Design and Implementation of an Android Host-based Intrusion Prevention System

Mingshen Sun[*] [†], Min Zheng[†], John C.S. Lui[†], Xuxian Jiang[§]
[†] The Chinese University of Hong Kong, [§] Qihoo 360 & NC State University
{mssun, mzheng, cslui}@cse.cuhk.edu.hk, jiang@cs.ncsu.edu

ABSTRACT

Android has a dominating share in the mobile market and there is a significant rise of mobile malware targeting Android devices. Android malware accounted for 97% of all mobile threats in 2013 [26]. To protect smartphones and prevent privacy leakage, companies have implemented various *host-based intrusion prevention systems* (HIPS) on their Android devices. In this paper, we first analyze the implementations, strengths and weaknesses of three popular HIPS architectures. We demonstrate a severe loophole and weakness of an existing popular HIPS product in which hackers can readily exploit. Then we present a design and implementation of a secure and extensible HIPS platform — "*Patronus*". Patronus not only provides intrusion prevention without the need to modify the Android system, it can also dynamically detect existing malware based on runtime information. We propose a two-phase dynamic detection algorithm for detecting running malware. Our experiments show that Patronus can prevent the intrusive behaviors efficiently and detect malware accurately with a very low performance overhead and power consumption.

1. INTRODUCTION

We now live in a mobile digital era and smartphones are becoming indispensable as they are being used as personal communication and computing devices. Unfortunately, smartphones are also becoming hackers' biggest target [44]. Android is an open source operating system for mobile devices. Its market share has reached around 80 percent of all smartphone shipments in the second quarter of 2013 [35]. Due to its large market share, it becomes the major target for hackers. A recent report [26] indicated that 97 percent of these threats targeted Android devices. Furthermore McAfee [39] showed that 17,000 new Android malware were found in the second quarter of 2013. All these indicate that Android devices are facing an explosive growth of malware threat.

Permission mechanism is the primary security protection in Android. When an Android user installs a mobile application (app for short), the system will alert the user about the required permissions of that app. But this mechanism does not provide an adequate level of security. As shown in [28, 31, 38, 16], there are plenty of Android permission abuses. Due to the working mechanism and the open source nature of Android system, it is easy to write malware variants which can bypass the detection of anti-virus software. Recently, researchers [59, 48] have shown that some of the existing anti-virus software cannot effectively detect malware variants which were generated by simple transformations. Most of the existing anti-virus engines are using static analysis and malware signatures [62, 32] for detection. To make up for the limitations of static analysis, several systems [24, 55, 8] aim to detect the *runtime behaviors* by using dynamic analysis. However, these systems are mainly for post-mortem analysis and with a high likelihood, the malware under investigation may have already infected many devices. Furthermore, for these dynamic analysis systems, how to automatically trigger the malicious behaviors is still an ongoing research problem [58, 47]. Therefore, the security community has been advocating the use of a *host-based intrusion prevention system* (HIPS), which is an installed software on a mobile device to monitor suspicious activities, and to block and report malicious behaviors by analyzing events at realtime. HIPS can be installed on mobile devices to provide protection against apps which have runtime malicious behaviors. HIPS can also dynamically intercept the apps at runtime and notify users when the malware invoke some dangerous application program interfaces (APIs). In fact, the use of HIPS is gaining popularity and has been implemented in products like Jinshan Mobile Duba [9], LBE [10], 360 Mobile Safe [13], etc.

In general, there are three approaches to implement HIPS in Android, they are: (1) *system patching*, (2) *application repackaging* and (3) *API hooking*. System patching is to modify the Android operating system with new permission management functions. Application repackaging is to disassemble a mobile app, add new policy enforcement to it, and then repackage the mobile app as a new app. API hooking is to intercept mobile app's API calls at runtime so as to inspect malicious behaviors. Each of these approaches has its own limitations and may bring new vulnerabilities. In Section 3, we present in detail the implementation issues as well as weaknesses of these three popular HIPS implementations.

Given the limitations of static and/or dynamic analysis, as well as weaknesses of existing HIPS products, we propose

[*]Part of this work was done during his internship at Qihoo 360.

ACSAC '14, December 08–12 2014, New Orleans, LA, USA

ACM 978-1-4503-3005-3/14/12 ...$15.00
http://dx.doi.org/10.1145/2664243.2664245

an enhanced HIPS called *Patronus*, which not only performs a secure policy enforcement, but can also dynamically detect existing malware using runtime information. To guarantee the ease of deployment, Patronus does *not* require any modifications on the Android firmware or mobile apps. Patronus performs runtime policy enforcement on the system level to inspect malicious behaviors. Moreover, Patronus provides a host-based runtime detection which can halt malicious behaviors execution. We make the following contributions:

- To the best of our knowledge, this is the first work which systematically analyzes three popular frameworks of HIPS and exposes various security vulnerabilities of these existing HIPS architectures. We also illustrate how to exploit the vulnerability of a popular HIPS product so as to bypass malware detection.
- We design and implement a secure architecture Patronus, which can prevent mobile malware intrusions and can detect malware at runtime. Patronus addresses the security issues we reveal in the current HIPS products and effectively prevent intrusions.
- We design and implement a two-phase detection algorithm based on the runtime intrusion information to detect existing malware. The algorithm can determine and prevent malware at runtime (online), while traditional signature-based static method can only achieve this in an offline manner.

The rest of the paper is organized as follows: Section 2 introduces the necessary background on Android. In Section 3, we present the strengths and weaknesses of three popular HIPS implementations. In Section 4, we propose our Patronus system, and describe the method to prevent intrusions as well as proposing a two-phase algorithm for dynamic malware detection. In Section 5, we present our experimental results to illustrate the effectiveness and performance of Patronus. Section 6 presents the related work and the conclusion is given in Section 7.

2. BACKGROUND

In this section, we introduce the architecture of Android system, its binder inter-process communication mechanism, as well as the Dalvik Virtual Machine and the Native Development Kit in Android. This background is essential to understand how one can create a HIPS for Android devices.

2.1 Android Architecture

Android is an open source operating system for mobile devices developed by Google. The system consists of five functional layers: (1) *kernel*, (2) *libraries*, (3) *runtime support*, (4) *application framework* and (5) *applications*. Figure 1 presents the Android architecture. Android utilizes Linux as the core kernel with various drivers for hardware communication. The libraries layer contains native libraries such as `libc` or `OpenGL` so as to support higher application layers. The runtime layer consists of the Dalvik Virtual Machine (DVM) with various runtime libraries. DVM is a special Java virtual machine to execute Android apps. Application framework layer contains basic services to provide activity management, SMS management, etc. Lastly, all Android apps are running on top of these layers.

Android apps are mostly written in Java using the Android SDK , and DVM is responsible for interpreting and executing these apps. Each app is running within its own DVM. The Android system assigns a unique ID for each

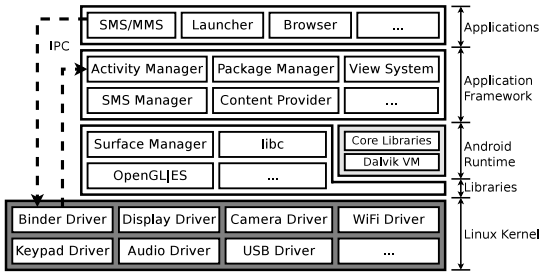

Figure 1: Architecture of Android system.

DVM to achieve process (or app) isolation. This sandboxing mechanism provides the basic security protection between apps. Android also provides a special inter-process communication mechanism called the *Binder*. Each app has to go through the Binder (which is in the kernel layer) so as to communicate with other apps or services.

Besides using DVM sandbox, permission is another security mechanism provided by Android. Only when an app has explicitly declared its permissions in its `AndroidManifest.xml` file, then the app can use the corresponding APIs. Moreover, when installing an app, the system will display a dialog window to alert users about the declared permissions of that app. However, most users usually accept these permission alerts without paying much attention at installation. Once the installation is completed, the system will not warn users in later invocations. Many Android malware take advantage that most users are negligent and simply accept the permission alerts, so the malware can often obtain enhanced permissions (e.g., sending premium SMS messages to subscribe chargeable service in the background).

2.2 Binder Mechanism

Binder is a specialized inter-process communication (IPC) mechanism in Android. Since apps are running in their own DVM sandboxes, they need to communicate through the Binder so as to utilize others' services. Figure 2 shows the basic flow of the Binder mechanism. For instance, if an app wants to send an SMS message, it should first (1) contact the `Service Manager` which contains the information of all registered services. The `Service Manager` will provide a handler to communicate with the `ISms Service` which is responsible for sending SMS messages. Once the app has the handler, it can (2) ask the `ISms Service` to send SMS messages, then (3) the `ISms Service` will process the request and send the message through the SMS Driver. Note that all communications have to go through the Binder by sending transactions with the required information (i.e., the parcel). Transaction is a communication procedure between two processes. In Android, the Binder transaction is used to send service requests (which is represented by a transaction code) to the corresponding processes. There are two stages to complete a transaction. First, the Binder will deliver a data parcel to the destination process containing the receiver information (i.e., transaction descriptor). Secondly, after completing the request, the received process will save the result in a reply parcel. In the above example of sending an SMS message, there are two transactions and they are completed in three steps. The first transaction requests for the ISms Service handler. The second transaction requests for sending an SMS message. In the second transaction, the data parcel contains the ISms Service descriptor (`com.android.internal.telephony.ISms`) and information

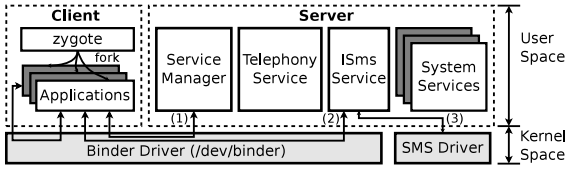

Figure 2: Binder IPC mechanism.

(e.g., destination address and text content) for sending an SMS message. In the third step, the `ISms Service` sends the request and saves the result in a reply parcel.

Note that this procedure is similar to the client-server communication model. Apps are on the client side and they are executing within their own sandboxes, while various services are on the server side and they are managed by the `Service Manager`. The `System Services` are a group of services related to the Android system (e.g., Account Manager Service, Battery Service and Network Management Service). Moreover, an app can export its service to the public by registering to the `Service Manager`. This special Binder mechanism not only provides a flexible inter-process communication framework, but also isolates apps and services so as to protect the system.

2.3 DVM & Native Development Kit

Dalvik Virtual Machine (DVM) is a special component in the Android system. The apps are written in Java and compiled into a bytecode file (i.e., `class` file) with Android SDK. These bytecodes will be converted to Dalvik executable files (i.e., `dex` file) before installation. To instantiate an app, the DVM will interpret and run the app in a *different* virtual machine. In order to integrate with other C/C++ native libraries, Android provides the Native Development Kit (NDK) for developers to implement parts of the app using native code such as C/C++. Moreover, Java Native Interface (JNI) provides a bridge between the Java code and C/C++ code so developers can invoke native code easily. Therefore, for the system, DVM can use native libraries (e.g., `OpenGL`, `libbinder`, `libc`) to support its execution.

3. HIPS

In this section, we discuss three popular frameworks to implement host-based intrusion prevention systems (HIPS): (1) *system patching*, (2) *application repackaging* and (3) *API hooking*. In particular, we present their implementations, strengths and weaknesses. We also demonstrate how to bypass a popular HIPS product.

3.1 System Patching

Before Android 4.3, there was no HIPS integrated in the Android OS, and users have experienced various forms of permission abuses since malware can take advantage of the system loopholes to achieve permission escalation. Because Android is an open source project, researchers and companies have developed patches for the Android system [53]. In fact, some third-party system images [7] have integrated with these patches and they can manage the permission usage for each app. Some mobile phone vendors (e.g., Huawei) also provide patched images for their phones and users can install these modified images on their devices. In Android 4.3, Google also provides system patching. For example, there is a hidden function called `App Ops` [45], which is a form of HIPS and by using `App Ops`, one can disable certain permissions of an app. However, this `App Ops` has been removed in Android 4.4.

To install these HIPS in Android, users need to update to Android 4.3, or install the patched system image from their vendors. However, as shown in the latest report [30], only 10% of Android phones are powered by Android 4.3 or 4.4. Furthermore, if users want to install any patched image, they have to find the appropriate third-party system images which match with their phone models. For this reason, many users often opt *not* to install these patches. These imply that there are a large percentage of smartphones which are vulnerable. Moreover, because one has to use `App Ops` to disable permission before launching the apps, systems like `App Ops` cannot prevent intrusions at runtime. Hence, even with system patches, many Android phones are still vulnerable to malware attack.

3.2 Application Repackaging

Android application package file (or `apk` file) is a zip file which contains Dalvik executable file (`classes.dex`) and other resource files (e.g., images or audio). An app needs to declare its permission usage in the `AndroidManifest.xml` file within the `apk` package so as to use the corresponding APIs. Thus, to prevent an app using sensitive APIs, one can delete certain permission declarations in the `AndroidManifest.xml` and repackage it as a new `apk` file. App Shield [5] is an app to manage app permission using this technique.

Since the Dalvik executable file is converted from the Java bytecode, it is easy to be reversed to readable codes. In fact, several tools [14, 4, 25] provide assembling and disassembling functions on `dex` files. Hence, one can disassemble the `dex` file into readable codes, modify the program logic and then assemble and repackage it into a new `apk` file. By using this repackaging method, HIPS can be implemented by inserting *stubs around sensitive Android APIs* so as to perform policy enforcement without modifying the Android system. For instance, if an app needs to get the current location by utilizing the GPS function, it has to use the `request LocationUpdates` API. So one can add code into the app to check the GPS permission before invoking this API. Aurasium [54] and RetroSkeleton [21] are two projects using this form of application repackaging to inspect sensitive APIs to reinforce the Android permission policy.

Although application repackaging does not need to modify the Android system, there are some major drawbacks. The first major drawback is on incomplete security coverage. Since a function can be implemented by different approaches (i.e., calling different APIs), so it is possible to miss policy enforcement unless we know *all possible APIs* in realizing a function. For example, using the Java Reflection, one can invoke APIs and bypass the stubs. Secondly, disassembling technique can only work on `dex` file. If application developers use native code (e.g., binary libraries written in C or C++) to implement some functionalities, application repackaging cannot monitor these native library calls. Hence, application repackaging cannot fully prevent malware intrusion. Furthermore, malware writers can exploit bugs in the disassembling tools [50, 3] to conceal their malicious functions. In summary, application repackaging *cannot* fully prevent malware intrusion.

3.3 API Hooking

Generally, API hooking is to intercept API calls in order

to inspect the behaviors of a calling app. For Android, apps must go through the Binder to call other services, so the hooking method can be implemented on the Binder communication. The basic flow of using API hooking to implement HIPS can be summarized as: (1) gaining root or system privilege; (2) injecting a shared library object file (i.e., `so` file) to the target process; (3) carrying out hooking on target APIs; (4) loading policy enforcement function. Let us describe in detail the Android API hooking process.

3.3.1 Overview of API Hooking

Due to the sandboxing protection, API hooking requires root or system privilege to hook on functions in the libraries of the target processes. There are several approaches [1, 29] to gain root or system privileges. Basically, these tools exploit the Android system and get root or system privileges. In order to ensure the security, the tools will also install an app called Superuser on the phones to manage the authorizations of higher privileges to certain apps. Furthermore, the system loopholes can be easily patched by using hooking methodology [41]. Hence, malware cannot exploit the loopholes to infect system.

The hooking operation has to be conducted in the native code. To achieve this, we inject a shared library file (`so` file) to achieve the hooking. Inline hooking [34] and `ptrace` are two methods to inject an `so` file. However, the implementation of inline hooking on ARM platform is more difficult than on the x86 platform. Therefore, we use `ptrace` to attach the target process so as to modify the registers in the target process to execute our shellcode. In the shellcode, we utilize `dlopen` and `dlsym` functions to inject an `so` file into the target memory. The shellcode will also invoke an entry function in the injected `so` file to carry out API hooking.

We can use the API hooking on Android to intercept any Java method. In particular, we can modify the global object (`gDvm`) in the `libdvm.so` library of DVM to intercept a target method. `gDvm` maintains the structure of every class and method at runtime. There is a variable called `insns` in `gDvm` for each method specifying the address of the corresponding Java method. So, we can find the target method in the loaded classes (`gDvm->loadedClasses`) and replace the method `insns` to point to the function in the injected `so`'s library. Hence, when one calls the intercepted APIs, the DVM will execute our own methods. After executing our methods, we can call back the original APIs. One can implement policy checking and enforcement in Java and compile them into a `jar` package. So the injected library can easily load the logics in the `jar` file.

Most of existing HIPS on mobile devices are implemented by "*hooking on the client side*". Let us elaborate on this approach and we will also illustrate its deficiency.

3.3.2 API Hooking on the Client Side

In Section 2, we explain the client-server model in the Binder communication. To realize HIPS via API hooking on the client side is to hook the shared libraries of the normal apps so as to perform policy enforcement. To do this, we need to first understand the workflow of the Binder transaction on the client side. To illustrate, consider that we use the following codes to send an SMS message:

```
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage(
    "phoneNumber", null, "message", null, null
);
```

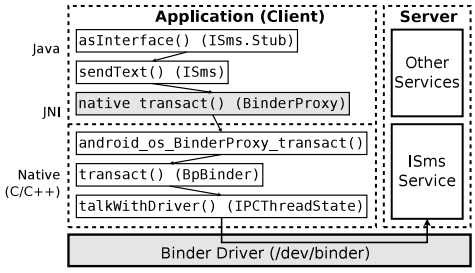

Figure 3: Call graph of ISms Binder transaction on the client side.

The first line is to query the ISms service (`ISms` handler) from the Service Manager. When calling the `sendTextMessage`, the system will call the `sendText` method of the `ISms` handler. Figure 3 depicts the partial function call graph of a Binder transaction for sending an SMS message. On the call path, there is a JNI method, `native transact`, which bridged with the `android_os_BinderProxy_transact` on the side of the native code. Finally, the transaction will end with the `talkWithDriver` function on the client side. Because the `native transact` method is the last method at the Java level, we can intercept it for hooking. Because it is a JNI method, the `insns` pointer in `gDvm` points to the address of the corresponding native function (`android_os_BinderProxy_transact`). Thus, we can store the original address, and replace the method pointer with our implementation inside the injected `so` file. After inspecting this transaction using our policy enforcement module, we can call back the original API, `android_os_BinderProxy_transact`. As shown in Figure 2, all apps are forked by a process called `zygote`. Therefore, instead of injecting an `so` file into any app, we can simply inject it to `zygote` at booting time. By using this approach, we can perform policy enforcement on any transactions which use the intercepted method.

Although the implementation of API hooking on the client side is straightforward and has been implemented in commercial systems [10, 9], this approach has a severe security limitation. This approach cannot ensure the security of the HIPS because the hooking operations are done in the *same sandbox* of the app. This implies that every operation can be modified and bypassed by the app itself. Therefore, it is dangerous if the malware writer discovers this HIPS implementation. Let us now demonstrate how to bypass this popular HIPS implementation.

3.4 Vulnerability of Existing HIPS Products

There are a number of existing HIPS products (e.g., [10, 9]) available in the market. In here, we show how to bypass these HIPS and elaborate on the vulnerability of the client-based hooking architecture (or *Client HIPS*). Client HIPS is one of the most popular Android anti-virus and protection architecture. For example, some HIPS products (e.g., LBE [10]) have more than ten million users. Moreover, Client HIPS has been pre-installed in a number of Android phones (e.g., Xiaomi smartphone).

By examining the memory structure of the installed apps and the services, we can determine whether app processes have been injected with an `so` file and a `jar` file. Figure 4 illustrates the memory structure of a normal app. There are injected `so` file and `jar` file which do not belong to the app. This implies that the Client HIPS is using the

```
# cat /proc/1459/maps                          <-- Memory structure of pid: 1459
...
6b811000-6b813000 r--s 00015000 b3:1d 927            <-- Mapped memory region
/data/data/com.lbe.security/app_hips/client.jar      <-- File path
6b831000-6b860000 r--p 00000000 b3:1d 1185
/data/dalvik-cache/data@data@com.lbe.security@app_hips@client.jar@classes.dex
6cc47000-6cc52000 r-xp 00000000 b3:1d 262788
/data/data/com.lbe.security/app_hips/libclient.so
...
```

Figure 4: Memory structure of an app.

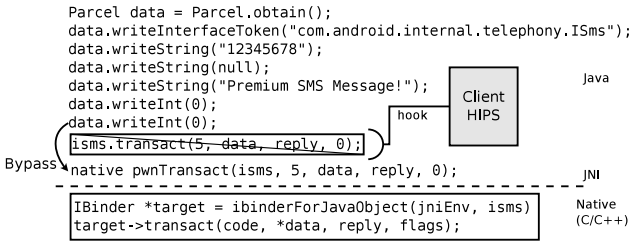

Figure 5: Workflow to bypass client HIPS.

client side API hooking to perform policy enforcement. By checking the `insns` value in the `gDvm`, the pointing address of `native transact` is `0x6CC4FED8`. In addition, from the mapped memory regions of apps, we find that the address of `native transact` resides in `0x6CC47000-0x6CC52000` region which mapped to `/data/data/com.client_hips/app_hips/libclient.so` file. Therefore, we can confirm that the system intercepts the `native transact` method in `android.os.Binder` class and points to the injected library.

Based on the discussion in Section 3.3, we know that the Client HIPS is in the same sandbox with the app. There are several ways to bypass the HIPS system. Firstly, we can modify the `insns` back to the original one to make sure that the transaction will not go through the policy enforcement. Secondly, because the `native transact` method is intercepted, malware writers can create their own transact implementation to bypass all the interceptions. Figure 5 demonstrates the workflow to bypass the Client HIPS. We implement our `native pwnTransact` method to send transactions to the target service directly. Therefore, Client HIPS cannot capture these transactions through intercepting `transact` method. In summary, malware writers can use the technique described above to determine the existence of Client HIPS (by examining the current mapped memory region `/proc/[pid]/maps` file). This shows that hooking on the client side *cannot* effectively prevent intrusions.

4. PATRONUS

In this section, we present our system, *Patronus*, a secure architecture which can prevent suspicious transactions and dynamically detect malware based on runtime transaction information. One advantage of Patronus is that we do not need to modify the Android system or repackage apps, and this facilitates easy deployment of our system. We present the system design of Patronus, and the methodologies used for intrusion prevention and dynamic detection.

4.1 API Hooking on the Server Side

As stated in Section 3.4, API hooking on the client side has a number of deficiencies. We also illustrated how to bypass detection if HIPS is implemented via API hooking on the client side. To overcome these security issues, Patronus complements the API hooking by realizing it *both* on the client side and the server side. Let us first present our design

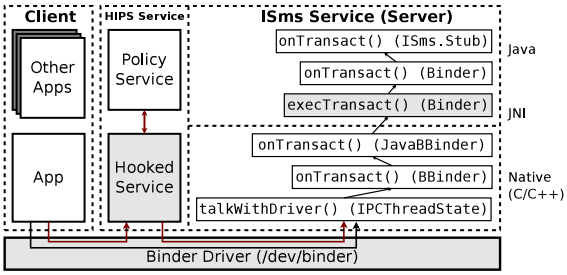

Figure 6: Call graph of ISms Binder transaction on the server side.

and implementation of API hooking on the server side.

There are two approaches to conduct API hooking on the server side. The first approach is via the *Java API hooking*, and the second approach is to *hook the* `Service Manager`. Figure 6 shows that the transaction goes through the Binder driver from the client side to the server side and will be executed by the `ISms` service. There is a JNI between the native method and the Java method. Therefore, the first approach of Java API hooking is to replace the address of this Java method, `execTransact`, in the DVM global object to our policy enforcement functions. Afterward, the hooking function will call back the original function if our policy accepts this transaction. Note that using this approach, some of the services could not be protected by the policy enforcement. Because we only hook the Java method of JNI, the interception will be invalid when the service uses native codes (e.g., camera service in Android is only implemented by the native codes). Therefore, this hooking method can only intercept *subset* of API calls.

The second approach of API hooking provides a more comprehensive solution, which is to intercept the `Service Manager`. In Section 2, we explained that the `Service Manager` manages all registered services. When an app wants to utilize a service, it should first communicate with the `Service Manager` to query for the handler of the service. Therefore, we can intercept the `Service Manager` and reply back to the app with a handler which points to the service of HIPS for policy enforcement. Figure 6 shows the basic flow of hooking on the `Service Manager` (as illustrated in red lines). When an app queries the `ISms` Service from the intercepted `Service Manager`, it will get the handler of HIPS' hooked Service, and when an app uses the handler to send an SMS message, the hooked Service will first check with the Policy Service, then an accepted transaction will return to the original `ISms` service to complete the SMS message sending operation.

Note that many malware have the network capability. However, hooking the Binder cannot monitor the network APIs because network function in Android relies on socket IPC, and the network APIs of Android (e.g., Apache HTTP Client and `HTTPURLConnection`) use socket to setup HTTP connections with servers. Hence, any app can communicate with a remote server without using the Binder if they have declared the `android.permission.INTERNET` permission in the `AndroidManifest.xml`. One approach to monitor network behaviors is to intercept socket related APIs. For example, `java.net.Socket` is a class used by other internet APIs. In Patronus, we use the technique of hooking on the client side to intercept the `connect` method in the `java.net.Socket` class to monitor the behaviors. Note that apps can also use network via the native `socket` function

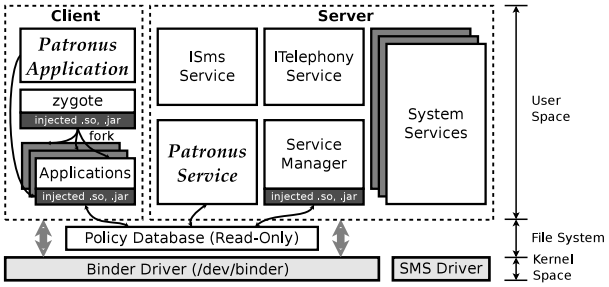

Figure 7: Architecture of Patronus.

call. Although we can use `ptrace` to inspect the `socket` and `connect` functions to cover this case, this will bring much overhead. Because we want to create a HIPS on mobile phones which are resource constrained, we use the technique of hooking on the client side to intercept the `connect` method in the `java.net.Socket` class to monitor the behaviors.

4.2 System Design of Patronus

In Section 3, we presented and demonstrated the weaknesses of existing host-based intrusion prevention system on Android. In order to overcome these security problems and provide better functionalities, we set the following two design goals for Patronus: (1) It should dynamically prevent suspicious malware intrusions and can detect malware using runtime information (i.e., transaction information); (2) the architecture needs to be *extensible* and provides a flexible methodology to detect malware without modifying Android system or the need to repackage apps. Figure 7 depicts the architecture of Patronus. The system consists of four components: (a) *Patronus application*, (b) *Patronus service*, (c) *injected files* and (d) *policy database*.

• **Patronus Application:** A process which is responsible for displaying the user interface, initiates the Patronus Service and injects `so` or `jar` files. Specifically, Patronus Application uses the `ptrace` to inject a shellcode so as to load the `so` or `jar` files, which will be injected in the `zygote` on the client side and the `Service Manager` on the server side.

• **Patronus Service:** A process which is launched as the start of the system and it is a *sticky service*, which means that the service will not be killed by the system even in the condition with limited memory resources. Hence, this process is always available and we use it to detect any malware which registers at installation. Patronus Service will intercept all transactions on the server side by receiving the redirected transactions from the hooked Service Manager so as to perform policy enforcement and dynamic detection. Furthermore, Patronus Service is the owner of the policy database process and it has permission to read and write the database. For malware, they cannot modify the policy database due to the file system protection of Linux system. Patronus Service will also record all runtime information (i.e., transaction parcels) in a temporary file so that the system can perform the dynamic detection.

• **Injected files**: There are two types of injected files. Injected files on the client side and injected files on the server side. Because malware can access injected files on the client side, these injected files only have the read permission on the policy database so as to avoid modification by the malware. The injected files on the client side use the method explained in Section 3.3.2 to intercept the `native transact` method. These injected files will first check the transaction policy in the database which resides on the client side. By checking the transaction policy, Patronus can inspect and prevent intrusions before they arrive at the server side. Using the approach of API hooking on the server side which we explained in Section 4.1, the injected files on the server side will intercept `execTransaction` method of the `Service Manager`. So whenever there is any query from the client side, it will reply the Patronus service handler back to all queries. Therefore the transactions on the server side will first arrive at the Patronus Service, and they will be checked based on the policy database before dispatching these transactions to the corresponding servers. In our current implementation, we implement our policy checking and enforcement in API hooking, and use the Android SDK (e.g., `Popup Window` class to show a popup window on the apps).

• **Policy database:** This database contains the policy enforcement rules for each type of transaction of all apps. In our current implementation, we store these rules in an `XML` file. Using the Patronus Service, users can block transactions from some untrusted apps. Specifically, Patronus App communicates with Patronus Service to write the user-defined policy into the policy database. Users can enable some trusted apps (e.g., Google apps) or deny some transactions (e.g., `sendText`, `requestLocationUpdates`) if they do not intend to request these sensitive services. Due to the security consideration, this policy database belongs to the Patronus Service user and can only be modified by the trusted Patronus Service. Hence, with the protection of Linux file system, malicious app cannot destroy the policy database.

With these four components, Patronus can inspect the transactions either on the client side or on the server side. By analyzing each transaction, Patronus can prevent intrusions and detect malware.

4.3 Intrusion Prevention

Patronus can inspect transactions dynamically and will notify users if it encounters some intrusive transactions. In the Android system, there are hundreds of transaction types. The intrusive transactions are the transactions which can invoke some dangerous behaviors, for example, some malware subscribe premium service by sending an SMS message, and some spyware steal the location information using GPS service. Table 1 lists some intrusive transactions with the transaction descriptors, codes and codes name. For instance, the `requestLocationUpdates` transaction in the `android.location.ILocationManager` can gain the location information based on GPS. When this intrusive transaction is not initiated by users or without users' permissions, Patronus will alert the users using a popup window so that the users can pay attention to this transaction. Users can also deny this transaction at runtime. Patronus will monitor the transaction stream based on the intrusive transaction list. It is important to note that users or security analysts can add and/or delete rules in the Patronus policy database to ignore some intrusive transaction from some apps. For example, a third-party SMS mobile app can send, read and write SMS messages. If users are certain about the authenticity and security of this mobile app, users can delete rules for this SMS related intrusive transactions in order to eliminate the popup alerts. Also, the list is defined by the system (i.e., anti-virus software) which contains the sensitive transactions. Legitimate transactions (e.g., opening window, changing Activity status) will not have any malicious behaviors.

Table 1: Intrusive Transaction List

TD	TC	TC Name
com.android.internal	4	sendData
.telephony.ISms	5	sendText
	...	...
com.android.internal	1	dail
.telephony.ITelephony	2	call
	28	getCellLocation
	...	...
android.location	1	requestLocationUpdates
.ILocationManager	5	getLastLocation
...	...	...

TD: Transaction Descriptor, TC: Transaction Code

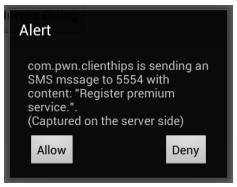

Figure 8: Alert of intrusive transaction (`sendText`).

Patronus uses two procedures to ensure the system security. The first is on the client side. Before the intrusive transaction goes through the Java layer and the native layer, Patronus will first inspect the transaction and check if there is any permission affiliated with this transaction. If there is no pre-defined policy, Patronus will pop up a window as an alert. The alert contains the transaction type, calling app of the transaction and transaction content (e.g., destination number and content of the SMS message) if needed. If the user allows this transaction, this transaction will proceed to the server side. The Patronus Service will not perform the second inspection. If the server finds that the transaction was not checked by the injected files on the client side, this implies that the app attempts to bypass the system or uses the native code. Then Patronus Service will perform policy enforcement to ensure the security of the transaction.

In Section 3.4, we illustrated several methods to bypass some existing HIPS. Patronus can easily detect and defend against these type of attacks. Firstly, if the malware attempts to bypass the transaction inspection on the Java layer, the system will perform the second inspection on the server side. Since the second verification is on the server side, the malware cannot bypass this step due to the process isolation feature in Android. Secondly, the Patronus policy database is also an important component of the system. The policy database can be read by any app but only Patronus has the write permission to the database. This mechanism guarantees that the policy database will not be compromised or contaminated by malware. Hence, Patronus provides a secured and complete intrusion prevention functionality. Consider the same mobile app (the one we discussed in Section 3.4) which bypassed the detection of a commercial HIPS product. When we execute this app on a mobile phone with Patronus, our system can intercept the intrusive transaction and the alert will popup, as shown in Figure 8.

4.4 Dynamic Detection

Besides intrusion prevention, Patronus can detect malware at runtime and block malicious behaviors before the malware infects the system. By using the runtime information to detect malware, Patronus can determine malware accurately and defend against obfuscation effectively. The dynamic detection consists of two steps: (1) *malware transaction forensics* and (2) *two-phase dynamic detection*.

4.4.1 Malware Transaction Forensics

Malware transaction forensics is a procedure to trigger the malicious behaviors and to collect the runtime transaction information. This can help the system to conduct dynamic detection. It includes two steps: *malware triggering* and *malware transaction tagging*.

Once the app is determined to be a potential malware (using existing static and dynamic analysis), the analysts can execute the malware on a test phone with the Patronus system. The analysts can manually trigger the malicious behaviors, and the system will record the transaction information (i.e., transaction descriptor and transaction code). If the transaction is an intrusive transaction, Patronus also records the content of the transaction parcel.

After collecting the transactions, the analysts can tag the malicious transactions in the set of suspicious intrusive transactions. The malicious transactions are used to invoke malicious behaviors by malware, such as sending premium SMS message, tracking location information and stealing contact information. Because the contents of transactions are evidence of malicious behaviors, these will be used to determine the type of malware we are analyzing. We want to emphasize that malware transaction tagging will not add more workload for analysts since we only tag malicious transactions and the system already filtered many other legitimate transactions. In our experiments, the number of intrusive transactions for a malware is less than five and most of them are tagged as malicious transactions. Let us now define *transaction footprint*, which we use for malware detection.

Definition 1. *A transaction footprint is a set of transaction information tuples* $S = \{T_1, T_2, \ldots, T_n\}$ *over runtime transactions where:*

- *The index represents the unique id of the combination of transaction descriptor and code.*
- *The transaction information tuple* $T_i = (N_i, F_i, C_i)$*, where* N_i *is the number of invocations of transaction* i*,* F_i *is the boolean flag tagging the malicious transaction* i*,* C_i *represents the content of the transaction parcel.*

We can use the transaction footprint for dynamic detection. Firstly, it describes the runtime behaviors of a group of similar malware. Secondly, it also contains evidence of malicious transactions. Due to these two factors, we propose a two-phase dynamic detection method to effectively detect the malware on an Android mobile phone at runtime.

4.4.2 Two-phase Dynamic Detection

The dynamic detection contains two phases: *correlation detection* and *transaction footprint comparison*. Because only the intrusive transactions can initiate malicious behaviors. For efficiency, the detection will be triggered when the system encounters an intrusive transaction. The system will use the transaction footprint collected at runtime and the malware footprint to detect the malware.

Phase one. Define V as the transaction vector over a transaction footprint S where $V = [v_1, v_2, \ldots, v_n]$, $(v_i = N_i)$. In phase one, we collect $V_{runtime}$ which is a transaction vector over the transaction footprint for the runtime app. Let

$V_{malware}$ be the transaction vector over the transaction footprint in malware transaction forensics. In Patronus, we use the *Pearson correlation* as the similarity score r so as to determine the correlation between the app and our malware samples. Formally, the similarity score r is:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}. \quad (1)$$
$$where\ X = V_{runtime},\ Y = V_{malware}$$

Because transactions of an app represent the behaviors when interacting with other apps, the higher r implies that the known malware in our database and the mobile app under testing are very similar. The system will calculate r with the transaction footprint in the malware database.

Phase two. If there is any r higher than a pre-defined threshold (the threshold we use in our experiment was set as 0.9), we treat the detected app as a suspicious app. For the suspicious app, we conduct the phase-two detection to ensure the accuracy of our detection. In phase two, Patronus will use the content of malicious transactions $[C_1, C_2, \ldots, C_i]$, $(F_i = true)$ in the transaction footprint S to perform further verification. The system will compare the decisive fields in the content of each intrusive transaction from the suspicious app with the selected footprints. The decisive fields are values indicating the malicious behaviors. In general, the content of a transaction contains several fields, the semantics of the fields for each transaction can be different. For example, `sendText` transaction has four fields which indicate the destination address, source address, text and intent flag. The decisive fields for a malicious transaction are values which determine the malicious properties. For example, the destination field (number of premium SMS service) is the decisive field of a malicious sendText transaction.

By using the above two-phase detection, Patronus can detect malware based on the runtime information effectively and prevent the malware before it infects the system or steal users' privacy. Since the transaction information we obtain have more semantic information than the low-level system calls using the `ptrace`-based dynamic analysis system, this makes Patronus achieving a much higher accuracy in detecting intrusive apps. Furthermore, because one transaction procedure usually only contains tens of system calls, the performance overhead of transaction-based detection will be less than system-call-based detection. Moreover, traditional static analysis systems have to scan the apps offline, while Patronus can perform detection online.

5. LARGE SCALE EVALUATION

In this section, we analyze the capabilities of intrusion prevention and dynamic detection on large number of mobile apps. We also evaluate the performance overhead and power consumption introduced by Patronus.

5.1 Capability Evaluation

We downloaded the top 500 legitimate mobile apps from Google Play and use them as the base for our evaluation database. In addition, we also collected three malware families from [2] and [11] including 213 `BaseBridge` samples, 9 `FakeAV` samples and 15 `MobileTx` samples. We also define an intrusive transaction list (`https://www.cse.cuhk.edu.hk/~mssun/pub/intrusive_transaction_list.pdf`) including 49 transactions within eleven services.

Table 2: Top 10 Intrusive Transactions

Transaction Name	Total #
CALL_TRANSACTION	3,508
REGISTER_RECEIVER_TRANSACTION	2,960
START_ACTIVITY_TRANSACTION	1,734
TRANSACTION_getDeviceId	1,732
GET_CONTENT_PROVIDER_TRANSACTION	1,400
QUERY_TRANSACTION	1,303
TRANSACTION_getSubscriberId	333
TRANSACTION_requestLocationUpdates	228
INSERT_TRANSACTION	139
TRANSACTION_getCallState	90

Table 3: Transaction Statistics

Service	Top 500	BaseBridge	FakeAV	MobileTx
PackageManager	0	0	0	0
Telephony	90	0	0	0
TelephonyRegistry	52	1	0	17
ContentProvider	4,996	131	0	0
LocationManager	228	7	0	25
ActivityManager	6,156	295	0	98
AudioService	9,781	1,453	37	40
Sms	0	0	0	49
PhoneSubInfo	2,208	622	0	0
NotificationManager	639	336	161	0
PhoneStateListener	0	0	0	0
Total transaction	724,185	66,229	548	3,920
Intrusive transaction	24,150	2,845	198	229
Percentage	3.33%	4.30%	3.61%	5.84%

To evaluate Patronus, we automatically install the top 500 legitimate apps and malware samples into Android device (Nexus 5) with Patronus installed. Then we use `monkey` tool [12] to generate 500 pseudo-random user events such as clicks, touches, gestures or system-level events, into the device to trigger various behaviors. Patronus will record the transactions which are in the intrusive transaction list. Table 2 shows the top ten intrusive transactions and their total numbers recorded in our evaluation. Most of these transactions are dangerous and related with privacy operation. For example, `CALL_TRANSACTION`, `GET_CONTENT_PROVIDER_TRANSACTION`, `QUERY_TRANSACTION` and `INSERT_TRANSACTION` are operations (e.g., querying and updating) on databases which may contain users' personal information. While `TRANSACTION_requestLocationUpdates` can get current location information. Table 3 depicted detailed statistics of these intrusive transactions on all our downloaded apps. Note that the percentage of intrusive transaction is only around 4%. Therefore, the alert notifications of intrusive transaction will not annoy users in practice.

To evaluate the capability of dynamic detection, we first conduct transaction forensics on the malware families in our evaluation database. We automatically run these malware samples along with 500 legitimate apps to analyze the effectiveness of our detection algorithm. During the testing, we also generated 500 pseudo-random user events such as clicks, touches, or gestures for each app so to trigger different routines. We calculate the distribution of correlation scores and they are illustrated in Figure 9. From this distribution, we choose 0.9 as our *threshold* in the two-phase detection which effectively separate malware and legitimate app. Table 4 shows the detection results of malware samples within the legitimate apps. We compute the true positive (TP), true negative (TN), false positive (FP), false negative (FN), precision and F-score to illustrate our results. From these results, we observe that only few legitimate apps is reported as malware in the first phase of detection. We like to

Table 4: Detection Results

Malware	# of Samples	TP	TN	FP	FN	Precision	F-score
BaseBridge	213	186	495	5	27	0.87	0.92
FakeAV	9	9	500	0	0	1	1
MobileTx	15	11	494	6	4	0.65	0.69

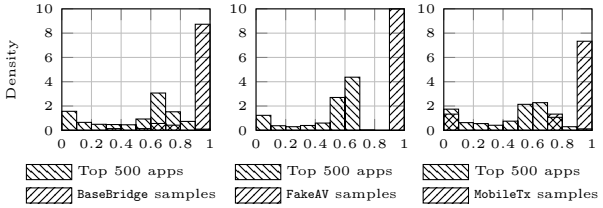

Figure 9: Distribution of correlation scores for BaseBridge, FakeAV and MobileTx samples.

note that for MobileTx families, some of the samples crashed in testing due to the bugs of the malware. Therefore, we obtained four false negative in our evaluation. In summary, Patronus can effectively prevent intrusions and detect existing malware.

5.2 Performance Evaluation

To measure the overhead introduced by Patronus, we use Quadrant Standard Edition v2.1.1 [6] to evaluate the general-purpose benchmark for CPU, memory, I/O, 2D and 3D graphics. The experiments are conducted on a LG Nexus 5 (Qualcomm Snapdragon 800 2.26GHz CPU, Adreno 330 450MHz GPU, 2300mAH battery) running Android 4.4.2 (KitKat) on a Linux 3.4.0 kernel. Table 5 shows the benchmark results. The "Baseline" means that the benchmarking was done on the device without Patronus. Compared with the baseline, Patronus only has a small impact on all but I/O intensive apps. However, we want to note that I/O operations are not the main operations for many mobile apps.

Because Patronus inspects transactions, we implement a transaction intensive app to evaluate its performance. The test app uses IPhoneSubInfo transaction to request device ID (transaction #: 1). The app will repeatedly send the transaction for one thousand times. We calculate the execution time of these transactions. The time on the device without Patronus is 890 milliseconds, while the execution time for a device with the Patronus installed increase to 988 milliseconds. This shows that the presence of Patronus will only bring about 11.1% performance overhead.

We also measure the battery overhead introduced by Patronus. We conduct two sets of experiments. In the first experiment, we check the battery usage of a fully-charged Nexus 5 in the standby mode for 24 hours. The device without Patronus has 94% battery left, and the device with Patronus installed shows 93%. This shows that Patronus only introduces a slight overhead. In the second experiment, we keep playing games on a fully-charged Nexus 5 for one hour. The results show that the device with Patronus uses 3% more battery than the original setting. In summary, Patronus not only provides a comprehensive intrusion protection, but also has a negligible impact on power consumption.

6. RELATED WORK

There is a growing interest on how to provide security and privacy on smartphones. For Apple's iOS, Wang *et al.* [51] discovers a severe security architecture problem. While most security issues are also observed on the Android platform.

Table 5: Benchmark Results

Test	Baseline	Patronus	Overhead
Total	8,914	8,285	7.1%
CPU	20,383	20,205	0.9%
Memory	14,354	13,211	8.0%
I/O	7,274	6,482	10.9%
2D	333	330	0.1%
3D	2,230	2,195	1.6%

Permission re-delegation problem is discussed in [20, 31] and some researchers propose possible solutions [28, 16, 22, 36]. Felt *et al.* [27, 15] systematically analyzes the permission abuse problem. AdRisk [33] reveals the potential privacy risks of advertisement libraries. Luo *et al.* [37] identifies the problem of Android WebView component. Zhou *et al.* [63] conducts a comprehensive study on the characterization and evolution of Android malware. Wu *et al.* [52] and Zheng *et al.* [60] analyze the security issues on the customized firmwares. Our study mainly focuses on the dynamic detection of Android malware and Android protection.

There are a number of works which propose to use HIPS on the PC platform [46, 43, 42]. However, the architectures of PC and mobile devices are very different. Therefore, few systems are proposed and targeted specifically for mobile operating systems. FireDroid [49] is a ptrace-based system call interception system. Because system calls are low level function calls interacting with kernel, one API call can generate a number of system calls. Therefore, ptrace-based system cannot accurately intercept all intrusive operations. TaintDroid [24] and PiOS [23] can track runtime privacy leaks on Android and iOS. DroidScope [55] provides a Dalvik semantic view for dynamic analysis. CrowDroid [18] is a detection system based on runtime system call. VetDroid [57] utilizes permission usage to study undesirable dynamic behaviors. AppIntent [56] aims to detect privacy leakage using symbolic execution to reduce the code search space. Android DDI [40] introduces the method to instrument the DVM. These systems are for malware dynamical analysis which are mainly used by malware analysts. However, our system aims to prevent intrusions on the user end. Aurasium [54] and RetroSkeleton [21] are two host-based systems which use repackaging technique. FlaskDroid [17] provides mandatory access control on SE Android. Compared with our system, they need to either repackage the apps or modify the system structure.

For static analysis, some systems aim to detect suspicious apps, zero-day malware and evaluate anti-virus software. DroidMOSS [62] and DroidRanger [65] can identify the potential repackaged apps or zero day malware in the third-party markets. DroidAnalytics [61] proposes a three-level signature to analyze malware to defend against obfuscation. ADAM and DroidChameleon [59, 48] discuss approaches to bypass anti-virus engines. MAST [19] analyzes the market-scale malware. Zhou *et al.* [64] proposes a system to find content leaks and pollution vulnerabilities. These systems can help the analysts to investigate the malicious behaviors at the code level.

7. CONCLUSION & FUTURE WORK

In this paper, we systematically analyze three popular frameworks of HIPS, and discuss their implementations, strengths and weaknesses. We also demonstrate how to bypass a popular HIPS software in the market. Moreover, we propose

a secure HIPS architecture and implement Patronus which can prevent intrusion and can dynamically detect malware. With the two-phase dynamic detection, Patronus utilizes runtime information to detect existing malware. We conduct extensive experiments to demonstrate the intrusion prevention and dynamic detection capabilities of Patronus, and demonstrate the Patronus only incurs small overhead in executing mobile apps. For future work, we plan to utilize cloud service along with uploaded runtime information collected by Patronus to improve the detection capability.

8. REFERENCES

[1] 360 one click root. http://shuaji.360.cn/root/index.html.
[2] Android Malware Genome Project. http://malgenomeproject.org.
[3] APKfuscator. https://github.com/strazzere/APKfuscator.
[4] Apktool. https://code.google.com/p/android-apktool/.
[5] App Shield. http://www.wandoujia.com/apps/com.gmail.exathink.appshield.
[6] Aurora Softworks quadrant standard edition. https://play.google.com/store/apps/details?id=com.aurorasoftworks.quadrant.ui.standard.
[7] cyanogenmod. http://www.cyanogenmod.org.
[8] DroidBox. https://code.google.com/p/droidbox/.
[9] Jinshan mobile duba. http://m.duba.net/.
[10] Lbe secrity guard. http://www.lbesec.com/.
[11] mobile malware mini dump. http://contagiominidump.blogspot.com/.
[12] monkey. http://developer.android.com/tools/help/monkey.html.
[13] Qihoo 360 mobile guard. http://shouji.360.cn/.
[14] smali/baksmali. https://code.google.com/p/smali/.
[15] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *CCS*, 2012.
[16] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.
[17] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security*, 2013.
[18] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. CrowDroid: behavior-based malware detection system for android. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.
[19] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: triage for market-scale mobile malware analysis. In *ACM WiSec*, 2013.
[20] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.
[21] B. Davis and H. Chen. RetroSkeleton: retrofitting android apps. In *MobiSys*, 2013.
[22] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smart phone operating systems. In *USENIX Security*, 2011.
[23] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in ios applications. In *NDSS*, 2011.
[24] 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 *OSDI*, 2010.
[25] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011.
[26] F-Secure. Threat report h2 2013.
[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS*, 2011.
[28] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *USENIX Security*, 2011.
[29] J. Forristal. Android: One root to own them all. In *Blackhat USA 2013*, 2013.
[30] Google. Platform versions. http://developer.android.com/about/dashboards/index.html.
[31] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.
[32] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day android malware detection. In *MobiSys*, 2012.
[33] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.
[34] N. Harbour. Win at reversing: Api tracing and sandboxing through inline hooking. In *DEFCON*, 2009.
[35] IDC. Apple cedes market share in smartphone operating system market as android surges and windows phone gains. http://www.idc.com/getdoc.jsp?containerId=prUS24257413.
[36] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *CCS*, 2012.
[37] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*, 2011.
[38] W. Luo, S. Xu, and X. Jiang. Real-time detection and prevention of android sms permission abuses. In *Proc. of the first international workshop on Security in embedded systems and smartphones*, 2013.
[39] McAfee Labs. McAfee threats report: Second quarter 2013. Technical report, McAfee Labs, 2013.
[40] C. Mulliner. Android DDI: Introduction to dynamic dalvik instrumentation. In *The 11th Annual HITB Security Conference in ASIA*, 2013.
[41] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda. Patchdroid: scalable third-party security patches for android devices. In *ACSAC*, 2013.
[42] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*.
[43] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Internet Society*, 2005.
[44] P. Olson. Your smartphone is hackers' next big target. http://edition.cnn.com/2013/08/26/opinion/olson-mobile-hackers, August 2013.
[45] A. Police. App Ops: Android 4.3's hidden app permission manager, control permissions for individual apps.
[46] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.
[47] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: automatic security analysis of smartphone applications. In *CODASPY*, 2013.
[48] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: evaluating android anti-malware against transformation attacks. In *ASIACCS*, 2013.
[49] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. Firedroid: hardening security in almost-stock android. In *ACSAC*, 2013.
[50] T. Strazzere. Dex education: Practicing safe dex. In *Blackhat USA 2012*, 2012.
[51] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: when benign apps become evil. In *USENIX Security*, 2013.
[52] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *CCS*, 2013.
[53] xda-developers. PDroid. http://forum.xda-developers.com/showthread.php?t=1357056.
[54] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *USENIX Security*, 2012.
[55] L. K. Yan and H. Yin. DroidScope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.
[56] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, 2013.
[57] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *CCS*, 2013.
[58] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proc. of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.
[59] M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In *DIMVA*, 2013.
[60] M. Zheng, M. Sun, and J. Lui. DroidRay: a security evaluation system for customized android firmwares. In *ASIACCS*, 2014.
[61] M. Zheng, M. Sun, and J. C. S. Lui. DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *TrustCom*, 2013.
[62] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the second ACM conference on Data and Application Security and Privacy*, 2012.
[63] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.
[64] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS*, 2013.
[65] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.