Hindawi
Mobile Information Systems
Volume 2020, Article ID 8863385, 9 pages
https://doi.org/10.1155/2020/8863385

Research Article

HAL-Based Resource Manipulation Monitoring on AOSP

Thien-Phuc Doan, Jungsoo Park, and Souhwan Jung

Communication Network Security Laboratory, Soongsil University, Seoul 06978, Republic of Korea

Correspondence should be addressed to Souhwan Jung; souhwanj@ssu.ac.kr

Received 25 September 2020; Revised 29 October 2020; Accepted 23 November 2020; Published 2 December 2020

Academic Editor: Vishal Sharma

Nowadays, Android malware uses sensitive APIs to manipulate an Android device's resources frequently. Conventional malware analysis uses hooking techniques to detect this harmful behavior. However, this approach is facing many problems, such as low coverage rate and computational overhead. To solve this problem, we proposed *HALWatcher*, an alternative technique to monitor resource manipulation on Android Open Source Project (AOSP). By modifying Hardware Abstract Layer (HAL) resource accessing interfaces and their implementation, we can embed more monitoring functions at critical methods that are in charge of transferring data between the Hardware Driver and the Framework Layer. Hence, *HALWatcher* provides a lightweight and high coverage rate system that can perform resource manipulation monitoring for Android OS. In this paper, we prove that the hooking technique is limited in detecting resource manipulation attacks. Besides that, *HALWatcher* shows an outperform detection rate with a low computational effort.

1. Introduction

Many studies have been published in recent years on malicious code on Android devices [1–5]. They have invested a huge effort to generate effective architectures to defend against Android malware [6]. Generally, a detection solution has two main parts: Malware analysis and Detection algorithm. The analysis part can be done by two main approaches: Static and Dynamic Approach. In the end, this part provides a set of patterns or features which are fed to the Detection algorithm.

Both Static and Dynamic approaches try to figure out the malicious behaviors of malware. Resource manipulation is one of the most popular harmful attacks. By different techniques, the malicious apps manipulate user's device resources, e.g., Camera, Phone, and SMS, to steal sensitive information or send messages to premium numbers without user awareness [7]. The privilege escalation attack is even more dangerous. They can take over device resources without user interaction. So there is a great need to design a detecting model for resource manipulation attacks.

Existing analysis techniques can be applied to solve this problem. One of the typical static analysis approaches is building a flowgraph based on critical API calls. Based on that, we can deduce what resources the malicious app manipulates. However, Android malicious samples have been obfuscated or encrypted using various evasion techniques. This difficulty significantly decreases detection accuracy. Dynamic behavior analysis seems to be a good supplementary solution. This approach uses hooking tools, such as xPosed, Frida, etc., to monitor and trace malware behaviors. There are many sensitive APIs that are related to controlling the device's resources. Hooking into all of these APIs may cause the computational overhead problem due to mobile devices' limited computing resources. Therefore, malicious apps are often crashed while the analysis is operating. Moreover, hooking tools are detectable due to its direct interference with the process's memory.

To address the limitation of hooking techniques, we design *HALWatcher*, a general method for monitoring Android hardware resources inside Hardware Abstract Layer (HAL). The idea is that HAL provides the interface for the communication between the Android Framework Layer, which handles the requests of getting resources from applications or processes, and the Hardware Driver inside the kernel layer. By modifying HAL interfaces and

implementation codes, we can keep track of all the manipulating hardware resources without any knowledge requirement about various vendors' hardware drivers. Moreover, *HALWatcher* does not require root permission because it is already working as a part of the Android system. Furthermore, the detection of *HALWatcher* in the system is almost impossible because of the various ways of modifying HAL in such a large number of developers. Although the flexibility is not high, our method dramatically reduces the amount of data collection work from dynamic analysis while also providing sufficient information for Android devices' protection service against bad actors.

HALWatcher architecture can be applied to develop various applications both in research and industry field. It is a useful technology to track malware behaviors, then constructing a complete dataset for dynamic analysis is achievable. Besides, this technology is suitable for all Android mobile device hardware because it only interferes with Hardware Abstract Layer. Therefore, developing a hardware resources manipulation system on real-world Android devices is uncomplicated. Moreover, root privilege is nonessential for *HALWatcher*, in which other hooking frameworks are strongly dependent.

In summary, this paper has the following contributions.

We demonstrated that the hooking techniques might not be useful for detecting resource manipulation attacks.

We proposed *HALWatcher*, an efficient and lightweight method to detect resource manipulation attacks. By modifying HAL, this module runs along with the Android system so that it is almost undetectable.

The rest of this paper is organized as follows: the background of HAL and resource manipulation monitoring is introduced in the second section. In the third section, we discuss how to detect resource manipulation attacks. After that, we present *HALWatcher*, a HAL based resource monitoring system running along with the Android OS. The implementation and the design of experiments are shown in the next section. Finally, we will discuss future work and conclusions about our work.

2. Background and Related Work

2.1. Resources Manipulation. Android malware analysis is well-studied nowadays [8]. Analytical techniques include dynamic analysis [9–13] and static analysis [5, 14–17]. Some frameworks seek to classify malicious code through application behavior following signature [9, 14], while others track data flow [15, 18]. Static analysis is the way to understand the application by finding the signatures of malicious code (e.g., permissions that the application has declared, APIs that the application uses). Dynamic analysis directly executes malicious apps in a sandbox environment [3], then collects the necessary information and organizes them to process. The data from dynamic analysis and static analysis are then fed into algorithms to assess application behavior. In particular, the application of machine learning and deep learning in the classification of malicious apps is trendy due to its high accuracy [6, 12, 19–22].

Data collection from the dynamic and static analysis has always faced many difficulties and obstacles [4]. Evasion techniques make it difficult for static analysis to locate the used APIs or to figure out the execution flow of data [15]. Meanwhile, dynamic analysis has difficulty finding ways to execute all the behavior of the application being analyzed automatically [10], along with a large amount of information that may not be needed after running malicious apps. However, we must recognize the flexibility that current dynamic analytical techniques are very high. The hybrid approach combines static analysis and dynamic analysis to solve the limitations of each technique [4, 23]. Wong et al. proposed IntelliDroid [24] that generates input for the dynamic analysis using the static analysis technique.

Malware behavior tracking is a common problem. One of the efficient ways to track malicious behavior is to detect resources that are manipulated by malware from an Android device. Almost all attackers' purposes are trying to steal sensitive data from the user by manipulating the phone resources such as CAMERA, MICROPHONE, PHONE, and SMS. Jiang et al. designed a resource management system architecture to collect data for behavior detection [25]. Static analysis is limited to detect unauthorized resource usage. Meng et al. constructed a graph-based model to describe the control flow of an application. However, their approach does not seem much effective with 89.5% precision [26]. Zhao et al. leveraged the power of Androguard to extract a set of sensitive APIs to represent the application's behavior [27]. The dynamic analysis uses hooking techniques. Using Java function hooking technology, Soewito and Suwandary successfully illustrate that their proposal is applicable to data leakage prevention [28]. Hooking technologies are easy to install and detectable to monitor resource manipulation. For instance, Frida, a hooking framework, interferes with the application's memory, which process it needs to analyze. The agent and then needs high privileged access because the Linux kernel does not allow any processes to interfere with each other's memory without authorization. Therefore, to use the hooking techniques, the device must be rooted, or the agent of the hooking framework must be attached to the application they want to analyze.

Frida framework has two ways to hook the function of target APK. First, it needs to run *frida server* inside the devices as root permission or nonroot permission with enough capability to access other processes 'memory. The *frida server* then modifies the memory to overwrite the functions which are specified in the JavaScript-based hooking script. The target app will use the overwritten function instead of the original function for its execution. For the second way of using Frida, we need to attach the *frida-agent.so* library into the target APK and repackage the APK. *frida-agent.so* then acts as *frida server* but with no root privileges requirement because the attaching agent is now a part of the application to fully access its memory.

Strace is a possible solution to hide from evasion malware. However, the massive log of the system call is quite complex to process.

2.2. Hardware Abstract Layer. A HAL (Hardware Abstract Layer)[1] defines a standard interface for hardware vendors to implement, enabling Android to be agnostic about lower-level driver implementation. Using a HAL allows you to implement functionality without affecting or modifying the higher-level or lower-level system. The legacy HALs is the old architecture for Android Nougat (7.0) and the previous versions. In Android 8.0 and higher, the architecture is designed to meet the requirements of modularity.

3. Resource Manipulation Attack Detection

Currently, the dynamic analysis approach can use the hooking technique for detecting resource manipulation attacks. Figure 1(a) describes the method of using Frida to keep track of the *SendSMS* function. To know whether the app manipulates SMS resources by requesting SendSMS or not, we hook into *sendTextMessage()*. The logging method is used in this example to gather the manipulation information. We found some disadvantages to this method.

Detectable. Hooking techniques require access to application memory during the analysis. Self-checking memory is one way to figure out the strange agents (e.g., *frida-agent.so*). Besides, the requirement of root privileges (i.e., in the case of *frida server*) makes it exposed to the Android system. Darvin claims in his blog[2] that there are many ways for an application to detect the existence of Frida inside the execution environment. Szczepanik et al. proposed an algorithm using stack-trace on detecting hooking tools[29].

Messy or Imperfect Data. Some SDK APIs call each other when the app requests a resource. Even the analyzer tries to reduce the number of sensitive APIs, but this is hard work. On the other hand, some APIs might be missed in the hooking list leading to the increasing of *False-positive* and *False-negative*.

Inapplicable to End-User Products. Most of the hooking techniques are applied in solving behavior analysis problems. It is hard to include these techniques into real end-user products due to the risk of misappropriation for wrong usage (i.e., bypass the protection mechanisms of apps).

Modifying HAL is the best choice for monitoring manipulation resources for many reasons. Firstly, all hardware resource requests go through HAL. Therefore, monitoring resources based on HAL gives a high coverage rate. Secondly, HAL is independent of the hardware driver. The monitoring module in HAL can work correctly for a wide range of Android devices. Last but not least, even the attack aimed to get rooted in the Android device, it cannot disable the monitoring module in HAL because this module is not running as a service, a part of the Android Operation System. We started to investigate the HAL source code and then came to these conclusions.

First, we can simply add more functions to monitor the manipulating resources with a small coding effort. Listing 1 shows a simple logging code of *sendSms()* function inside HAL. *Line 5* is the only code that we need to add. On the other side, Frida needs more effort (i.e., see Listing 1 to hook into *sendTextMessage(),* which will request for sending SMS (i.e., the same resource of example in Listing 1).

Second, the information collected from HAL interfaces or functions is sufficient for detecting resource manipulation attacks. There are multiple Android APIs that act the same behavior. For instance, both *sendTextMessage()* and *sendMultipartTextMessage()* can be used to send SMS through radio network. Moreover, *sendTextMessage()* have 2 different overloading methods. Therefore, there is a need to develop two hooking functions for each *sendTextMessage()* to cover all the resource manipulation APIs. Besides, HALWatcher performs monitoring procedure accurately by adding one line of code (i.e., for logging) into the *sendSms()* implementation function (i.e., for the sendSms interface) as shown in Listing 2.

4. *HALWatcher*: Resource Manipulation Monitoring Module

HALWatcher, as shown in Figure 2, then works as a part of the Android Operation System. Therefore, there is no requirement of the rooted system or repackaging the target application. All of the installed packages from the Play store or other Vendor market can be monitored. Besides, the process generated from a Remote Code Execution (RCE) attack is also under monitoring. Moreover, because of the diversity of vendor Android firmware (or ROM) types, the detection of *HALWatcher* is almost impossible. Our model generates information whenever the resource requests the hardware. Therefore, the amount of information (e.g., logs) is significantly reduced but ensures that all resource manipulation behavior is recorded and reported. In the next subsection, we will give some examples of how to build *HALWatcher* in many types of hardware resources.

Based on the previous section's conclusions examining the HAL source code, we provide a detailed design for *HALWatcher*. First of all, all requests to access hardware information and resources will start from the Framework Layer, namely, the Java Native Interface (JNI). We consider the Malware or RCE attack in equal measure because all hardware resource manipulating requests must go through the JNI. The information will then be moved down to the Hardware Abstraction Layer (HAL). In HAL, interfaces are feature independent; that is, there are no interfaces that share the same purpose. At the critical methods of each resource type (which we discuss in more detail in the next section), we embed one or more code lines to record any action and related information about the manipulated resource. These code lines are called *resource monitoring modules*. Listing 2 shows an example of one resource monitoring module. At line number 5, we add one line of code into the *RadioImpl::sendSms* interface to monitor the SMS resource by logging whenever this interface is called. In short, all resource monitoring modules are developed following three steps: figure out resources implementing interface source code in HAL, embed monitoring functions into the interfaces, manage, and send monitoring information to monitoring service.

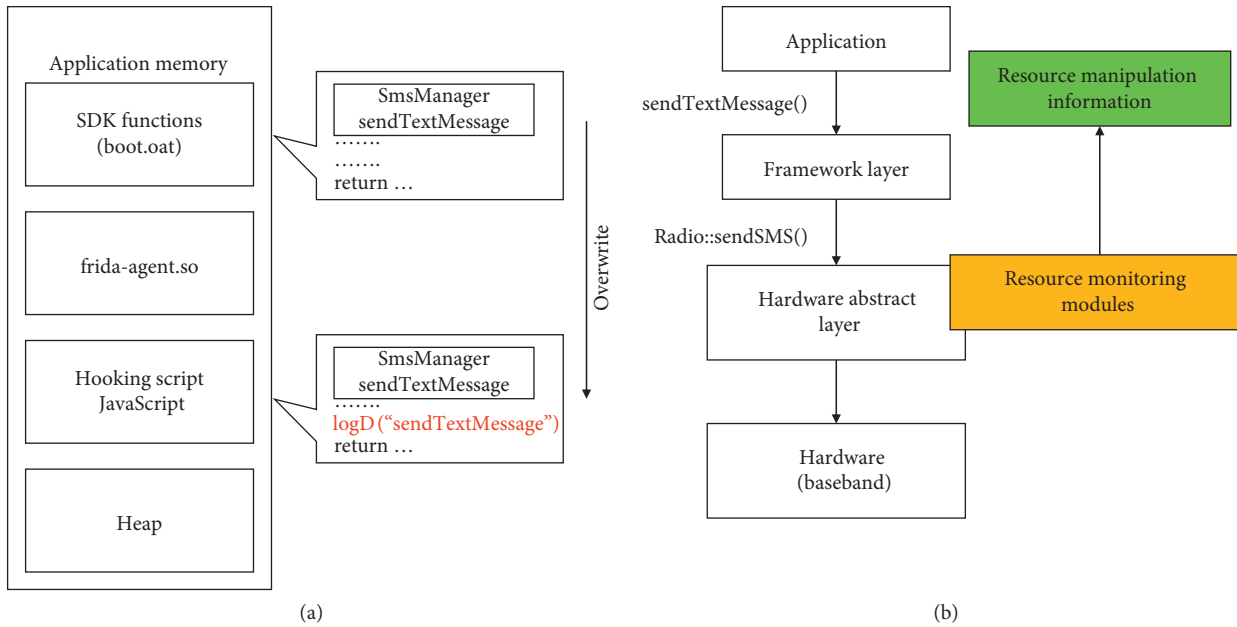

Figure 1: Frida vs. *HALWatcher* architecture for SMS resource monitoring. (a) Using Frida to monitor whether sending SMS API is called. Note that there are several APIs to send SMS. (b) HAL based resource monitoring.

```
(1) var hook = Java.use( android.telephony.SmsManager );
(2) hook.sendTextMessage.overload( java.lang.String , java.lang.String , java.lang.String ,
    android.app.PendingIntent , android.app.PendingIntent ).implementation =
(3) function(arg_0, arg_1, arg_2, arg_3, arg_4){
(4)   var olog = Java.use( android.util.Log );
(5)   olog.d( sendTextMessage is called );
(6)   return this.sendTextMessage(arg_0, arg_1, arg_2, arg_3, arg_4);
   }
```

Listing 1: Example of Frida hooking into *sendTextMessage()* function (Android SDK).

```
(1) Return<void> RadioImpl:sendSms(int32_t serial, const GsmSmsMessage& message) {
(2) #if VDBG
(3)   RLOGD( sendSms: serial %d , serial);
(4) #endif
(5)   RLOGD( [%d] [HALWatcher] RIL_REQUEST_SEND_SMS: serial %d ,(int)time(NULL), serial);
(6)   dispatchStrings(serial, mSlotId, RIL_REQUEST_SEND_SMS, false,
(7)     2, message.smscPdu.c_str(),
(8) message.pdu.c_str());
      return Void();
   }
```

Listing 2: Example of HAL modifying in *sendSms()* function (HAL).

5. Resources Manipulation Monitoring Module in HAL

The resource monitoring modules are basically located in many critical HAL interfaces and implementation functions. In this work, we illustrate *HALWatcher* in monitoring SMS, PHONE, and CAMERA resources in detail. For other resources, we conduct a list of modules' locations of *HALWatcher* for further works.

5.1. SMS and Phone. Both SMS and Phone permissions on the Android device allow the application of the right to access carrier service. Android communicates with carrier providers through SIM (subscriber identity module), which needs a radio baseband device to run radio service. Each baseband device has its own vendor's driver represented as "*libril-vendor.so*" file. Android HAL performs a RIL (Radio Interface Layer) connecting Android Framework and vendor's driver. Therefore, all resource usage related to SMS and

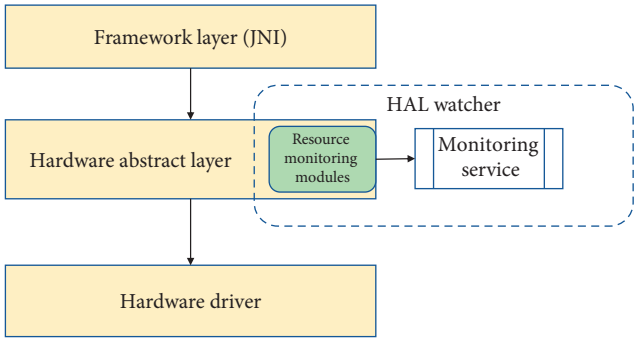

Figure 2: *HALWatcher* general design.

Phone permissions go through RIL. While analyzing the RILD, we found two types of RIL command: *REQUEST* and *UNSOL*. *REQUEST* command is used by the Android Framework Layer (i.e., *rilj*), to request data (e.g., signal strength) or functions (e.g., send SMS, conduct a call). *UNSOL* stands for *unsolicited* responses, which originate from the baseband (e.g., new SMS).

5.2. Camera. Recently, sensitive information leaked through Camera. The Android application might run as a service that has no activity screen. In that case, the malicious application or vulnerable application attacked by an intruder can handle a camera resource silently without any notification to the user. HAL Camera interfaces have been implemented inside *hardware/camera/device/1.0/default/CameraDevice.cpp*.

These interfaces provide methods to communicate to Camera hardware driver such as *getCameraInfo(), dumpState()*, etc. Some focus on managing the memory resources for Camera device (e.g, *CameraHeapMemory()*), others open or close Camera device (e.g., *Camera::open(), Camera::close()*), and others provide normal task of the Camera functions such as *startRecording(), stopRecording(), takePicture(), cancelPicture()*. In order to keep an eye on Camera resources, we create logs about the function when the Camera is opened and closed. We also keep a log for the working time of the Camera because of the irregular using period.

5.3. Other Resources Monitoring. Similar to Camera resources, other resources also have the implementation code inside/hardware/resource_name. Table 1 shows the list of hardware resources and their implementation source code. Regarding resource monitoring, we can add more features than logging the needed information for those resources.

5.4. Resources Manipulation Monitoring Service. This service is responsible for getting data from the monitoring module inside HAL. By using the logging method, this component is not required because the log data can be got from *logcat* command. HAL resources manipulation module does not log for any sensitive data of the user or the phone so that this log data can be public. The service can be any application inside or outside the phone, which is the only convenient purpose for the user or analysis researcher.

6. Implementation and Evaluation

6.1. Implementation. We implemented our approach using AOSP version 9.0.0_r47 on a Hikey960 board[3]. The limitation of the Hikey960 is that it does not support full hardware that usually exists on a real phone (e.g., Vibrator). Therefore, to prove that the HAL modifying method for resource monitoring is possible, we focused on SMS and PHONE resources. These resources HAL interface are implemented in hardware/ril/ as known as Radio Interface Layer Deamon, which stands in the middle of the communication between the Android Framework Layer (RILJ) and the Hardware driver (i.e., carrier baseband).

Hikey960 board has a list of hardware devices that need the corresponding HAL modules to work with. These components are defined in *hikey/device-common.mk* and *hikey/hikey960/device-hikey960.mk* config file. Because of the limitation of hardware that the Hikey960 board supports, we can only see the impact of HAL modifying when we change the source code of the component that we listed in Table 2.

Hikey960 does not have a SIM card reader. Therefore, we used the Huawei 4G E173 USB stick as a SIM card reader. Then we added the Huawei lib-ril (i.e., the driver that supports E173 USB stick to work as a baseband device) at the driver layer of the final compiled AOSP. Typically, the Linux kernel will accept USB as a storage device. Therefore, to make the kernel recognize the dongle as a 3 G/4G USB device, we switched the USB mode of the device using *usb_modeswitch* tool. Then we needed to customize the Hikey960 kernel (i.e., kernel 4.9) and add more kernel module that supports the *usb_modeswitch* function. We also customized the RIL daemon to automatically switch the USB device to PPP mode before loading the driver.

We evaluate our method by modifying directly to log information that goes through radio methods in RIL. In total, we customize several requesting methods, which are most related to send SMS and conduct phone calls. We also can hook the other implementation functions in the same way.

6.2. Evaluation

6.2.1. High Coverage Rate. We used the default Messaging of AOSP to send normal SMS, Premium SMS, conduct Phone calls, and send USSD. Then, we tried to send an SMS without using the application. We found a way to send SMS messages through *iSms service*-a default service in AOSP. We denote that Frida can not work in this situation of monitoring SMS resources. The iSms service is called by service call command through ADB shell[4] with the form: *adb shell service call isms 7 i32 0 s16* "*com.android.mms.service*" *s16* "*+1234567890*" *s16* "*null*" *s16* "*Hello*" *s16* "*null*" *s16* "*null*". To prove that *HALWatcher* can work without root privileges, we removed/*xbin/su* binary and built a *non-userdebug* version of AOSP to unroot the ADB shell. Finally, we ran *Trojan-SMS* on both

Table 1: HAL implementation source code related to some group of Dangerous and Protection permission.

Resource hardware	Permission level	HAL interface
CAMERA	Dangerous	Hardware/interface/camera/device/1.0/default/
LOCATION	Dangerous	Hardware/interface/GNSS/1.0/default/
PHONE/SMS	Dangerous	Hardware/ril/
SENSORS	Dangerous	Hardware/interface/sensors/1.0/default/
AUDIO	Dangerous	Hardware/interface/audio/core/2.0/default/
NFC	Protection	Hardware/interface/nfc/1.0/default/
BLUETOOTH	Protection	Hardware/interface/bluetooth/1.0/default/
WIFI	Protection	Hardware/interface/wifi/1.2/default/
VIBRATOR	Protection	Hardware/interface/vibrator/1.0/default/

Table 2: Hikey960 HAL components.

Hardware	Package name
WIFI	https://android.hardware.wifi@1.0-service
AUDIO	https://android.hardware.audio@2.0-impl https://android.hardware.audio.effect@2.0-impl https://android.hardware.broadcastradio@1.0-impl https://android.hardware.soundtrigger@2.0-impl
PHONE/SMS	rild
DRM	android.hardware.drm@1.0-impl
BLUETOOTH	android.hardware.bluetooth@1.0-service.btlinux
POWER	android.hardware.power@1.0-impl
LOCATION	android.hardware.gnss@1.0-impl
KEYMASTER	android.hardware.keymaster@3.0-impl
SENSORS	android.hardware.sensors@1.0-service

Table 3: *HALWatcher* vs. Frida in resource manipulation monitoring.

Test cases	HALWathcer	Frida
Ability to hook into the process which		
Send SMS with normal app on rooted device	100%	100%
Send SMS with normal app on nonrooted device	100%	0%
Trojan-SMS request sendSMS on rooted device	100%	76%
Trojan-SMS request sendSMS on nonrooted device	100%	93%
Send SMS using ADB shell on rooted device	100%	0%
Send SMS using ADB shell on nonrooted device	100%	0%
Log size retrieved in the test of (#line)		
Normal SMS apps	1	6.8
Trojan-SMS apps	1.84	3.84

rooted and nonrooted systems to prove the limitation of Frida hooking.

The result in Table 3 shows that 100% of data can be logged using *HALWatcher*. For some samples of *Trojan-SMS*, they detect root device so that the app immediately crashes. Some malware samples use obfuscation techniques, so the *frida-agent.so* may be wrongly embedded and leads to crash the app after spawning. Obviously, *HALWatcher* performs resource monitoring better than the Frida framework.

6.2.2. Compact and Complete Data. In comparison with hooking techniques, *HALWatcher* is less flexible (i.e., the need for rebuilding AOSP). However, this method gives compact and complete data that can be used for real-time hardware resources manipulation and malware analysis. To observe this possibility, we compared *HALWatcher* with Frida by hooking into sensitive APIs that require the use of SMS [27]. We looked at two datasets, benign and malicious applications (e.g., FakePlayer). For *HALWatcher*, we recorded RIL requests related to SMS. Both Frida and *HALWatcher* used the same logging method, which logged only the called function's name and the timestamp of the calling. We ran and triggered the app to send SMS, then terminate the apps.

The result in Figure 3 shows that the log from *HALWatcher* is less than hooking techniques, while the testing application manipulates the same hardware resources. Listing 3 shows the example of different log sizes between *HALWatcher* and Frida on monitoring send SMS resources. The log size retrieved from hooking on the normal app is much larger than malware. We found that the normal message application has call *getSubscriptionId()* API

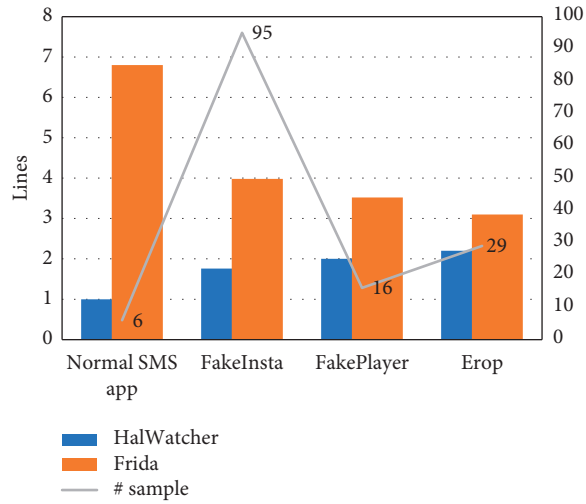

Figure 3: Average log size retrieved by *HALWatcher* and Frida.

```
(a) HALWatcher log of the default SMS app
(1)   [1590389070] [HALMonitor] RIL_REQUEST_SEND_SMS: serial 485
(b) Frida hooking log of the default SMS app
(1)   [1590389069] [android.telephony.SmsManager] [getSubscriptionId]
(2)   [1590389069] [android.telephony.SmsManager] [getSubscriptionId]
(3)   [1590389069] [android.telephony.SmsManager] [getSubscriptionId]
(4)   [1590389069] [android.telephony.SmsManager] [getSubscriptionId]
(5)   [1590389069] [android.telephony.SmsManager] [sendTextMessage]
(6)   [1590389069] [android.telephony.SmsManager] [sendMultipartTextMessage]
```

Listing 3: Example of log content of *HALWatcher* (a) and Frida (b) c. Low computational effort.

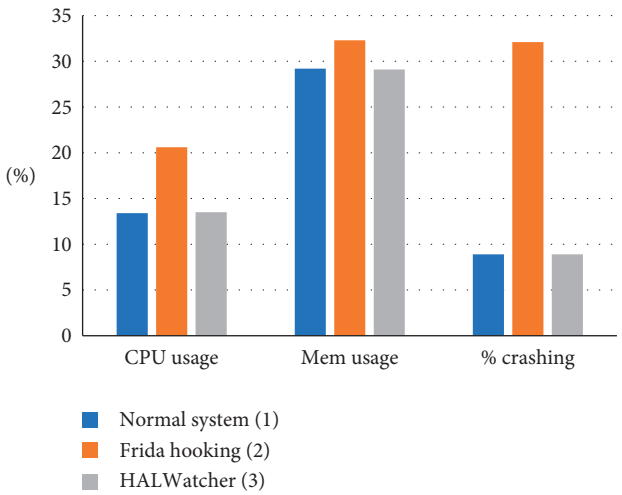

Figure 4: Computational evaluation result for (1) Normal system, (2) System with Frida hooking framework, (3) System with *HALWatcher*.

multiple times before calling *sendTextmessage()*. Moreover, the default Messaging app of AOSP calls both *sendTextmessage()* and *sendMultipartMessage()*. Note that the log is always collected from the start of opening the applications.

We compared the total CPU usage and Memory usage in three scenarios: (1) run the samples without Frida hooking and *HALWatcher*; (2) run the samples with Frida hooking; (3) run the samples with *HALWatcher*. For Frida hooking on (2), we conducted hooking progress on the target process only.

As shown in Figure 4, both CPU and Memory usage rates in (2) are more 5% higher than (1) and (3). We denote that the hooking script is injected into only the target application. However, in a practical resource monitoring system, we should implement instrumentation for all running processes. At that time, the system might be crashed (i.e., 5% extra computational resource for each process). Meanwhile, the indicators are almost no different in (1) and (3) whether the sample is malware or benign. Besides, the crashing samples rate is dramatically increased while we were running the test. This crashing happens because some samples are not suitable in AOSP 9.0; some samples are only crashed while we start Frida for hooking progress.

7. Conclusions and Discussion

Resource manipulation attacks are the most widespread malicious behaviors of Android malware. Current solutions, including static and dynamic analysis, are not efficient

enough to detect this attack. *HALWatcher* is a new approach that modifies Hardware Abstract Layer to monitor resources. This approach addresses the limitations of hooking techniques. *HALWatcher* provides a high coverage rate in monitoring hardware resources with low computational effort. In addition, *HALWatcher* can be applied to build a protecting mechanism in real-world devices because of the nonrooted environment requirement. However, *HALWatcher* faces some limitations. First, there is a need for strong knowledge about the Hardware Abstract Layer development to extend and deploy *HALWatcher*. Second, *HALWatcher* is only capable of monitoring hardware resources, not for others, which are already stored in system storage (e.g., CALENDAR, CONTACT, PHOTO, etc.). For our future research, we plan to research the new approach to monitoring other system resources that can integrate with *HALWatcher* to make a complete resource manipulation defending framework.

Data Availability

The source code and log file can be found at https://github.com/josebeo2016/HALModifying.

Disclosure

This paper is a revised version of the presented Poster: "HAL Based Resource Manipulation Monitoring on AOSP" in WISA 2020, Jeju, South Korea.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (no. 2020-0-00952, Development of 5G Edge Security Technology for Ensuring 5G+ Service Stability and Availability) and supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2019-0-00477, Development of android security framework technology using virtualized trusted execution environment).

References

[1] M. Fan, J. Liu, X. Luo et al., "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, 2018.

[2] L. Nguyen-Vu, J. Ahn, and S. Jung, "Android fragmentation in malware detection," *Computers & Security*, vol. 87, Article ID 101573, 2019.

[3] N.-T. Chau and S. Jung, "Dynamic analysis with Android container: challenges and opportunities," *Digital Investigation*, vol. 27, pp. 38–46, 2018.

[4] A. T. Kabakus and I. A. Dogru, "An in-depth analysis of Android malware using hybrid techniques," *Digital Investigation*, vol. 24, pp. 25–33, 2018.

[5] H. Zhou, W. Zhang, F. Wei, and Y. Chen, "Analysis of android malware family characteristic based on isomorphism of sensitive API call graph," in *Proceedings of the IEEE Second International Conference on Data Science in Cyberspace*, pp. 319–327, DSC), Shenzhen, China, June 2017.

[6] J. Qiu, S. Nepal, W. Luo et al., "Data-driven android malware intelligence: a survey," in *Machine Learning for Cyber Security. Lecture Notes in Computer Science*, X. Chen, X. Huang, and J. Zhang, Eds., Springer International Publishing, Midtown Manhattan, New York, pp. 183–202, 2019.

[7] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 95–109, NW Washington, DC; USA, 2012.

[8] S. Y. Mahmud, A. Acharya, B. Andow, W. Enck, and B. Reaves, "Cardpliance:${$PCI$}$${DSS}$ compliance of android applications," in *Proceedings of the 29th ${$USENIX$}$ Security Symposium (${$USENIX$}$ Security 20)*, San Diego, CA, United States, May 2020.

[9] 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, Illinois, USA, October 2011.

[10] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the ${$OS$}$ and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the Presented as Part of the 21st ${$USENIX$}$ Security Symposium (${$USENIX$}$ Security 12)*, pp. 569–584, Berkeley, CA, August 2012.

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

[12] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer DL-Droid, "DL-Droid: deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, Article ID 101663, 2020.

[13] A. De Lorenzo, F. Martinelli, E. Medvet, F. Mercaldo, and A. Santone, "Visualizing the outcome of dynamic analysis of Android malware with VizMal," *Journal of Information Security and Applications*, vol. 50, Article ID 102423, 2020.

[14] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, "Automated synthesis of semantic malware signatures using maximum satisfiability," 2020, https://arxiv.org/pdf/1608.06254.pdf.

[15] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[16] C. Zheng, S. Zhu, S. Dai et al., "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 93–104, CA, USA, October 2012.

[17] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "ANASTASIA: ANdroid mAlware detection using STatic analySIs of Applications," in *Proceedings of the 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, Larnaca, Cyprus, November 2016.

[18] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma, "A combination method for android malware detection based on control flow graphs and machine learning algorithms," *IEEE Access*, vol. 7, pp. 21235–21245, 2019.

[19] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[20] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20, Fajardo, PR, USA, October 2015.

[21] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[22] M. Ijaz, M. H. Durad, and M. Ismail, "Static and dynamic malware analysis using machine learning," in *Proceedings of the 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pp. 687–691, Islamabad, Pakistan, January 2019.

[23] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague, "A5: automated analysis of adversarial android applications," in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pp. 39–50, Scottsdale, AZ, USA, November 2014.

[24] M. Y. Wong and D. Lie, "IntelliDroid: a targeted input generator for the dynamic analysis of android malware," *National Down Syndrome Societ*, vol. 16, pp. 21–24, 2016.

[25] J.-G. Jiang, Z.-S. Liu, M. Yu, and C. Liu, "A resource management system design for malware behavior detection," in *Proceedings of the Computer Science, Technology and Application*, pp. 467–474, World Scientific, Changsha, China, March 2016.

[26] G. Meng, R. Feng, G. Bai, K. Chen, and Y. Liu, "DroidEcho: an in-depth dissection of malicious behaviors in Android applications," *Cybersecurity*, vol. 1, no. 1, p. 4, 2018.

[27] C. Zhao, W. Zheng, L. Gong, M. Zhang, and C. Wang, "Quick and accurate android malware detection based on sensitive APIs," in *Proceedings of the 2018 IEEE International Conference on Smart Internet of Things (SmartIoT)*, pp. 143–148, Xi'an, China, August 2018.

[28] B. Soewito and A. Suwandaru, "Android sensitive data leakage prevention with rooting detection using Java function hooking," *Journal of King Saud University - Computer and Information Sciences*, 2020, Published online July 21, 2020.

[29] M. Szczepanik, M. Kędziora, and I. Jóźwiak, "Android methods hooking detection using dalvik code and dynamic reverse engineering by stack trace analysis," in *Theory and Applications of Dependable Computer Systems. Advances in Intelligent Systems and Computing*, W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, Eds., Springer International Publishing, Midtown Manhattan, New York, pp. 633–641, 2020.