A Survey on Various Threats and Current State of Security in Android Platform

PARNIKA BHAT and KAMLESH DUTTA, National Institute of Technology, India

The advent of the Android system has brought smartphone technology to the doorsteps of the masses. The latest technologies have made it affordable for every section of the society. However, the emergence of the Android platform has also escalated the growth of cybercrime through the mobile platform. Its open source operating system has made it a center of attraction for the attackers. This article provides a comprehensive study of the state of the Android Security domain. This article classifies the attacks on the Android system in four categories (i) hardware-based attacks, (ii) kernel-based attacks, (iii) hardware abstraction layer-based attacks, and (iv) application-based attacks. The study deals with various threats and security measures relating to these categories and presents an in-depth analysis of the underlying problems in the Android security domain. The article also stresses the role of Android application developers in realizing a more secure Android environment. This article attempts to provide a comparative analysis of various malware detection techniques concerning their methods and limitations. The study can help researchers gain knowledge of the Android security domain from various aspects and build a more comprehensive, robust, and efficient solution to the threats that Android is facing.

CCS Concepts: • **Security and privacy** → **Operating systems security**; **Mobile platform security**; *Malware and its mitigation*;

Additional Key Words and Phrases: Android, malware, intra library collusion, malware detection, privilege escalation

ACM Reference format:
Parnika Bhat and Kamlesh Dutta. 2019. A Survey on Various Threats and Current State of Security in Android Platform. *ACM Comput. Surv.* 52, 1, Article 21 (February 2019), 35 pages.
https://doi.org/10.1145/3301285

1 INTRODUCTION

In the past few decades, the number of smartphone users has grown exponentially and it is expected to reach 2.87 billion users by 2020 [1]. Availability of high-speed Internet on mobile phones and increased social media activity are the primary causes for the accelerated growth of smartphones, especially those that are Android based. The cost competitiveness in the smartphone market has also contributed to this growth and has made smartphones available to the middle- and lower-income groups as well. This availability and the features of the latest smartphones have made the population dependent on these phones for their daily work, such as paying household bills through phone banking or transferring essential documents. The operating system is an

This work is supported by research grant from the Ministry of Human Resource Development (MHRD), Government of India.

Authors' addresses: P. Bhat and K. Dutta, National Institute of Technology, Hamirpur, Anu Road, Hamirpur, Himachal Pradesh HP, 177005 India; emails: bhatparnika.kdnith@gmail.com, kd@nith.ac.in.

0360-0300/2019/02-ART21 $15.00
https://doi.org/10.1145/3301285

integral part of these smartphones and is responsible for the functioning of the device, which makes it prone to attacks that take control of the device. Hence, it becomes essential for developers of malware detection tools to have an idea of the loopholes and vulnerabilities present in various parts of the operating system.

According to the International Data Corporation (IDC) [2], the Android operating system covers around 85% of the world's smartphone market. Android gained popularity around 2010. Since then it has been ruling the smartphone market. According to the mobile overview report by Scientiamobile [3], Android 6.0 (Marshmallow) is the most popular version of the smartphone operating system.

Because of its increasing popularity, Android is drawing the attention of malware developers and cyber-attackers. Each year the number of Android malware families is increasing at a higher rate. These attacks vary regarding their functioning, the part of the system targeted, and the vulnerabilities exploited. This article aims to provide comprehensive knowledge of the attacks in respect to these factors. The researchers can use this study to develop sophisticated malware detection tools, for Android, which can thwart the malware developers from harming the system using any of these vulnerabilities. Many kinds of malware are discussed in this article. Most of the times, the malware developers emphasize on developing malware that they can use to extract profit. For example, Ransomware [4] is one such malware family that once installed in the user's system, gives the control of the system to the attacker. The attacker then demands a huge ransom from the victim for releasing the data that otherwise attacker threatens to leak to the outside world. Ransomware is no longer limited to just windows or PC; it is evolving as a threat for smartphones as well as IoT devices. Other than Ransomware, Android malware families that are popular are spyware, bots, Adware, Potentially Unwanted Applications (PUA), Trojans, and Trojan spyware. PUA is the topmost Android malware detected by Quick Heal [5]. The attackers use third-party application stores and repacked applications to propagate these malicious programs. They extract the private information of the user, without his/her consent, and then send it to the server; later, they use this information for black marketing and social engineering purposes.

Of all the malware, Trojans hold a significant share in spreading malicious content. The latest malware detection statistics have shown an increase in the number of mobile Ransomware and banking Trojans. In 2016, Kaspersky Lab detected 128,886 lakh mobile banking Trojans and around 261,214 mobile Ransomware Trojans [6]. Malware developers are continually working in the direction of finding new methods to unearth loopholes in the Android system. Although the latest Android systems now feature robust security mechanisms, malware developers are finding new techniques to evade the security policies of the system. Gugi is an example of a banking Trojan that can exploit the security policies of Android Marshmallow. The Google Play store has dozens of Android apps that are malicious and have already infected millions of Android phones. For example, researchers found that Judy, which affected around 36.5 million Android users [7], was present in around 40 applications in the Google Play Store. South Korean Company Kininwini developed these malicious applications. The majority of malicious applications are designed to gain root access to the system, and the malware developers craft them so well that it is tough to detect them, and they keep on working in silence; users have no idea that the malware has endangered their system. After gaining control of these systems, these malicious applications can perform dangerous actions and can also infect other devices connected to the infected system. Android malware has become a marketplace for hackers and malware developers. They are earning a considerable amount of money from this source. The Dark Web is one such example; it is a marketplace for criminals to buy and sell mobile malware, and the malware developers sell it as a component of software packages. On the Dark Web, malware and some other tools that are capable of evading many operating systems like Android and iOS are available at a lower price, thereby gaining the interest of hackers and causing an increase in criminal activities [8]. Malware has become a severe threat to modern society, which is increasingly becoming dependent on smart technologies for day-to-day activities.

This article provides a comprehensive analysis of various threats, challenges, and solutions relating to security in the Android environment as specified below.

Section 2 discusses a wide range of attacks targeting three essential components of the Android device, which include the hardware, kernel, and applications. The article emphasizes vulnerabilities in the system, attack strategy, and possible solutions to the attacks. This section also provides a brief description of the security measures adopted in the latest Android version Oreo and the conceptualization of possible attack on it. The following Section 3 discusses in detail the security loopholes in the Android environment that are exploited by malware developers such as the permission system, security protocols, and poor developer practices. After a detailed perusal of articles and current attack trends, one can observe how attacks remain prevalent due to technical glitches and loopholes found in the areas mentioned above. One key takeaway from the study is the fact that lack of awareness about the use of security protocols and poor developer practices are the primary cause of security threats to Android. Further, Section 4 provides the analysis of defensive mechanisms such as antivirus and static, dynamic, and hybrid approaches to the malware discussion. Last, Section 5 talks about the present scenario of the Android system regarding security and future research directions.

Unlike previous surveys [9, 10], this article not only discusses the attacks and methods to defend them but also focuses on the underlying vulnerabilities in the Android system and the poor developer practices that contribute to the proliferation of attacks. This study also pays particular attention to the problems in the Android infrastructure. The article puts forward problems present in the permission system of Android and how this leads to various vulnerabilities and attacks, such as privilege escalation. The article outlines various attacks relating to libraries in combination with privilege escalation attacks; these kinds of attacks have become rampant recently, but they had not been addressed in previous surveys [11], for example, attacks relating to intra-library collusion. This article also provides possible solutions to the problems arising due to intra-library collusion. This study also discusses the role of the certificate authority and its importance for secure SSL connection. This article also presents a picture of the current state of the defensive mechanism of Android against malware. This article discusses the various types of anti-virus software and also gives a comprehensive analysis of static and dynamic approaches to malware detection along with their limitations. Android is currently one of the most used operating systems in the world, having various components and compatible applications running in the thousands. This study tries to cover many areas of this vast system and provide a comprehensive analysis of it from a security perspective.

2 ATTACKS ON THE ANDROID SYSTEM

This article classifies attacks into four different categories based on the components of the Android device targeted:

- Hardware-based attacks
- Kernel-based attacks
- Hardware abstraction layer-based attacks
- Applications-based attacks

Figure 1 shows the outline of the Android system Architecture. As represented, there are large numbers of applications running on the system. These can be pre-installed applications, which are present by default in the system, as well as third-party applications that are installed by users from

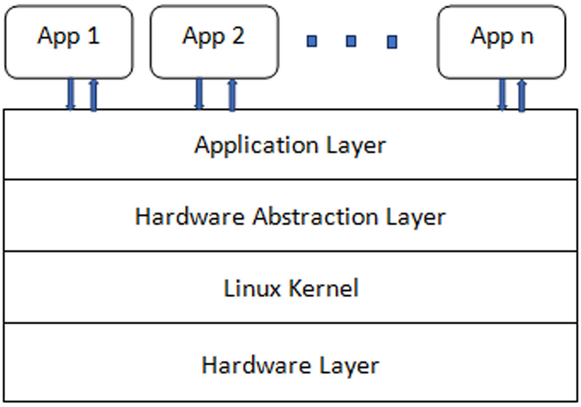

Fig. 1. Android layers overview.

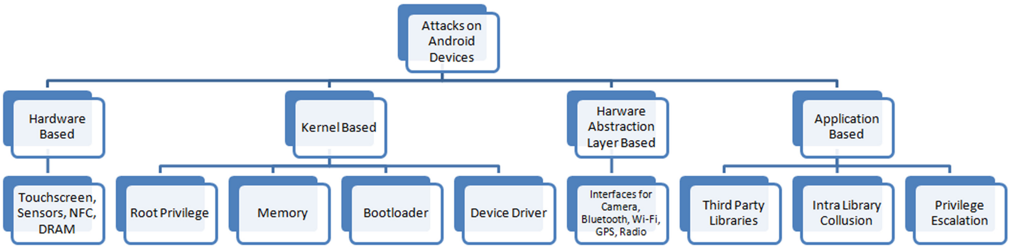

Fig. 2. Attacks classification.

various sources. All these applications use the Application Programming Interface (API) provided by application layer to interact with the system. The Hardware Abstraction Layer (HAL) consists of several library modules. Each library module provides an interface for specific hardware components. The kernel is core of the system; it is responsible for proper functioning of all the components in the system. The hardware layer consists of hardware components that are in-built into the system. Each hardware component has a specific interface that is used for interaction with the component. The malware developers devise various approaches to gain control of the system, which may include an attack on one or several of these layers, exploiting their vulnerabilities. This section provides a detailed explanation of these attacks and various vulnerabilities exploited by the malware developers. Figure 2 provides a further classification of the attacks that are also discussed with examples in this section.

2.1 Hardware-based Attacks

In this layer, attackers target the vulnerabilities found in the hardware components. Most of the vulnerabilities exist because of the preinstalled vulnerable components in the device provided by the manufacturers. It is tough to protect the system from attacks that target such components. These vulnerabilities are very difficult to patch, because the patches are available with the distributors and carriers only. Another possible way to attack the Android system is to exploit the security weaknesses found in the repaired components. Shwartz et al. [12] have shown how using repaired Android components like touchscreen, sensors, and near-field communications (NFC) readers from unauthorized vendors rather than from original vendors, called original equipment manufacturers (OEMs), can put the integrity of the system at stake. They have shown in their article how attacks like touch injection and buffer overflow use unauthorized components to exploit the vulnerabilities of the device and attack the system. Authors have also pointed out that designers must keep this aspect in consideration when implementing a system where replaced components fall inside the trust periphery of the system. It must be essential for the system to check the integrity of the communication between the replaced component and central processor of the system. Also, the component driver's source code should not implicitly assume that the component hardware is authentic and trustworthy.

A Rowhammer attack is an example of a hardware-based attack. The attacker uses malicious JavaScript in the browser (like Firefox) to attack an Android phone. The preliminary requirements of this attack are the installation of a malicious application in the targeted device and access to graphics processing unit (GPU) used in the device, and to speed up the attack the attackers can also use the attack speed chip's cache. To launch this attack, the attacker needs to access a specific memory location or a row in memory inside a dynamic random access memory (DRAM) chip. Hamming this row number of times causes a small amount of electrical current to leak from the target location. It results in a change in the state of memory bit or memory leak [13]. GLitch is the latest form of Rowhammer attack that allows an attacker to hack the target device remotely. GLitch is so named because attackers use WebGL, a library that renders interactive graphics code within any browser that is compatible with WebGL. These graphics can trigger a well-known glitch found in memory chips, namely DDR3 and DDR4. Attackers launching a Glitch attack exploit the GPU. The main reason for using the GPU is that its cache can be easily administered by the attacker, which gives them easy access to target rows [14], which is otherwise tricky if CPU is used instead of GPU, as used in previous versions of a Rowhammer attack. Another attack that falls under the same category is Deterministic Rowhammer Attacks on Mobile Devices (Drammer). The target component is DRAM chips. Attackers use the malicious application to exploit DRAM chips and can get access to the kernel of the devices. To date, there is no patch available to safeguard Android from these attacks entirely.

RAMpage (CVE-2018-9442) is a vulnerability, found in hardware components (LPDDR2, LPDDR3, or LPDDR4) of RAM, that implements attacks like Rowhammer. The attackers use the same strategy of using a malicious application to exploit these vulnerabilities. The strategy is to break the memory allocator ION (unified memory management interface). Once the attacker executes the attack successfully, he or she can get access to the restricted memory location in the device. The attacker can leak sensitive information from the victim's device and can also get complete control over the system [15]. GuardION is an open source technique proposed by Veen et al. This tool mitigates the threat caused by attacks like Drammer. GuardION protects the Android kernel by defending DMA allocations from these attacks. DMA allocations are shielded using rows that are empty, which results in isolated bitflips. Thus this technique helps in reducing Drammer and similar attacks [16]. QuadRooter vulnerability as the name suggests is a combination of four vulnerabilities that poses a threat for Android devices that use Qualcomm chipset. The four vulnerabilities are

- Linux IPC router binding any port as a control port (CVE-2016-2059)
- Ashmem vulnerability (CVE-2016-5340)
- Use After Free Due to Race Conditions In KGSL (CVE-2016-2503, CVE-2106-2504).

Attackers can launch a privilege escalation attack using a malicious application and gain root access by exploiting any of these four vulnerabilities. The drivers that manage communication between different components of the chipset for their devices are also at high risk. Since the vulnerability is present in the drivers installed in the devices, the patch to fix the vulnerability can

Table 1. Hardware-based Attack

Attack	Category of Attack	Vulnerability Exploited	Strategy	Security Measure
Rowhammer attack	DMA-based Rowhammer attack	Rows in DRAM	• Install malicious applications • Hammer specific row in DRAM • Cause memory leakage	GuardION to an extent
Glitch attack	Rowhammer attack series	Loopholes in DDR3 and DDR4	• Use WebGL to get interactive graphics code in browser used in the target system • Use graphics to exploit loopholes found in DDR3 and DDR4	GuardION to an extent
Drammer attack	Rowhammer attack series (DRAM components)	DRAM chip components	• Install malicious applications • Exploit vulnerable components • Gain root access	GuardION to an extent
RAMpage attack	Rowhammer attack series (RAM components)	Components of RAM: • LPDDR2 • LPDDR3 • LPDDR4	• Install malicious applications • Exploit vulnerable components • Break memory allocator (ION) • Gain access to the sensitive memory location • Leak information	GuardION to an extent
QuadRooter Attacks	Qualcomm components-based attack	• Linux IPC router binding any port as a control port • Ashmem vulnerability • Use After Free Due to Race Conditions In KGSL	• Install malicious applications • Exploit vulnerable components • Gain access to root	NA (Patches can be provided by the distributors only)

only be availed from distributors or carriers of the component, which in turn is provided to the distributors or carriers by Qualcomm [17]. Analysis of these attacks is summarized in Table 1.

2.2 Kernel-based Attacks

A kernel is the heart of the Android system. Attacks targeting kernel are aimed at getting root access to the system and can cause severe damage. The Linux kernel [18] is directly or indirectly responsible for every action performed on the system and also for the management of different components of the Android system. The Linux kernel also takes care of Android runtime environment functionalities. Attackers are continually working in the direction of exploiting vulnerabilities in the kernel and related components like device drivers, memory, and Android Runtime (ART). Android has come out with many security features that include Android Sandbox, secure Inter-Process Communication (IPC), Encrypted File System, Cryptographic APIs, and so on [18]; still, many vulnerabilities continue to be exposed. In the recent past, the Android system has faced highly severe attacks exploiting vulnerabilities found in all these security features.

Next, the article will discuss attacks exploiting vulnerabilities found in different components of kernel.

2.2.1 Attacks Targeting Root Privilege. These attacks target the vulnerabilities found in the kernel to exploit privileges. For example, the zero-day local privilege escalation attack exploits Linux kernel vulnerability (CVE-2016-0728). The error was found in the way the objects were referenced by the keyctl(), a critical management function of the Linux kernel. The flaw, when exploited, results in memory leakage, which is used by attackers to gain unprivileged access to the system. This flaw affected Linux kernel version 3.8 and higher [19].

Gooligan Attack: Malware called Gooligan exploits the zero-day vulnerability (CVE-2013-6282) [20]. The zero-day vulnerability is called “zero-day” because it is the duration during which vulnerability is exposed and used by the attackers while there is no patch provided by the vendor. The CVE-2013-6282 vulnerability exists due to the error found in kernel API (put_user/get_user). Attackers inappropriately perform read/write operations on the kernel memory using a malicious application. Exploitation of this vulnerability allows attackers to escalate privileges and execute any malicious code. The gooligan attack persists; attackers are using the vulnerabilities VROOT and Towelroot, which are found in Android version 4-Froyo and also in version 5-Gingerbread [21].

DroidKungfu Attack: DroidKungFu malware[22], categorized under root exploit attack, is capable of bypassing the Android Sandbox security mechanism. The motive of this malware is to silently root into the mobile device and unlock all system files and functions. The malware is transmitted using repackaged applications. It exploits root privileges to use system resources without requesting for required permissions. DroidKungfu malware uses the exploits known as RageAgainstTheCage (CVE-2010-EASY) and Exploid (CVE-2009-1185) to escalate privileges. Exploid exploits the vulnerability found in the udev, which is a root-level code, by sending a NETLINK udev event message. The message tricks the udev into running an arbitrary binary as root when triggering a hotplug event. The RageAgainstTheCage exploited the Android Debug Bridge daemon(adb)'s functionality. The adb process starts with root privileges, and it has to call setuid() to lower its privileges to shell privileges. The exploit uses the fork function to start multiple processes. When the number of processes for the malware reaches the maximum permissible number RLIMIT_NPROC, the exploit kills the adb process. The adb process again restarts with root privilege and tries to call setuid() to lower its privilege, however, at that point, the exploit has already started RLIMIT_NPROC-1 processes other than adb process. So the call to setuid() fails and adb keeps running as root. The exploits come along with a repackaged application in encrypted form, and they get decrypted after getting installed on the device. While the user uses the application, the Trojan exploits the vulnerabilities in the background to perform actions such as download additional applications and adware without user's permission. The Trojan can also add new service and receiver into the application. Once the system is booted, the malware sends a notification to the receiver. The receiver can then initiate the service without user permission. Eventually attacker's gains unprivileged access to system resources. Security Enhanced (SE) Android prevents such attacks from exploiting the Linux kernel [23].

2.2.2 *Attacks Targeting Memory.*

Code Reuse attack exploits memory corruption vulnerability for attacking system. We can define Memory corruption vulnerability as a situation where malicious code can alter the memory without giving instructions explicitly. The drivers [24] and generic syscalls are privy to this vulnerability. The simplest approach used by the attackers to attack the kernel is to look for a bug in kernel code. The information about these bugs is available on Common Vulnerabilities and Exposures (CVE) site. These loopholes or bugs are used by attackers, to add malicious content in the kernel address space or redirect the execution of the kernel to the address space where malicious content is present. These attacks require the knowledge about the location of the targeted component in the kernel. Return-Oriented Programming (ROP) [25], Jump-Oriented-Programming (JOP) [26], return-to-libc [27] and ret-2-user [28]are a few examples of such attacks. These attacks exploit the address space in kernel and address space of its code segment. A lot has been done to harden Android security with regards to these vulnerabilities. Researchers have come up with well-crafted security features, which include randomization of addresses; both kernel address space and user address space. The method is called Address Space Layout Randomization (ASLR) [29]. The ASLR method randomly chooses the location in memory, where the executable program is to be loaded, thereby preventing the attacks that target these vulnerabilities to gain root access to the system and steal the data.

Return-oriented Programming Attack: Return-oriented Programming is a complex code reuse attack. Architectures like X86, ARM (Advanced RISC Machines) are vulnerable to this attack. Android is based on the ARM architecture, which makes it vulnerable to these attacks. The attackers use the code of an existing program for exploitation. This approach does not require attackers to inject any malicious code as it reuses existing code. The existing code here implies to the executable instructions that are also known as "Gadgets." Gadgets are generally machine-level instructions that are present within the libraries preceding the "return" instruction. Tools available to find the Gadgets are ROPgadget [30], Rp++ [31], Ropeme [32], Ropc [33], Nrop Gadgets [34]. These tools list the available Gadgets; after finding the Gadgets, the address of the first instruction in the Gadget is saved on the stack to overwrite the saved, extended instruction pointer on the stack, this process changes the return address. Gadgets are executed indirectly by controlling the flow of stack. Buffer Overflow attack is a classic example of execution of such attacks. In Buffer Overflow attack a program is used to overwrite memory location adjacent to the boundary of the buffer thus making it possible for attackers to execute any random code. Return-oriented Programming attack circumvent the security offered by the Data Execution Prevention (DEP) [35], ALSR and other techniques that are used to protect memory. Stack Canaries [35, 36] can be used to detect ROP and similar attacks that are based on buffer overflow of the stack. In the Stack Canary approach, a small canary (integer) value is selected and placed before the address of stack return pointer. Canary value selection takes place at the beginning of the program. Whenever buffer overflow occurs Canary value gets overwritten along with the stack return pointer. Every time a routine calls the stack return pointer, it checks the first Canary value, if the value is different from the initial value at the beginning of the program; it indicates that buffer overflow has occurred, which means, the value of the stack return pointer has changed. This method will stop the execution of any malicious activity thus making it challenging for attackers to execute the buffer overflow attack in the presence of stack canaries. To breach the security offered by Stack Canary, attackers have to look for some source for information leakage. Some other defensive techniques are ASLR, ROP Defender [37], KBouncer [38], Dynamic Binary Instrumentation (DBI) [39], Instruction set randomization [40], and so on. Researchers [36] have shown that return instruction is not compulsory to execute Return-oriented Programming attacks. JOP [41], SOP [42], BROP [43], SROP [44] are variants of ROP attack. JOP, unlike ROP, does not rely on the address of stack pointer or return instruction.

Jump-oriented Programming Attack: For JOP attacks, as the name suggests, jump instructions are used. One gadget dispatcher and a program counter are the two main components required to perform this attack. A dispatcher is used to send functional gadgets and then control their execution. Role of the program counter, which is a register, is to point to the dispatch table, which contains the addresses and contents of the gadgets. Attackers with the use of these components overwrite the stack and a function pointer. Another approach followed by attackers for launching JOP attack is to overwrite setjmp buffer [41]. Control-flow integrity (CFI) [45], Data-flow integrity (DFI) [46], Program shepherding [47], and a few other similar techniques can be used to defend against ROP and JOP attack. However, these techniques require a significantly large number of resources and codebase, which makes it cumbersome to deploy such defensive techniques. The attackers have also introduced String-oriented Programming (SOP), an extension to ROP and JOP attacks. In SOP, the attacker uses format string exploit [42] to overwrite the Global Offset Table entries or function pointers. In a format string attack, the attacker exploits a string that is controlled by the user and passes that string as an argument to a function that belongs to print class. Tokens like %x, %p, %n, %s are passed with the strings as an argument. These tokens and their combinations can be used by the attacker to collect details about the implementation of the stack or can be used to fetch any other relevant information. There are two ways in which attackers can get control over the execution flow. First is a direct method in which flow is redirected to any arbitrary code by overwriting the stack pointer. Depending upon the position of the buffer, either ROP or JOP can be used to overwrite the stack pointer. ROP is usable when the buffer location is on the stack, but in the case where the buffer is located on the heap then JOP is used. Another method is an indirect approach to control the flow of a program. In the latter method, the primary motive is to overwrite an entry in the Global Offset Table to gain control of the program. The overwriting process is similar to JOP and ROP. ASLR, Stack Canaries, and DEP are existing security measures adopted for protecting Android against SOP attacks. The ROP and similar attacks require knowledge about the binaries to exploit it.

Blind ROP Attack: A Blind ROP (BROP) [43] attack can be launched even without knowledge of target code or binaries. This attack is able to exploit a stack buffer remotely without using binaries. This method can hack servers that are based on open source platforms, and this method can also hack closed binary operations. This hacking can take place even without knowing the target source code and binaries. Two requirements to execute this attack are (a) knowing the vulnerability in the stack and the information about how to exploit it and (b) a server application that will restart every time after it crashes. This known vulnerability in the stack is exploited to launch a buffer overflow attack using known methods. The buffer overflow causes the server to crash. The attacker must be able to crash the server *n* number of times and every time server is supposed to restart. The period during which server restarts is utilized by attackers to find enough instructions/binaries (gadgets) to trigger and control arguments of "write" system call of the victim system. Attackers then embed sufficient malicious binaries to exploit the system. In this way, attackers can remotely attack the system and hack it. Randomizing the address space of the server when it restarts, sleep on crash, and Control-flow integrity (CFI) are few techniques that are used to mitigate a BROP attack.

Sigreturn-oriented Programming Attack: Another variant of ROP is SROP [44]. This method breaches security offered by ASLR, DEP and Stack Canaries. The malware developers use SROP for exploiting source code and also for introducing a backdoor into the target system. The method of attack is similar to ROP, but the preconditions for launching the SROP attack are different from that of ROP. Unlike ROP, in SROP only a single gadget is to be known for exploitation, e.g., sigreturn or syscall. The gadgets have mostly fixed positions in the memory. The attack mechanism involves placing a fake structure of signal context [48] on the stack (call stack) and then overwriting a return address with the address of a known gadget. This gadget will help the attacker in accessing either syscall or sigreturn. Further, SROP is platform independent, and it can attack against different operating systems; this makes it even more critical attack. Technologies present to defend system from these attacks are ASLR, Signal cookies [49], Vsyscall emulation [50, 51], Return-Address Protection (RAP) [52], and Control-flow Enforcement Technology (CET) [53].

Returning to Libc Attack: This is an approach that can exploit the weakness in the system to impose buffer overflow attack on a stack that is non-executable. As the stack is non-executable, it is not possible to inject malicious code and execute using the traditional buffer overflow method. To overcome this problem, the attackers use the function provided by the library to execute the return to libc attack. In this method, the malware overwrites the return address, using a stack overflow method, to a libc function, and if it passes right arguments to this function, then the function will execute the code bypassing the security provided by non-executable stack [27]. Recently, a return-to-art attack is also used by attackers to attack new ART architecture. This attack is very much like a return-to-libc attack [54]. ASLR, Stack smashing, and No eXecute(NX) bit methods safeguard the system from the return-to-libc attack to an extent.

Return-to-user Attack: ret2usr attack is also an example of code reuse attack. In this attack, the malware application overwrites the kernel pointers. Attackers then have control over the execution flow, and they redirect it to the address of arbitrary code in the user space. It is essential to isolate the kernel and user address space to increase the kernel security. Researchers have proposed many methods, which prevent the random transfer of control flow between the kernel and userspace. There exists defensive methods for ret2usr attack like kGuard, Supervisor Mode Execution Protection (SMEP)/Supervisor Mode Access Protection (SMAP), Privileged Execute-Never (PXN)/ Privileged Access-Never (PAN), and Non-executable pages (KERNEXEC)/Userland-kernel separation (UDEREF) [28]. These methods isolate both address spaces but only explicitly and not implicitly. So attackers are exploiting implicitly shared address space to breach the present defensive techniques. The ret2dir attack does the same by exploiting the information given in Page Frame Numbers (PFN's). Researchers [55] have proposed an eXclusive Page Frame Ownership (XPFO) scheme that helps in protecting the kernel from ret2dir attack.

ROP attacks and its variants are still persistent and are continuously exploiting weaknesses in security policies employed by the Android system [35]. Attackers have found various mechanisms to bypass the security offered by ASLR and other security techniques. Many factors contribute to the proliferation of these attacks one of them is using security techniques like ASLR require a large number of resources to perform well, which degrades the performance of the device and also increases its cost. Hence, most of the systems do not adopt these techniques. Also, the Zygote process used by application runtime architecture in the Android system to fork each user-based application weakens the ASLR security. Each application has a systematic memory design that challenges the concept of randomization used in ASLR technique [56]. In Android Oreo a Kernel Address Space Layout Randomization (KASLR) technique is used to reduce attacks at the kernel level. KASLR randomizes the space used to place kernel code during boot time [57]. It renders help in preventing the attacks mentioned above. Every time the system is booted, this technique randomizes the location of kernel code, this helps in safeguarding the kernel from code reuse attacks [58]. However, researchers [59] have shown that this KASLR technique can be evaded by a timing attack named DrK, which exploits a hardware feature called Intel Transactional Synchronization Extension (TSX) found in latest CPUs. This attack can de-randomize the location of kernel code. Usually, OS is supposed to intervene when a hardware exception occurs, but TSX suppresses such exceptions from being notified to the OS. It results in abortion of the transaction and invoking of the abort handler. The time difference in accessing mapped and unmapped memory locations in the kernel during abort handling is calculated by the DrK to de-randomize the kernel code. So, it will not be surprising in future if attackers explore any such vulnerability in the Android environment that can put the credibility of KASLR in a questionable state. Other than KASLR,[60] the latest advanced security features on Android Oreo:

- PXN - Prevents a process from executing a code for which it does not have privilege.
- PAN - Prevents from redirecting a kernel pointer to userspace.
- Post-init read-only memory - Prevents from overwriting important kernel pointers.

However, even these advanced security features fail to prevent against root exploitation technique called Kernel Space Mirroring Attack (KSMA). The KSMA can bypass both PAN and PXN and reach the kernel space to exploit kernel vulnerabilities. The KSMA exploits minor design fault of ARM virtual memory system's virtual memory translation descriptor. In violation of the principle of least privilege, the data access permissions for stage 1 of the Exception level 0 (user mode), provide read/write capability to the unprivileged user. Though the permission seems useless because of PAN; attackers have successfully exploited this vulnerability to read and write kernel data virtual address without using syscalls, from user mode. It shows that even the latest Android Oreo devices can easily be rooted [61].

2.2.3 Attacks Targeting Bootloader. Bootloader is a vital program that is responsible for performing some crucial tasks whenever a system is switched on like initiating the operating system and some other kernel processes. In current mobile phones bootloaders, apart from performing essential functions, are also responsible for maintaining the security of the device. One of the primary tasks of the bootloader is to ensure the integrity of the Chain of Trust (CoT) policy of the system. According to CoT policy, the bootloader has to verify the integrity of processes in each stage during booting before the execution of the processes. The booting process is supposed to be invulnerable to kernel attacks where attackers gain complete control over the kernel, and it should be able to maintain CoT policy of system even in a compromised system. That is a situation where the bootloader is expected to perform its intended functions while taking malicious input from attackers. Many factors hinder the bootloaders from performing its functions and also make them vulnerable to attacks. Most of the vulnerabilities exist due to flaws found in coding and implementation of bootloaders. Redini et al. [62] have discussed few such vulnerabilities and how attackers exploit them. They have proposed a hybrid method called BOOTSTOMP. It is a multi-tag taint analysis method that comprises features of both static and dynamic method to defend the system's bootloader from attacks by identifying the hidden vulnerabilities that are present in it. CVE-2014-9798 and CVE-2015-8893 are vulnerabilities found in the Qualcomm bootloader. The attacker can impose denial of service attack on the system using a malicious application to exploit this vulnerability of bootloader in which it does not verify the address relationship in between Tags and Aboot. Information about the patch is available on the Android security bulletin site [63]. BOOTSTOMP was able to identify CVE-2014-9798 vulnerability present in previous versions of Qualcomm devices that could be exploited by the attackers. BOOTSTOMP also exposed severe other security vulnerabilities in the Huawei and NVIDIA'S bootloader. These vulnerabilities allow attackers to run arbitrary code, perform privilege escalation, corrupt memory, and impose buffer overflow or denial of service attack on the system. It is crucial for developers to design bootloaders keeping in consideration these security issues [64]. Running arbitrary code by initiating memory corruption in bootloader code, attackers can gain control of Trusted Execution Environment of the system. Bricking and Unsafe unlock are other method used by attackers to attack the system [62].

2.2.4 Attacks Targeting the Device Driver. The device driver is an interface that enables communication between software and hardware. In a system, there are many hardware components, and for each component, there is a unique driver installed in the system. Each driver has specific functions. These functions allow the operating system and other programs to interact with the hardware and accordingly eliminate the need of knowing complex hardware details.

Device driver vulnerability allows an attacker to breach the driver and ultimately crash the system by gaining control over the kernel or make the system unusable. One of the main reasons for security vulnerabilities in the device driver is that developers implement drivers with the only objective that driver must function accurately without giving focus on the possible vulnerabilities in the code that can be exploited by attackers. When a company releases a driver in the market, attackers can easily find out the security gaps in the code. So it is vital for developers to implement drivers keeping in consideration security features that need to be employed [65]. In Android, bugs are mostly found either in a root driver or drivers of applications like Camera, Wifi, or Bluetooth, which are pre-installed in the device [66]. There are large numbers of Android phones in the market, and they have different hardware components with different drivers. There is always a notable gap between the time any vulnerability is exposed and the patch to fix it is released. So, attackers effectively utilize this time gap for exploiting the vulnerability found in the driver and launch attack against the system. Such attacks are called time of check to time of use (TOCTOU) attacks. This vulnerability arises when driver and user have direct access to user data buffer. So there are chances that both may be accessing and updating data buffer simultaneously. Attackers are also imposing attacks like denial of service, buffer overflow attack, privilege escalation attack, network traffic-based attacks on the system by exploiting the vulnerabilities found in the driver code. In 2016, researchers of Zlab explained how two vulnerabilities, CVE-2016-2435 and CVE-2016-2411, found in NVIDIA Video and MSM Thermal driver, respectively, affected Android version 6.0 (Marshmallow) in a Nexus device. Attackers use a malicious application to gain root access to the system by exploiting NVIDIA video driver vulnerability (CVE-2016-2435). Attackers can corrupt memory values and, finally, launch a privilege escalation attack against the system. After gaining the complete access over the system, the attacker can disclose users' confidential data or collapse the system by exhausting the resources. CVE-2016-2411 is a security flaw that includes two drivers, Qualcomm power management kernel and MSM Thermal driver. A cluster_id is passed to MSM Thermal driver function msm_thermal_process_ voltage_table_req from userspace without validation. The cluster_id is given an out-of-range value by attackers that results in heap overflow. Eventually, attackers can gain root access by privilege escalation and can also fail the security offered via SELinux [67]. The implementation flaws are present in both the kernel and external driver (example Qualcomm, PowerVR). CVE-2018-9417 and CVE-2018-5838 are the latest vulnerability reported [68] that are found in a kernel USB driver and OpenGL ES driver a Qualcomm component, respectively. Both vulnerabilities allow attackers to launch a privilege escalation attack against the system by using the malicious application to run arbitrary code. Shen [69] has shown a step-by-step procedure to exploit Trusted Execution Environment (TEE) used by Huawei HiSilicon. A vulnerability in driver /dev/tc_ns_client is exploited to get access to kernel privileges. Attackers execute arbitrary code and use a "_FPC_readImage" system call to get images of fingerprints from the device, bypassing the RTOSck (TEE's kernel) security test. In Reference [66] the author has discussed common flaws in driver codes and how the attackers exploit them. Developers must strictly follow the list of "dos" and "don'ts" while designing the device driver to maintain the security of the system [65]. Analysis of few of these attacks is summarized in Table 2.

2.3 Hardware Abstraction Layer-based Attacks

This layer includes library modules that implement interfaces for Camera, Bluetooth, Wi-Fi, GPS, Radio, and other components in the Android system [18]. Attackers target the interfaces and use these components to peek into the system. In Broadcom Company's Wi-Fi chipset, which is present in most Android and iOS devices, attackers found a highly dangerous vulnerability that allowed the attacker to get control of the complete system by using Booby-Trapped Signals [70]. Later a patch for this vulnerability was released. Researchers [71] have exposed severe vulnerabilities in Wi-Fi security protocol WPA2 that is used in most of the modern operating systems, including Android version 6.0 (Marshmallow) and higher. This vulnerability is present even in iOS, Windows, and OpenBSD. Researchers have also listed the vulnerabilities found in the WPA2 that are exploited by attackers by using a novel method of attack known as Key Reinstallation Attack (KRACKs). For exploiting this vulnerability, the attacker needs to be within the Wi-Fi range of the target system. This attack can be used to steal the confidential data of the victim and can even decrypt the encrypted file in the system. Attackers can use applications to spy on the user's data using Audio Channels, GPS, Camera, and other components. Google Play Services offers some unsafe features to the applications that can be exploited by the intruders. Some of these features are by default enabled by the service. If the user disables them, then it will affect the normal functioning of the fundamental features of the system. A brute-force attack is used to intrude into the Android file system, which stores the system's private keys. The attack targets the kernel by cracking the system passwords [72].

Table 2. Kernel-based Attacks

Attack	Category of attack	Vulnerability exploited	Strategy	Security measure
Gooligan attack	Root privilege	Kernel API put_user/get- _user	• Malicious applications • Inappropriate validation of memory • Privilege escalation • Root access to the system	NA
Droid KungFu attack	Root privilege	• Root Privileges of adb • Vulnerability in udev	• Repackaged applications for malware transmission • Silent mobile device rooting • Unlocks all system files and functions • Installs itself without any user interaction	Security Enhanced (SE) Linux in Android • KASLR • PAN • PXN
ROP attack	Code reuse	Vulnerable Executable code present in libraries (binary, shared)	• Finding the gadgets using tools or using objdump and grep command • Controlling extended instruction and stack pointer. • Overwriting the return address and make the stack pointer, point to the location of the arbitrary code	• ASLR • Stack Canary protection • ROP Defender • KBouncer • DBI • Instruction set randomization • CFI • DFI • Program shepherding
JOP attack	Code reuse	Vulnerable jump instructions	• Finding the gadgets • Controlling the flow of gadgets using Dispatcher • Ovrerwriting: (a) Stack (b) Function pointer (c) Setjmp buffer	• ASLR • CFI • DFI • Program shepherding
SOP attack	Code reuse	Format Strings	• Collect information about stack • Overwrite (a) Stack function pointers (b) GOT entry	• ASLR • Stack Canaries • DEP
Return to usr attack	Code Reuse	• Shared address space between kernel and user	• Overwriting kernel pointers • Redirecting execution flow to arbitrary code address	• kGuard • SMEP/SMAP • PXN/PAN • KERNEXEC/UDEREF
Kernel Space Mirroring Attack	Permission Exploitation	Read Write data access to EL(0) user mode	• Read Write from EL(0) and EL(1), break kernel isolation • Bypassing PXN and PAN and post init read only memory • Accessing kernel from EL(0)	NA
Attack on Boot Loader	Denial of Service	Non-verification of addresses	Sending flood of traffic to the target system.	BOOTSTOMP up to an extent
TOCTOU Attack on Driver	Zero-day	Driver and User with direct access to data buffer	• Gaining control over the Device Driver • Corrupting memory values and launching privilege escalation attack using device driver	NA

First, the attacker works on gaining comprehensive knowledge of the security measures and weaknesses in Android's encryption system and then uses that information to break through the system and gain control over the kernel. It is done by repeatedly guessing the pair of keys until the exact match is found. Attackers these days are using jailbreaking or rooting tools like Aircrack-ng to guess the passwords. After gaining control over the kernel, the attacker has the full access to all the resources and data stored on the system. A robust encryption mechanism is needed to tackle such attacks. Lately, Android systems incorporate strong encryption algorithms and Key Derivation Process (KDP) employing SCRYPT function for generating key-based password [73]. This process makes it challenging for attackers to execute a brute-force attack, as it will require intensive knowledge and a significant amount of memory and resources for computation.

The Sturdy Android Keychain Management system also plays a crucial role in improving the defenses against brute-force attacks. A two-factor authentication keys system is incorporated into Android Oreo, which further strengthens Android's security [21]. Malware known as Invisible Man [74] gets into the system when the user installs fake Flash updates, disguised as authenticated ones, from the unauthorized website. This malware, once installed into the system, can get the bank details of the user by acting like a key-logger. Simultaneously, it installs itself as the default text message application and grants itself access to send and receive text messages in the system. Such records can be used by attackers, as well as analyzers, for spying on the user's data. Malicious applications, installed into the system, can also use the Audio channels to steal the password. Such applications send deceitful commands, through the speaker of the victim's system, to its microphone, which behaves as confused deputy. Talk Back Accessibility Service of Android is used to accomplish the purpose of this attack, as it reads out the password typed by the victim to the attacker [75–77]. AuDroid [75] is a policy enforcement technique that is employable with SELinux for preventing and detecting attacks that use audio channels for exploiting the system. SemaDroid [78] is a method to protect from attacks, like covert channel attacks [79], that target the sensor data, which includes data related to Camera, GPS, Microphone, Motion Sensors, and other sensors in the system. They enforce user policies that put restriction on the applications installed on the system from collecting the critical information from sensors. Such policies can control the type of information applications are collecting from the host device. It can defeat any stealthy attack on the sensors, thereby protecting the user's private information.

Security of HAL has been substantially improved in Android Oreo. Now, different device drivers have separate HALs. Each driver is controlled only by the HAL to which it is assigned. When a process requests for a device driver, it gets a particular HAL, and together they run in a sandbox. Thus, the process gets only necessary privileges that are required to complete its job. This way the processes will not be able to access device drivers directly, and they can also not access device drivers that they have not requested. It helps in limiting privilege escalation attacks and even phishing attacks, which use combined privileges on different components for spying or stealing users' data [80]. Analysis of these attacks is summarized in Table 3.

2.4 Application-based Attacks

malware developers exploit the vulnerabilities of default applications, as well as the applications installed into the system, from Play Stores and websites. Applications request for the privileges, from the system, at the time of installation, for smoothly running, later on. Various researchers have suggested that the applications that users install from websites request for unnecessary permissions, which they do not require to performing their job. Attackers take advantage of this situation and use attacks like privilege escalation [81], for exploiting the target system, at different levels. Attackers are also misusing the flaws in native libraries and the third-party libraries found in the applications. They use these libraries to request for combined permissions. Escalated

Table 3. HAL-based Attacks

Attack	Category of attack	Vulnerability exploited	Strategy	Security measure
Key Reinstallation Attacks	Replay attack	WPA2 vulnerability of allowing reconnection using same key value for third handshake in the four way handshake mechanism	• The attacker repeatedly resends the third handshake of other device to the target to reset the encryption key. • The target accepts same value to encrypt the data to be sent. • The attacker can then identify the keychain used to encrypt that data by matching multiple packets.	• Patches for this vulnerability is given in Android Security Bulletin 2017-11-06
Brute-force attack	Encryption system	• Weakness in the Android Encryption System • Weak Passwords and Authentication System	• Guessing the right pair of keys using Open source password cracking tools	• Strong passwords and encryption system • Sturdy Android Key chain Management system and Key Derivation Process • Two-factor authentication keys system • CFI • DFI • Program shepherding
Invisible man attack	Key-logger (sensor-based attack)	Security weakness in Android accessibility services	• Attacker lures victim to visit a mobile banking site. • Attacker launches phishing attack to steal victim's login credentials.	• User must always download applications from trusted site or Google's Play Store and not from any third-party sites that are not authorized.
Audio channel attacks	Media-based attacks	Weak security enforcement policies for accessing audio channels	• Attackers use services like Talkback to steal passwords from victim's device	• AuDroid-policy enforcement
Covert channel attack	Sensor-based attack	Exploit permission system and Inter-Process Communication	• Attackers use malicious applications to retrieve data from the sensors • Attackers steal sensitive data	• Semadroid • Security policy enforcement • Process Isolation • Strong permission System

privilege rights give attackers access to any component of the victim's system, including the kernel. Attackers can even gain root privileges and can cause irreparable damage to the system. After sneaking into the system, attackers can spy on the user's data for personal motive or can sell it to third parties for money. Privilege Escalation attacks and attacks based on libraries, both have placed Android security in peril. Malicious applications are used by the attackers to breach into the Android system and steal vital details. Runtime Information Gathering Attack (RIG) is an attack through which attackers can accumulate information from the victim's system, during the runtime of the malicious application. The malicious applications exploit the permissions, given to the application, at the installation time to perform this attack. Researchers [82] have shown how this attack can pose a severe threat to the Internet of Things, administered by Android. They have also implemented the technique, called App Guardian, which helps in preventing from the RIG attacks. App Guardian keeps a check on the suspicious application running and stops all other processes running in the background. Once the process stops the executing, it restarts the stopped processes and cleans the memory used during runtime by the suspicious application so that no trace of information is left for the attackers to steal from the system. The application-based attacks are targeting the resources of the Android system like memory, CPU, files, disk, battery to damage the system.

The malicious application can consume more memory, battery, and processing power than their regular counterparts. Malicious applications can exhaust the system and bring it to a standstill. Attackers can use multimedia for targeting the battery of the system. Well-designed multimedia files, which remain stealthy, are used by attackers for such attacks. These files consume more battery power, and the system will get exhausted quickly, and its processing power will slow down. It will not be able to perform any normal operation and will ultimately collapse. Such attacks are the example of energy-based attacks [83].

2.4.1 Library-based Attacks. Libraries, used for supporting security features for protecting the Android environment, cannot be trusted fully, as even they are not flawless. There are errors in the code, which can pose serious security issues when used. In March 2017, Android Security Bulletin [84] released the list of security updates and vulnerability, in which they ranked remote code execution vulnerability in OpenSSL and BoringSSL as critical, since it allows the attacker to corrupt memory at the time of processing data as well as the file. Attackers accomplished this task by using a carefully designed file. From time to time patches and updated versions, to provide a solution for the vulnerabilities, are released. OpenSSL has released OpenSSL 1.1.0f and OpenSSL 1.0.2lf [85] that include bug fixes. Undoubtedly, the security of an application is dependent on the libraries included in the programming. Any vulnerability in the library will be included in the utilizing application.

2.4.2 Third-party Library Attacks. Lately, third-party libraries have become extremely vulnerable to attacks. Poor developer practice is one of the significant causes, for this kind of vulnerability. In 2016, buffer overflow vulnerability was reported in the getaddrinfo() function; this function was used to search domain name included in the GNU C library. The solution to the problem came 7 months after its initial reporting, which gave enough time for attackers to peek inside the system and cause severe damage, which was sometimes irrecoverable. According to researchers, this security loophole proved fatal for the system, and it even allowed attackers to perform malicious activities (CVE-2015-7547) from a remote system [86]. Software companies have faced the consequences of the vulnerabilities caused by third-party libraries.

These vulnerabilities provide privilege to the attackers to gain easy access to the Android device, breach the privacy of the user, exposing the user's private information to the outside world. Attackers can even initiate Code injection attacks [93, 94]; they can take control of the system by performing malicious acts like account hijacking [95], or an attacker can remotely perform malicious activities by linking compromised Dropbox account to the target system [96]. Libraries are an integral part of the .apk file. They are included in the application code to ask for the permissions that are required to run the applications on the system. Third-party libraries hold a significant share in the Android application code (higher than 60%) [89]. Various applications use 20 or more third-party libraries [97]. These third-party libraries included in the source code the request for the permissions. Among these are some permissions that, once granted, can provide open access to the system. The main reason for this can be a lack of consistency between the description of the application and its functionalities. This inconsistency is the result of a large number of permissions requested by the third-party libraries, thus opening a channel for the malicious applications to intrude into the system [89]. The tool called WHYPER is used to analyze this inconsistency. WHYPER is a tool based on Natural Language Processing (NLP); it looks for the sentences in application code that can explain why the application requires the requested permission. It helps in establishing a relationship between the description of an application and the need for the requested permission. For evaluation purposes, WHYPER used a dataset of 581 popular applications and three necessary permissions, i.e., ADDRESS_BOOK, READ_CALENDAR, and RECORD_AUDIO. The reason for using these three permissions was that they maintain the integrity of a critical security resource.

Table 4. Analysis of Third-party Library Detection Tools

Detection tool/year	Worked with	Technique	Obfuscation	Performance	Limitation
AdDetect [87] 2014	Package Dependency Graph (PDG)	Hierarchical Agglomerative Clustering	Yes	95.34% accuracy is achieved in detecting Ad libraries [87]	• Insensitive to library versions[88]
WuKong [89] 2015	Sub-package level features	Clustering	Yes	For Filtering third-party libraries 60% of accuracy level is achieved in removing sub-packages from one lakh Android applications.	• Detection mechanism can be evaded by complex obfuscation algorithms. • Fails to detect app clones, when small dataset is used and when cloned app is composed of multiple small applications or when it has comparatively more code than usual.
LibRadar [90] 2016	API features	Multi-level clustering [89]	Yes	LibRadar takes 10 milliseconds to detect library	• High processing time required for extracting unique features from each library. • Insensitive to exact library versions [88]
LibSift [91] 2016	Package Dependency Graph (PDG)	Multi-level clustering	No	95.61% accuracy achieved in detecting libraries.	• Fails against familiar obfuscation technique: Renaming of packages.
LibScout [92] 2017	Original library SDKs (compiled .jar/.aar files)	Library Profile Matching	Yes	In Reference [88], the author has used a dataset of 98 distinct libraries and 12,118 apps from which LibScout is able to identify 2,028 and 260 libraries accurately.	• In libraries, LibScout is unable to identify changes made at patch-level [88] • Large dataset is required for better analysis and results.

Results showed that WHYPER was able to achieve accuracy of 97.3% [98]. From a security point of view, it is essential for the Android security mechanism to be able to detect and analyze the library code separately from the application code. There are many techniques available for detecting library code, but, like any other techniques, malware developers can evade them with the help of the code obfuscation method. WuKong [89], LibRadar [90], LibScout [92], AdDetect [87], and LibSift [91] are few library detection tools. Analysis of these tools is summarized in Table 4.

Using an updated version of libraries in application code is of prime importance, as most of the security flaws arise because developers often use the outdated version of libraries or they slowly adapt the updated version. This time gap between identification of vulnerability, patch release, and adaptation of that patch is utilized by the attacker for intruding into the system and causing damage to the system. Leisurely adapting the patches is not the only difficulty; the obfuscation technique used by the attacker can help them bypass the security mechanism or the tool used for library detection effortlessly. The most commonly used method for detecting the third-party library is merely looking for the packages inside the library and matching their name with the previously known packages. However, this is not a reliable method and will fail in the case when malware developers use a code obfuscation method like the renaming of the identifier, as in this the method package name that the detector searches will not exist due to renaming [99]. To tackle the problem of code obfuscation, where an identifier name is changed or obfuscated, the malware detectors should use library detection techniques that employ machine-learning approaches. For example, PEDAL (Privilege De-escalation) [100] is a tool proposed by B. Liu et al., where the classifier is trained to detect the libraries by extracting features from the library SDKs code and using the information based on the association between the packages. Library detection technique proposed in Reference [99] consists of a Profile Matching Algorithm. According to the author, it can identify library version, used in the application accurately and can withstand code obfuscation like control-flow randomization or API hiding, as the detection technique is not based on library code but on matching the profiles. These profiles represent the association information of the packages. The proposed Profile Matching Algorithm consists of two parts; in the first part profiles are extracted from the database, which is constructed from original library SDKs and in the second part profiles are compared to find a match for library version. The algorithm assigns scores from 0 to 1, based on finding an exact, partial, or no match. The result shows that the developers adapt to the updated versions of vulnerable libraries very slowly, giving attackers enough time to harm the system and thus putting the user's privacy at risk of getting exposed.

2.4.3 Intra-library Collusion. Intra-library collusion [101] occurs when an individual library obtains a combined set of privileges assigned to multiple applications on a system. It occurs because of the sharing of libraries among several applications, especially third-party applications. Each application in Android has a different set of permissions or privileges assigned to it; since the individual library works as a shared library between multiple applications, it gets access to the entire set of permissions assigned to those applications [102]. Malware attackers are exploiting this security vulnerability as, such flaws allow access to the user's private information, with the help of the collective privileges, gained by shared libraries. After analyzing 15,000 applications, the .com/facebook library (11.9%), Google Analytics libraries (9.8%), and Flurry libraries (6.3%) were found to be holding a significant share among the libraries that used "Intra-library collusion." Revocation of privileges is not of much help. It is challenging to detect this flaw, because malicious activity, mostly, does not take place at the user's end but at the servers belonging to a third party [101]. The security model of an Android operating system does not perform privilege separation in-between applications and their in-built libraries or Ad libraries. The security model of an Android operating system does not perform privilege separation in-between applications and their in-built libraries or Ad libraries. Due to this, all the permissions allowed to an application are by default accessible to its in-built libraries. Sometimes, permissions are added intentionally by developers, to allow these libraries to run smoothly on the system. Some of the libraries especially Ad libraries take undue advantage of intra-library collusion. They require access to the Internet for downloading the content to be advertised Ad libraries are supported by ad networks, which act as an intermediate between advertising agents and application developers. The Ad libraries, taking advantage of the Intra-library collusion flaw, gain added permissions like being able to access information regarding the location of the user or personal information of the user like International Mobile Equipment Identity (IMEI). The attackers can use the SIM card to access Such information, which can help the ad library in extracting information regarding the user. Added permissions give these Ad libraries the privilege to gain access to the stored files in the system, record voice or audio messages or even use the information from other applications installed in the system. Analysis of all this information helps the Ad libraries interpret user's interest and lure them by fetching and downloading the particular customized content. The purpose of the Ad libraries is to attract users to click the Ads, thereby increasing the number of clicks and hence increasing the revenue. Because of the Intra-library collusion, Ad libraries are proving beneficiary for the advertising industry, analytics, and the malware developers. Malware developers are using this flaw to get into the system and harm the targeted system or fetch private information of the user. Attackers use this information to threaten the user or sell this information for economic profit [88, 102].

2.4.4 Privilege Escalation Attacks. Time and again, attacks exploiting privilege escalation vulnerabilities are reported, that too with a much higher rate. For example, Dirty COW (CVE-2016-5195), well-known privilege escalation vulnerability, was found in the Linux kernel versions 5, 6, and 7. It was exposed in 2016 by the researcher Phil Oester. COW stands for "Copy-on-Write," used by Linux for reducing the repetition of objects in the memory. For exploiting this vulnerability, one necessary condition is that the attacker must have a connection with the host server. Then, after gaining unprivileged access to the target file, a race condition is created to exploit Copy-on-Write method that allows the attacker to write on the file, which is otherwise available for reading only [103]. The patch for this vulnerability is included in Linux kernel 7.3 and later versions [104].

On October 2, 2017, the Android Security Bulletin re-leased [84] listed highly severe privilege escalation vulnerabilities in the Android framework (CVE-2017-0806) and its media framework (CVE-2017-0812). A security patch was released soon after these vulnerabilities were exposed, which addressed these issues. Although no exploitation was reported based on these vulnerabilities till date, reportedly, the vulnerability found in Android framework could have allowed the application, with malicious content to acquire added permissions, which were otherwise not assigned to them. For this, they do not require any consent from the user. Another vulnerability found in the Android media framework could have allowed system access to any attacker, having remote access to the system. The attacker will use a specially designed file to run malicious content on the targeted system. For this, the attacker requires access to a process, with all the necessary privileges assigned, so that it can provide a favorable environment required for running the malicious code. Broadly, there are two types of privilege escalation attacks: confused deputy attack [76] and collusion attack [105]. The former type, exploit the vulnerability of the target system, and the latter can implement the malicious application, attacking the system. In Reference[106], the authors discussed methods that can be useful for ensuring access control at the user level.

2.4.5 Privilege Escalation Attacks Using Third-party and Advertising Libraries. Third-party libraries are the easiest approach that attackers are using for exploiting privilege escalation vulnerability at different levels in the Android system. Taylor et al. [102] analyzed third-party libraries in more than 30,000 smartphones. They found that per day from a single smartphone, 2.4 times private information is leaked by ad libraries, using permissions assigned to third-party libraries and that the average user has their data sent to 1.7 different ad servers per day. From the security perspective, it has become essential to detect and prevent system and users from privilege escalation attacks. For this purpose, it is necessary to provide the applications and libraries with limited privileges, and they should not be allowed to use each other's permissions or any other permission not assigned to them. Analysis of few of these attacks is summarized in Table 5. The next section will discuss about various factors and loopholes that contribute to the increase of malicious activities.

3 FACTORS CONTRIBUTING TO THE INCREASE OF MALWARE

One of the main reason behind the massive growth of Android malware is the underlying architecture of the Android system and its growing application market [11]. The Android operating system provides an open source platform where any developer can develop applications. This open source nature of Android makes it more vulnerable to attacks and exploitations from malware developers.

3.1 Permission System Exploitation

As the Android system continues to evolve, so does its permission system, which is becoming exposed to malware attacks. Malware developers are taking advantage of weaknesses of not only permission system but also exploiting Android infrastructure susceptibility to attacks. the Android

Table 5. Application-based Attacks

Attack	Category of attack	Vulnerability exploited	Strategy	Security measure
Runtime Information Gathering Attack (RIG)	Permission system sxploitation	Default permission System	Attacker use malicious applications to exploit the permissions	App Guardian
Energy-based attacks	Denial of service	Exploiting complex interrelationship between hardware, software and network components in device	Attacker use malicious applications to consume resources in the victim's device and bring it to standstill.	Stronger permissions system
Cloak and dagger attack	Permission system exploitation	Default permission System	The attacker takes control over User Interface Feedback Loop using default permission.	Stronger permissions system
Man-in-the-middle attack	Network-based attack	Non-authentication of the two parties in communication.	The attacker pretends to be one of the trusted hosts	TLS Public Key Infrastructure
Code injection attack	permission eystem exploitation	Third-party libraries Vulnerability	The attacker gains excess permissions using third-party libraries and injects malicious code into the system.	PEDAL (Profile Matching Algorithm)
Ad-libraries attack	Permission eystem exploitation	Intra library collusion	The attacker exploits the additional permissions acquired by the library due to intra-library collusion	The developers should provide permissions in a justifiable way and not just so that the application runs smoothly
Dirty COW	Permission system exploitation	Privilege Escalation Attacks	The attacker first establishes a connection with the target then creates a race condition to exploit Copy-on-Write method to write on an otherwise read-only file	Patch is included in Linux kernel 7.3 and later versions

system comes with its in-built security mechanism based on a Linux kernel [107]. The permission system forms the critical component of the Android security mechanism. It restricts the applications from accessing user's personal information like user's identification number, bank account number, and contact numbers. For an application, to utilize any of the system's resources and interface, it requires separate permissions [108]. Presently, 146 permissions [109] have been included in the permission system to provide a flexible environment to the applications to run on the latest versions of the Android system. The Android Manifest Permission file, used for static analysis of applications, includes the permissions required by the application. The present permission system includes risky permissions that are complex and makes it difficult, not only for the user's to understand [110], but also becomes a challenging task for system's security mechanism to restrict the requesting application access from accessing user's confidential information. During installation, applications request for permission to access different components of the system as and in case the user does not grant all the permissions, the application does not run. So, users accept such permissions, to be able to use the application; giving knowingly or unknowingly consent to the attackers to get into the system. Attackers can use benign as well as malicious applications for this purpose.

Research suggests that malicious applications generally request for more number of permissions than benign applications [11]. All these permissions are coarse-grained, so users are left with no other options than to accept them. Until Android Marshmallow (API 23), users had to accept all the permission requests of the application at the time of installation [111]. After crossing this checkpoint, the malicious application can gain root access to the system by exploiting kernel-level vulnerabilities and can initiate privilege escalation attacks [81].

Cloak and Dagger attack is a case of permission system exploitation of the Android system. It is the result of design vulnerabilities in Android. This attack needs two permissions, SYSTEM ALERT WINDOW, and BIND ACCESSIBILITY SERVICE for getting complete control over the User Interface (UI) Feedback Loop. The system grants these permissions to the malicious application without taking approval from the user. With the help of these permissions, the application takes control over the visuals of the phone and carries on malicious activities without the user even noticing them. Researchers [112] have uncovered various loopholes in the Android system design that, if exploited, can be fatal for the system.

3.2 Lack of Awareness Regarding Security Protocols and Poor Developer Practices

Android developers, while developing the application, do not give much emphasis on secure communication of data. Android SDK provides packages for the Internet connection that can support Internet connection via both HTTPS and HTTP. Developers' lack of coding skills and partial understanding of SSL can lead to a compromise, with the security of the application. Using SSL certificates, which are signed by the Android trusted Certificate Authority, and Android's application interface that is in-built, helps in securing the data from intruders. Because of the high cost involved in debugging, Android applications using SSL certificates, developers prefer using development servers consisting of unauthorized certificates for debugging. Not validating or verifying the certificates and host Names is also a part of poor developing practices [113]. HTTPS and the TLS/SSL are a set of standardized protocol that the applications must use for the exchange of data. Using HTTP instead of HTTPS for URL connection and WiFi, and Bluetooth for Internet connection, further adds to the problem of information leakage. It encourages Man-In-The-Middle attack, where the attacker tries to sniff the data between the two parties, by pretending to be one of the trusted host. The developers should use TLS Public Key Infrastructure for authenticating the two parties in communication; however, its complex architecture demotivates them from using it [114]. Rather than sending data over HTTPS and in plaintext format, proper use of SSL helps in sending the data in a ciphertext format, protecting it from the attackers. Bouncy Castle Crypto APIs [115] and OpenSSL [85] are open source platform that supports cryptographic libraries and provides the toolkit for SSL/TLS protocol. Users are generally unaware of the consequences of using HTTP and of the importance of proper usage of security protocols in the application.

Android developers are continuously working in the direction of making the Android system more secure. Recently, to make Android Oreo more secure, TLS version fallback is eliminated from HttpsURLConnection. This feature was used in previous versions to support HTTPS stack for connecting to the servers in which TLS protocol version implementation was not proper. In case, TLS handshake failed to connect, then HttpsURLConnection used to disable the newer version of TLS protocol, from retrying the handshake. Disabling the TLS protocol results in downgrading of protection. It makes the system vulnerable to attacks that is why in Android Oreo, TLS version fallback is removed to avoid such reattempts to establish connection [116].

Lack of documentation by developers can be misleading for the users and for other developers who are not directly involved in coding but with different modules of application development. Developers' lack of knowledge regarding security protocols can prove fatal for the Android system. Error in the application coding does not mean it is malicious, but it can serve as a gateway for malware developers. If the developer does not write SSL/TLS code correctly or does not follow proper steps for establishing the secured connection, then it can lead to cryptographic attacks.

3.3 Non-Verification of SSL Certificate

For establishing a secure connection via SSL, the first and foremost step is to authenticate the server certified by a trusted Certificate Authority. It is essential to check the validity of the server's authentication, and sometimes a certificate is revoked by the CA in the case where a server has been compromised or is not behaving as per the norm. In Reference [117], rules for developing and validating certificates are defined based on the X.509 protocol. This document also explains how to check whether the CA has revoked the certificate. The status of the certificate can be checked using the Online Certificate Status Protocol (OCSP) [118]. OSCP provides up-to-date information regarding the status of the certificates as compared to the Certificate Revocation List (CRL) [117]. In Reference [119], researchers found that many critical applications and libraries were not verifying the SSL certificate at all. A list of software found violating the SSL certificate validation included Amazon's EC2 Java library and all cloud clients based on it: Amazon's and PayPal's merchant SDKs, integrated shopping carts, and AdMob code. For example, AdMob provided code that used data-transport library cURL for connecting SSL to AdMob's server, however, the code would turn off the SSL certificate validation. In this article, the author has also listed the "dos" and "don'ts" for the application and SSL library developers. Further, while analyzing 137 of 200 applications [113], which used SSL, it was found that 84 applications used SSL incorrectly. Of 137 applications, in 58 applications, TrustManager used Trust All strategy, and in 13 applications, HostnameVerifier used the Allow All strategy. The remaining 13 applications used both the strategies (Trust All and Allow All). SSL code uses TrustManger and HostnameVerifier classes for verifying and validating server. If the application uses Trust All with Trust-Manager, then it will not require verification of the server. Similarly, if it uses Allow All with the HostnameVerifier approach, then it will not require validation of server.

Vulnerable cryptographic keys and digital certificates were found to be the root cause of such attacks. SSL Blacklisting and Pinning are two methods offered by Android for protecting the system from the threat caused by breached CAs (e.g., Comodo, DigiNotar) and from the use of certificates issued deceitfully. SSL Blacklisting allows blacklisting of certificates and CAs as well. SSL Pinning restricts the number of CAs to be trusted [120].

3.4 Android Application Security Vulnerabilities

Google has found security flaws in more than 275,000 applications found on the Google Play Store [121]. The App Security Improvement Program notifies the developer quickly whenever it finds a vulnerability. This program started with scanning the Google apps for embedded Amazon Web Services (AWS) credentials and Keystore file. The revelation of credentials and keys were posing grave threats to users' private information and for data transfer at that time. In reference [122], the authors have provided a list of the latest security loopholes flagged to developers on Google Play along with the respective vulnerability and solution.

This section has highlighted the common factors that are responsible for the proliferation of attacks that were discussed in Section 2. The next section discusses defensive tools and techniques used to defend the Android system from various attacks and malicious activities.

4 ELABORATION OF DEFENSIVE MECHANISMS

This sections will elaborate various defensive mechanism and tools. There are two main approaches for detection and prevention of malicious activity or application. The first mechanism is static, in which the tool can detect the malware or malicious activity before the execution of an application. The second mechanism is dynamic, which detects the malicious activity at runtime. These following subsections will elaborate defensive mechanisms.

4.1 Tools for Preventing Privileges Escalation Attacks

eXtended Monitoring on Android (XManDroid) [19] prevents privilege escalation attacks dynamically based on the system policy already defined in the system database. For this purpose, researchers have designed the Policy Check Algorithm, which is embedded in the Decision Maker component of the XMandroid framework. This algorithm keeps track of Inter-component Communication (ICC) among the applications. With the help of policies defined in the policy database that are the security rules, it decides whether it should allow the action or reject it. User confirmation is also taken into account whether to allow ICC call. XMandroid helps in preventing an ICC-based privilege escalation attack. A sample of 50 applications was taken in which XMandroid dynamically observed 11,970 ICC calls that occurred during runtime. The total number of cache hits = 11,592 and cache miss = 378. According to the researchers, the tool performed consitently in cases of ICC between applications as well as in case of ICC with content and service providers.

The Extending Android Permission Model (Apex) [123] helps users grant permission to applications for accessing resources during runtime selectively. For this purpose, an application installer named Poly is implemented, which is an extension to the already-existing installer in the system. It provides users with an easy interface to imply constraints on access to resources requested by the applications at installation time.

AdSplit (separating smartphone advertising from applications) [124] separates the advertising libraries from the host application, both run in a completely separate environment. Unique UID and separate permissions are given to applications and the advertising libraries by the UNIX process so that applications need not request for additional privileges for the advertising libraries. Then, for coordination between two processes, an advertising service is there. It keeps track of user actions, performed at UI and sends it to related advertising activity. With the help of a centralized advertising service component, AdSplit can identify any fake UI event, and this component is also responsible for proper display of advertisements. This framework combines two security modes HTML and smartphone applications, which is the current need for smartphone security, as most of the applications and libraries use Java and HTML script.

Like AdSplit, AdDroid (A Privilege Separated Advertising Framework) [125] also wholly separates the advertising component from the parent application by integrating advertising component into the Android framework itself rather than the parent application. It helps in restricting the advertisement component from gaining access to any security-critical information, even if the parent application has the privilege of accessing the same information. The AdDroid framework achieves this goal by introducing the advertising API (AdDroid API), which is an extension of the Android API, it is responsible for data and application. AdDroid also includes two new permissions, ADVERTISING, and LOCATION_ADVERTISING. For using AdDroid, applications need to request for any of these two permissions. These permissions give requesting application access to the advertising API calls; rest AdDroid handles all the decisions regarding advertising and events related to user interactions. Of 473 advertising libraries of corresponding applications, the functionality of 456 advertising libraries was satisfied by the AdDroid permission system. Thus, a 96.4% satisfactory rate was achieved.

Component-Level Access Control (Compac) puts a restriction on the privileges, given to the third-party elements at component level. It achieves this by letting the system decide that request from third-party elements for accessing the private information need to be accepted and which not. The system can make the decision based on the information excavated from the application's components, using runtime information of Java package. Compac gives privilege to the users and system to assign a subset of permission to the application component rather than giving similar access rights to all the components of the application. This approach helps in enhancing the

Table 6. Analysis of Tools Preventing Privilege Escalation Attacks

Tool/ year	Goal	Application and library isolated	User confirmation in privilege selection	Threats addressed	Limitation
Apex [123] 2010	Privilege selection and fine-grained access control	No	Yes	• Prevents leakage of private information from system resources by applications	Permission check can be applied to limited extent by Apex as it does not have access to all the components of the applications.
XMandroid[19] 2011	Fine-grained access control	No	Yes	• Detects ICC- based escalation attacks • Collision attacks	Taking confirmation from users may be risky sometimes, because user's lack of knowledge regarding security rules can result in bad decisions.
AdSplit [124] 2012	Privilege selection	Yes	No	• Protects against click fraud	Applications are still able to send private information to the advertisements running as uniquely identified individual processes.
AdDroid [125] 2012	Privilege selection	Yes	No	• Grayware • Benign-but-buggy applications • Vulnerable advertising networks	Lack of flexibility than other existing models.
Compac [126] 2014	Privilege selection	No	Yes	• Implicit Invocation Attack • Inter-component Code Injection Attacks	Difficult to identify Package Forgery attack (Java package name can be forged that can result in bypassing the restrictions on the package.).
PEDAL [100] 2015	Privilege selection	No	Yes	• Identifies obfuscated code in ad libraries.	Efficiency of separator is reduced if public API code is obfuscated also fails to classify ad libraries in case developers merge self created ad libraries with app logic.

security of the Android model, in context of the attacks, caused due to third-party libraries or components. The approach achieved 97.4% success rate in performing privilege escalation (as per Antutu benchmark). The researchers considered a dataset of 18,566 applications in which 16,000 was benign data and remaining 2,566 was malware [126].

Unlike AdDroid, in Privilege De-escalation (PEDAL) [100] researchers have included a module, called separator that allows users to assign privileges to ad libraries selectively. This module forbids ad library from inheriting permissions, which are granted to their parent applications by the system at the time of installation. PEDAL achieved 98% accuracy rate in detecting ad libraries. Further analysis of these tools is given in Table 6.

A number of tools and techniques has been developed so far by researchers to minimize the unnecessary permissions requested by the applications and their respective ad libraries. These tools can identify and protect the system from different types of privilege escalation attacks. Based on the user experience and attacks, these tools decide which permission request should be entertained and which one should be denied. These tools serve as an extension to the already existing security model of Android. Android Security Model comprises of security features like Application Sandbox, Cryptography, and IPC [127]. Even then attackers are continually working to explore

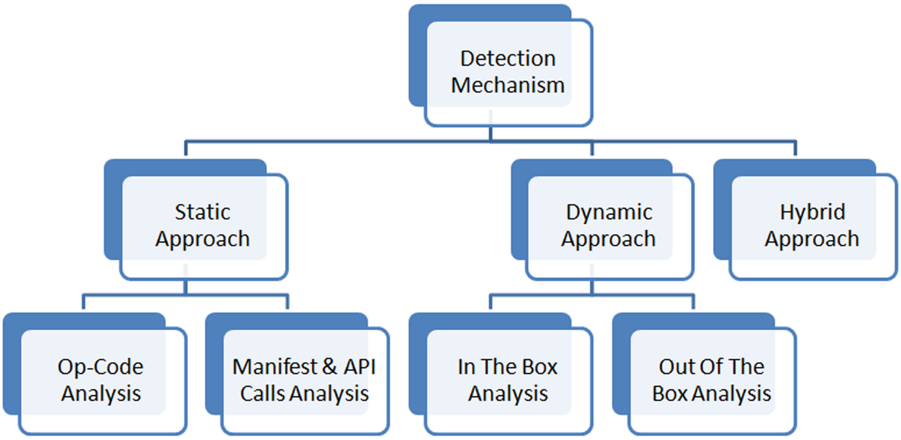

Fig. 3. Detection approaches.

the vulnerability of these defensive mechanisms against privilege escalation attacks. For example, app-in-the-middle attacks can bypass Android Sandbox [128]; such attacks pose a serious threat to the user's private content.

Completely blocking the ad libraries will not give an appropriate solution as this is a source of revenue for advertising industry. So it is necessary to extend the Android security model, and this can be achieved by using these defensive tools and increasing permission constraints. Application developers also need to minimize the number of permission requests they include in the application they develop. It will help in reducing the ambiguity caused due to the large number of permissions requested at the installation time, which are mostly tricky for the users to understand. The naive users just want to use the applications. So they accept all the permissions, without giving it much thought. It gives attackers and analysts the opportunity to peek into the system, which can prove fatal for the system.

4.2 Static and Dynamic Approaches to Analyze and Detect Attacks

The malware detection tools use static and dynamic detection mechanisms for identifying attacks and blocking them from penetrating into the system. Static detection is a passive approach where the tool extracts features from the application file without executing the application. This methodology is resource and time efficient, as the application is not required to be executed. A dynamic approach is an active approach; it identifies attacks in the real environment. Unlike a static approach, it is not easy to evade dynamic detection tool with the help of code obfuscation attack. A static approach monitors features like permissions, API calls, .dex files for opcodes, and metadata. The dynamic approach analyzes features, while the application is running; it includes network traffic, battery usage, CPU utilization, IP addresses opcodes. Most of the dynamic and static detection mechanisms incorporate machine-learning techniques for fine results. Once the model is trained, using machine-learning Algorithm, it can work automatically in the detection of attacks and human intervention is not required. It helps in improving the detection of attacks that are otherwise not detectable during manual analysis. Moreover, data mining helps in identifying the class of attacks that are previously unknown, with the help of ample experience. This technique helps in proper resource utilization. Figure 3 provides a classification of these approaches.

There are broadly two categories of static detection: (i) Op-Code analysis and (ii) Manifest and API Calls analysis. In Op-Code analysis or N-Gram Opcode analysis [132], the detection tool extracts N-Gram Op-Codes from the dataset of malware and benign applications. The dataset is used to classify application using Pattern Recognition and machine-learning techniques. One of the most widely used classifier for this purpose is Support Vector machine. In Manifest and API Calls analysis [131], the tools use machine learning to learn features such as permissions, intents, component deployment from the Manifest file, and API Calls details from the dataset of malicious and benign applications, these data are then used to classify the target applications. Dynamic analysis is of two types: (i) in-the-box analysis and (ii) out-of-the-box analysis [11]. In case of in-the-box analysis, data collection and analysis is performed on the same privilege level as the malware. The detection tools use ART for this purpose.

Data collection and analysis can be meddled with by malware, as it shares the same privileges as the analyzer. This approach allows interception of data at the OS level, access to memory architecture, libraries, API, and other methods. However, at the same time, it makes the critical data vulnerable to attacks. In the out-of-the-box analysis approach, analysis is done taking the security of the system as well as that of the analyzer into consideration. To achieve this, a virtual environment is created to deceive the attackers. It allows the analyzing technique to understand the nature of the attack securely. VM-based analyses and emulators are key techniques used for performing out-of-the-box analysis. Attackers have also explored this detection mechanism and have found the methods to evade such emulated environment and use strategies to remain undetected. Such attacks are stealthy and are often difficult to detect. Dynamic analysis mechanism performs better in identifying attacks that use code obfuscation technique as it analyzes the program in runtime environment. Static analysis approach is better equipped to identify previously known attack for which the detection mechanism has been trained. Researchers have also developed a hybrid approach, a combination of the approaches mentioned above for improving the detection mechanism. A malware detection system can be host based or network based. In a host-based system, the detection tool is incorporated in the system itself and can better analyze malicious activities occurring within the system as well as defend the system from outside attacks. A network-based detection system is responsible for monitoring the whole network to which host is connected and has access to complete network as well as to the system. A network-based detection mechanism cannot analyze the malicious activities occurring within the system like at the kernel level. Such details are hidden from the network-based detection mechanism. Moreover, a network-based detection mechanism requires more software and hardware resources as compared to the host-based detection mechanism and is costlier as well. As host-based detection mechanisms are located on the system itself, they are easy targets for attackers. A network-based detection mechanism fails against previously unknown attacks. A network-based detection mechanism has to analyze heavy network traffic, so its performance suffers from the intense workload and leads to a poor detection rate. The main aim of these detection mechanisms is to reduce the false alarms raised during analysis and detection. Researchers have developed a huge variety of analysis and detection tools that are capable of identifying a wide range of attacks. Table 7 summarizes some of these tools. In the table, ML denotes for "machine learning."

4.3 Anti-virus or Anti-malware Tools

In 2012, Google introduced Google-Bouncer [135], an anti-virus system to automatically detect malicious content in applications on the Android platform. Researchers have exposed many security weaknesses present in Google-Bouncer [136, 137]. Google is continuously trying to improve the security features of Google-Bouncer. However, attackers are also continually working in the direction of finding methods to breach this security mechanism. In 2017, malware named Judy [7] was able to bypass the Google-Bouncer security system. The malware itself clicks on ads in the background without knowledge of the user, thereby earning revenue for the hackers. Further,

Table 7. Comparative Analysis of Static, Dynamic, and Hybrid Detection Tools

Tool/ approach	Year/ ML	Data source	Worked with	Attacks/ malware detected	Limitation
TaintDroid [129] Dynamic	2010 No	Applications from Android market	IPC (data flow)	Malicious intents and activities	• Control Flow is not considered • Outburst of tainted information • No tracking of tags on Direct-Buffer objects.
Andromaly [130] Dynamic	2012 Yes	Artificially created malware	Components of Android and application framework	Unwanted Information, Theft-of-service, Information theft and Denial-of-Service (DoS)	Worked on Artificial Dataset and with limited variety
DroidMat [131] Static	2012 Yes	Contagio mobile + Google Play Store	App-specific manifest file + API calls + intents	ADRD, DroidDreamLight, and DroidKungFu	Low Performance in detecting Android malware with just one sample.
Q.Jerome et al. [132] Static	2014 Yes	Static Genome Project+ Google Play Store	Opcode	Goodware	Approach fails against advance bytecode-level obfuscation and Proguard obfusctor used by attackers
DroidSafe [133] Static	2015 No	Applications from three Red-Team Organizations	API calls	Malicious information flows	Java native methods, dynamic class loading, and reflection cannot be handled. Information that is not considered sensitive by the tool can be excavated by attackers.
CuckooDroid [134] Hybrid	2015 Yes	Applications from Android market	Opcodes, CPU usage, number of packets sent over network, number of running process and battery level	Zero-day malware identified with a low false negative rate.	Large number of resources are required

Google-Bouncer is effective only when applications are downloaded from the Google Play Store; problems arise if the user downloads applications from a third-party or unauthorized sources [138]. It exposes the operating system to attacks. So, to protect the the Android system from such attacks, large numbers of anti-virus products for Android (more than 50) are also available in the market. These tools detect and protect the system from malware. The detection mechanism of these anti-virus applications can be broadly classified into four categories [139]:

- API based
- Dataflow based
- Interaction based
- Signature based

Researchers have shown how malware developers can fail these detection mechanisms. These anti-virus products do not provide a robust solution and are easily evadable. For example, a signature-based anti-virus product looks for a particular pattern or signature in the program, and, if found, the program is considered malicious and, otherwise, benign. anti-virus applications, just like any other application, are prone to attacks themselves. For example, the Automatically Bypassing Android Malware Detection System (AVPASS) developed by researchers can evade most anti-virus products, with a minimum detection ratio of 5.8% (3.42/58). AVPASS uses a binary obfuscation method to infer the features and rules of anti-virus running on the targeted system. After learning the rules, it can behave in a way such that the anti-virus tool is not able to detect the malicious application, as they behave like any other normal application. Code obfuscation techniques are used by malware developers to create polymorphic and pligomorphic malware that can easily bypass such anti-virus applications and tools, thus limiting the utilization area of these anti-virus applications. Moreover, many anti-virus tools are freely available and can be downloaded from app stores, which can themselves be malicious programs, and even in case of genuine anti-virus applications, once installed they become an integral part of the system. The attacker can then look for the vulnerabilities in these programs to exploit them and gain access to the system. Zero-day exploits, used by malware developers for attacking the system, are extremely hard to detect by anti-virus applications and other tools developed for malware detection. Much work is required in the direction of making strong anti-virus products that are almost impenetrable for the malicious content to pass through.

5 FUTURE SCOPE AND DISCUSSION

As discussed in this article, the Android platform still has security vulnerabilities that need to be improved. Following are some of the steps that can be taken to provide a more secure Android environment to its users.

1. The application developers should restrict the number of permissions to safeguard the system from privilege escalation attacks. The requested permissions should be specific to the tasks that the application needs to perform. It will help in reducing attacks that use third-party libraries and dd libraries. Also, Google should come out with a more robust permission system to stop its exploitation. In Reference [140], the author listed some of the standard permissions and explained how they are a potential threat for the system. In this case, even a selective privilege technique fails. Hence, only a more robust permission system is the solution that will also require a review of current standard permissions.
2. The developers should mandatorily follow the security protocols like an SSL certificate that helps in securing the data.
3. A secure system also requires vigilant users. The users should follow the directions issued regarding strong passwords, phishing sites, and not sharing personal information like phone banking details. They should also verify whether the permissions requested by an application are related to its job.
4. Further, Google Play should employ a robust static detection mechanism within its framework using pattern recognition, machine learning, and artificial intelligence. Google Play has already started verifying applications using machine learning based on “peer groups,” which automatically groups applications with similar functions into a group and compares their static features such as the permissions these applications request. If some applications request a higher number of permissions or strange permissions from their peer groups, then the tool flags them as malicious. The tool should also employ Op-Code analysis using SVM classifiers to detect malware, as this detection process has provided accuracy of nearly 99% in various research.
5. Android should not allow its users to install applications from any place other than the Google Play Store. The applications installed from other sources cannot be verified, which makes them a potential threat to the system.
6. Though it is possible to detect much malware using pattern matching and machine learning on the Google Play Store, some malware may still go undetected such as malware that uses

code obfuscation or a newly developed attack strategy such as KSMA. However, a dynamic detection approach can help detect these attacks based on anomaly detection and gather information about them that can help in preventing them. The information can subsequently be forwarded to Google, and Google can use the features of new malicious application to further train its classifier, thus making the process automatic. One of the main reasons for not using dynamic detection mechanisms is that it takes a significant amount of resources during runtime, which may slow down the device. However, the processing power of a smartphone is increasing multifold each year; it is not as challenging now to employ a dynamic detection mechanism in each device as it was a few years ago. The latest Snapdragon 845 processor has 25% better processing power than its predecessor, Snapdragon 835, as claimed by Qualcomm and tests conducted by tech reviewer magazine CNET [141]. Developers may also utilize this rapid increase in processing power to develop malware detection tools . However, researchers need to focus on the efficiency and accuracy of these tools.

6 CONCLUSION

The main idea of this article is to present the current state of the Android security domain. Researchers and developers are formulating novel techniques for protecting the Android system from attackers' malevolent inclinations to destroy it. This article explains attacks on different layers of Android in detail along with their present solutions, which will help researchers and software developers formulate their strategies for future malware detection and prevention. The article includes tabular explanations of various threats and detection approaches to provide a better understanding and comparative analysis. Some of the important findings of this article include the necessity of using original hardware products to keep the system secure; the need for a robust two-step malware detection approach; employing both static and dynamic detection strategies, which can detect known as well as unknown attacks; and making it compulsory for developers to follow important security protocols. Unlike other surveys, this article also provides an in-depth analysis of various vulnerabilities in the Android system exploited by attackers. The Android operating system will continue to evolve over time for a number of reasons and so will various applications running on it. There is always a huge gap between the two. This leads to research gaps to which there is no complete solution provided by present tools. Continuous research and innovation is required in this area to deal with the latest threats.

REFERENCES

[1] Mobile threat report. 2014. Pulse Secure Mobile Threat Center (MTC). Retrieved from https://www.pulsesecure.net/lp/mobile-threat-report-2014.

[2] IDC. Smartphone OS market share. 2017. Retrieved from https://www.idc.com/promo/smartphone-market-share/os.

[3] Mobile overview report. 2017. Scientiamobile. Retrieved from https://www.scientiamobile.com/page/wpcontent/uploads/2017/05/MOVR-Q1-2017-Final.pdf.

[4] Trojans, ghosts, and more mean bumps ahead for mobile and connected things. 2017. McAfee. Retrieved from https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2017.pdf.

[5] Annual threat report. 2017. Quick Heal. Retrieved from http://dlupdate.quickheal.com/documents/others/Quick_Heal_Annual_Threat_Report_2017.pdf.

[6] Roman Unuchek. Kaspersky Security Bulletin. Mobile malware evolution 2016. 2017. Retrieved from https://securelist.com/mobile-malware-evolution-2016/77681.

[7] Jason Murdock. Judy' could be the largest malware campaign ever found on google play. 2017. International Business Times. Retrieved from http://www.ibtimes.co.uk/judy-could-be-largest-malware-campaign-ever-found-google-play-store-1623508.

[8] Ericka Chickowsk. Cybercrime: A black market price list from the dark web. 2016. Retrieved from https://www.darkreading.com/cloud/cybercrime-a-black market-price-list-from-the-dark-web/d/d-id/1324895?.

[9] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. 2015. Android security: A survey of issues, malware penetration, and defenses. *IEEE Commun. Surv. Tutor.* 17, 2 (2015), 998–1022.

[10] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda. 2014. Evolution, detection and analysis of malware for smart devices. *IEEE Commun. Surv. Tutor.* 16, 2 (2014), 961–987.

[11] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The evolution of Android malware and Android analysis techniques. *ACM Comput. Surv.* 49, 4, Article 76 (Jan. 2017), 41 pages. DOI: http://dx.doi.org/10.1145/3017427

[12] Omer Shwartz, Amir Cohen, Asaf Shabtai, and Yossi Oren. 2017. Shattered trust: When replacement smartphone components attack. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT'17), Vancouver, BC*. USENIX Association. Retrieved from https://www.usenix.org/conference/woot17/workshop-program/presentation/shwartz.

[13] Dennis Fisher. Rowhammer, Android and the future of hardware attacks. 2018. Retrieved from https://duo.com/decipher/rowhammer-android-and-the-future-of-hardware-attacks.

[14] Swati Khandelwal. GLitch: New 'rowhammer' attack can remotely hijack Android phones. 2018. Retrieved from https://thehackernews.com/2018/05/rowhammer-android-hacking.html.

[15] Adam Conwayl. Every Android device is susceptible to a hardware vulnerability called RAMpage. 2018. Retrieved from https://www.xda-developers.com/android-hardware-vulnerability-rampage/.

[16] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. 2018. GuardION: Practical mitigation of DMA-based rowhammer attacks on ARM. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 92–113.

[17] Swati Khandelwal. QuadRooter: New Android vulnerabilities in over 900 million devices. 2018. Check Point Software Technologies Ltd. Retrieved from https://blog.checkpoint.com/2016/08/07/quadrooter/.

[18] Android Developers Blog. 2018. Retrieved from https://developer.android.com/guide/platform/index.html#hal.

[19] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, F. Thomas Fischer, and Ahmad-Reza Sadeghi. 2011. XManDroid: A new android evolution to mitigate privilege escalation attacks. Technical Report, Center for Advanced Security Research Darmstadt.

[20] Zero-day vulnerability in google Android. 2013. Retrieved from https://www.zero-day.cz/database/253/.

[21] Android malware googligan/ghost push. Cyber Swachhta Kendra. 2017. Retrieved from https://www.cyberswachhtakendra.gov.in/alerts/gooligan.html.

[22] Xuxian Jiang. 2011. Security alert: New sophisticated Android malware droidkungfu found in alternative chinese app markets. NC State University. Retrieved from http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[23] Security-enhanced Linux in Android. 2017. Android Developers Blog. Retrieved from https://source.android.com/security/selinux/.

[24] Android security bulletin, December. 2017. Android Developers Blog. Retrieved from https://source.android.com/security/bulletin/2017-12-01.

[25] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07), New York, NY*. ACM, New York, NY, 552–561. DOI: http://dx.doi.org/10.1145/1315245.1315313

[26] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11), New York, NY*. ACM, New York, NY, 30–40. DOI: http://dx.doi.org/10.1145/1966913.1966919

[27] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11), Berlin, Heidelberg*. Springer-Verlag, Berlin, 121–141.

[28] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. USENIX Association, Berkeley, CA, 39–39. Retrieved from http://dl.acm.org/citation.cfm?id=2362793.2362832.

[29] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, Article 40, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195686

[30] Jonathan Salwan. 2011. ROPgadget—Gadgets finder and auto-roper. ROP. Retrieved from http://shell-storm.org/project/ROPgadget/.

[31] Axel Souchet. 2017. rp++ is a full-cpp written toor. Retrieved from https://github.com/0vercl0k/rp.

[32] Black Hat USA. Payload already inside: Data re-use for ROP exploits. 2010. Retrieved from http://ropshell.com/ropeme/.

[33] PAKT, Patroklos Argyroudis, and Edward J. Schwartz. Ropc—A turing complete rop compiler. 2013. Ropc. Retrieved from https://github.com/pakt/ropc.

[34] Aurelien Wailly, Axel Souchet, Jonathan Salwan, Anthony Verez, and Tiphaine Romand. 2016. Automated return-oriented programming chaining. Retrieved from https://github.com/awailly/nrop.

[35] Vivek Parikh and Prabhaker Mateti. 2017. ASLR and ROP attack mitigations for ARM-based Android devices. In *Security in Computing and Communications*. Springer, 350–363.

[36] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, New York, NY, 559–572. DOI: http://dx.doi.org/10.1145/1866307.1866370

[37] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*. ACM, New York, NY, 40–51. DOI: http://dx.doi.org/10.1145/1966913.1966920

[38] Vasilis Pappas. 2012. kBouncer: Efficient and transparent ROP mitigation. Retrieved from http://www.cs.columbia.edu/ vpappas/papers/kbouncer.pdf.

[39] Z. Huang, T. Zheng, Y. Shi, and A. Li. 2012. A dynamic detection method against ROP and JOP. In *Proceedings of the 2012 International Conference on Systems and Informatics (ICSAI'12)*. 1072–1077.

[40] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM, New York, NY, 272–280. DOI: http://dx.doi.org/10.1145/948109.948146

[41] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*. ACM, New York, NY, 30–40. DOI: http://dx.doi.org/10.1145/1966913.1966919

[42] Mathias Payer and Thomas R. Gross. 2013. String oriented programming: When ASLR is not enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)*. ACM, New York, NY, Article 2, 9 pages. DOI: http://dx.doi.org/10.1145/2430553.2430555

[43] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*. IEEE Computer Society, Washington, DC, 227–242. DOI: http://dx.doi.org/10.1109/SP.2014.22

[44] Erik Bosman and Herbert Bos. 2014. Framing signals - A return to portable shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*. IEEE Computer Society, Washington, DC, 243–258. DOI: http://dx.doi.org/10.1109/SP.2014.23

[45] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (Nov. 2009), 40 pages. DOI: http://dx.doi.org/10.1145/1609956.1609960

[46] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 147–160. http://dl.acm.org/citation.cfm?id=1298455.1298470

[47] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, 191–206. http://dl.acm.org/citation.cfm?id=647253.720293

[48] Thomas Petazzoni. Elixir cross referencer. 2018. Retrieved from https://elixir.bootlin.com/linux/latest/source/arch/x86/include/uapi/asm/sigcontext.h.

[49] Scott Bauer. 2016. SROP Mitigation: Signal cookies -Article –LWN.net. Retrieved from https://lwn.net/Articles/674861/.

[50] Jonathan Corbet. 2011. On vsyscalls and the vDSO -Article –LWN.net. Retrieved from https://lwn.net/Articles/446528/.

[51] Alessandro. 2015. Hack.lu 2015 - Stackstuff 150: Why and how does vsyscall emulation work. Retrieved from https://toh.necst.it/hack.lu/2015/exploitable/StackStuff/#why-and-how-does-vsyscall-emulation-work.

[52] The PaX Team. RAP: RIP ROP. 2015. Retrieved from https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIPROP.pdf.

[53] Intel. Control-flow enforcement technology preview. 2016. Retrieved from https://software.intel.com/sites/default/files/managed/4d/2a/ control-flow-enforcement-technology-preview.pdf.

[54] Mingshen Sun, John C. S. Lui, and Yajin Zhou. 2016. Blender: Self-randomizing address space layout for Android apps. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'16)*.

[55] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. Ret2Dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, 957–972. Retrieved from http://dl.acm.org/citation.cfm?id=2671225.2671286.

[56] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. 2014. From zygote to morula: Fortifying weakened ASLR on Android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*. IEEE Computer Society, Washington, DC, 424–439. Retrieved from https://doi.org/10.1109/SP.2014.34.

[57] Jake Edge. 2013. Kernel address space layout randomization. KSLR. Retrieved from https://lwn.net/Articles/569635/.

[58] Sami Tolvanen. 2017. Hardening the kernel in Android oreo. Android Developers Blog. Retrieved from https://android-developers.googleblog.com/2017/08/hardening-kernel-in-android-oreo.html.

[59] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM, New York, NY, 380–392. DOI: http://dx.doi.org/10.1145/2976749.2978321

[60] Kaer Morhe. 2017. PaX/grsecurity –> KSPP –> AOSP kernel: Linux kernel mitigation checklist(WIP). Hardened GNU/Linux. Retrieved from https://github.com/hardenedlinux/grsecurity-101-tutorials/blob/master/kernel_mitigation.md.

[61] Yong Wang. 2018. KSMA: Breaking Android kernel isolation and Rooting with ARM MMU features. KSMA. Retrieved from https://www.blackhat.com/docs/asia-18/asia-18-WANG-KSMA-Breaking-Android-kernel-isolation-and-Rooting-with-ARM-MMU-features.pdf.

[62] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. BootStomp: On the security of bootloaders in mobile devices. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, Vancouver, BC, 781–798. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/redini

[63] Android Developer Blog. Android Security Bulletin, July 2016. Retrieved from https://source.android.com/security/bulletin/2016-07-01.

[64] Swati Khandelwal. 2015. Mobile bootloaders from top manufacturers found vulnerable to persistent threats. Black Hat USA 2015. Retrieved from https://thehackernews.com/2017/09/hacking-android-bootloader-unlock.html.

[65] Don Marshall. 2018. Driver security checklist. Microsoft. Retrieved from https://docs.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-checklist.

[66] Android Developers Blog. Security updates and resources. 2018. Retrieved from https://source.android.com/security/overview/updates-resources.

[67] NDAY-2017-0105: Elevation of privilege vulnerability in MSM thermal driver. 2017. Retrieved from https://blog.zimperium.com/nday-2017-0105-elevation-of-privilege-vulnerability-in-msm-thermal-driver/.

[68] Android Developer Blog. Android Security Bulletin, July 2018. Retrieved from https://source.android.com/security/bulletin/2018-07-01.

[69] Di Shen. 2015. Exploiting trustzone on Android. Black Hat USA 2015. Retrieved from https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf.

[70] Dan Goodin. 2017. Android devices can be fatally hacked by mali-cious Wi-Fi networks Broadcom chips allow rogue Wi-Fi sig-nals to execute code of attacker's choosing. Ars Technica. Retrieved from https://arstechnica.com/information-technology/2017/04/wide-range-of-android-phones-vulnerable-to-device-hijacks-over-wi-fi/.

[71] Mathy Vanhoef and Frank Piessens. 2017. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, New York, NY, 1313–1328. DOI: http://dx.doi.org/10.1145/3133956.3134027

[72] Hacking. Popular tools for brute-force attacks. 2017. Retrieved from http://resources.infosecinstitute.com/popular-tools-for-brute-force-attacks/.

[73] P. Teufl, A. Fitzek, D. Hein, A. Marsalek, A. Oprisnik, and T. Zefferer. 2014. Android encryption systems. In *Proceedings of the 2014 International Conference on Privacy and Security in Mobile Systems (PRISMS'14)*. 1–8.

[74] Iain Thomson. 2017. Invisible man' malware runs keylogger on your Android banking apps. Retrieved from https://www.theregister.co.uk/2017/08/02/banking_android_malware_in_uk.

[75] Giuseppe Petracca, Yuqiong Sun, Ahmad Atamli, and Trent Jaeger. 2016. AuDroid: Preventing attacks on audio channels in mobile devices. *CoRR* abs/1604.00320 (2016). arxiv:1604.00320

[76] Norm Hardy. 1988. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (Oct. 1988), 36–38. http://dl.acm.org/citation.cfm?id=54289.871709

[77] Android Developers Blog. Testing your app's accessibility. 2017. Retrieved from https://developer.android.com/training/accessibility/testing.html.

[78] Zhi Xu and Sencun Zhu. 2015. SemaDroid: A privacy-aware sensor management framework for smartphones. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*. ACM, New York, NY, 61–72.

[79] Ahmed Al-Haiqi, Mahamod Ismail, and Rosdiadee Nordin. 2014. A new sensors-based covert channel on Android. *The Scientific World Journal* 2014 (2014).

[80] Jeff Vander Stoep. 2017. Shut the HAL Up. Android Developers Blog. Retrieved from https://android-developers.googleblog.com/2017/07/shut-hal-up.html.

[81] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. 2014. A taxonomy of privilege escalation attacks in Android applications. *Int. J. Secur. Netw.* 9, 1 (Feb. 2014), 40–55. DOI: http://dx.doi.org/10.1504/IJSN.2014.059327

[82] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. 2015. Leave me alone: App-level protection against runtime information gathering on Android. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. 915–930.

[83] U. Fiore, F. Palmieri, A. Castiglione, V. Loia, and A. De Santis. 2014. Multimedia-based battery drain attacks for Android devices. In *Proceedings of the 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC'14)*. 145–150.

[84] Android Developers Blog. Android security bulletin, October. 2017. Retrieved from https://source.android.com/security/bulletin/2017-10-01.

[85] OpenSSL Software Foundation. The legion of the bouncy castle. 2017. Retrieved from https://www.openssl.org.

[86] Stan Wisseman. 2017. Third-party libraries are one of the most insecure parts of an application. Techbeacon. Retrieved from https://techbeacon.com/third-party-libraries-are-one-most-insecure-parts-application.

[87] A. Narayanan, L. Chen, and C. K. Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *Proceedings of the 2014 IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP'14)*. 1–6.

[88] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17), Dallas, TX*. 2187–2200.

[89] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A scalable and accurate two-phase approach to Android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. ACM, New York, NY, 71–82. DOI: http://dx.doi.org/10.1145/2771783.2771795

[90] Z. Ma, H. Wang, Y. Guo, and X. Chen. 2016. LibRadar: Fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C'16)*. 653–656.

[91] C. Soh, H. B. K. Tan, Y. L. Arnatovich, A. Narayanan, and L. Wang. 2016. LibSift: Automated detection of third-party libraries in Android applications. In *Proceedings of the 2016 23rd Asia-Pacific Software Engineering Conference (APSEC'16)*. 41–48.

[92] LibScout: Third-party library detector for Java/Android apps. 2017. Github, Bibtex entries: [bib-ccs16] [bib-ccs17]. Retrieved from https://github.com/reddr/LibScout.

[93] Google ASI. Vungle support, security vulnerability in Android sdks prior to 3.3.0. 2016. Retrieved from https://support.google.com/faqs/answer/6313713.

[94] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the Network and Distributed System Security Symposium*.

[95] Mohit Kumar. 2014. The hacker news. Facebook sdk vulnerability puts millions of smartphone users' accounts at risk. Retrieved from http://thehackernews.com/2014/07/facebook-sdkvulnerability-puts.html.

[96] Devdatta Akhawe. 2015. Security bug resolved in the dropbox sdks for Android. Dropbox Blog. Retrieved from https://blogs.dropbox.com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android/.

[97] Song Luan Kevin Ku Mike Ville-na Bharadwaj Ramachandran Richmond Wong Janne Lindqvist Norman Sadeh Jialiu Lin, Shahriyar Amini, and Joy Zhang. 2014. Privacygrade: Grading the privacy of smartphone apps. CMU CHIMPS Lab.

[98] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, 527–542. Retrieved from http://dl.acm.org/citation.cfm?id=2534766.2534812.

[99] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM, New York, NY, 356–367. DOI: http://dx.doi.org/10.1145/2976749.2978333

[100] Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*. ACM, New York, NY, 89–103. DOI: http://dx.doi.org/10.1145/2742647.2742668

[101] V3 Newsdesk. Shared library security flaw enables Android apps to access personal information without the right permissions. 2017. Retrieved from https://www.v3.co.uk/v3-uk/news/3015611/shared-library-security-flaw-enables-android-apps-to-access-personal-information-without-the-right-permissions.

[102] Vincent F. Taylor, Alastair Beresford, and Ivan Martinovic. 2017. Intra-library collusion: A potential privacy nightmare on smartphones. CoRR abs/1708.03520.

[103] How bad is dirty cow? 2016. Linux Foundation. Retrieved from https://www.linuxfoundation.org/blog/how-bad-is-dirty-cow/.

[104] Redhat customer portal. Dirtycow-cve-2016-5195, 2017. Kernel Local Privilege Escalation. Retrieved from https://access.redhat.com/security/vulnerabilities/DirtyCow.

[105] Roman Schlegel, Kehuan Zhang, Xiao yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. 2011. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*. The Internet Society. http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#SchlegelZZIKW11

[106] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. 2012. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 224–238.

[107] Android Developers Blog. System and kernel security. 2017. Retrieved from https://source.android.com/security/overview/kernel-security.

[108] Pooja Singh, Pankaj Tiwari, and Santosh Singh. 2016. Analysis of malicious behavior of Android apps. *Proc. Comput. Sci.* 79 (2016), 215–220. DOI: http://dx.doi.org/10.1016/j.procs.2016.03.028

[109] Android Developers Blog. Manifest.permission. 2017. Retrieved from https://developer.android.com/reference/android/Manifest.permission.html.

[110] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS'12)*. ACM, New York, NY, Article 3, 14 pages. DOI: http://dx.doi.org/10.1145/2335356.2335360

[111] Mu Zhang and Heng Yin. 2016. *Android Application Security— Semantics and Context-Aware Approach*. Springer. DOI: http://dx.doi.org/10.1007/978-3-319-47812-8

[112] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. 2017. Cloak and dagger: From two permissions to complete control of the UI Feedback loop. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP'17)*. 1041–1057.

[113] Vasant Tendulkar and William Enck. 2014. An application package configuration approach to mitigating Android SSL vulnerabilities. *CoRR* abs/1410.7745 (2014). arxiv:1410.7745 http://arxiv.org/abs/1410.7745

[114] Xuetao Wei and Michael Wolf. 2017. A survey on HTTPS implementation by Android apps: Issues and countermeasures. *Appl. Comput. Inf.* 13, 2 (2017), 101–117. Retrieved from http://www.sciencedirect.com/science/article/pii/S2210832716300722.

[115] The legion of the bouncy castle. 2017. Tau Ceti Co-operative Ltd.

[116] Tobias Thierer. 2017. Android O to drop insecure TLS version fallback in HttpsURLConnection. Android Developers Blog. Retrieved from https://android-developers.googleblog.com/2017/04/android-o-to-drop-insecure-tls-version.html.

[117] D. Cooper, S. Santesson, S. Farrell Trinity, S. Boeyen, R. Housley, and W. Polk. 2008. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Network Working Group. Retrieved from https://tools.ietf.org/html/rfc5280.

[118] A. Malpani S. Galperin M. Myers, R. Ankney and C. Adams. 1999. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol—OCSP. Network Working Group. Retrieved from https://tools.ietf.org/html/rfc2560.

[119] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 38–49. DOI: http://dx.doi.org/10.1145/2382196.2382204

[120] Android Developer Blog. Security with HTTPS and SSL. 2017. Retrieved from https://developer.android.com/training/articles/security-ssl.html.

[121] Lucian Constantin. 2017. Google pushed developers to fix security flaws in 275,000 Android apps. IDG News Service. Retrieved from http://www.pcworld.com/article/3159972/security/google-pushed-developers-to-fix-security-flaws-in-275000-android-apps.html.

[122] Android Developer Blog. App security improvement program. 2017. Retrieved from https://developer.android.com/google/play/asi.html.

[123] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*. ACM, New York, NY, 328–332. DOI: http://dx.doi.org/10.1145/1755688.1755732

[124] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating smartphone advertising from applications. *CoRR* abs/1202.4030 (2012). arxiv:1202.4030 http://arxiv.org/abs/1202.4030

[125] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege separation for applications and advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*. ACM, New York, NY, 71–72. DOI: http://dx.doi.org/10.1145/2414456.2414498

[126] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. 2014. Compac: Enforce component-level access control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY'14)*. ACM, New York, NY, 25–36. DOI: http://dx.doi.org/10.1145/2557547.2557560

[127] Security Tips 2018. (2018). Android Developers Blog. Retrieved from https://developer.android.com/training/articles/security-tips.html.

[128] Ionut Arghire. 2017. App-in-the-middle attacks bypass Android sandbox: Skycure. Information Security News. Retrieved from http://www.securityweek.com/app-middle-attacks-bypass-android-sandbox-skycure.

[129] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[130] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": A behavioral malware detection framework for Android devices. *J. Intell. Inf. Syst.* 38, 1 (Feb. 2012), 161–190. DOI: http://dx.doi.org/10.1007/s10844-010-0148-x

[131] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu. 2012. DroidMat: Android malware detection through manifest and API calls tracing. In *Proceedings of the 2012 7th Asia Joint Conference on Information Security*. 62–69.

[132] Q. Jerome, K. Allix, R. State, and T. Engel. 2014. Using opcode-sequences to detect malicious Android applications. In *Proceedings of the 2014 IEEE International Conference on Communications (ICC'14)*. 914–919.

[133] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-flow analysis of Android applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.

[134] Xiaolei Wang, Yuexiang Yang, Yingzhi Zeng, Chuan Tang, Jiangyong Shi, and Kele Xu. 2015. A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection. In *Proceedings of the 6th International Workshop on Mobile Cloud Computing and Services (MCS'15)*. ACM, New York, NY, 15–22. DOI: http://dx.doi.org/10.1145/2802130.2802132

[135] Chloe Albanesius. 2017. Google 'bouncer' now scanning Android market for malware. PC Magazine.

[136] Ellen Messmer. 2012. Black hat demo: Google bouncer malware detection can be beaten. Retrieved from https://www.infoworld.com/article/2617648/application-security/black-hat-demo--google-bouncer-malware-detection-can-be-beaten.html.

[137] Mohit Kumar. 2012. Researchers bypass Google bouncer Android security. Retrieved from https://thehackernews.com/2012/06/researchers-bypass-google-bouncer.html.

[138] Marie Black. 2018. Do you need antivirus on Android? Retrieved from https://www.techadvisor.co.uk/how-to/google-android/do-you-need-antivirus-on-android-3668607/.

[139] Jinho Jung, Chanil Jeon, Max Wolotsky, Insu Yun, and Taesoo Kim. 2017. AVPASS: Automatically bypassing Android malware detection system. Black hat events USA. Retrieved from https://www.blackhat.com/docs/us-17/thursday/us-17-Jung-AVPASS-Leaking-And-Bypassing-Anitvirus-Detection-Model-Automatically.pdf.

[140] Deepak Abbot. 2017. Android users, beware! Apps do not need your permission to violate your privacy. Inc42 Blog. Retrieved from https://inc42.com/resources/android-users-beware-apps-privacy.

[141] CNET. Snapdragon 845 speed comparison. 2018. Retrieved from https://www.cnet.com/news/snapdragon-845-speed-versus-galaxy-note-8-pixel-2-xl/.

Received March 2018; revised October 2018; accepted November 2018