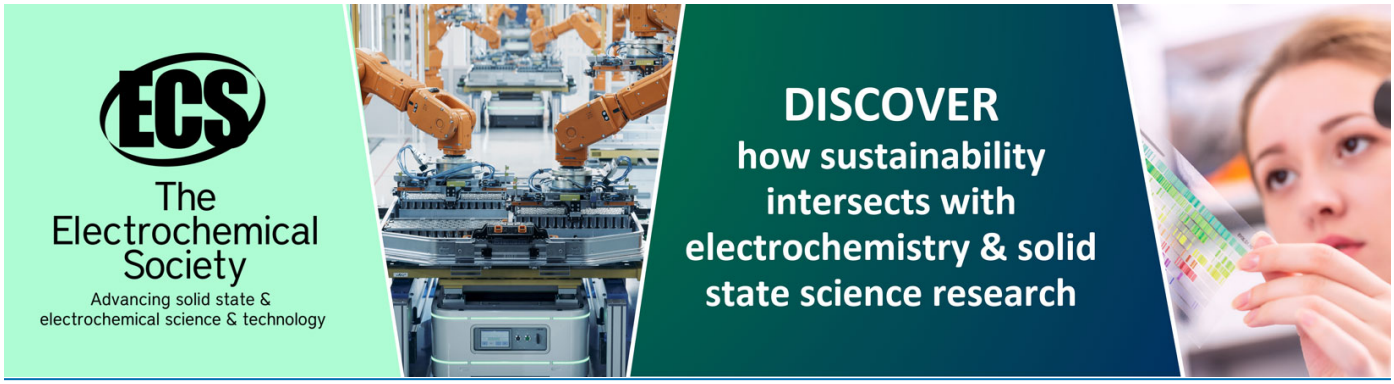
Enhancing the Security of Mobile Device Management with Seccomp



Enhancing the Security of Mobile Device Management with Seccomp

Yingjiao Niu[*†‡], Yuewu Wang[*†], Shijie Jia[*†], Quan Zhou[*†], Lingguang Lei[*†], Qionglu Zhang[*†‡] and Xinyi Zhao[*†‡]

[*]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Building C3, No. 65, Xingshikou Road, Haidian District, Beijing, China
[†]Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Building C3, No. 65, Xingshikou Road, Haidian District, Beijing, China
[‡]School of Cyber Security, University of Chinese Academy of Sciences, No. 380, Huaibei Village, Huaibei Town, Huairou District, Beijing, China
Email: {niuyingjiao,wangyuewu,jiashijie,zhouquan,leilingguang,zhangqionglu,zhaoxinyi}@iie.ac.cn

Abstract. Mobile devices today have been increasingly used to store and process sensitive information, and Mobile Device Management (MDM) solution is provided in Android systems to manage the mobile devices. MDM solution enforces device policies through calling Device Administration APIs provided in Android system. However, these APIs are usually implemented in user space, thus make the system more vulnerable to be attacked. Potential adversary may bypass these secure policies by breaking through the isolation between processes. In this paper, we propose an approach based on the seccomp mechanism of Linux kernel to enforce device management policies on Android. Seccomp can restrict the system calls of Android process to enforce device management policies in a more secure way. Experiments results demonstrate that our work may work well and incur negligible overhead.

1. Introduction

Mobile devices (e.g., smart phones and tablets) are increasingly ubiquitous nowadays due to their portability and mobility. In particular, mobile devices have been widely utilized to operate corporate emails and customer information in workplace[1], contributing to a Bring Your Own Device (BYOD) trend[2] and thus leaving large amounts of sensitive personal/corporate data in these devices, which may lead to a multitude of data security issues[3]. According to the Verizon Mobile Security Index 2019 report[4], one in three organizations in the US have been suffered a data breach in 2018, and moreover, in 25% of companies, at least one mobile device has experienced a mobile cryptojacking incident, a new kind of malicious code.

To raise the bar of the malicious attacks on the mobile devices in workplace, Google provides Device Administration APIs[5] to carry out mobile device management (MDM) in enterprise environments since Android 2.2. Device Administration APIs are the foundation of MDM system, which allows the BYOD administrator to allow or restrict the device owners to access various functions of the devices, such as Camera, Bluetooth, WIFI, etc[6][7]. MDM provides complete lifecycle management for mobile devices from all the aspects of device registration, activation, utilization, and elimination[8]. Thanks to the great convenience of MDM, it has received great

popularity. According to the Verified Market Research[9], the MDM market was valued at USD 3.37 billion in 2019 and is projected to reach USD 18.97 billion by 2027.

However, the realization scheme of the MDM would inevitably introduce security vulnerabilities to the mobile devices. In Android, the user space is more vulnerable to attack compared with kernel space. According to a report published by Google[10], the proportion of non-kernel bugs accounts for 3/5 of security vulnerabilities on Android devices. What's worse, the introduced vulnerabilities could be exploited by attackers to conduct privilege escalation attacks. For example, CVE-2020-0105[11] leverages a missing permission check in key_store_service.cpp to elevate privileges, allowing apps to use keyguard-bound keys when the screen is locked. CVE-2020-0183[12] can be used to elevate privileges because of an incomplete reset in *BluetoothManagerService*. Unfortunately, by analyzing the source code of the Device Administration APIs, we found that the realization scheme of the MDM (by calling the APIs provided by Android) truly lies in the user space (e.g., the function of disabling camera is implemented by the permission check in the "*cameraService*", which is a native process and lies in system library layer), which opens a door for the adversary to attack the mobile devices.

In this paper, we tackle the aforementioned limitations of MDM and propose a novel device management scheme to enhance the security of MDM. Our proposed scheme is based on the seccomp mechanism, which is introduced since Android O[13] to restrict the system calls of Android process. The principle of the seccomp mechanism is that when a seccomp filter is assigned to a process, the seccomp would specify the available system calls for the process, and the invocations of the systems calls, which are not on the filter list, will be forbidden. Inspired by the principle of the seccomp, in this paper, we utilize the availability of the system calls to realize the management of the device. Specifically, to restrict a device function (e.g., Camera, Bluetooth, WIFI), we would remove a certain key system calls from the available system call list of the corresponding process, which is responsible for communicating with the lower-level kernel driver. Then we can add the removed system calls again to access the restricted device function.

To enhance the security of MDM by utilizing seccomp, when we would like to restrict a certain device function (e.g., Camera, Bluetooth, WIFI), we need to solve the following problems: 1) determining the target process which is used to interact with the kernel driver; 2) selecting the key system calls which are indispensable for the target process; 3) configuring the seccomp filters for the target process. For the first problem, in Android, as all the applications send the requests to the corresponding driver through HAL service (e.g., Camera, Bluetooth and WIFI), which sits between the driver and the higher-level Android framework. Therefore, we select the HAL service as the target process. For the second problem, as the HAL service communicates with the kernel driver by taking advantage of the Binder IPC calls[14], and in the Binder mechanism, the ioctl system call is used to send and receive binder messages. Therefore, we select the ioctl system call as the key system call. Note that, as the HAL service runs in its own process, thus if we remove the ioctl system call from the available system call list, it will not affect the normal operation of the whole system. For the third problem, we introduce a kernel module to achieve the cross-process seccomp filter configuration. Moreover, we provide a friendly notification to the device owner when a certain device function may be prohibited due to the device policy, then the device owner could determine whether to accept the prohibition or not.

We implement a prototype and verify the effectiveness of our proposed method on different models of mobile phones with different system versions. Specifically, we take the Camera, Bluetooth and WIFI as example. The experimental results show that our method could provide a practical way to enhance the security of the MDM. In addition, our method introduces negligible performance overhead.

The remainder of this paper is organized as follows. Section 2 introduces the necessary background knowledge. Section 3 details the MDM policy enforcement. Following is the design and implementation of our proposed scheme in section 4. Section 5 discusses the evaluation of our system. Finally, we conclude the paper in Section 6

2. Background

In this section, we first provide necessary background on Android MDM solutions and then introduce the seccomp mechanism, which is highly related to the design and implementation of our method.

2.1. Android MDM

Android 2.2 introduced Device Administration APIs for developers to enforce a system wide security policy and to dynamically adapt the features based on the device's current security level. Mobile Device Management (MDM) solutions are provided in the way of the Device Policy Manager API, which allows the developers to control functions like password complexity, lockout timing, remote wipe and remote lock. Device Policy Manager exposes part of the functionality of the underlying system service, namely, *DevicePolicyManagerService*. Same with most system services, *DevicePolicyManagerService* is started by *system_server* process and runs within it as the system user. Therefore, *DevicePolicyManagerService* could execute almost all privileged actions, which makes it possible for users to enable and disable the device administrators on demand[15]. The security policies have different granularity and can be enforced either for the current user or for all users on a device. Some policies are not enforced by the system at all, as the system may only notify the declaring administration application, which is then responsible for taking an appropriate action.

2.2. Seccomp Mechanism

Seccomp filter provides a mean for a process to specify a filter for the incoming system calls. The filter is expressed as a Berkeley Packet Filter (BPF) program, as with socket filters[16]. Seccomp has three working modes, i.e., *seccomp-disabled*, *seccomp-strict* and *seccomp-filter*[17]. In the disabled mode, the process is not under the protection of the seccomp mechanism. In the strict mode, only four system calls could be invoked, i.e., read, write, _exit and sigreturn. In the filter mode, the available system calls can be defined by constructing a BPF program[18] (i.e., a seccomp filter). The seccomp filter works like the socket filter, except that it checks the system call invoked by a process, including the system call number and system call arguments. A BPF program is composed of BPF instructions. Each instruction is 64 bits, with a fixed format, which includes the actual filter code, the jump offset when the filter codes return true, the jump offset when false is returned, and a generic value[19].

In Android kernel, the list of available system calls is represented as a *seccomp_filter* data structure. Each *seccomp_filter* structure contains a *sock_filter* member, which is an array consisting of the BPF instructions. Each process has a *task_struct* structure that contains a *seccomp* structure, in which the *mode* flag defines the seccomp state of the process. If the process is protected by the seccomp mechanism, the *filter* field of the *seccomp* structure points to the first *seccomp_filter* structure. It is possible for a process to be attached with multiple *seccomp_filters*, and all *seccomp_filters* are organized in a one-way linked list. Each *seccomp_filter* structure contains a *sock_filter* member, which is an array consisting of the BPF instructions.

3. MDM Policy Enforcement

In this section, we discuss how the camera policy is implemented and enforced in the MDM as an example. The analyzed code is based on Android 10.0.0_r39.

The main class for managing policies enforced on a device is *DevicePolicyManagerService*, which provides *setCameraDisabled* API for device administrators to disable all the cameras on the device. When a device administrator calls the *setCameraDisabled* API, the propagation of camera restriction policy travels from *DevicePolicyManagerService* to *UserManagerService* and then finally to *AppOpsService*. Specifically, *DevicePolicyManagerService* pushes the camera restriction to *UserManager* via *addUserRestriction* interface. The *AppOpsService* rules define a mapping from UID/package name to the allowed operations and offer an interface to the application to retrieve the current configuration for checking whether the application is allowed to perform the corresponding operation.

In the following we will introduce how the application is prohibited from using the camera when it is disabled through the device policy.

In the Android system, camera service provides the functions of using the camera device (e.g., taking a picture, recording a video). The workflow of the disabled camera policy enforcement is shown in Figure 1, which consists of 8 steps. Getting an instance of the Camera object is the first step and the recommended way is to use the *Camera.open()* method which is encapsulated by the framework layer. The initial entry in the framework layer is CameraManager's *openCamera* method, which finally calls the *openCameraDeviceUserAsync* method. Subsequently, the *openCameraDeviceUserAsync* method calls the lower-level native code *native_setup(),* which is implemented in *android_hardware_Camera.cpp* to obtain access to the camera service.

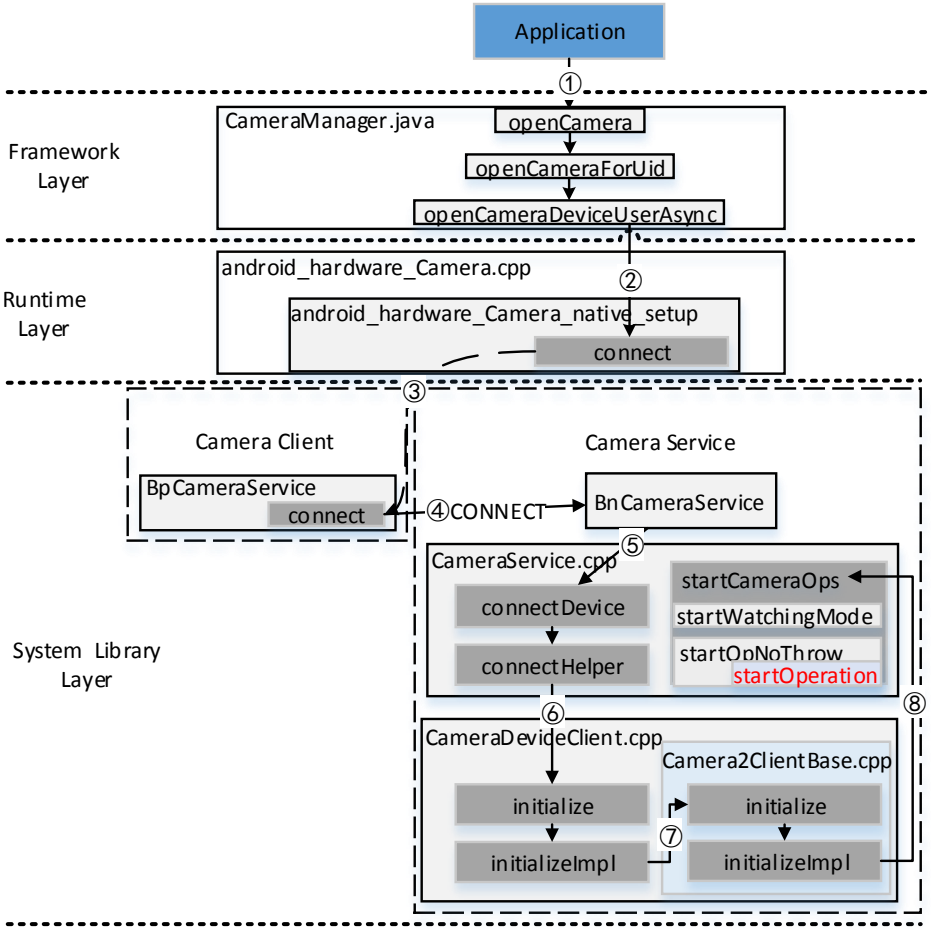


Figure 1. The workflow of the disabled camera policy enforcement.

To communicate with the camera service, the camera client in the system library layer first obtains the camera service instance by *getService()*, which establishes the connection between the client and the service by the binder. The client side gets camera from *BpCameraService::connect*, which sends a CONNECT command to *Bncameraservice*. When the *Bncameraservice* receives the CONNECT command, it calls *connectDevice* in the *CameraService.cpp* to obtain the remote device (step 5). Actually it is the *connectHelper* method that is invoked to implement the connection logic. Step 6 and Step 7 generate and initialize a specific *CameraDeviceClient* instance. In the *InitializeImpl* method of *Camera2ClientBase.cpp*, the *startCameraOps* method will be called to perform permission check operations. The *startCameraOps* method mainly completes the following tasks. Firstly, the *AppOpsManager::startWatchingMode* class, which is provided by *AppOpsService*, is used to monitor for changes to the operating mode for the given operation. Secondly, *startOperation* class is called to detect whether the given application could perform the operation.

From the above analysis, we can conclude that the caller's permission is checked in the *camera service*, which is a native process. As any application can run native codes such like C/C++ and

bypass the permission check, thus the current MDM solution can not prevent the attacks of accessing resources by native code directly, (e.g., transplantation attack[20]).

4. Design and Implementation

To achieve precise device management based on the seccomp mechanism, we restrict the available system calls of the target process, which is used to connect to the kernel driver, in this way, we can prevent the applications from accessing the underlying resource of the system.

The architecture of the system is illustrated in Figure 2, which consists of three modules, i.e., the *Policy Configuration*, the *Policy Enforcement* and the *Failure Indication*. The *policy configuration module* is responsible for receiving the device management strategy. The *policy enforcement module* provides seccomp-based management to enforce device security policy, which includes *BPF constructor submodule* and *seccomp filter loader submodule*. *The failure indication module* is designed to display reminder messages to users if an operation is restricted due to the device policy. In the following, we will introduce the implementation of the *policy enforcement module* and the *failure indication module* in detail.

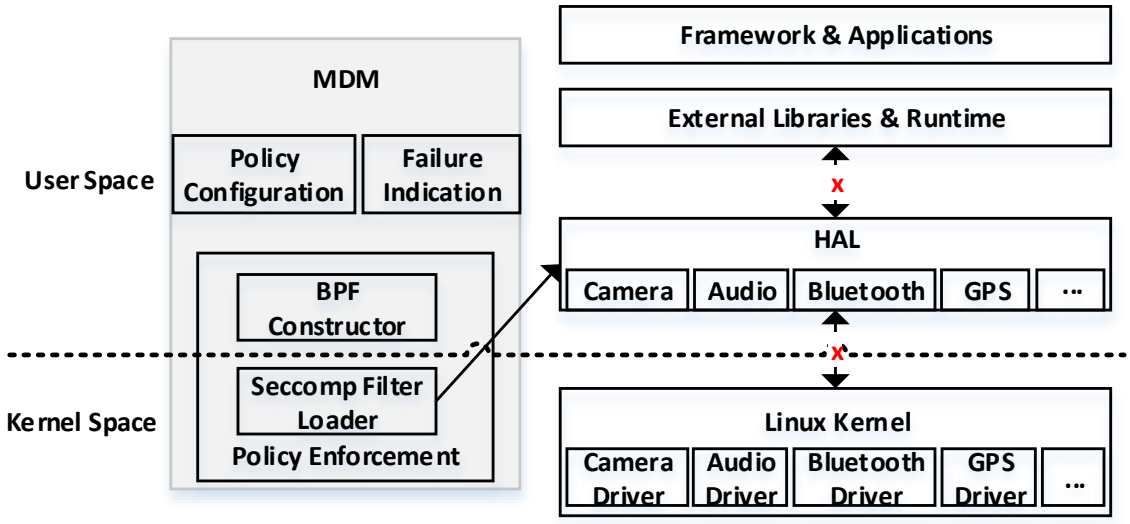


Figure 2. The architecture.

4.1. Policy Enforcement

4.1.1. BPF constructor. BPF constructor is implemented in the user space. Since the Android system doesn't provide an interface to achieve the cross-process seccomp filter configuration, we use two macros (i.e., *BPF_JUMP* and *BPF_STMT*) to convert the available system call list to the BPF filter instructions in the user level process.

As aforementioned before, the seccomp structure is a member of the task_struct, if we would like to change the seccomp policy of the process, we should first get the pointer of the task_struct. As all HAL services are registered and launched by the *init* process when the system starts, thus we can use the *pidof* interface provided by the kernel to obtain the pid of the target service with the packagename as a parameter. Then the *pid_task()* function is called to get the task_struct of the process.

4.1.2. Seccomp filter loader. To load the seccomp filter for the target HAL service, we need first convert the BPF filter built in the user space to the kernel data structure. As some codes of the BPF filter program are different from the codes in seccomp filter, thus we use *sk_ck_filter()* and *seccomp_check_filter()* in the kernel module to transform the BPF filter program into a seccomp BPF filter program.

We take a list of excluding ioctl system call as an example to illustrate the procedure of constructing a seccomp filter. The BPF instructions in the second subfigure of figure 3 are the output of the BPF constructor. As figure 3 shows, after *sk_ck_filter()* and *seccomp_check_filter()* are called in the third and fourth subfigure of figure 3, then the code flag will be checked. The number of 29 in the second subfigure of figure 3 represents the system call number of ioctl.

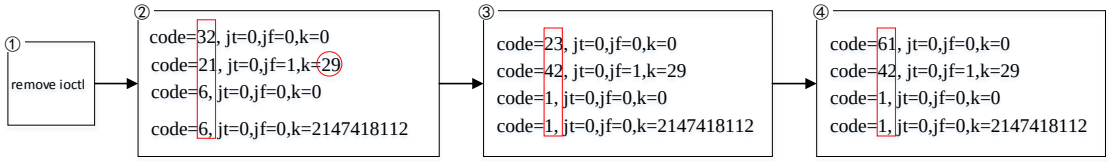


Figure 3. A sample of the seccomp_filter structure constructing procedure.

After the *seccomp_filter* structure is created in the kernel, the seccomp filter loader modifies the *filter* pointer, making it point to the newly constructed *seccomp_filter* structure. Since the HAL services are not protected by the seccomp mechanism during initialization, the working mode of the seccomp in these processes is *seccomp-disabled*. In order to start the seccomp policy of the target process, we should set the working mode to *seccomp-filter*, more specifically, setting the "mode" flag to "SECCOMP_MODE_FILTER" and setting "task_struct->thread_info->flags" to "TIF_SECCOMP".

4.2. Failure Indication

The failure indication module provides users with prompt information when an operation is prohibited by the device policy. And the user can decide whether to accept the prohibition. Compared with the existing solutions, which prevents all the applications from accessing the system resources, our solution can achieve more refined management of the device.

We make use of socket for cross-process communication to implement message push and result feedback. The client side is responsible for intercepting and capturing the disabled system call information. For each system call invoked during the execution of the process, the system first calls the *secure_computing* function to check whether the system call is in the list of the available system calls. The returned value of -1 in the *secure_computing* would result in the process being killed immediately. In order to display a prompt message to the user and let the user decide whether to perform the operation, if the returned value is -1, the process that invoked the system call will not be immediately terminated, and then sends a request to the server. After the server accepts the request, it displays the analysis information to the user, and feeds the user's request back to the client side. At last, the client side decides whether to allow the execution of the forbidden system call based on the returned result.

5. Evaluation

In this section, we make an evaluation for our prototype system in the aspects of effectiveness and overall performance. As the principle of managing different functions in the mobile phone are almost the same, thus we take Camera, Bluetooth and WIFI as examples for evaluation.

5.1. Effectiveness

To verify the practicability and effectiveness of our method, we implement our method on different mobile phones shipped with different Android system versions in the real world. Specifically, we choose 4 different phones and 3 different Android system versions. The result is illustrated in **Table 1**. We can see that our method could manage the devices functions on all the evaluation platforms.

Table 1. Result on different Android system versions and models.

Android system Version	Model	Result		
		Camere	Bluetooth	WIFI
8.1.0	Nexus 5X	✓	✓	✓
	Pixel 2	✓	✓	✓
9.0.0	Pixel 2	✓	✓	✓
	Pixel 3	✓	✓	✓
	Pixel 3a	✓	✓	✓
10.0.0	Pixel 3	✓	✓	✓
	Pixel 3a	✓	✓	✓

5.2. Overall Performance

We evaluate the performance impacts of our method on the running systems. The performance consumption is mainly brought by the following two operations, i.e., dynamically enforcing the seccomp filter for the target process according the device management strategy, and displaying prompt information for users. To evaluate the time consuming on the above two aspects, we run our method on the selected evaluation platforms for 10 times, and the average results are shown in **Table 2** and **Table 3** respectively. We can see that the time of configuring the seccomp filter is less than 4ms and the time of displaying information is about 2s.

Table 2. Time consuming on configuring the seccomp filter.

Android system Version	Model	Result (ms)		
		Camere	Bluetooth	WIFI
8.1.0	Nexus 5X	2.3	2.2	1.9
	Pixel 2	2.7	2.3	1.9
9.0.0	Pixel 2	2.4	2.7	2.4
	Pixel 3	2.1	2.5	2.8
	Pixel 3a	2.1	3.3	2.4
10.0.0	Pixel 3	2.1	2.4	2.6
	Pixel 3a	2.4	3.5	2.9

Table 3. Time consuming on information display.

Android system Version	Model	Result (s)
8.1.0	Nexus 5X	1.99
	Pixel 2	1.72
9.0.0	Pixel 2	1.93
	Pixel 3	1.63
	Pixel 3a	1.20
10.0.0	Pixel 3	1.18
	Pixel 3a	1.36

6. Conclusions

In this paper, we introduce a method to enhance the security of the Mobile Device Management on Android. Unlike the existing approaches, our scheme is based on the seccomp mechanism to restrict the system calls in the kernel space, which avoids the common vulnerabilities of the schemes in the user space. Furthermore, we provide a friendly restriction notification to the device owner, achieving fine-grained device management. The experimental evaluation confirms the efficiency of our proposed method.

7. References

[1] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce: Enforcing complex, data-centric, systemwide policies in android," in *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*, 2014, pp. 40–49.

[2] O. Zungur, G. Suarez-Tangil, G. Stringhini, and M. Egele, "Borderpatrol: Securing BYOD using fine-grained contextual information," in 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019, 2019, pp. 460–472.

[3] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices," in 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015, 2015.

[4] Verizon, "Mobile security index 2019 executive summary," https://enterprise.verizon.com/resources/reports/mobile-security-index/2019/executive-summary/, 2019.

[5] "Device administration," https://developer.android.com/guide/topics/admin/device-admin, 2019.

[6] OPENSYS, "Android android-10.0.0_r39," http://aosp.opersys.com/xref/android-10.0.0_r39/xref/frameworks/base/core/java/android/provider/Settings.java#9583,2020.

[7] X. Cai, X. Gu, Y. Wang, Q. Zhou, and Z. Cao, "Enforcing ACL access control on android platform," in Information Security - 20th International Conference, ISC 2017, Ho Chi Minh City, Vietnam, November 22-24, 2017, Proceedings, 2017, pp. 366–383.

[8] N. Group, "Android mdm," https://www.nccgroup.com/us/about-us/newsroom-and-events/blog/2012/march/android-mdm.-part-i-build-up/, 2012.

[9] VERIFIED, "Global mobile device management market size by type, by deployment type, by geographic scope and forecast," https://www.verifiedmarketresearch.com/product/mobile-device-management-market/, 2019.

[10] T. Spring, "What's new in android 8.0 oreo security," https://threatpost.com/whats-new-in-android-8-0-oreo-security/128061/, 2017.

[11] "Cve-2020-0105 detail," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0105, 2020.

[12] "Cve-2020-0183 detail," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0183, 2020.

[13] P. Lawrence, "Seccomp filter in android o," https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html, 2017.

[14] F. Tcymbal, "Project treble. what makes android 8 different?" https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Project-Treble.-What-Makes-Android-8-Different_-Fedor-Tcymbal-Mera-Software-Services.pdf, 2018.

[15] N. Elenkov, "Android security internals: An in-depth guide to android's security architecture," pp. 217–218, 2014..

[16] "Seccomp bpf," https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[17] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: split-phase execution of application containers," in Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings, 2017, pp. 230–251.

[18] J. Schulist, D. Borkmann, and A. Starovoitov, “Linux socket fltering aka berkeley packet flter (bpf),” https://www.kernel.org/doc/Documentation/networking/filter.txt, 2019.
[19] PLUMgrid, “Bpf - in-kernel virtual machine,” https://events.static.linuxfound.org/sites/events/files/slides/bpf_collabsummit_2015feb20.pdf, 2015.
[20] Z. Zhang, P. Liu, J. Xiang, J. Jing, and L. Lei, “How your phone camera can be used to stealthily spy on you: Transplantation attacks against android camera service,” in Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015, 2015, pp. 99–110.