DroidBarrier: Know What is Executing on Your Android

ABSTRACT

Many Android vulnerabilities share a root cause of malicious unauthorized applications executing without user's consent. In this paper, we propose the use of a technique called process authentication for Android applications to overcome the shortcomings of current Android security practices. We demonstrate the process authentication model for Android by designing and implementing our runtime authentication and detection system referred to as DroidBarrier. Our malware analysis shows that DroidBarrier is capable of detecting real Android malware at the time of creating independent processes. A performance evaluation of DroidBarrier unveils its low performance penalties.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms

Security

Keywords

Android system security, authentication, malicious processes, Android malware

1. INTRODUCTION

An important feature of the Android operating system is that it relies on mature technologies such as the Linux kernel. In particular, Android's Dalvik runtime system relies on Linux process creation when launching an application or a service, making the runtime system as the parent process of all user application processes in Android.

With the assistance of the Linux kernel, Android implements a fundamental security feature called application sandboxes. Android's approach is to install each application with an isolated sandbox to protect its data from unauthorized accesses by other applications, by means of file system permissions and creating independent processes for each application at runtime. Similar to other Linux-base systems, Android suffers from numerous vulnerabilities that cause the Android's application sandboxes to fail. For example, the `Gingerbreak` exploit affected the popular Android 2.3 enabled a malicious application to gain `root` privileges and completely bypass Android's application sandboxes. This privilege escalation is used to establish attacks on various system and user resources. Malicious applications can then exploit other vulnerabilities above the Linux kernel level such as the ones in Android's inter-component communications [7, 15]. A malicious application may launch attacks to misuse system resources with the goal of spying on users, stealing private user data, and causing financial loss [16].

To combat the security vulnerabilities in Android, current state of the art focuses on a wide range of approaches. There have been proposals for using virtual layers to separate execution domains [3], using control-flow to limit access to data [13], and protecting inter-component communications in Android's Binder [11]. Despite Android's heavy reliance on its Linux-based security sandboxes, with a few exceptions (e.g., [25]), existing security solutions barely attempt to enhance the security capabilities of the Linux kernel in Android. Prior work specifically targeting traditional Linux-based systems (e.g., [2, 20, 29]) are not directly applicable to Android. This limitation of security solutions for Linux-based systems is because of major differences in the system architecture, application and security models used in Android, despite the reliance on a modified Linux kernel.

In this paper, we present a technique, referred to as *process authentication for Android applications*, that complements the Android's sandbox mechanism and provides strong protection for system resources against malicious and unauthorized applications. The concept of authentication has been previously applied and used by (i) Quire [11] for providing provenance proof on inter-component communications in Android, (ii) A2 [2] for authenticating native Linux applications, and (iii) the authenticated system calls [27] for preventing misuse of system calls. However, developing an authentication model for Android applications imposes unique technical challenges since the Android framework primarily runs Java applications running in virtual machines and makes use of features such as application services that run in background. These and other Android features require rethinking application and process authentication, which is the subject of our work.

Our key observation is that critical vulnerabilities in Android share a root cause: *malicious applications that are in-*

stalled and executed without the user's consent. Our process authentication model addresses the problem by regarding application processes as individuals that must be authenticated before using system resources. In this model, legitimate applications are given credentials that are used for authentication at runtime. When enforcing process authentication, unauthorized processes that do not possess credentials fail to authenticate. This failure results in denying access to critical system services provided by the kernel.

The main property of our process authentication model is enabling the detection of unauthorized processes at runtime. To demonstrate this property, we design and implement DroidBarrier, a runtime system, that enforces a mandatory authentication on all processes for any application in Android. Using this mandatory authentication enforcement, DroidBarrier *guarantees* the detection of processes that fail to authenticate and prevents their subsequent attacks. Our technical contributions are summarized below.

1. *Process authentication model.* We present a process authentication model and discuss the security requirements and guarantees of our model. We discuss a general set of operations needed to implement our model at runtime.

2. *Runtime system.* We design a runtime system called DroidBarrier that is capable of detecting unauthorized processes. Our runtime system mediates the authentication between a process and the kernel. Our design does not require modification of existing applications or Android's Dalvik runtime system.

3. *Implementation.* We implemented and tested DroidBarrier for a physical Android device. Our implementation consists of patches to the kernel, a set of tools for process monitoring, authentication, and a lightweight access control system in the kernel.

The fundamental difference between our approach and related work such as [1, 2, 3, 11] is that our process authentication for Android is a general and scalable model specifically for authenticating any Android native and Dalvik application processes, taking into the consideration that applications might comprise of several heterogeneous processes for UI activities, background services, or native tasks. We achieve this generality by using the process abstraction to separate the identities of processes using uniquely issued secure credentials that are managed by DroidBarrier.

Per our evaluation, DroidBarrier detects and stops malicious processes of three major Android malware categories with hundreds of instances. According to our performance experiments on a physical Android tablet, DroidBarrier has negligible performance penalties in process creation. It also shows a maximum and a minimum I/O performance penalty of 12.92% and 3.76%, respectively.

The rest of this paper is organized as follows. In Section 2, we present a motivating example, followed by our security analysis and the description of the process authentication model. Section 3 describes the details of DroidBarrier and its functions. We describe the implementation of DroidBarrier and our experiments in Sections 4 and 5, respectively. Finally, we discuss the related work in Section 6 and conclude in Section 7.

2. MODEL AND OVERVIEW

In this section we present the problem and an overview of the solution. We also discuss our security model and present the process authentication model.

2.1 Problem Statement and Overview of the Solution

We address the problem of *preventing stealthy installation and execution of malicious applications in Android-enabled devices.* This is a root cause of many malicious attacks that exploit other vulnerabilities in Android, with attack goals such as stealing private data and incurring financial loss.

Motivating example attack. We conducted a runtime analysis of three sets of Android malware `DroidKungFu`, `BaseBridge`, and `AnserverBot`. Each of these malware sets have a number of variants with nearly identical malicious activities with differences in their user interface elements. According to our findings, and the results in [33] and [34], these malware sets rely on malicious shell scripts (running in independent processes) to perform a privilege escalation exploit. In Section 5.1, we discuss that our runtime system, DroidBarrier, is capable of detecting and stopping these applications at the time of creating processes that fail to authenticate.

To demonstrate the problem, we construct an example attack scenario that is common to these (and other) malware categories. As depicted in Figure 2.1, the remote attack exploits two vulnerabilities that result in execution of malicious applications without the user's permission. In the first phase of the attack, a vulnerable client is exploited to execute a payload that downloads a native malicious executable. In the second phase, the downloaded native binary exploits a vulnerability in a system task (e.g., Android debugging bridge daemon) to gain `root` privileges. Having `root` privileges, the native binary bypasses Android's sandbox and installs applications on the file system without asking for the user's permission.

Once the malicious application is installed with all the requested permissions defined in its manifest file, it can conduct further attacks. The key element to this attack strategy is the ability to install applications without the user's explicit authorization. DroidBarrier does not directly detect the payload that is downloaded to install malicious applications. However, DroidBarrier is designed to prevent such installations by means of detecting their unauthenticated processes, thereby foiling this form of attack.

Challenges and Goals. To develop a solution for protecting the system from execution of unauthorized malicious applications, we face two technical challenges:

- The kernel only enforces file system permissions, and provides memory isolation for processes. In Android, the kernel lacks advanced capabilities to detect possible misuse of `root` privileges.

- Because Android application processes execute in virtual machines and are managed by the Android runtime, monitoring application processes inside the kernel, without modifying the runtime, faces a semantic gap.

To address these challenges and develop a mechanism that can detect installation and execution of unauthorized applications, this work has the following goals:

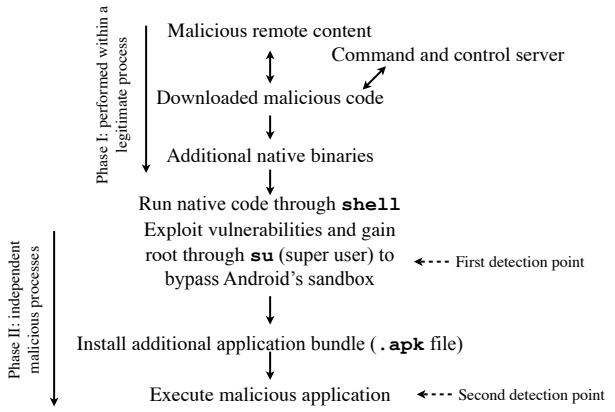

Figure 2.1: A remote attack sequence to bypass Android's sandbox and install malicious applications without user's permissions. In the first and second detection points, the application creates processes that can be detected by DroidBarrier.

- Providing a mechanism for the kernel to authenticate processes and unveil the existence of malicious applications.
- Detecting malicious processes at runtime with high confidence about their origins. In our detection, we reconstruct the semantics between the kernel and Android's runtime by monitoring process forks and the runtime's subsequent loading of application class files into the newly created processes.

In our process authentication model, the kernel enforces a mandatory authentication on every process. The user designates, in advance, which applications are allowed to run, and thus, has full control on the execution. Our runtime system, DroidBarrier (Section 3.2), implements our process authentication model. DroidBarrier continuously monitors low level process creation functions in the kernel to effectively enforce the mandatory authentication policy.

2.2 Security Model

Security assumptions and trust model. We trust the kernel's code and the isolation of memory provided by the kernel used in Android. We assume that the integrity and confidentiality of kernel's memory are preserved. Further, we assume that Android's system software and processes do not intentionally contain malicious functionality.

Attack model. We target the following methods of attacks. We first categorize the attacks according to their installation approach.

- **Remote attacks.** As shown in the example attack in Figure 2.1, remote exploitation attacks start by remotely exploiting a vulnerability in an application. For example, attackers can exploit the many vulnerabilities in Android's WebView API (that applications use to show specific web pages) to trigger drive-by-downloads [23] and install malicious applications. These attacks can download the application and use system vulnerabilities to bypass user's permission for installing the application.
- **Physical attacks.** Using a physical communication channel such as Android Debug Bridge (adb) daemon to install malicious applications [32]. In this case, the attacker has physical access to the device and the attack goal is to install malicious applications without the user's knowledge. These attacks either require that the device is not password protected or existence of a system vulnerability to bypass the password protection.

According to their execution dependencies attacks are further categorized into two main classes:

- **Dependent attacks**. The malicious code runs in a compromised application's processes. Thus, the malicious code depends on another legitimate application to continue execution.
- **Independent attacks**. The malicious code needs to run in at least one independent process that is created by a native code or a Dalvik application.

In this paper, we specifically target independent (physical or remote) attacks that require installation of malicious applications with full capabilities. independent attacks are widely seen in Android malware samples. As pointed out in [34], there are powerful malware species (such as `DroidKungFu`) that cannot achieve their effects if they execute within the context of a compromised legitimate process. These attacks need (i) additional functionality that is not provided in the context of a legitimate application, and, (ii) they need to continue execution when the compromised legitimate process is terminated.

In case of dependent attacks (such as return-oriented programming [10]) that complete with no independent processes, the malicious payload may be limited to achieve all its functionality within the boundary of a legitimate application. We realize the importance of such attacks, however, they are out of the scope of our current attack model.

2.3 Process Authentication Model

To achieve our goals for protecting Android from malicious applications that could be installed without the user's consent, we use a process authentication model that can detect malicious application executions. Our process authentication model (also referred to as authentication model in this paper) regards application processes as individual principals that must be authenticated at runtime. Our mandatory authentication provides legitimate applications with valid *application credentials*, and, detects malicious applications that lack such credentials.

Our goal is to authenticate individual application processes at the time of creation. Since applications are loaded in separate processes at runtime, our authentication model requires each process (that is created by an Android application) to be authenticated by the operating system. Inspired by authentication mechanisms in communications protocols, we assign a credential to each application. There are various ways to instantiate application credentials. A credential is a secure piece of information that can strongly authenticate a process and bind it to its application code at runtime. We define a secure application credential below.

Definition 1. *A secure application credential (SAC) is a unique secret issued to an application by a trusted process.*

Each SAC is associated with exactly one installed Dalvik or native application.

Our authentication model's policy is to enforce a mandatory authentication for each process. The authentication is based on using the secure application credential (also referred to as the credential in this paper) provided to the application that created the process. We also define two special processes called the *verifier process* that represents the authenticator (i.e., the operating system), and, the *registrar process* that is responsible for registering application credentials.

DEFINITION 2. *The verifier process π is a trusted process that has the authority of authenticating other processes.*

DEFINITION 3. *The* registrar *process ρ is authorized to perform the registration of an application by associating an application bundle with a unique SAC. The registration of the application is performed at install time with explicit user permission.*

To implement a secure authentication method and prevent attacks on the credentials, we define the following set of *credential registration requirements.*

- *Unique credential set.* For every legitimate Android application bundle, there is exactly one credential $\gamma \in \Gamma$, where Γ is a unique set of credentials.
- *Protected credential set.* Access to the set Γ (in memory or on the file system) is restricted to the verifier process π and the registrar process ρ.
- *Hard to regenerate.* It is computationally hard to regenerate an application credential $\gamma \in \Gamma$ created by ρ.
- *Preventing replay attacks.* Eavesdropping and replaying of a credential shall not be possible.

These requirements provide the basis for detecting and preventing the execution of malicious processes through a number of operations and properties that we discuss next.

2.4 Authentication properties and operations

The operating system is responsible for using application credentials to authenticate processes. The rationale behind our authentication model is determined by the following *Process Authentication (PA)* properties:

1. User processes (other than π and ρ) may not create or modify credentials, or, assign credentials to other processes and applications.
2. If a process fails to authenticate itself with a valid secure application credential, then the process is potentially malicious.
3. A process may not be authenticated with more than one credential.
4. A process may not inherit its *authentication status* from parent processes or any other process. Sibling processes are authenticated with a shared application credential.
5. A Dalvik application process is always a child of the `zygote` process. Native processes must either be a child of an Android system process or a Dalvik application process.

The PA properties ensure that processes are bound to proper application credentials such that a malicious process (under our attack model in Section 2.2) does not bypass the authentication, or, spoof other legitimate processes.

Operations. Our authentication model has three core operations: credential registration, process authentication, and runtime detection.

1. *Credential registration.* The user of a mobile device requests the registration of an application bundle, which is performed by the registrar process ρ. A credential is *valid* if (i) it is generated according to the credential registration requirements, (ii) it is registered in an application bundle, and (iii) it is recorded in a credential database maintained by the registrar.
2. *Process authentication.* In this operation, the operating system exchanges an authentication request and response with a verifier process π (defined earlier) to validate the authenticity of a process and bind it to a registered credential. Process authentication fails if (i) the process has no application credential, or, (ii) the process possesses an invalid credential.
3. *Runtime detection.* A runtime detection operation monitors process creations and enforces the mandatory authentication on all processes. A process that fails in process authentication is flagged as malicious and is denied execution rights.

These operations ensure proper authentication of all processes and detection of unauthorized ones. In the following section, we present the design of DroidBarrier that implements these operations in the Android operating system.

Our model may complement (i) a classification procedure (for example, [8, 12, 14]) that precede the credential registration operation, which provides the user more information about the application, and, (ii) a sophisticated access control system (such as Android's SELinux [30]) that uses our strong authentication and detection to properly identify application process and enforce fine-grained access rights based on the specified policies.

3. SYSTEM DETAILS

We design *DroidBarrier* to realize our authentication model. DroidBarrier implements the operations described in Section 2.4. In the following sections, we present the core components of DroidBarrier and discuss some of the alternative design approaches.

3.1 Credential Registration and Protection

To perform the registration of credentials, DroidBarrier includes a component referred to as the *credential registrar* (also referred to as the registrar in this paper). As depicted in Figure 3.1, credential registrar's task is to generate credentials for applications that the user designates as legitimate.

Establishing the trustworthiness of applications is an important procedure that can be executed before registering an

application with a valid credential. To establish this trustworthiness, there exists a number of classification techniques using static analysis [12, 14, 17], dynamic analysis [33] of the application, or based on experts' knowledge. In this work, we do not explicitly address this problem.

Credential generation. The registrar generates a credential that is computationally hard to guess. Among many ways to generate a credential, the registrar can use a strong pseudo random number generator.

There are two alternative approaches to our authentication mechanism based on secret credentials. First, registering a public checksum (e.g., a hash of application's class file) of an application bundle, and, recomputing the checksum at runtime to establish the authenticity and integrity of the application. Our approach is to use a *secret* credential, which eliminates the need for recomputing the checksum. Although by checking the checksum one can determine if the application bundle's integrity was violated, we choose to protect application bundles (as described below) for verifying and *preserving* their integrity and disabling possible denial of service. The second alternative approach is to use developer signatures to establish trust. In fact, Android uses developer signatures, but without an actual verification of the signature. Our design eliminates the need for third party certifications and verifications of public keys yet delivers the required level of trust.

To correctly establish the authenticity of an application, the registrar uses a two-way registration method. First, the registrar generates and stores a credential γ in A's bundle, forming a new application bundle A^*. We refer to an application bundle with an embedded credential, as a *protected application bundle*. This design choice is important to bind processes to specific executables in our runtime detection system, and, to protect the credential γ from attacks (described below). Second, the registrar stores a copy of γ in a *SAC database*. The SAC database implements the unique set Γ, which is the set of reference for validating any credential. Maintaining a copy of γ in the SAC database prevents forgery and replay attacks on the credentials. Note that an application's credential γ is invalidated if γ is removed from Γ and if the application is reinstalled or deleted.

Credential protection. To fulfill the specification of credentials (Section 2.3) and the PA properties (Section 2.4), our design must fully protect the credentials generated by the registrar to preserve their integrity, without relying on file system permissions. Our approach to this problem is to enforce access restrictions on the protected application bundles and the SAC database. To maintain integrity and confidentiality, we disable any process, other than the verifier process and the registrar, from read/write access to protected application bundles and the SAC database. We enforce this protection using DroidBarrier's kernel-side components by intercepting all open system calls and preventing unauthorized access to the SAC database (Section 4).

3.2 Runtime System

DroidBarrier includes a *process monitor* (Figure 3.1) to track the creation of processes by the Android's runtime. The process monitor relies on the authentication decision by DroidBarrier's *authenticator*. To maintain compatibility, we design these components as part of the Linux kernel without modifying Android's runtime. Our design strategy is to detect process creations, bind them to specific application bundles, and authenticate the processes according to registered credentials.

Process Monitoring and Runtime detection. The process monitor's policy is to regard every new process as unauthorized until it is authenticated. The detection strategy of the process monitor is to check the authentication status of applications at the time of creation. A process's status is either authenticated or unauthenticated.

The technical challenge in the design of DroidBarrier's runtime system is the semantic gap between the kernel and Dalvik virtual machine, which runs Android applications. Android's runtime system process, `zygote`, forks a new process when the user wants to run an application. At this point, it is not clear to the kernel which application is loaded in the newly created process. To reconstruct the semantics, in addition to monitoring process creations, the process monitor keeps track of file accesses by `zygote` to bind the loaded class file of the application to the newly forked process. When the newly forked process is bound to a Dalvik class file, the process monitor's runtime detection is complete and the process's identifier (PID) is sent for authenticator to proceed with the authentication.

Process authentication. DroidBarrier needs a reliable mechanism for authentication to prevent stealing credentials and spoofing legitimate processes. Android requires applications to be signed by developers [4]. However, as discussed earlier, developer signatures do not provide any reliable assurance about the authenticity of applications [4, 32]. We follow a design choice to mediate the authentication between DroidBarrier and user applications. Using our authentication mediation strategy, DroidBarrier performs the process authentication operation in three stages:

1. *Kernel-side checking.* As shown in Figure 3.1, a kernel-side component, authenticator (denote as AT), receives an authentication request from process monitor (denote as PM) for a process P. AT maintains a *status list* L, which records the authentication status for each process, *authenticated*, or, *unauthenticated*. If P's status is unauthenticated, AT sends an authentication verification request to the verifier process π. The format of the request is $(P.PID, P.path)$, where $P.PID$ is P's process ID and $P.path$ is the file path for the Dalvik class file that created P.

2. *Verification of credentials.* π loads the credential c from the protected application bundle on $P.path$ and the corresponding credential c' from the SAC database. If $c = c'$, then π sends a success response message back to AT.

3. *Status update.* When AT receives the response message from π, AT updates P's authentication status in L, accordingly.

3.3 Security Analysis

Security guarantees. DroidBarrier guarantees that all processes are authenticated. This property ensures that stealthy applications that were installed without the user's consent are detected. Independent remote or physical attacks described in Section 2.2 are detected as soon as a process with no valid credential is created. For example, when a malicious application tries to gain `root` privileges, it must create independent processes to run exploits through

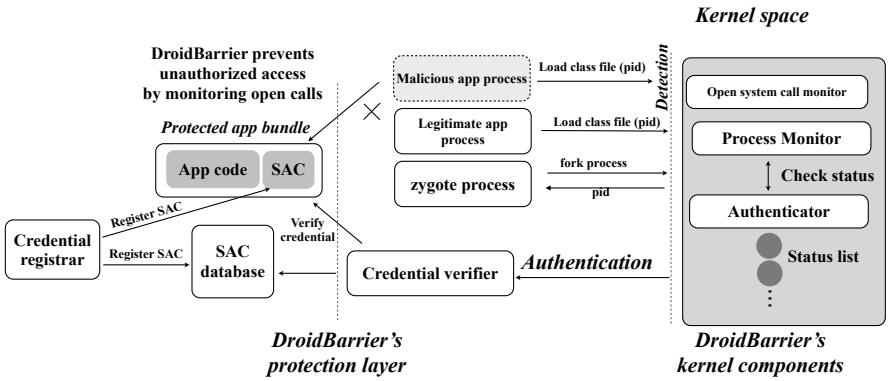

Figure 3.1: DroidBarrier performs three operations: credential storage and protection, runtime monitoring, and authentication of processes. Applications with valid credentials are allowed normal execution.Applications must obtain valid secure application credentials to ensure normal execution of their processes. An unauthenticated process is considered potentially malicious. Depending on system policies, DroidBarrier may kill a process or limit its activities.

shell scripts or native code. DroidBarrier requires mandatory authentication and denies access to malicious processes that fail to authenticate. As discussed in Section 5.1, all the instances of Android malware tested under DroidBarrier, failed to continue normal execution.

Security of credentials. To secure application credentials, we study various attacks that can occur on the credentials and provide solutions accordingly. An important attack on DroidBarrier is to steal or corrupt application credentials. Attacks on credentials can occur in a number of ways:

- An attacker may include a credential in a malware's application bundle according to our credential specifications. This attack fails since the fake credential created by malware is not registered with the SAC database.
- Under our attack model (Section 2.2), an attacker may attempt to violate confidentiality by stealing the credentials or violate integrity by corrupting them. We protect credentials by disabling read/write access to protected application bundles on the disk (Section 3.1). Read and write accesses on protected application bundles are only permitted for the `zygote` process to allow loading the class files. Note that readability constraints on applications do not prevent loading class files. That is, we especially allow `zygote` to load class files, after authenticating `zygote`.
- We prevent memory attacks on credentials by mediating the authentication (Section 3.2). Thus, credentials are never stored in application heaps or stacks at runtime.

Protecting DroidBarrier code and data. DroidBarrier's process monitor and authenticator execute in kernel mode. This guarantees the isolation of process monitor and authenticator data from user space attacks under the reasonable assumption of a trusted Linux kernel (Section 2.2).

The process monitor and the authenticator communicate directly on shared data structures. The authenticator and the verifier process communicate through a shared memory that is managed by the kernel. This shared memory is not accessible by any other process and may not be read from or written to. This protection is implemented in DroidBarrier's kernel-side components without relying on Linux kernel's permission system. The verifier component is protected in two ways. The executable code is kept in a protected application bundle with an additional restriction of disallowing the Android application manager to remove or reinstall it. At runtime, the verifier process is automatically created by DroidBarrier's kernel-side components when the `zygote` process is created. Further execution of the verifier application is denied.

Limitations of DroidBarrier. DroidBarrier specifically targets processes that are created by malicious applications. This is a critical category of attacks that is used by modern Android malware [34]. However, embedded attacks that run entirely within the boundaries of a legitimate process cannot be detected by our current mechanism. In general, any malicious application that possesses valid credentials (by hijacking legitimate applications, or granted by mistake) bypasses our security guarantees.

DroidBarrier's code and data may be subject to kernel-based attacks either by rootkits or malicious kernel modules. In principle, rootkit attacks (such as return-oriented rootkits [19] and system call obfuscation [31]) can cause kernel integrity and confidentiality violations. Although DroidBarrier is not designed to specially prevent rootkits, assuming that the kernel is initially free of malicious code, DroidBarrier prevents further malicious code executions. For instance, modern rootkits need to use return-to-user attacks [21] and run the code stealthily in user space. DroidBarrier detects this type of attack when the prerequisite of a successful attack is to first run a user space task that can receive the execution, for example, from a `NULL` dereferencing in the kernel. To protect DroidBarrier from malicious kernel modules, DroidBarrier can be extended to authenticate kernel modules at the time of loading. This extension is left for a future work.

4. IMPLEMENTATION

We describe a prototype of DroidBarrier implemented in C

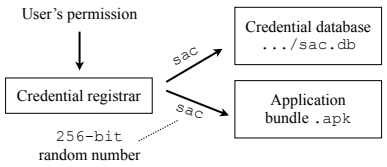

Figure 4.1: The registrar creates new credentials for applications and uses the credentials to label the apk bundles.

using Android's native APIs. Our implementation involves an extension to the Linux kernel for process monitor and authenticator (described in Section 3.2). We use Android Honeycomb 3.2 with the Linux kernel version 2.6.36.4 for our implementation platform. Below, we present a highlight of our implementation features.

1. Kernel modification is minimal and hardware independent.
2. DroidBarrier does not modify the original semantics of process management and creation functions in Linux.
3. Our prototype is compatible with Android's runtime, and all other Android operating system components.

In the following we first describe the implementation details of credential registrar followed by process monitor, authenticator, and verifier.

Credential registrar. Figure 4.1 shows the registration process. We implemented a credential registrar that installs application credentials for an Android application bundle. The registrar uses a random number generation function to generate a credential and labels the application bundle (with the `apk` extension) with the newly created credential. The registrar also records the new credential in a sequential database of keys and application names for reference at authentication time. Our labeling of the `apk` bundle does not alter the functionality of the application in any way. The label only includes an additional header and the credential in a format that DroidBarrier can recognize.

Credential protection. We protect credentials (Section 3.1) by implementing a light-weight access control system in the kernel. As depicted in Figure 4.2, we place checkpoints in the beginning of `do_sys_open`, which enables us to intercept all open system calls. Before returning a file descriptor `fd` to the requesting process, we check for two conditions. First, we check if the requested file path is a protected application bundle, by reconstructing the full file path from the process's current directory. Then, we verify if the requested file path is an executable. If the condition is true, we check if the requesting process is the registrar, the credential verifier or `zygote`. According to our policies (Section 2.4), DroidBarrier denies access to executables for all other processes. Second, we check the file path against our credential database. For the credential database, we only return an `fd`, if the calling process is either the credential verifier or the registrar.

Process monitor. We implement the process monitor (Figure 4) by inserting check points in specific kernel functions. To maintain performance, we avoid using existing Linux APIs to trace kernel functions (such as `kprobe` or `ptrace`). To monitor the creation of processes by the Android's runtime, we insert check points in `do_fork`, for Android's Dalvik

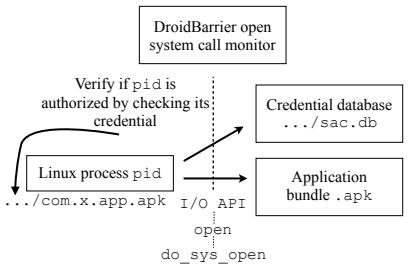

Figure 4.2: DroidBarrier protects credentials by monitoring open system calls and checking for permissions.

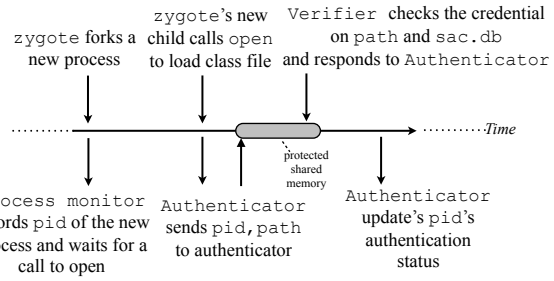

Figure 4.3: Monitoring of a process starts at the time zygote forks the process. Process monitor prepares an authentication pool for authenticator, which sends verification requests to the verifier process. The interactions above the time line are in user space.

applications, and `do_execve`, for native applications loaded directly by the kernel. The `zygote` process calls `fork()` to create a new process and load a Dalvik class in it. We record the generated `pid` in `do_fork` and track `zygote`'s subsequent system calls so that we bind the new `pid` to its application bundle.

DroidBarrier tracks the loading of Dalvik class files through our check point in the `do_sys_open` function. When the process `pid` (that must be the child of `zygote`) loads a class file from the file system, the authentication operation can proceed to verify `pid`'s credentials. Since the authentication is asynchronous and waits for the open system call, DroidBarrier maintains a dynamic pool of awaiting authentication processes (an array `requests` of type `struct auth_request`) to perform the authentication.

Authenticator and verifier. The authenticator creates and manages a shared memory with the verifier process in user space. The shared memory is not visible to any process. We implement this restriction in the check point in `do_sys_open` function. DroidBarrier checks for permissions before the open system call returns a file descriptor. The permission check maintains a list of permissions for specific file descriptors. Access to the shared memory file descriptor is denied to all processes except the verifier (for which we record the `pid`). The authenticator continuously checks the `requests` pool for fresh authentication requests. Authenticator sends each request in a special string format to the verifier using a first come first served schedule. Once the verifier checks the credentials for the process it sends a response back on the same shared memory to the authenticator. The authentica-

tor informs the process monitor about completed authentication via updating the status of the request in `requests`.

The verifier's task is to verify the credentials on a file path received from the authenticator. The verifier opens the corresponding file path and searches for credentials. If one found, the verifier reads the credential and checks if the credential exists in the SAC database. The SAC database is a sequential file on the disk containing application names, file paths, and credentials. If the credential matches the process file path, the verifier creates a success message and sends it to the authenticator.

5. EVALUATION

We evaluate DroidBarrier by testing its detection capabilities against Android malware samples. Also, we present experiments for evaluating the performance of DroidBarrier.

5.1 Detection of Malicious Applications

According to Android's application model, every application bundle must run in at least one process independent of other applications. Thus, to detect malicious Android applications, we design DroidBarrier to detect malicious applications based on their process creations. According to the analysis performed in [34], about 36.7% of the applications (in a collection of 1260 malware instances) contained embedded privilege escalation exploits that execute through hidden shell scripts requiring their own independent processes.

We analyzed three chosen sets of Android malware `DroidKungFu`, `BaseBridge`, and `AnserverBot`. These malware (provided by Android genome project[1]) have 96, 122, and 187 variants, respectively. Our chosen malware categories use sophisticated social engineering, root vulnerability exploitation, and financial attacks, found in many other similar malware sets [34]. Nevertheless, our malware analysis is not limited to the chosen sets and is applicable to all malware with privilege escalation attacks as described in [34]. The three analyzed malicious applications share a core functionality that results in their detection by DroidBarrier. That is, they try to gain escalated privileges by running an exploit and subsequently calling the `su` utility. The `su` utility is not included in Android by default, but the malicious application can install it after gaining installation privileges that bypass user's permissions.

DroidBarrier successfully detected all the samples that we tested from these three sets. All the samples tried to run `su`, which resulted in processes that failed to authenticate. `DroidKungFu` tried to run other unknown executables that were also immediately detected at the time of process creation. Thus, based on our results, we conclude that DroidBarrier can detect a malicious application that

- is installed through drive by download,
- uses privilege escalation attacks, or
- is physically ported to the device.

While DroidBarrier detects critical malicious applications, it cannot detect a malicious application if

- the user grants credentials to it, or
- it can exploit existing legitimate applications to achieve its goals.

[1]http://www.malgenomeproject.org/

We observed that the malicious applications that we tested stopped working, and sometimes their processes were automatically killed due to malfunctioning as a result of denying access to their required services by DroidBarrier.

5.2 Performance Evaluation

We evaluate the performance of our implementation prototype described in Section 4. Our performance evaluation investigates the following:

1. DroidBarrier performs permission checks prior to creating a file descriptor in the open system call. How do these permission checks affect the I/O performance in the kernel?
2. What is the performance penalty on process creations?
3. How does the I/O performance in Dalvik applications are affected by DroidBarrier?

In the rest of this section, we describe our experimental setup followed by three sets of experiments for I/O and process creation. Our results show efficient performance of DroidBarrier with low extra overhead in I/O performance, and, negligible performance penalty in process creation.

Experimental setup. Our evaluation focuses on the effect of DroidBarrier on the performance of the Linux kernel in Android. We run all the experiments on a Samsung Galaxy Tab 10 (with model number P7510) running Android Honeycomb 3.2 with the Linux kernel version 2.6.36.4. For our evaluations, we port the benchmarking suites lmbench[2] and UnixBench[3] to Android.

For our evaluations we needed a benchmarking suite that can accurately measure I/O and process creation. There are existing suites such as lmbench[4] and UnixBench[5] that conveniently run on Linux distributions for x86 machines. Per our research, these suites have not been ported to embedded Linux running on the ARM architecture. Thus, we modified some of the existing code in lmbench and UnixBench for compatibility with the ARM Linux.

When running the modified kernel with DroidBarrier, we periodically simulate the authentication of processes as if a user is consistently launching new applications. When performing experiments, there was about a total of 130–150 running Linux processes.

I/O performance. I/O performance experiments show a consistently efficient performance of DroidBarrier with the performance penalty not exceeding 13%. For measuring I/O performance, we evaluate the performance of the I/O operations write, read, open, and close system calls as well as the performance of piping. Note that for protecting secure application credentials DroidBarrier only includes a monitoring on the open system call. For each measurement, the I/O operation is repeated 10000 times continuously. We measure the time taken to perform the whole 10000 calls to the I/O function. The experiments on read and write include calls to open and close. For open and close, in each iteration we call open followed by a close. We performed 250 runs of each loop to collect an average performance value.

[2]http://www.bitmover.com/lmbench/
[3]http://code.google.com/p/byte-unixbench/
[4]http://www.bitmover.com/lmbench/
[5]http://code.google.com/p/byte-unixbench/

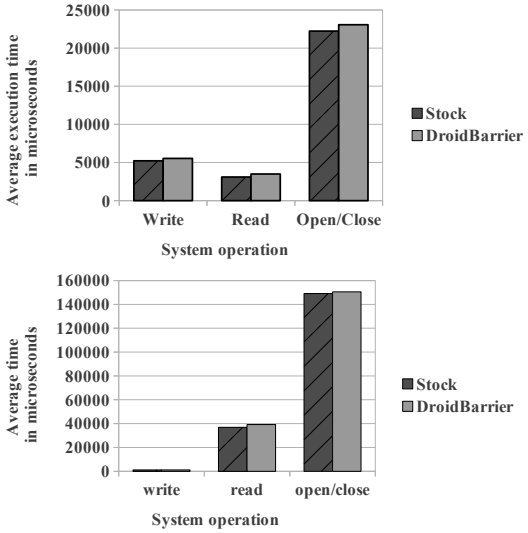

Figure 5.1: (a) Four I/O operations: write, read, open, and close. Maximum penalty is 12.92. (b) Four I/O operations: write, read, open, and close for Dalvik applications. Maximum penalty is 6.26.

In the results of our experiments (depicted in Figure 5.2), we have an average maximum of 12.92% performance penalty for the read calls, an average minimum of 3.76% performance penalty for open/close, and for write calls there is an average of 6.01%.

To examine the performance of Dalvik applications under DroidBarrier, we conducted an I/O experiment by making file writes, reads, open and close using the `BufferedWriter` and `BufferedReader`. The open and close calls involve opening a file using `FileWriter`, which is used to create a `BufferedWriter` and we close the `BufferedWriter` subsequently. We developed an application to perform 1000 iterations of each I/O operation (we call open and close calls together in one iteration). The results of Figure 5.2 shows the average performance for 200 runs of all experiments.

Our experiments show a maximum penalty of 6.26% in `BufferedReader`, an average performance penalty of 1.48% in `BufferedWriter`, and a minimum performance penalty of 0.927% when opening a file using `FileWriter`.

Process creation performance. We measure the fork system call in loops of 1000 iterations. In each iteration we fork a child, and, a child exits immediately, and, we perform a total of 100 runs of the experiments. Android heavily uses fork for process creations. Since DroidBarrier performs the authentication asynchronously, we do not see major performance downgrade for forking a process. The average performance penalty is 0.041% (depicted in Figure 5.2), which is insignificant.

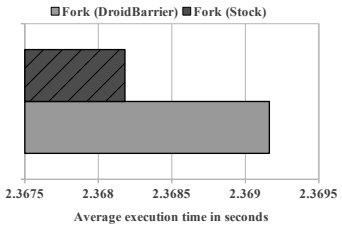

Figure 5.2: Performance of fork system call evaluated in loops of 1000 iterations. DroidBarrier asynchronously authenticates processes to avoid major delays at the time of creating processes. It has an insignificant performance penalty of 0.041% in process creation.

6. RELATED WORK

Quire is a cryptographic solution that annotates interprocess communications (IPC) in Android to provide provenance assurance to the receivers of the IPC messages [11]. Quire provides authentication only at the IPC and remote procedure call (RPC) levels. Quire's strategy is to authenticate every IPC and RPC and enable the applications to use the provenance information provided by Quire. Our authentication model in DroidBarrier uses a different strategy. First, we authenticate a Linux process as one unit. This authentication can provide provenance at any lower level such as function calls, IPCs, and RPCs. Second, under DroidBarrier, only registered applications can successfully authenticate, and as a result, unauthenticated processes are considered potentially malicious.

A2 [2], uses a challenge-response protocol to authenticate applications. A2 [2] is designed to work with Linux processes created for native C applications. DroidBarrier authenticates Android's interpreted applications by reconstructing the semantics between the Linux kernel and Android's runtime (Section 3.2). DroidBarrier substantially improves and advances the techniques discussed in A2 by (i) an authentication model that specifically addresses the challenges for authenticating Dalvik applications, (ii) an authentication mediation strategy that reconstructs the semantic gap between the Dalvik runtime and the kernel, and (iii) a runtime system that monitors Android's `zygote` for process creation and enforcing authentication.

VMWare Mobile Virtualization Platform (MVP) [3] is a type 2 hypervisor and is capable of isolating restricted and normal execution environments. Bare Metal Hypervisor [18] runs security sensitive applications in trusted and isolated environments. Also, Cells [9] runs multiple virtual phones using a shared underlying physical phone to provide isolation. Other systems such as TrustDroid [6] isolate applications in isolated logical domains. In DroidBarrier, we do not use virtualization to provide isolation at runtime. This helps in achieving better compatibility.

Mandatory access control (MAC) systems complement DroidBarrier to provide fine-grained authorization. Android's SELinux [30], for example, can benefit from DroidBarrier's strong authentication guarantees. Moreover, TOMOYO Linux implements a MAC system based on behavioral analysis [24]. Paranoid [26] is a system that takes a novel direction by delivering a cloud-based security solution for Android. User-driven access control [28] is another promising direction that uses the principle of least privileges and incorporates the user's decision in granting permissions.

Static analysis can also complement DroidBarrier by providing a classification of applications at registration time. Kirin [14] uses security requirements engineering techniques for classifying malicious applications. RiskRanker [17] analyzes Android applications for finding zero-day Android malware. CHEX [22] is another tool capable of detecting

component hijacking attacks in poorly engineered Android applications. Barrera et al. [5] present a methodology for investigating permission usage in Android applications. Finally, general static analysis techniques such as [12] can also be used to provide information on installing applications.

7. CONCLUSIONS

We presented a general model for providing high assurance authentication for application processes running on an Android-enabled device. We achieve the high assurance by developing an authentication model that uses secure application credentials, maintained and protected by our runtime system, to authenticate processes and bind their identity to legitimate applications installed on the device. Our authentication approach guarantees protecting the system from execution of malicious applications that may exploit the many system and application vulnerabilities to be installed on the device. Our future work will focus on authenticating interprocess communications and authenticating access to applications' assets.

8. REFERENCES

[1] H. Almohri, D. Yao, and D. Kafura. Process authentication for high system assurance. *IEEE Transactions on Dependable and Secure Computing*, PP(99), 2013.

[2] H. M. Almohri, D. Yao, and D. Kafura. Identifying native applications with high assurance. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 275–282, New York, NY, USA, 2012. ACM.

[3] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, Dec. 2010.

[4] D. Barrera, J. Clark, D. McCarney, and P. C. Van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of Android. In *Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices*, SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

[5] D. Barrera, H. G. u. c. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17^{th} ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[6] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on Android. In *Proceedings of the 1^{st} ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'11, pages 51–62, New York, NY, USA, 2011. ACM.

[7] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2011.

[9] C. Dall, J. Andrus, A. Van't Hof, O. Laadan, and J. Nieh. The design, implementation, and evaluation of cells: A virtual smartphone architecture. *ACM Trans. Comput. Syst.*, 30(3):9:1–9:31, Aug. 2012.

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[12] K. O. Elish, D. Yao, and B. G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Proceedings of the Workshop on Mobile Security Technologies (MoST)*, May 2012. In conjunction with the IEEE Symposium on Security and Privacy.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. 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, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[14] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[15] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, Jan. 2009.

[16] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[17] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[18] K. Gudeth, M. Pirretti, K. Hoeper, and R. Buskey. Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 33–38, New York, NY, USA, 2011. ACM.

[19] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages

383–398, Berkeley, CA, USA, 2009. USENIX Association.

[20] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM symposium on Access control models and technologies*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. ACM.

[21] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240, New York, NY, USA, 2012. ACM.

[23] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the Android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 343–352, New York, NY, USA, 2011. ACM.

[24] J. Park, B. Kim, S.-R. Kim, J. H. Yoon, and Y. Cho. Performance analysis of security enforcement on Android operating system. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 282–286, New York, NY, USA, 2011. ACM.

[25] Y. Park, C. Lee, C. Lee, J. Lim, S. Han, M. Park, and S.-J. Cho. RGBDroid: a novel response-based approach to Android privilege escalation attacks. In *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*, LEET'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[26] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[27] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated system calls. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 358–367, June 2005.

[28] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 224–238, Washington, DC, USA, 2012. IEEE Computer Society.

[29] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[30] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-powered mobile devices using SELinux. *Security Privacy, IEEE*, 8(3):36–44, may-june 2010.

[31] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In *Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'11, pages 214–233, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current Android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[33] L. K. Yan and H. Yin. DroidScope: seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[34] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, may 2012.