect benign applications from malicious applications. Chex [16], Droidchecker [17], AAPL [18], and Amandroid [19] use various methods to check if applications have component-hijacking vulnerabilities. These methods perform the data-flow analysis to find out whether an application can be exploited maliciously. These researches focus on how to prevent unauthorized accesses through component-hijacking attacks on vulnerable applications. The purpose of their researches is different from ours.

2.2. Dynamic Analysis-Based Approaches. Many dynamic analysis-based methods [4–10, 20] have been also proposed so far. Isohara et al. [4] proposed a system that analyzes kernel-level behaviors for malware detection. System call invocations are monitored and checked whether the system call traces are matched with preidentified malware signatures, that is, system call traces. A signature is expressed as a regular expression that is related to specific malicious activities. Droidscope [5] provides the semantic views of malware through dynamic analysis. Their system monitors invoked system calls as well as changes of processes, threads, and memories to generate kernel-level semantic views. In addition, Dalvik instruction traces are extracted and used in taint analysis to generate Dalvik-level semantic views. Andrubis [6] is a dynamic malware analysis system that can analyze application behaviors. This system monitors system call invocations, and API invocations, and also performs the taint analysis. Taintdroid [7] conducts the data-flow analysis of applications and detects the information leakage of sensitive data. These approaches modified the Android kernel to monitor the behaviors of applications such as instruction sequences, system call invocations, API invocations, and changes of system resource usages.

Shabtai et al. [8] and Schmidt et al. [9] proposed anomaly detection systems in mobile devices. Their systems monitor and analyze mobile devices using various features to identify abnormalities of device usages. The features used in their systems include CPU usages, the number of sent packets, battery levels, and so on. There are some malware that conducts stealthy attacks such as user information leakages, and this kind of malware can avoid these detection systems because the stealthy attacks hardly affect the system metrics used in their systems.

AASandbox [10] is a malware detection system that uses both static analysis and dynamic analysis, and system call frequencies of both benign applications and malware are collected and compared for dynamic analysis. There is no classification nor detection algorithm used in their paper, and its results just show the possibility to detect malware using this proposed system.

Crowdroid [20] is a framework that uses system call traces to distinguish between benign applications and malware using the k-means algorithm. The framework uses the client-server model as our framework does. The k-means algorithm has high overheads to examine application behaviors at

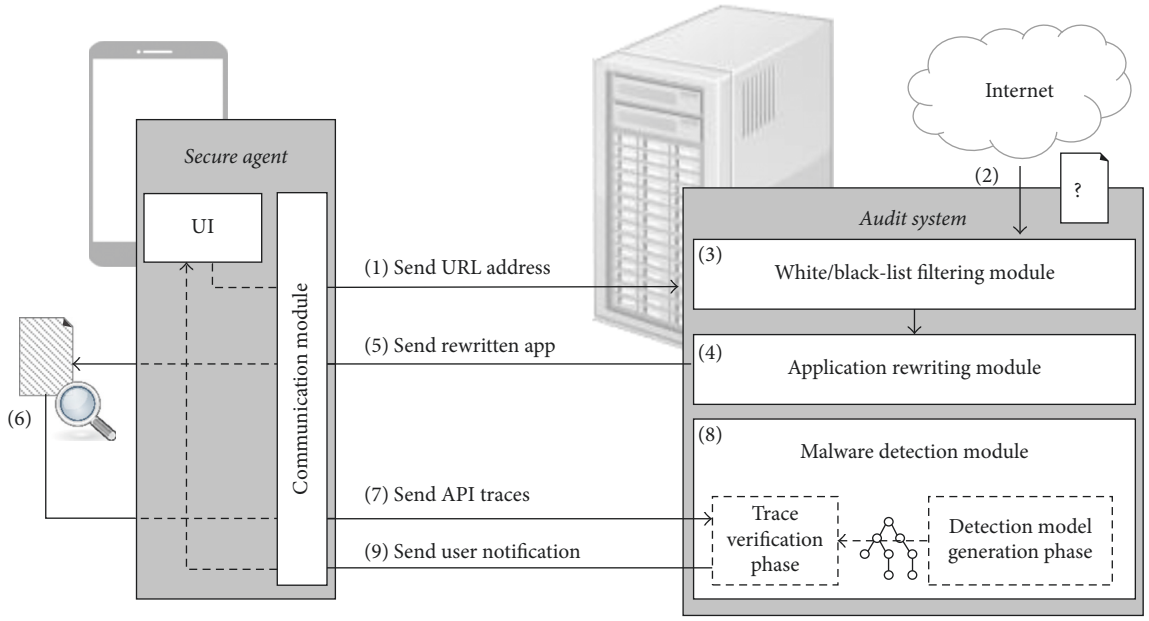

Figure 1: The overall architecture of malware detection framework.

runtime, and the framework cannot be used in real-time detection because the framework needs a certain amount of time to collect the system call frequencies. Our proposed framework uses a detection model to examine application behaviors finely at runtime while perserving high efficiency.

There exist researches [21–25] that use application rewriting techniques to enforce security policies in Android. I-ARM-Droid [21] inserts a reference monitor into an Android application, and the method to embed the reference monitor code in an application is introduced in [21]. Hao et al. [22] proposed an application rewriting system for API-level access control, and they studied on the effectiveness of the application rewriting technique for access control. Schreckling et al. [23] proposed a system that uses the application rewriting technique to prevent Android applications from abusing unnecessary permissions. The system analyzes what permissions should be declared for the specific application components and provides users a tool to activate or to deactivate permissions of applications. Aurasium [24] repackages applications to observe behaviors for the purpose of detecting information leakage of sensitive data. A rewritten application is isolated in its own sandbox, and its API invocations are monitored by their proposed system. Boxify [25] supports application sandboxing to prevent malicious activities such as privilege escalation. The system hooks Bionic libc to intercept system calls and to check whether each system call is valid or not. These researches based on the application rewriting technique only focus on how to monitor applications for the reference monitoring, and the collected information from the monitoring system is not delivered securely, which can be used as an additional attack vector.

3. Proposed Malware Detection Framework

Our framework consists of two parts: the server-side component (called as an audit system) and the client-side component (called as a secure agent). The overall architecture of the framework is shown in Figure 1. When a user wants to download an application, the user informs the secure agent of URL (uniform resource locator) information. Then, the secure agent delivers the URL information to the audit system. The audit system downloads an application from the URL, and checks whether the application is a known application or not, using whitelist and blacklist. If an application is known as a benign application, the application is transmitted to the secure agent and is allowed to be installed on the user device. If an application is known as malware, the audit system raises an alert to the secure agent. Otherwise, the application is rewritten in the audit system for further analysis.

The purpose of the application rewriting mechanism is to insert self-monitoring code into an application. After the audit system rewrites an application, the rewritten application is transmitted to the device, and the secure agent installs the rewritten application. The self-monitoring code extracts and delivers the API invocation traces of the application to the secure agent, and the traces are transmitted to the audit system. Each transmitted trace is examined with the detection model in the audit system. If any parts of the traces are matched with traces of malware, the audit system notifies results to the secure agent, and the secure agent generates alarms on the device.

3.1. Whitelist/Blacklist Filtering Module. After the audit system downloads an application, the audit system checks whether the application is included in the whitelist or the blacklist of known applications or not. The purpose of the filtering module is to avoid unnecessary inspections of known applications or known malware. We designed the filtering module using a hash algorithm for fast comparison. We calculate a hash value of each known benign or malicious application and store it in the whitelist database or the

blacklist database. The file size of each application is also stored together to mitigate the hash collision problem. By comparing the hash value and the size of an application with those in the whitelist database or the blacklist database, the audit system can identify the known applications.

3.2. Application Rewriting Module. The application rewriting process consists of three major steps: (1) the application package is unzipped and the dex files are decompiled, (2) self-monitoring code to monitor invocations of dangerous APIs is inserted into the decompiled dex files, and (3) the rewritten files are recompiled and repacked to a new application package. The dex file is decompiled with the modified version of the baksmali tool [26] to produce smali files, and Apktool [27] is used to repack the files into a new application package. The dangerous APIs and the self-monitoring code are explained in the next subsections.

3.2.1. Dangerous APIs. In our framework, an application behavior is defined as a part of the API invocation trace of an application.

Monitoring all the API invocations can cause high overheads on the applications, and the size of the rewritten application will be increased significantly due to a large number of injected monitoring codes. Therefore, our framework decided to monitor only certain APIs that are mostly utilized to perform malicious activities, and, as a starting point, we selected APIs that were presented in [28] as dangerous APIs. Additionally, we investigated APIs that are explained in the Android Developers reference [29] and selected dangerous APIs which might be used in malicious activities.

Our selected dangerous APIs are classified into seven categories: APIs that access users' sensitive data, APIs that perform network activities or file activities, APIs that modify or access the device information, APIs that access or send SMS messages or Emails, APIs that access or execute services or installed applications, APIs that are used in API reflection or dynamic code loading, and APIs that affect the permission information.

3.2.2. Self-Monitoring Code. We adapted the hooking technique of the application rewriting framework introduced in [21] to monitor API invocations. The application rewriting module generates a class that contains the self-monitoring methods, and the class is imported into the original application.

The example of a smali file that contains the definition of the class is depicted in Figure 2.

The self-monitoring methods replace dangerous APIs in the original code. The self-monitoring method is invoked instead of the specified dangerous APIs, and it conducts three principal functions: delivery of the names of the invoked dangerous APIs to the secure agent, identifying the types of dangerous API, and the invocation of the original dangerous APIs.

(1) *Invocation trace delivery to the secure agent*: Each self-monitoring method contains code to deliver the names of the invoked dangerous APIs to the secure agent. Invocation traces are delivered using intent message communications. Sending an intent message whenever a dangerous API is invoked can degrade the performance of the mobile device, so the imported class contains a trace buffer to collect the traces to reduce the communication overhead. When the buffer becomes full, the buffer is wrapped in an intent message, and the message is sent to the secure agent. If a rewritten application communicates with the secure agent using intent messages insecurely, then the communication can be modified by malware. To prevent the attack on vulnerable intent-based communications, explicit intent messages with random tags are used to deliver the API invocation traces securely. The explicit intent message can specify its destination, and only the specified application can receive the messages while the other applications cannot eavesdrop the messages. By checking random tags in the messages, the secure agent can identify applications and also avoid the denial of service attacks.

(2) *Identifying the types of APIs*: The application rewriting module scans the dangerous API invocations and replaces invocation code of the APIs with the self-monitoring methods. Naturally, the dangerous APIs that originally supposed to be executed can be identified without any additional process. However, in the self-monitoring method, the name of API to be invoked is delivered as described previously. In the case of the JAVA reflection API, the additional process is needed to identify the name of API because the name of API is dynamically determined at runtime. To monitor the reflected API, the self-monitoring method for java.lang.reflect.Method.invoke() method gets the first parameter that contains the reflected API name. A lot of Android malware use the Java reflection API to hide their behaviors, so this process must be performed to improve the accuracy of API monitoring.

```
.class public Lsf/containterCls;
.super Ljava/lang/Object;
…
.method static sfMonitor_1(LContext;LString;)LObject;
 //get parameters of the original API
  …
  sget-object v0, Lsf/containterCls;->buffer
  //get buffer obj
  const/16 v1, 0x01
  invoke-static {v1}, LInteger;->valueOf(I)LInteger;
  move-result-object v1
  invoke-virtual {v0, v1}, LArrayList;->add(LObject;)Z
  //add the dangerous api number to the buffer
  invoke-static {}, Lsf/containterCls;->IsBufferFull()Z
  move-result v0
  if-eqz v0, :cond_0
  //check whether the buffer is full
  invoke-static {}, Lsf/containterCls;->sendIntent()V
  // the trace in the buffer is sent to the secure agent using intent msg
  …
  :cond_0
  invoke-virtual {p0, p1}, LContext;->getSystemService(
                                          LString;)LObject;
  //original API call
  move-result v0
  return-object v0
  //pass the return value
.end method
…
```

Figure 2: The example of the self-monitoring code.

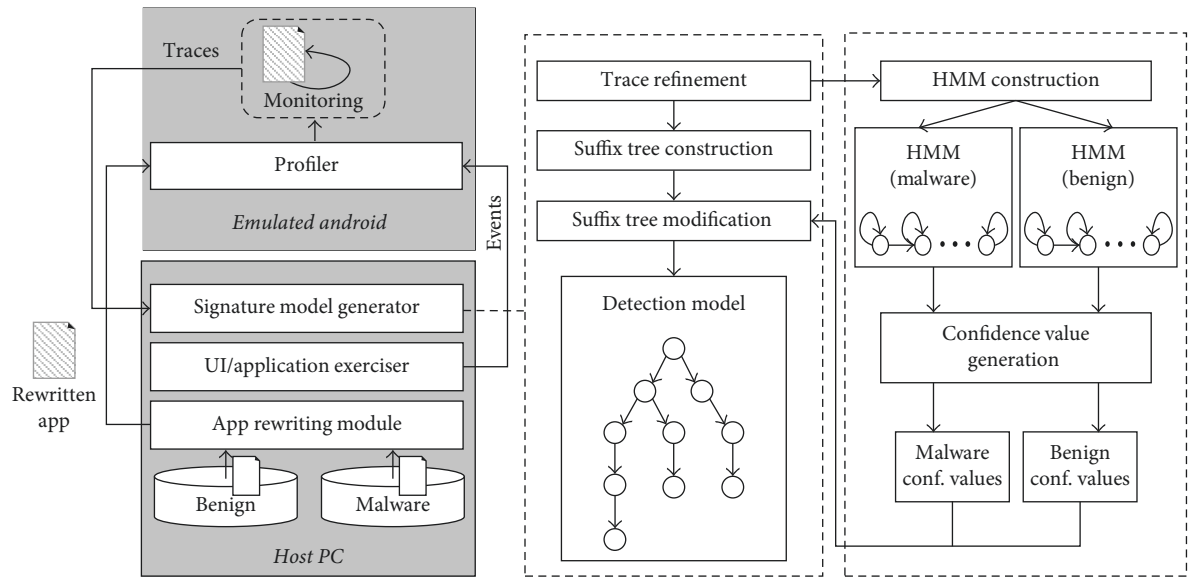

Figure 3: Malware detection model generation process in the detection model generation phase.

(3) *Invocation of the original APIs:* After the name of the dangerous APIs is collected, the original dangerous APIs are invoked and their return-values are also recorded by the self-monitoring code.

3.3. Malware Detection Module. The audit system examines the API invocation traces of an application using the detection model, and the detection model is built from known malware and known benign samples. We designed the detection model based on the suffix tree which represents the dangerous API invocation traces' degree of maliciousness and benignness. The suffix tree [30] is a data structure that is used to index all the substrings which are contained in a set of strings. The suffix tree can provide linear-time solutions to search substrings, and the suffix tree can only indicate which malware or benign samples are included in the input trace. So, in our detection method, we altered the suffix tree to have not only trace labels, but also each trace's confidence value that can be used in the malware detection. The malware detection module is mainly composed of two phases: the detection model generation phase and the trace examination phase. The details about these phases are explained in the next subsections.

3.3.1. Detection Model Generation Phase. To generate the detection model, we extracted API invocation traces from known malware and benign applications. The left part of Figure 3 shows how the API invocation traces of applications are extracted. Each application is rewritten by the application rewriting module, and it is executed by the UI/Application exerciser. While an application is executed, its API invocation traces are stored. The explanation about the application execution for the API invocation trace collection is given in Section 4.2.

After the API invocation traces of all the application samples are extracted, the detection model is generated as described on the right side of Figure 3.

All the API invocation traces are refined before the suffix tree is constructed. There are many consecutive elements in each trace, and those repeating elements are merged. For example, $[API_1, API_1, API_3]$ is refined to $[API_1, API_3]$.

The trace refinement process reduces the size of the API invocation trace of applications, and the decreasing rate of trace size of 1,000 malware and 1,000 benign samples was about 70.34% and 68.72% on average in our experiment.

In addition, the refinement process alleviates the effects of consecutive APIs on detection accuracy. Traces containing many consecutive elements were frequently extracted from both malware and benign applications. So, the repeating elements in a trace could cause false alarms in malware detection. After the trace refinement, the suffix tree is constructed with the refined traces using Ukkonen's algorithm [31].

In the left side of Figure 4, the example of the suffix tree is described. Each edge in the suffix tree is labeled with a subtrace of traces, and it is used as a transition condition. Each node has a list of trace labels whose subtraces are same with a certain prefix of the concatenation of all the edge labels in the path from the root to the node. If a trace is inputted to search the same trace in the suffix tree, nodes are visited by following the edges that are same with prefixes of the input trace. When a certain leaf node is reached, the node traversal ends. This means that the traces that reach a leaf node include a subtrace of samples indicated in the node. For example, if $[API_2, API_3, API_4, API_5]$ is the input trace, then the transition state will move along the shaded nodes in the Figure 4, and its destination node will be the leaf node that has mt.1 trace which means that the input trace is a subtrace of mt.1 $(= [API_1, API_2, API_3, API_4, API_5])$.

It is noted that even if a trace is smaller than an edge's trace and is a prefix of an edge's trace, a transition also occurs. The node followed by the edge is reached, and the node traversal ends. It happens because of the path compression in the Ukkonen's algorithm that merges the consecutive nodes that have only a single descendant to

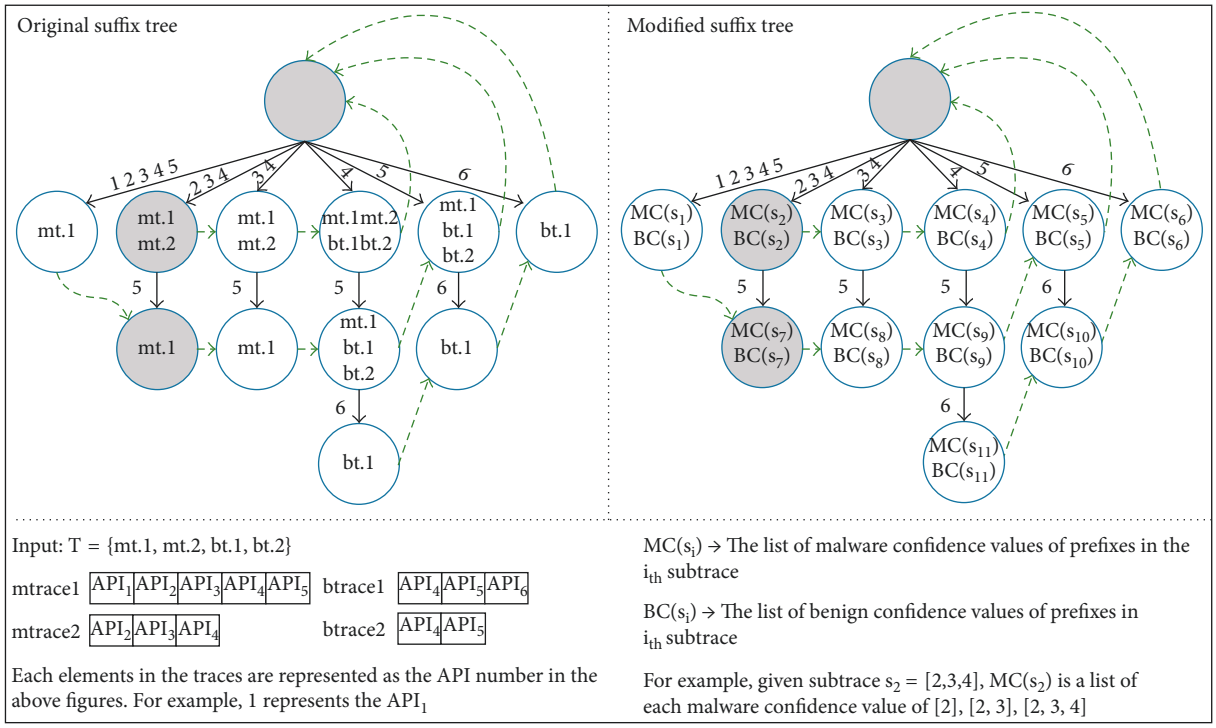

Figure 4: Example of suffix tree generation/modification: dotted line edges denote failure transition, and solid line edges denote state transitions. Ukkonen's algorithm is utilized to construct the suffix tree.

one. So, it is considered that the trace is matched to a certain implicit node.

After the initial suffix tree is constructed, the suffix tree is modified for the further malicious trace detection. While traversing the nodes in the suffix tree, the list of trace labels is replaced with the confidence values of prefixes in each edge's trace for both malware and benign applications. The example of the modified suffix tree is described in the right side of Figure 4.

We defined two confidence values to measure the input trace's maliciousness, and they are called as a malware confidence value and a benign-ware confidence value each. The document frequency ratio and the log-likelihood value are used to calculate the confidence values. The document frequency ratio is the proportion of malware or benign-ware invocation traces that include the subtrace to the total malware or benign-ware traces, and the malware or benign-ware log-likelihood value of a subtrace is determined using malware or benign-ware's HMM (Hidden Markov Model).

The HMM [32] is an algorithm which is widely used in various fields such as speech recognition, bioinformatics, and so on. For given observed variables, the HMM algorithm can predict the state transition as well as can estimate the probability of matches between the given data and the predefined model.

The HMMs for both malware and benign applications should be prepared to evaluate the log likelihood of each subtrace. For the HMM generation, the parameters such as the number of states, emission probabilities, transition probabilities, and initial probabilities are specified first.

Generally, there is no rule to specify the number of states, but we tried to find the number of states whose transition probabilities are evenly distributed after the model generation. As a result, the number of states is specified as five. The emission probabilities and initial probabilities are randomly selected. The observed variables are the dangerous API invocations in the trace.

Table 1: Notations for the confidence value calculation.

s	An observed sequence of dangerous API invocations, $(x_1, x_2, \ldots, x_T)$
dfr	Document frequency, the number of documents the term occurs in
A	Transition probability distribution, $\{a_{1,2}, \ldots, a_{N-1,N}\}$
B	Emission probability distribution, $\{b_1(x_1), \ldots, b_N(x_P)\}$
π	Initial probability distribution, $\{\pi_1, \ldots, \pi_N\}$
λ	Hidden Markov model, (A, B, π)
N	The number of states
K	The number of traces
P	The number of dangerous API invocations

Once the model is initialized, the parameters are reestimated using the Baum–Welch algorithm that is widely used as EM (expectation maximization) algorithm. The Baum–Welch algorithm [33] searches a locally optimal model that fits with the observed data. The parameters are updated until the maximum likelihood values of the updated models are converged. After the parameters are estimated completely, then the likelihoods of each subtrace are computed using the HMMs.

The malware and benign-ware confidence values are computed as described in (1–5), and the notations in the equations are explained in Table 1. As described in (5), the

```
1: procedure Trace Examination (trace, gmc_sum, gbc_sum, lmc_queue, lbc_queue)
2:     trlen←len(trace)
3:     swlen←len(sw)    ▷ get the len. of the sliding window
4:     i←0
5:     while i < trlen − swlen do
6:         sw←trace[i : i + swlen − 1]
7:         mc, bc←get_conf_from_suffixtree(sw)
8:         lmc_queue.enqueue(mc)
9:         lmc_queue.dequeue()
10:        lbc_queue.enqueue(bc)
11:        lbc_queue.dequeue()
12:        if lmc_queue.isfull() and lbc_queue.isfull() then
13:            lms←sum(lmc_queue)/lmc_queue.size
14:            lbs←sum(lbc_queue)/lbc_queue.size
15:        end if
16:        gcount←gcount + 1
17:        gmc_sum←gmc_sum + mc
18:        gbc_sum←gbc_sum + bc
19:        gms←gmc_sum/gcount
20:        gbs←gbc_sum/gcount
21:        if lms − lbs > lth or gms − gbs > gth then
22:            send_alarm(clientscok)
23:        end if
24:        i←i + 1
25:    end while
26: end procedure
```

Algorithm 1: Trace examination procedure.

confidence values of a trace, $\mathrm{mc}(s)$ and $\mathrm{bc}(s)$, are the results of the multiplication of the normalized log-likelihood values and the document frequence ratios. The min-max normalization is used to scale the log-likelihood values, $\mathrm{llh_m}(s)$ and $\mathrm{llh_b}(s)$. The document frequency, $\mathrm{dfr_m}(s)$ or $\mathrm{dfr_b}(s)$, is the proportion of the malware or benign-ware traces to the whole traces that contains s as a subtrace. The log likelihood, $\mathrm{llh_m}(s)$ or $\mathrm{llh_b}(s)$, is the probability of the observed sequence for the malware model or the benign-ware model. The log likelihood is computed in a recursive way using the forward algorithm. $\alpha_T^{(\log)}$ in (2) is a forward vairable, and the initial forward variable and the recursive forword vairable in the forword algorithm are defined in (3) and (4).

$$\mathrm{dfr_m}(s) = \frac{\mathrm{df_m}(s)}{\mathrm{df_T}(s)}, \qquad \mathrm{dfr_b}(s) = \frac{\mathrm{df_b}(s)}{\mathrm{df_T}(s)}, \tag{1}$$

$$\begin{aligned} \mathrm{llh_m}(s) &= \log P\left(s|\lambda_{\mathrm{m}}\right) = \left[\log \mathrm{sum}_{i=2}^{N-1}\left(\alpha_T^{(\log)}(i) + \log a_{i,N}^{\mathrm{m}}\right)\right], \\ \mathrm{llh_b}(s) &= \log P\left(s|\lambda_{\mathrm{b}}\right) = \left[\log \mathrm{sum}_{i=2}^{N-1}\left(\alpha_T^{(\log)}(i) + \log a_{i,N}^{\mathrm{b}}\right)\right], \end{aligned} \tag{2}$$

$$\alpha_1^{(\log)}(i) = \log a_{1,i} + \log b_i(x_1), 2 \le i \le N-1, \tag{3}$$

$$\begin{aligned} \alpha_{t+1}^{(\log)}(j) = &\left[\log \mathrm{sum}_{i=2}^{N-1}\left(\alpha_T^{(\log)}(i) + \log a_{i,j}^{m}\right)\right] \\ &+ \log b_j\left(x_{t+1}\right), 1 \le t \le T, 2 \le i, j \le N-1, \end{aligned} \tag{4}$$

$$\begin{aligned} \mathrm{mc}(s) &= \frac{\mathrm{dfr_m}(s) * \left(\mathrm{llh_m}(s) - \min_{0 \le i \le k}\left(\mathrm{llh_m}(s_k)\right)\right)}{\max_{0 \le i \le k}\left(\mathrm{llh_m}(s_k)\right) - \min_{0 \le i \le k}\left(\mathrm{llh_m}(s_k)\right)}, \\ \mathrm{bc}(s) &= \frac{\mathrm{dfr_b}(s) * \left(\mathrm{llh_b}(s) - \min_{0 \le i \le k}\left(\mathrm{llh_b}(s_k)\right)\right)}{\max_{0 \le i \le k}\left(\mathrm{llh_b}(s_k)\right) - \min_{0 \le i \le k}\left(\mathrm{llh_b}(s_k)\right)}. \end{aligned} \tag{5}$$

3.3.2. Trace Examination Phase. Once the detection model is generated, the audit system can examine the trace transmitted from the secure agent using the detection model. The confidence values in the detection model are used to compute the scores for the trace examination. The trace examination process is described in Algorithm 1.

First, the trace from the client is scanned with a sliding window, and each subtrace within the sliding window is searched in the suffix tree. The confidence values of each subtrace are retrieved from the suffix tree, and the malware and benign-ware confidence values are accumulated to calculate local scores and global scores.

Table 2: Malware samples used in the experiment.

ADRD	Anserverbot	Asroot
BaseBridge	BeanBot	Bgserv
CoinPirate	CruseWin	DogWars
DroidCoupon	DroidDeluxe	DroidDream
DroidDreamL	DroidKungFu1	DroidKungFu2
DroidKungFu3	DroidKungFu4	DroidKungFuS
DroidKungFuU	Endofday	FakeNetfilx
FakePlayer	GGTracker	GPSSMSSpy
GamblerSMS	Geinimi	GingerMaster
GoldDream	Gone60	HippoSMS
Jifake	KMin	LoveTrap
NickyBot	NickySpy	Pjapps
Plankton	RogueLem	RogueSPP
SMSReplic	SndApps	Spitmo
Tapsnake	Walkinwat	YZHC
Zitmo	Zsone	jSMSHider
zHash		

The local malware score or the local benign score represents the maliciousness or benignness of the traces in the prespecified local scoring period, and it is calculated by averaging the confidence values in the local scoring period. The global malware score or benign-ware score represents the maliciousness or benignness of the traces in the whole period, and it is also calculated by averaging the confidence values from the beginning to the present. After the local or global scores are calculated, it is checked whether the difference between the local scores and the global scores exceeds a certain threshold. If either of the comparison results exceeds the threshold, then the trace matching procedure concludes that the application is malware, and the alarm is sent to the client.

There are three parameters that affect the malware detection in the trace examination procedure: the size of sliding window, the local scoring period, and the score threshold. We have some experiments in order to find parameters that make the malware detection accuracy high, and the experimental results are explained in Section 4.3.1.

4. Experiments

4.1. Dataset. We evaluated our framework in terms of detection accuracy, runtime overhead, and the size overhead of rewritten applications. 1,260 malware samples from the Mal-Genome project [34] were used in the experiments, and the malware families are listed in Table 2. Meanwhile, 1,033 benign applications were downloaded from Google Play Store [35] between February and March 2016, and these applications are used in the experiments as benign samples. We implemented a tool that downloads applications from the official Google market, and to find out whether an application is benign or not, the online malware scanning service of VirusTotal [36] is used. The collection tool uses the VirusTotal APIs to request the tests and to get the scanning results. If all the virus-scanning tools consider an application as benign, then the application is stored as a benign sample.

4.2. Automatic Application Executions. The experiment to select suitable parameters that affect the accuracy was conducted first, and the experiment to measure performance metrics in best case was conducted to show the final results. In the experiments, 1,033 benign applications and 1,260 malware were used, and 80% of applications (826 benign applications and 1,008 malware) were used as the training dataset in the detection model generation, and the others were used as the test dataset.

Before conducting the experiments, we measured code coverages of applications to find out how many system events are required to observe application behaviors. To measure the code coverage, we rewrote 107 applications and added additional code in the applications. The code is inserted in each basic block, and the code logs unique values that are assigned to basic blocks. Whenever a basic block is executed, its unique value is logged. The rewritten applications were executed by MonkeyRunner [37], and by increasing the number of the events from 1,000 to 15,000, the number of the executed basic blocks was measured. The code coverage might be affected by the number of repeated executions. Therefore, we also executed the applications up to three times while measuring the number of unique basic blocks executed.

Figure 5 shows the results of the code coverage measurements. As shown in Figure 5, the code coverage rate is not increased linearly with the increasing number of the system events or the increasing number of executions. Even though there are possibilities of increasing number of events when applications are executed multiple times, the training time for malware detections is limited in practice. To find out practical parameters, we investigated the minimum number of events that produce the maximum code coverage rates for each application. We also computed the code coverage difference by the number of executions. As a result, the average of the minimum number of the events was 8,604, and when the applications were executed twice, the code coverage was increased significantly; 1.4% of the code coverage rate is increased. Therefore, in the training phase for malware detections, we executed the applications twice up to 9,000 system events.

4.3. Malware Detection Accuracy

4.3.1. Parameter Selection. To select the suitable parameters for detection, we measured the accuracies of our framework with different parameter values. The size of sliding window, the local scoring period, and the score threshold are parameters that are related to the detection accuracy. We specified the range of each parameter, and the malware detection accuracies were measured with the combinations of possible parameter values in the specified range. The size of the sliding window was in the range of 3 to 20, the local scoring period was in the range of 1 to 50, and the score threshold was in the range of −1 to 1. In addition, the malware detection accuracies were measured in various cases where the local score is only used or both local and global scores are used. Hereafter, the case that the local score is only used in the detection is referred to the local-only case, and the other case is referred to the local-global case.

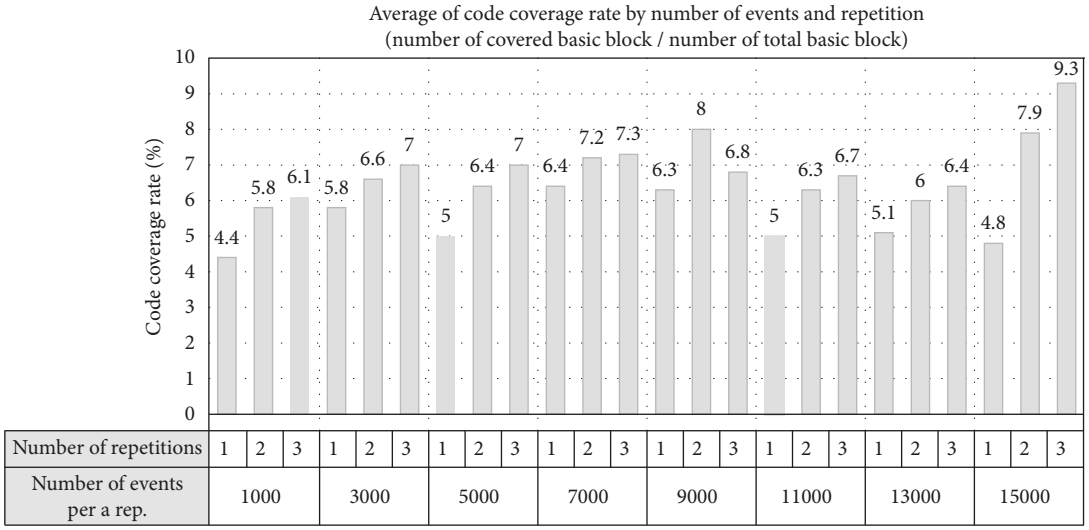

Figure 5: Basic block code coverage rate of the 107 applications.

Figure 6: Heap maps of the malware detection accuracy by the size of sliding window and the local scoring period (the min acc, 0.705, is expressed as red, and the max acc, 0.96, is expressed as green). (a) ACCs by the size of sliding window. (b) ACCs by the local scoring period. (c) ACC heat map of the local only case. (d) ACC heat map of the local-global case.

As shown in Figures 6(c) and 6(d), the two large heat maps show how the malware detection accuracy is changed with different parameter values of the size of the sliding window and the local scoring period. Each heat map is the result either when both of the local and global scores are used or when only the local score is used. To express the color of each cell in the heat maps, the maximum value among the accuracies of the different threshold is selected, while the size of the sliding window and the local scoring period is fixed. The selected maximum value is converted to the color of each cell. The highest value is represented by the green color, and the lowest value is with the red color.

The heat map from the accuracy measurement of the local-global case has more green-shaded cells compared to the other heat map. This means that malware detection accuracies are higher when both local and global scores are used together. When the size of the sliding window was 5 and the local scoring period was 8, the highest accuracy value, that is, 0.96, was obtained among the accuracy results in all the cases that we observed.

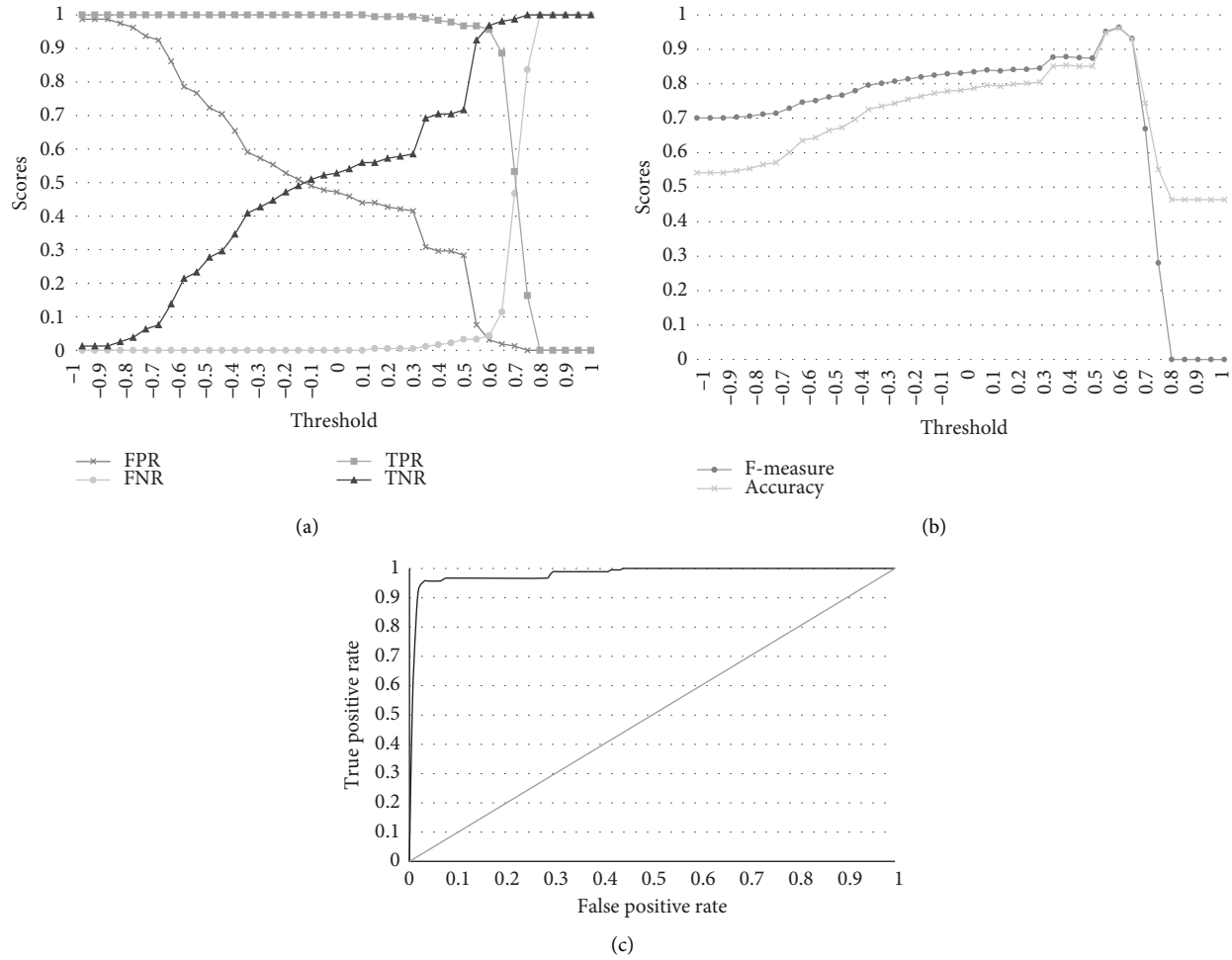

Figure 7: The evaluation results in terms of the malware detection performance. (a) FP/FN/TP/TN rate by the threshold. (b) F-measure/Accuracy by the threshold. (c) ROC (Receiver Operating Characteristic) Curve.

To figure out the impact of the size of the sliding window and the local scoring period, the measured detection accuracies are rearranged according to each parameter. Figures 6(a) and 6(b) show the rearranged heat map results. In order to show the results in a comprehensive manner, we expressed each cell in each heat map using the average or maximum value of the accuracies measured when a parameter is fixed to a certain value. The average and maximum values are calculated by scanning horizontally or vertically large heat maps represented in Figures 6(c) and 6(d).

There is no big difference between the local-only case and the local-global case in terms of the accuracy change by different size of the sliding window. The overall accuracy is increased by including the global score in the detection, but the accuracy change pattern still remains similar. The accuracy tends to be high at a certain size of the sliding window in both cases. The global score is less affected by the size of the sliding window, whereas the local score is highly associated with the size of the sliding window.

The sliding window whose size is in between 4 and 11 gives high accuracies in our malware detection. The detection process produced the best accuracy when the sliding window size was 5. In addition, the detection accuracy is lowered when the sliding window size is approaching 20, and this tendency is similar when the sliding window size is above 20.

In the local-only case, the accuracy increases as the local scoring period decreases. After the global score is applied, a range of the local scoring period showing a relatively high accuracy was discovered from the results. In detail, when a local scoring period in the range of 4 to 13 was used, the detection accuracy was measured as relatively high. The best local scoring period was 8.

4.3.2. Malware Detection Performance. We evaluated the malware detection performance of our proposed framework from various aspects when the size of the sliding window and the local scoring period were set to 5 and 8, respectively. Figure 7(a) shows the false positive rate, false negative rate, true positive rate, and true negative rate as the score threshold is varied. In addition, the F-measure and accuracy results are also described in Figure 7(b). When the score threshold was 0.6, our proposed framework produced the best result in the evaluation metrics. In the best case, the false

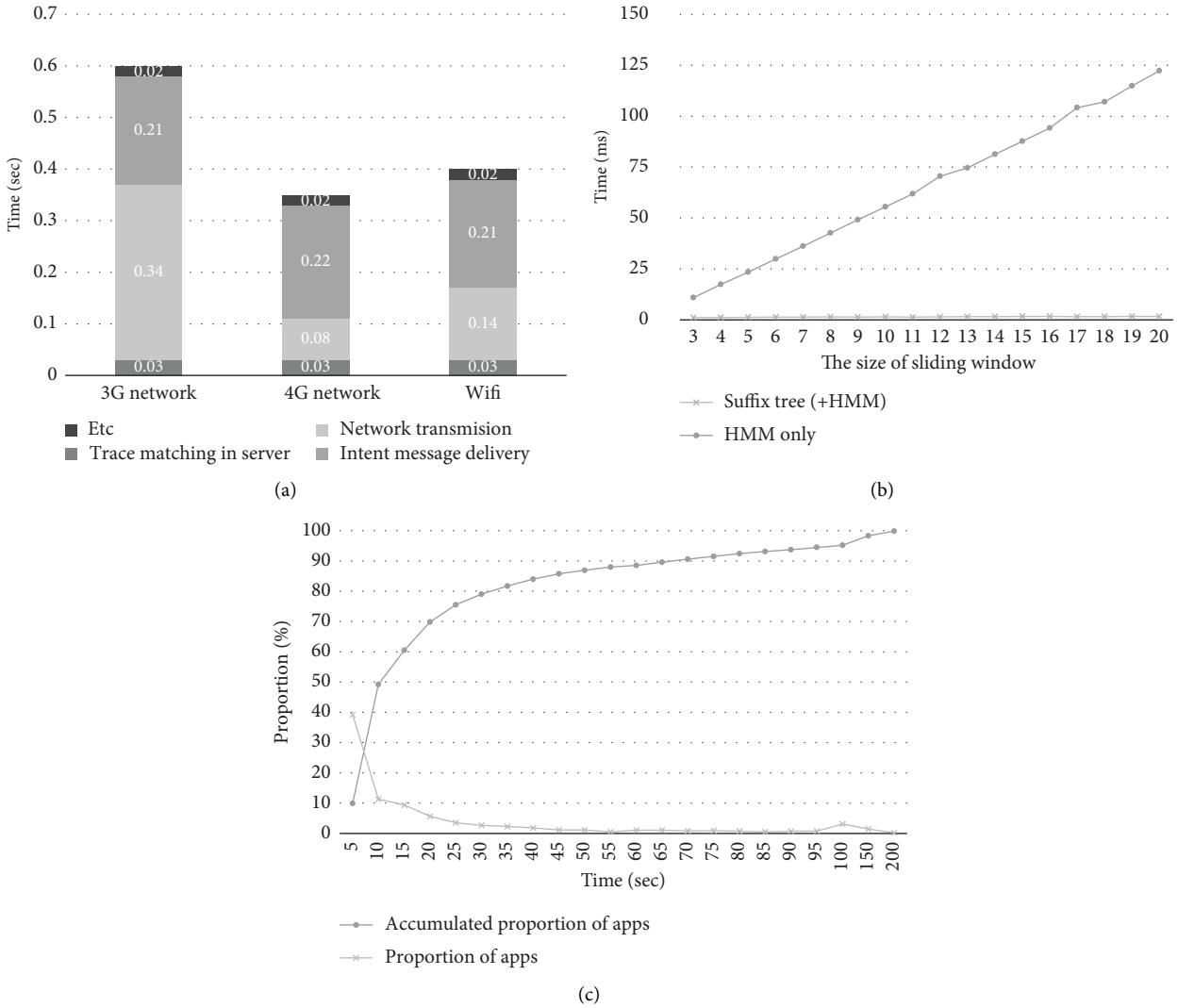

Figure 8: The evaluation results in terms of time overhead. (a) Trace processing time measurement. (b) Local and global scoring time measurement. (c) Application rewriting time measurement.

positive rate, false negative rate, true positive rate, and true negative rate were about 0.03, 0.04, 0.96, and 0.97, respectively, and the F-measure and accuracy were about 0.96.

As shown in Figure 7(a), when the score threshold was in between −1 and 0.1, the false negative rate and the true positive rate were 0 and 1, respectively. When the score threshold was less than 0.1, all the samples were determined as malware.

As the score threshold increased from 0.1, the accuracy was improved gradually. The false negative rate slightly increased from 0, and the true positive rate slightly decreased from 1. At the same time, the false positive rate decreased rapidly from 0.44 and the true negative rate rapidly increased from 0.56. When the score threshold was close to 0.6, the true positive rate and true negative rate became 0.96 and 0.97 each. At the same time, the false positive rate and false negative rate also became 0.03 and 0.04 each. When the score threshold was 0.7 or more, the true negative rate and false positive rate remained 1 and 0, respectively, due to the fact that most of the samples were classified as benign samples.

In addition to these evaluation metrics, the ROC (Receiver Operating Characteristic) curve is depicted in Figure 7(c). The ROC curve is a graphical plot that illustrates the robustness of malware detection in our proposed framework as the score threshold is varied. The ROC curve is created by plotting the true positive rate against the false positive rate with various score thresholds. As shown in Figure 7(c), the plotted points are above the diagonal, which represents good detection results. The curve line is close to the upper left corner of the ROC area, which indicates that the malware detection test of the framework has high discriminating capabilities. In detail, the AUC (area under the curve) was about 0.98. The AUC is the evaluation metric to summarize the performance of the classifier.

4.4. Run-Time Overheads. The end-to-end overhead, API monitoring overhead, and application rewriting overhead were measured to evaluate the efficiency of our framework. The end-to-end overhead shows how efficiently our client-server based framework works to process a trace generated

from an application, and the API monitoring overhead shows the time added by the self-monitoring in the rewritten application. From these two evaluation factors, it is possible to estimate the time required for the detection process in our framework. Even though the application rewriting is performed in the preprocessing phase, we also measured the application rewriting time to show the usability of the framework. We used Samsung Galaxy S4 with an Exynos 5 Octa 5410 CPU and 2 GB RAM. The operating system of the device was Android 5.0.1. We also used a Windows 7 machine with an Intel Core i5-4570 CPU and a 32 GB RAM as the audit system.

(1) *End-to-end time*: We had two kinds of experiments to evaluate the end-to-end time overheads of our framework. In the first experiment, we measured the end-to-end time from when the secure agent receives the dangerous API trace to when the secure agent receives the examination result from the audit system. One hundred applications were used in the experiment, and the time overhead was measured in three network environments: 3G network, 4G network, and wifi network. Since the secure agent and the audit system communicate over the public network, the SSL (Secure Socket Layer) protocol is used in order to protect the communications from potential attacks. The detailed setting of SSL is TLS_RSA_WITH_AES_128_CBC_SHA. In addition, the size of trace which is transmitted from the client was set to 20, and each transmitted trace is scanned using the sliding window of length 5. The local scoring period was set to 8.

As shown in Figure 8(a), the average time to process a transmitted trace whose length is 20 did not exceed 0.6 seconds in all the network environments. The average time for the network transmissions in 3G network was longer than the others, and there were no significant differences between 4G network and wifi network. Naturally, the difference between the end-to-end times of different network environments depends on the network transmission time. Except for the network transmission, the trace processing in the server took the longest time. The server, that is, the audit system, performs many jobs such as the reception of the dangerous API invocation traces, encryption and decryption for the communications, the trace examination, the transmission of the examination results, and so on. Among these jobs in the server, the trace examination procedure was evaluated through the additional experiment.

Our framework detects malware based on the local and global scores, and each score is calculated based on the confidence values. Therefore, the execution time of examination procedure depends on the computation of the confidence values of the given trace. In order to measure the effect of the suffix tree, we repeatedly measured time to calculate the local score and the global score in the second experiment. We assumed that the audit server has the trace of length 296 which is delivered from the client, and the trace from the client is examined with the different size of sliding windows. The trace in the audit server was the real trace of the AnserverBot malware. In addition, the local scoring period was set to 8, and the number of symbols in the HMMs was specified as five. The scoring time was measured 20 times for each size of the sliding window, and the average value of scoring time according to the size of sliding window is shown as graphs in Figure 8(b).

Table 3: Scoring time using suffix tree and HMM.

The size of sliding window	Suffix tree-based scoring time, $t1$ (ms)	HMM-based scoring time, $t2$ (ms)	$t1/t2$ (ratio)
3	1.1	10.9	9.6
4	1.1	17.3	15.5
5	1.2	23.4	19
6	1.3	29.8	23.2
7	1.3	36.1	27.1
8	1.5	42.6	29
9	1.3	49.1	37.3
10	1.4	55.5	38.7
11	1.3	61.8	48.3
12	1.4	70.4	49.5
13	1.6	74.6	47.3
14	1.5	81.3	52.9
15	1.7	87.6	52.6
16	1.6	94.1	58.2
17	1.6	104.1	66.1
18	1.6	107	66.3
19	1.7	114.9	67.4
20	1.7	122.2	71.5

When the suffix tree that contains the condence values is used, all the measured times are reduced significantly. The scoring time when the HMM is used alone was about 10–72 times of the scoring time when the suffix tree is used. Table 3 shows the measured average time of both cases. The suffix tree-based scoring time ranged from 1.1 to 1.7 ms, whereas the HMM-based scoring time ranged from 10.9 to 122.2 ms. Assumed that the amount of the traces from many clients can be very large, the trace examination should be performed as fast as possible. Therefore, the suffix tree containing the precomputed condence values is suitable to be utilized in the trace examination procedure.

(2) *Application rewriting time*: We measured the time for the application rewriting including the application package decompiling and the self-monitoring code insertion. 1,617 applications were used in these experiments. The measured time ranges from 1.52 seconds to 197.41 seconds, and its average was 23.81 seconds. As shown in Figure 8(c), about 80% of the applications were rewritten within 30 seconds.

(3) *Self-monitoring time*: We also measured the overheads of the rewritten applications which are generated with the added self-monitoring methods. To evaluate the performance degradations caused by API invocation monitoring, we implemented a program that repeatedly invokes FileWriter.append() to write a string to a file. The buffer size which is same with the transmission trace unit in the secure agent was set to 20. Table 4 shows that the time for API invocations are increased by the self-monitoring method. As a result, the average of the increased time per one invocation was about 0.013 ms.

4.5. Size Increase of Rewritten Applications. The application rewriting module inserts the self-monitoring code into the dex file of the application. 1,645 applications were used in our experiments. We measured the size increases of the dex

Table 4: API execution time overhead by self-monitoring.

		Number of repetitions				
		1000	2000	3000	4000	5000
Original app	Total time (ms)	28	63	150	145	197
	Average time per an invocation (ms)	0.028	0.032	0.05	0.036	0.039
Rewritten app	Total time (ms)	59	75	152	205	255
	Average time per an invocation (ms)	0.059	0.038	0.051	0.051	0.051

Table 5: Size increase by app rewriting.

Percentage increase (=size increase/original size)	Number of applications
$0\% \le$ P.I. $\le 0.5\%$	1285
$0.5\% \le$ P.I. $\le 1.0\%$	237
$1.0\% \le$ P.I. $\le 1.5\%$	68
$1.5\% \le$ P.I. $\le 2\%$	13
$2.0\% \le$ P.I. $\le 2.5\%$	7
$2.5\% \le$ P.I. $\le 3.0\%$	1
$3.0\% \le$ P.I.	34

files in the rewritten applications and calculated the percentage of the increased size of each dex file. The average of the size increase was 25.17 kB. The minimum size increase and the maximum size increase were 12.56 kB and 47.13 kB, respectively. The percentage increases of the applications are shown in Table 5, and the percentage increases of most applications were less than 1, which is negligible.

4.6. Comparison with Other Methods. We had an experiment to compare our framework with other detection methods. In the experiment, we evaluated n-gram frequency-based frameworks using general classification methods which are widely used in dynamic analysis-based detection. The n-gram frequency of the API trace is measured, and it is inputted to the classification algorithm. The bi-gram whose size of n-gram is 2 was used in the frameworks.

Table 6 shows the performance result of our framework and the n-gram frequency-based detection methods. As shown in the table, we used three different kinds of classification algorithm in the n-gram based frameworks, and the sliding window size and the local scoring period of our framework were set to 5 and 8 each. 1,033 benign applications and 1,260 malware were used in this experiment, and 80% of total applications (826 benign applications and 1,008 malware) were used as the training dataset, and the others were used as the test dataset.

The results show that our framework produces higher detection accuracy compared to the other methods. In addition, the n-gram frequency methods using classification algorithms are not designed to be used in the runtime detection, and it needs the sufficient amount of API traces for more accurate detection. To achieve the accuracies presented in Table 6, 475 API invocations on average were collected and used in the audit system. It means that traces of length 20 should be transmitted through the network at least 20 times. In contrast, our framework can detect the malware in near real time with the relatively small amount of traces while showing higher detection accuracy. On average, our framework was able to detect the malware using only 4% of traces that were used in n-gram based methods.

Table 6: Comparison with n-gram frequency-based detection methods.

Method	TPR	FPR	TNR	FNR	Accuracy
n-gram (RandomForest)	0.95	0.05	0.95	0.05	0.95
n-gram (DecisionTree)	0.93	0.07	0.93	0.07	0.93
n-gram (NaiveBayes)	0.76	0.25	0.75	0.24	0.76
Ours	0.96	0.03	0.97	0.04	0.96

5. Discussion

Rewritten applications are resigned with a new signature. Therefore, if the rewritten application tries to update itself, then it would be failed because of the different signatures. If the updated version of an application is available to download from the online market, then it is possible to apply our mechanism to the updated application for further analysis. In another case, to mitigate the update problem, a simple heuristic solution can be applied to our framework, as follows: the secure agent prohibits the autoupdate of applications, and if the new dangerous API traces from the applications are not generated in a certain period of time or a user wants to update the application directly, then the original applications are provided by the audit system. There is no proper period of time that guarantees that an application is not malicious, and so the period should be specified by considering two factors: usability and security.

6. Conclusion

In this paper, we proposed a novel Android malware detection framework that uses the application rewriting technique for application behavior monitoring without any kernel modifications. Furthermore, we designed the framework as a client-server model and deployed the analysis processes for malware detection on the server-side. The client component only monitors applications, and the server component examines the traces using the suffix tree algorithm. In the detection process, we designed the suffix tree-based model to have the confidence values that are precomputed using the document frequency ratios and the likelihood values from the HMM. In addition, we also proposed the scoring method to make the final decision for the malware detection. We made efforts for our framework not only to have the high detection accuracy, but also to be efficient enough to examine the application behaviors at runtime.

In the evaluation, we conducted the experiments to show the performance of our proposed framework in terms of

detection accuracy, runtime overhead, and size overhead of rewritten application. The framework could achieve about 96% malware detection accuracy in the best case, the average time for processing each trace from the secure agent and the audit system was less than 0.6 seconds in three different network environments. In addition, the newly proposed detection method only took 1-2 ms to process about three hundred subtraces. This means that our score based detection barely affects the end-to-end time overhead.

7. Future Work

The event generations to trigger behaviors of applications still remain as the future work. We figured out the appropriate way to improve the code coverage with MonkeyRunner. However, if there is an elaborate way to generate the events considering execution paths in an application, the performance of dynamic analysis can be improved significantly. Some existing researches on automated testing [38, 39, 40] have been proposed, but their approaches use heuristic ways to generate events. We plan to apply the symbolic execution methods to our framework to trigger application behaviors.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (no. 2017-0-00388, Development of Defense Technologies against Ransomware) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (no. NRF-2016R1A2B4015254). This work was supported by the Human Resources program in Energy Technology (No. 20174010201170) of the Korea Institute of Energy Technology Evaluation and Planning (KETEP) grant funded by the Ministry of Trade, Industry and Energy of the Korea government.

References

[1] N. Islam and R. Want, "Smartphones: past, present, and future," *IEEE Pervasive Computing*, vol. 4, no. 13, pp. 89–92, 2014.

[2] G DATA Security Blog, *8,400 New Android Malware Samples Everyday*, G DATA Software, Bochum, Germany, 2017.

[3] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pp. 329–334, Hangzhou, China, May 2013.

[4] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the Seventh International Conference on Computational Intelligence and Security (CIS)*, pp. 1011–1015, Hainan, China, December 2011.

[5] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pp. 569–584, Bellevue, WA, USA, August 2012.

[6] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Vee, and C. Platzer, "Andrubis 1,000,000 apps later: a view on current android malware behaviors," in *Proceedings of the 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pp. 3–17, Wroclaw, Poland, September 2014.

[7] W. Enck, P. Gilbert, S. Han et al., "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 1–29, 2014.

[8] 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.

[9] A. D. Schmidt, F. Peters, F. Lamour, C. Scheel, S. A. Camtepe, and S. Albayrak, "Monitoring smartphones for anomaly detection," *Mobile Networks and Applications*, vol. 14, no. 1, pp. 92–106, 2009.

[10] T. Blasing, 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)*, pp. 55–62, Nancy, France, October 2010.

[11] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: automated familial classication of Android malware," in *Proceedings of the ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, , San Diego, CA, USA, January 2014.

[12] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu, "Droidmat: Android malware detection through manifest and API calls tracing," in *Proceedings of the Seventh Asia Joint Conference on Information Security (Asia JCIS)*, pp. 62–69, August 2012.

[13] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2014.

[14] S. Y. Yerima, S. Sezer, and G. McWilliams, "Analysis of Bayesian classication-based approaches for android malware detection," *IET Information Security*, vol. 8, no. 1, pp. 25–36, 2014.

[15] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1105–1116, Scottsdale, AZ, USA, November 2014.

[16] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 229–240, Raleigh, NC, USA, October 2012.

[17] P. P. Chan, L. C. Hui, and S. M. Yiu, "Droidchecker: analyzing android applications for capability leak," in *Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks*, pp. 125–136, Tucson, AZ, USA, December 2012.

[18] K. Lu, Z. Li, V. P. Kemerlis et al., "Checking more and alerting less: detecting privacy leakages via enhanced data-ow analysis and peer voting," in *Proceedings of the 2015 Network and*

Distributed System Security (NDSS), San Diego, CA, USA, February 2015.

[19] F. Wei, S. Roy, X. Ou et al., "Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329–1341, Scottsdale, AZ, USA, November 2014.

[20] 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, Chicago, IL, USA, October 2011.

[21] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: a rewriting framework for in-app reference monitors for android applications," *Mobile Security Technologies*, vol. 2012, no. 2, pp. 1–7, 2012.

[22] H. Hao, V. Singh, and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, pp. 25–36, Hangzhou, China, May 2013.

[23] D. Schreckling, S. Huber, F. Hohne, and J. Posegga, "Uranos: user-guided rewriting for plugin-enabled android application security," in *Proceedings of the International Workshop on Information Security Theory and Practices*, pp. 50–65, Heraklion, Crete, Greece, June 2013.

[24] R. Xu, H. Sadi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pp. 539–552, Bellevue, WA, USA, August 2012.

[25] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: full-edged app sandboxing for stock android," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pp. 691–706, Washington, DC, USA, August 2015.

[26] Baksmali: https://baksmali.com.

[27] Apktool: http://code.google.com/p/Android-apktool/.

[28] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: mining api-level features for robust malware detection in android," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, pp. 86–103, Sydney, Australia, September 2013.

[29] Package Index: https://developer.android.com/reference.

[30] M. Farach, "Optimal suffix tree construction with large alphabets," in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pp. 137–143, Miami Beech, FL, USA, October 1997.

[31] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[32] L. R. Rabiner and B. H. Juang, "An introduction to hidden Markov models," *IEEE ASSP Magazine*, vol. 3, no. 1, pp. 4–16, 1986.

[33] L. R. Welch, "Hidden Markov models and the Baum–Welch algorithm," *IEEE Information Theory Society Newsletter*, vol. 53, no. 4, pp. 10–13, 2003.

[34] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, CA, USA, May 2012.

[35] Google Play Store: https://play.google.com/store/apps.

[36] VirusTotal: https://www.virustotal.com/.

[37] MonkeyRunner: https://developer.android.com/studio/test/monkeyrunner.

[38] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pp. 224–234, Saint Petersburg, Russia, August 2013.

[39] T. Azim and I. Neamtiu, "Targeted and depth-rst exploration for systematic testing of android apps," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 641–660, 2013.

[40] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 204–217, Bretton Woods, NH, USA, June 2014.