

CopperDroid: Automatic Reconstruction of Android Malware Behaviors

Kimberly Tam*, Salahuddin J. Khan*, Aristide Fattori[†], and Lorenzo Cavallaro*

*Systems Security Research Lab and Information Security Group
Royal Holloway University of London

[†]Dipartimento di Informatica
Università degli Studi di Milano

Abstract—Mobile devices and their application marketplaces drive the entire economy of the today’s mobile landscape. Android platforms alone have produced staggering revenues, exceeding five billion USD, which has attracted cybercriminals and increased malware in Android markets at an alarming rate. To better understand this slew of threats, we present CopperDroid, an automatic **VMI-based** dynamic analysis system to **reconstruct the behaviors of Android malware**. The novelty of CopperDroid lies in its agnostic approach to identify interesting OS- and high-level Android-specific behaviors. It reconstructs these behaviors by observing and dissecting **system calls** and, therefore, is resistant to the multitude of alterations the Android runtime is subjected to over its life-cycle. CopperDroid *automatically* and *accurately* reconstructs events of interest that describe, not only well-known process-OS interactions (e.g., file and process creation), but also complex intra- and inter-process communications (e.g., SMS reception), whose semantics are typically contextualized through complex Android objects. Because CopperDroid’s reconstruction mechanisms are agnostic to the underlying action invocation methods, it is able to capture actions initiated both from Java and native code execution. CopperDroid’s analysis generates detailed *behavioral profiles* that abstract a large stream of low-level—often uninteresting—events into concise, high-level semantics, which are well-suited to provide insightful behavioral traits and open the possibility to further research directions. We carried out an extensive evaluation to assess the capabilities and performance of CopperDroid on more than 2,900 Android malware samples. Our experiments show that CopperDroid faithfully reconstructs OS- and Android-specific behaviors. Additionally, we demonstrate how CopperDroid can be leveraged to disclose additional behaviors through the use of a simple, yet effective, app *stimulation* technique. Using this technique, we successfully triggered and disclosed additional behaviors on more than 60% of the analyzed malware samples. This *qualitatively* demonstrates the versatility of CopperDroid’s ability to improve dynamic-based code coverage.

I. INTRODUCTION

With more than a billion Android-activated devices [25] and over a billion of monthly-active Android users [19], mobile platforms have clearly become ubiquitous with trends showing such a pace is unlikely to slow down. Application

marketplaces, such as Google Play, drive this entire economy of mobile applications (apps). For instance, with more than 50 billion downloaded apps [39], Google Play has generated revenues exceeding 5 billion USD [23] in 2013. Such a wealthy and unique ecosystem, with high turnovers and access to sensitive data, has unfortunately spurred an alarming growth in Android malware. Privacy breaches (e.g., access to address book and GPS coordinates) [47], monetization through premium SMS and calls [47], and colluding malware to bypass 2-factor authentication schemes [12] have become real threats. Recent studies also report how easily mobile marketplaces have been abused to host malware or seemingly legitimate applications embedding malicious components [46].

The nature of Android apps makes it difficult to rely on standard, traditional, dynamic system call malware analysis systems *as is*. While Android apps are generally written in the Java programming language and executed on top of the Dalvik virtual machine (VM) [7], native code execution is possible, for instance, via the Java Native Interface (JNI). This mixed execution model seems to suggest the need to reconstruct, and keep in sync, different semantics through virtual machine introspection (VMI) [20] for both the OS and Dalvik views, as recently shown in [43]. More recently, Zhang et al. stressed this concept further in [43] and pointed out that traditional system call analysis is **ill-suited** to characterize the behaviors of Android apps as it misses high-level Android-specific semantics and fails to reconstruct inter-process communications (IPC)¹ and remote procedure call (RPC) interactions, which are essential to understanding Android application behaviors.

In a significantly different line of reasoning from [17], we observed that system call invocations remains central to both low-level OS-specific and high-level Android-specific behaviors. However, a naive analysis of system calls would miss the rich semantic of Android-specific behaviors. This is where the novelty of our approach lies; our techniques enable seamless and automatic dissection of complex inter-process communications resulting in the automatic deserialization of complex Android objects. This enables us to reconstruct behaviors of Android applications at multiple levels of abstraction from a single point of observation (i.e., system calls). More importantly, our simplified approach makes the analysis agnostic to the runtime system, freeing the analysis engine from playing catch-up with each change to the system (our techniques work transparently on systems using the Dalvik VM and ART).

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author’s employer if the paper was prepared within the scope of employment.
NDSS ’15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
<http://dx.doi.org/10.14722/ndss.2015.23145>

¹Android IPC is also known as inter-component communication (ICC) [17]. We will use IPC and ICC interchangeably throughout the text.

Our framework, titled CopperDroid², is an approach built on top of QEMU [6] to *automatically* perform out-of-the-box (VMI-based) dynamic analysis and reconstruct the behaviors of Android malware. Through CopperDroid, we demonstrate that all behaviors manifest themselves *through the invocation of system calls* [17], and that we can *faithfully* reconstruct Android malware behaviors regardless of whether it is initiated from Java or native code. However, to automatically and reliably reconstruct system call semantics, including IPC, RPC, and (complex) Android objects, is a challenging task. In fact, high-level Android object information is not directly available at the system call level. Moreover, to guarantee transparency against inner changes (or whole replacement) of the Android runtime (e.g., Dalvik VM, ART), its *direct introspection* must be avoided. To address this challenge, we introduce the concept of the *unmarshalling Oracle*, which seamlessly recreates complex Android objects to enrich the semantics of the reconstructed OS- and Android-specific behaviors. This is where the real value of CopperDroid lies. A preliminary description of CopperDroid, focused on introducing basic analysis capabilities (e.g., system call tracking), has already appeared in our workshop paper [36]. In this paper, we present our mature research efforts, including the following contributions:

1) *Automatic IPC Unmarshalling*: We introduce CopperDroid and present the design and implementation of a novel, practical, oracle-based technique to *automatically* and *seamlessly* reconstruct Android-specific objects involved in system call-related IPC/ICC and RPC interactions. Our approach avoids manual development efforts and transparently addresses the challenge of dealing with the ever increasing number of complex Android objects introduced in different Android releases. The Oracle allows CopperDroid to perform large-scale, automatic, and faithful reconstruction of Android apps behaviors (Sections IV), suitable to enable further research, including Android malware detection.

2) *Value-based Data Flow Analysis*: To abstract sequences of related low-level system calls to higher-level semantics (e.g., network communications, file creation) and enrich our reconstructed behavioral profiles, we automatically build data dependency graphs over sets of observed system calls (including those referring to IPC/ICC and RPC mechanisms) and perform value-based forward slicing to cluster data-dependent system calls. This gives us the ability to automatically recreate the resources associated with a stream of sliced system calls. Moreover, this further simplifies the understanding of the behavioral profiles by summarizing its semantic, and provides access to the reconstructed resources, which can be fed back to CopperDroid, downloaded for additional inspection, or analyzed by complementary systems (Section V-B).

3) *Behavioral Reconstruction*: We provide a thorough evaluation of CopperDroid’s behavioral reconstruction capability on more than 2,900 Android malware samples provided by different sources [11], [30], [49]. Furthermore, our experiments show how a simple yet effective malware *stimulation* strategy (Section V-A) allows us to disclose an average of 25% of additional behaviors on more than 60% of the analyzed samples, qualitatively improving dynamic analysis behavioral reconstruction capabilities with a very limited effort and negligible overhead (Section VI).

It is our belief that CopperDroid’s unified reconstruction significantly contributes to the state-of-the-art reconstruction of Android malware behavior. Although our system could have been built on top of DroidScope [43], a general-purpose VM-based out-of-the-box framework to build dynamic analysis for Android, its source code was not available when we began our development. Furthermore, DroidScope offers basic hooking mechanisms and relies on keeping a synchronized 2-level VMI (for OS and Dalvik VM semantics), which makes it complex and harder to port onto different versions of Android OSes (for instance, VMI-related offsets tend to vary more frequently in the Dalvik VM rather than in the kernel). Our approach, on the other hand, is unaffected by such changes. *VetDroid* [44] presents a framework to construct permission-use behavior graphs, which highlight how applications use permissions to access system resources, and how such resources are utilized by the application. Although an interesting approach, VetDroid requires a quite intrusive modification of the Android system (both Dalvik VM, Binder, and Linux kernel), which hampers the ability to easily port the system to different Android versions. In addition, VetDroid builds on top of TaintDroid and, therefore, inherits its drawbacks [10], [37].

Conversely, CopperDroid’s unified analysis does not require complex introspection, but only needs to collect the system calls invoked by the processes running on the monitored system. Hence, all analyses are performed outside the VM. This flexibility allows our system to be largely decoupled from any specific Android environment, enabling seamless integration across different Android versions. For instance, we have successfully run CopperDroid on Froyo, Gingerbread, Jelly Bean, KitKat, and the newest Lollipop (i.e., Android 5.0) version with no modification to the Android system and minimal, automatically-generated, alterations to CopperDroid. This is particularly remarkable as Lollipop has substituted the existing Dalvik’s just-in-time compiler (in all previous Android OS versions) with the new, faster, ahead-of-time compiler ART [34]. While all Dalvik-level based analysis engines will be affected by this change, CopperDroid was *unfazed*.

The enhancements presented in this paper are central to CopperDroid’s VMI-based system call-centric analysis—whose automatic IPC and RPC dissection and Android-specific objects (and thus behaviors) reconstruction are a key aspect. Furthermore, we evaluated CopperDroid on a large and diverse datasets to demonstrate the range of behaviors (e.g., shell execution, IPC) it can abstract.

II. BACKGROUND: THE ANDROID SYSTEM

Android applications are typically written in the Java programming language and then deployed as Android Packages archive (APKs). Each application runs in a separate userspace process [2] as an instance of the Dalvik virtual machine (DVM) [7] and usually with a distinct user and group ID. Although isolated within their own sandboxed environment, these applications can interact with other applications and the system through well-defined APIs. Every APK is also considered to be a self-contained app that can be logically decomposed into one or more components (e.g., activities, services, and broadcast receivers). Each component is generally designed to fulfil a specific task (e.g., GUI-related actions, notification receiver) and is invoked either by the user or the OS.

²Based on an informal British term for police officers as well as the metal.

Activities, services, and broadcast receivers are activated by intents, i.e., asynchronous messages exchanged between individual components to request an action. Activity and service intents specify actions to be performed. Conversely, broadcast receiver intents define the received event and are delivered to the interested receivers.

Components must also be declared within the Android manifest, a XML file that must be included in every APK and contains a number of interesting information that can indeed provide preliminary insights about an app’s maliciousness [48]. A manifest also declares the set of permissions the application requests, along with the hardware and software features the application uses. In addition, a manifest may include intent filters, i.e., the set of intents the app is willing to handle.

A. Inter-process communications and remote procedure calls

The Android system implements the principle of least privilege by providing a sandbox for each installed application. Therefore, one process may not manipulate the data of another process and can only access system components if it explicitly requested the corresponding permission(s) in the manifest. Nevertheless, apps often need a way to communicate to each other, e.g., an app can request the permission to send SMS through the appropriate service using a remote method call.

The Android system relies on Binder as its optimized inter-process communication (IPC) and remote procedure call (RPC) mechanism. When IPC is initiated from a component \mathcal{A} to a component \mathcal{B} (both may be two components of the same app) the calling thread in \mathcal{A} will wait until the next available thread in the thread pool of \mathcal{B} replies with the results. The calling thread returns as soon as it receives such a result. The data sent in the transaction is a *Parcel*, a buffer of often serialized (marshalled) flattened data and meta-data information. Dispatching of the message between \mathcal{A} and \mathcal{B} takes place by means of a *ioctl* system call handled by the Binder kernel driver. When a service needs to provide a binding, it must define a client-server interface that allows applications to bind and interact with it; such an interface is called *bound service*. If the service is used by other apps, or across separate processes, and requires multithreading, then the interface is usually defined by the Android Interface Definition Language (AIDL). Thanks to the AIDL of a service, clients know how to communicate with it, i.e., which remote methods can be invoked and what parameters (and types) they take. Any kind of request and data exchanged among clients and services go through Binder, whose thorough analysis therefore allows us to identify Android-specific behaviors (e.g., sending an SMS and accessing private information). Even when passing-by-reference rather than data marshalling is used, data is still sent through IPC channels in a flattened Binder.

III. OVERVIEW OF COPPERDROID

In our framework, an unmodified Android system runs on top of a modified Android emulator (CopperDroid emulator), which is built on top of QEMU [6], as shown in Figure 1. To this end, we have enhanced (i.e., instrumented) the Android emulator to enable system call tracking and support our out-of-the-box system call-centric analyses. It is worth pointing out that *all our analyses* are executed outside the CopperDroid

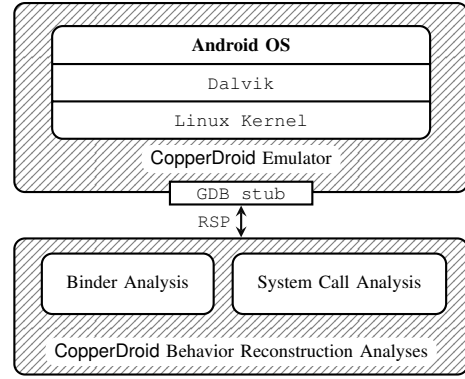


Figure 1: CopperDroid architecture. An unmodified Android image runs on top of CopperDroid emulator (a modified QEMU emulator), which collects system call information that are sent to CopperDroid analyses to perform out-of-the-box system calls (including Binder) behaviors reconstruction.

emulator; we rely on well-known virtual machine introspection (VMI) [20] techniques to fill the semantic gap between our emulator and the Android OS³. This provides a transparent environment to *automatically* perform out-of-the-box dynamic behavioral analysis on any kind of Android application (and, for this work, we are specifically interested in Android malware). To this end, CopperDroid presents a *unified* analysis to characterize low-level OS-specific and high-level Android-specific behaviors.

A. Tracking System Call Invocations

Tracking system call invocations is at the basis of virtually all dynamic malware behavioral analysis systems [24], [27], [41]. Most—if not all—of such systems implement a form of VMI to track system call invocations on a virtual x86 CPU. Although similar, the Android ARM architecture presents a few characteristics that may challenge VMI-based system call invocations tracking and are therefore worth elaborating on.

The ARM ISA provides the *swi* instruction for invoking system calls, which causes user-to-kernel transition by triggering a software interrupt. To track system call invocations, we instrument QEMU to intercept when the *swi* instruction is executed. That instruction is not (dynamically) binary translated and can therefore be easily intercepted when QEMU handles the software interrupt. Of course, it is also of paramount importance to detect when a system call is about to return in order to save its return value and enrich our analysis with additional semantic information. Usually, the return address of a system call invocation instruction *swi* is saved in the link register *lr*. While it seems natural to set a breakpoint at that address to retrieve the system call return value, a number of system calls may actually not return at all (e.g., *exit*, *execve*). The generic approach CopperDroid adopted is to intercept CPU privilege-level transitions. In particular, CopperDroid detects whenever the *cpsr* register switches from supervisor to user mode (*cpsr_write*), which allows it to retrieve system call return values, if any (further details are available in [36]).

³We refer the reader to our preliminary workshop paper [36] for details.

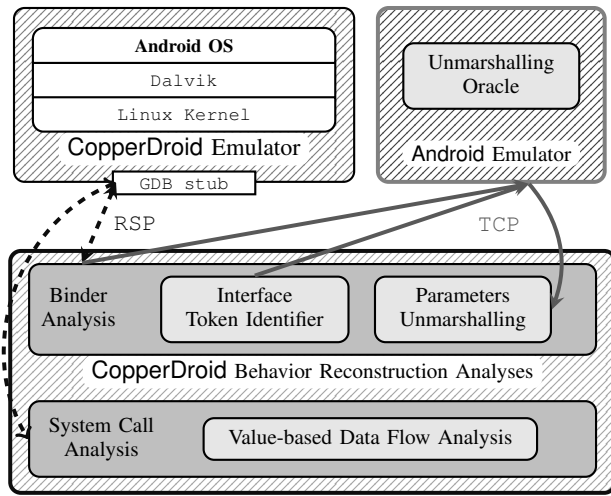


Figure 2: CopperDroid overall architecture, which highlights the unmarshalling Oracle running on a **vanilla** Android emulator (top right), the CopperDroid emulator (top left), and the out-of-the-box analyses CopperDroid implements to dynamically reconstruct the overall behaviors of Android malware (bottom). Specifically, CopperDroid emulator tracks system calls and their arguments, which are sent to the out-of-the-box analyses. This captures Binder transactions too (`ioctl` system calls), whose behavior is reconstructed thanks to the automatic interaction with the unmarshalling Oracle.

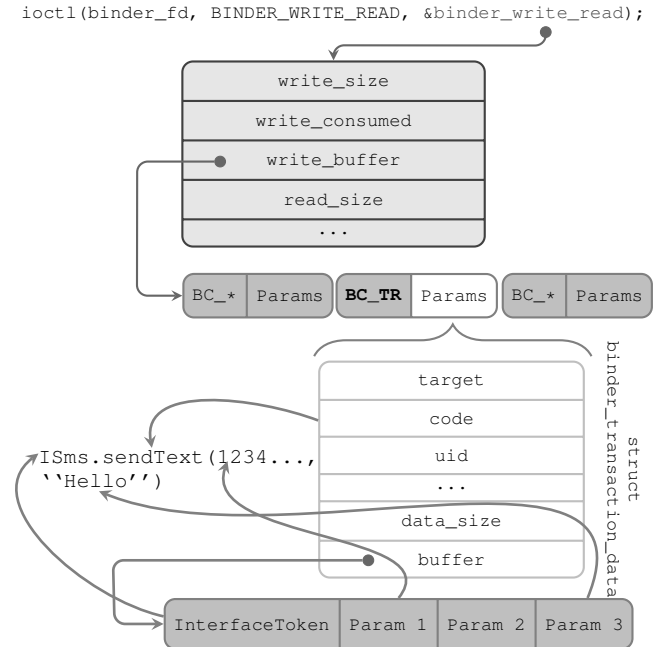


Figure 3: An example Binder payload corresponding to a send SMS action. CopperDroid initially parses the payload of the `ioctl` system call (Binder interaction), and sends the extracted (potentially-marshalled) arguments to the unmarshalling Oracle, which automatically unmarshalls them to reconstruct the send SMS behavior of the action under analysis.

B. Behavior Reconstruction

Based on the observation that all behaviors are eventually achieved through the invocation of system calls, CopperDroid’s VMI-based system call-centric analysis faithfully describes Android malware behavior whether it is initiated from Java or native code. Thus, tracking and analyzing system calls provides a unique observation point to reconstruct OS- and Android-specific behaviors. Any interesting application behavior is in fact characterized by system calls and their parameters, which includes high-level semantic, extracted from Binder unmarshalled data. This perspective highlights the strength of our unified analysis: a mere system call tracking would not provide any behavioral insights if it were not combined with Binder information and automatic (complex) Android objects reconstruction, as outlined in Section IV.

To analyze OS-specific events (e.g., write to a file, network connection creation), CopperDroid relies on a value-based data flow analysis (Section V-B) to abstract a sequence of (non-necessarily consecutive) system call invocations to high-level semantics (e.g., file system access, TCP communication). Additionally, to reconstruct Android-specific behaviors CopperDroid introduces an unmarshalling Oracle, which allows automatic deserialization of all IPC Binder transactions.

Specifically, CopperDroid intercepts specific `ioctl` system call invocations (Binder interactions) and parses their payload (see Figure 3). This blob of (potentially-marshalled) Binder-related system call arguments is then sent to the unmarshalling Oracle, a userspace Java application that runs on top of an unmodified Android emulator (see Figure 2). The unmarshalling Oracle relies on a finite number of unmarshallable

primitives, Java reflection (provided by the Android runtime), and the IBinder structure to deserialize Binder communications and return them to CopperDroid for further analysis. This allows for an easy, human-readable representation of the IPC message’s contents, which is of paramount importance to reconstruct and understand Android-specific behaviors, as shown in Figures 4 and 7. These Figures also demonstrate how Android malware low-level OS-related behaviors (e.g., writing to a file, creating a process, sending network data) are of course achieved through system calls and therefore intercepted and automatically reconstructed by CopperDroid’s unified analysis.

IV. AUTOMATIC IPC UNMARSHALLING

The unmarshalling Oracle is a Java application that runs in a vanilla Android emulator alongside the CopperDroid emulator. The purpose of the Oracle is to receive each Binder method signature and its arguments, in the form of a marshalled Parcel blob, and to return a custom representation of the method and all parameter values (see Figures 2 and 3). By default, the Oracle is queried offline, i.e., after all the execution traces have been collected. The data redirection from CopperDroid’s emulator to the Oracle is split into two sets of data of 1) marshalled data derived from binder communication by CopperDroid’s binder analysis and 2) the signature of the method, whose argument types corresponds to the sequence of marshalled data (see Figure 2).

As outlined in Section II-A, the Android system heavily relies on IPC/ICC and RPC interactions to carry out tasks. Therefore, tracking and dissecting the communications that happen over this channel are a key aspect for reconstructing

high-level Android-specific behaviors. Although recently explored to enforce user-authorized security policies [42], to the best of our knowledge, CopperDroid is the first approach to carry out a detailed analysis of such communication channels to comprehensively characterize OS- and Android-specific behaviors of malicious applications.

As systematically evaluating the Oracle’s reconstruction capabilities on every possible object (over 300+ AIDL objects alone) is a very challenging task, we introduce a representative example which triggers the main aspects of the Oracle’s unmarshalling capabilities. Let us consider an app that sends an SMS as our running example. The Java code that corresponds to the SMS behavior (e.g., creation and execution of an SMS intent) can be seen in Figure 4 (a). Such code typically includes a call to the `sendTextMessage` method of `SmsManager`, with the destination number (e.g., “123456789”) and the SMS text (e.g., “Hello”) as parameters. It is also optional to include a `PendingIntent`, which is broadcasted when the message is successfully sent to its destination.

`PendingIntent` objects are usually passed by reference, rather than being directly marshalled. To keep track of such data, CopperDroid is also aware of any creation of shared memory (for example, to store a `PendingIntent`) due to several `ioctl` calls sent from the client application to the `IActivityManager` (specifically the `getIntentSender` method). Indeed, all Binder transactions, including SMS, are fired by invoking the two `ioctl` system calls, as shown in Figure 4 (b). For example, when an SMS is sent, we would see one `ioctl` to locate the SMS service and the other to invoke the `sendTextMessage` method. It is the latter that is sent to the Oracle for unmarshalling. Furthermore, if the second `ioctl` includes a pass-by-reference parameter (e.g., a handle to a `PendingIntent`) CopperDroid locates a third `ioctl` with the actual referenced object (e.g., `PendingIntent` saying “SENT”) being sent to the `IAccountManager`.

From a high-level perspective (e.g., Java methods), sending an SMS by executing `sendTextMessage` (last line of Figure 4 (a)), roughly corresponds to obtaining a reference to an instance of the class `SmsManager`, the phone SMS manager, and sending the SMS out by invoking the method `sendTextMessage` as seen in the figure. This includes the necessary method parameters including the destination phone number and the text message as the method arguments. On a lower level, this corresponds to locating the Binder service `isms` and remotely invoking its `sendText` method with proper arguments. From this low-level perspective, the same actions correspond to the sender application invoking two `ioctl` system calls on `/dev/binder`: one to locate the service and the other to invoke its method. CopperDroid thoroughly introspects the arguments of each binder-related `ioctl` system call to reconstruct the remote invocation. This allows us to identify the invoked method and its parameters, and to infer the high-level semantic of the operation. In particular, we focus our analysis on Binder *transactions*, i.e., IPC operations that actually transfer data (also responsible for RPC). To identify them, CopperDroid parses the memory structures passed as a parameter to the `ioctl` system call and identifies Binder transactions (`BC_TRANSACTION`) and replies (`BC_REPLY`). (see [36] and Figure 3).

However, just intercepting transactions is of limited use

```
PendingIntent sentIntent = PendingIntent.getBroadcast(
    SMS.this, 0, new Intent("SENT"), 0);
SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage(
    "123456789", null, "Hello", sentIntent, null);
```

(a) SMS send with `PendingIntent` behavior at Java level.

```
ioctl(0x14, BINDER_WRITE_READ, 0xbec93e8) = 0
ioctl(0x14, BINDER_WRITE_READ, 0xbec69508) = 0
ioctl(0x14, BINDER_WRITE_READ, 0xbec693e8) = 0
```

(b) SMS send `ioctl`s with `PendingIntent` behavior. The third parameter points to a `binder_write_read` data structure, which eventually leads to a `binder_transaction_data` data structure, as shown in Figure 3.

```
BINDER_TRAN (from binder_transaction_data)::
sentIntent =
    [android.app.PendingIntent = Intent("SENT")]

BINDER_REPLY (from binder_transaction_data)::
sentIntent =
    [android.app.PendingIntent{Binder:
        type = BINDER_TYPE_HANDLE,
        flags = 0x7F|FLAT_BINDER_FLAG_ACCEPTS_FDS,
        handle = 0xa,
        cookie = 0xb8a58ae0}]

BINDER_TRAN (from binder_transaction_data)::
com.android.internal.telephony.ISms.sendText(
    destAddr = "123456789", srcAddr = None,
    text = "Hello",
    sentIntent =
        [android.app.PendingIntent{Binder:
            type = BINDER_TYPE_HANDLE,
            flags = 0x7F|FLAT_BINDER_FLAG_ACCEPTS_FDS,
            handle = 0xa, cookie = 0x0}],
    deliveryIntent = null

Oracle::
com.android.internal.telephony.ISms.sendText(
    destAddr = "123456789", srcAddr = None,
    text = "Hello",
    sentIntent =
        [android.app.PendingIntent("SENT")],
    deliveryIntent = null
```

(c) Simplified SMS send (including `PendingIntent`) reconstruction produced by CopperDroid and the Oracle, using the `binder_transaction_data` retrieved from the `ioctl`.

Figure 4: Excerpt of CopperDroid reconstructed SMS send with `PendingIntent` (for send confirmation) using captured `ioctl`s, AIDL generated files, and correlated binder handles generated by `IActivityManager`.

when it comes to understanding Android-specific behaviors. In fact, the raw `ioctl`-provided Binder data that flows throughout transactions are flattened and marshalled into a single buffer. Moreover, every interface a client and service agree upon has its own set of predefined methods’ signature, and, as the Android framework counts more than 300 of these AIDL interfaces, manual unmarshalling is unfeasible.

To understand the invoked method and the unmarshalling procedure for its parameters, we extended CopperDroid with the following. First, we let it identify the *InterfaceToken* specified in the payload (see Figure 2, Interface Token Identifier block). This is then used to find the AIDL description, if

available, of the interface CopperDroid needs to associate the numeric code to the invoked method and, therein, understand the types of its parameters. This step is necessary because, even if Parcel methods can create easily unmarshallable streams of bytes (including metadata to associate bytes to types), payloads are often marshalled directly and only the receiver knows exactly how to unmarshall them. Our solution, therefore, parses the AIDL files (using a modified AIDL parser) to automatically generate signatures of each method for each interface. These signatures are specific to a given version of Android and are used to assist in the unmarshalling process. Such signatures are then sent to an *unmarshalling Oracle* running separate from, but with the same Android OS version, the CopperDroid emulator (see Figure 2). Along with the AIDL signatures, marshalled data extracted from the `ioctl` transaction is also sent to the Oracle to determine the values of the method parameters through automatic unmarshalling of binder data, as outlined next.

A. AIDL Parser

An AIDL file defines a given interface detailing its methods, parameters and return values types. The Android platform includes an AIDL parser which, given an AIDL file, will produce Proxy and Stub classes. The Proxy is used on the client side and matches the method calls that a client would call (in terms of method name, parameters and return value). The Stub, used on the server side, utilizes the transaction code in order to perform the appropriate actions for the given method call. The reason for this is that all Binder calls go through the Binder device driver as I/O controls and while the functions on the client side (Proxy) match those in the client, the server (Stub) needs to efficiently map a given call to its method. The actual parcel data is held in the buffer field of the `binder_transaction_data` structure (see Figure 3).

While the AIDL process works fine for marshalling data between clients and servers during normal runtime, it is necessary for CopperDroid to combine components of the Proxy and Stub in order to unmarshall the objects post-analysis. Furthermore, it is also necessary to implement a parcel reader that understands how to unmarshall parameters from the marshalled data. Therefore, CopperDroid includes a *modified* AIDL parser that obtains the method names, parameters and return values types (i.e., usually utilized in the Proxy at runtime) to build a mapping between transaction codes and methods. It then combines this information with the parcel reader class mentioned earlier to automatically produce handling code for a given method. CopperDroid utilizes this code to extract the necessary information from any Binder call during later analysis. All this automatically-generated AIDL information is stored in multiple Python files, preserving the mapping of all interface names to parcel data extraction routines. For example, “com.android.internal.telephone.ISms” is mapped to the `db_parcel_ISms` function call. As this process is only needed once per Android OS version, it can be done before analysis and does not induce overhead during analyses.

B. Oracle Input

We extract two different pieces of data per `ioctl` call containing the `BINDER_WRITE_READ` flag. First, a blob of marshalled parameters is taken from the `buffer` field, as

Data: Marshalled binder transaction and data types (determined through the AIDL)

Result: Unmarshalled binder transactions

```

1 while data → marshalled do
2   determine type of marshalled item;
3   if type → primitive then
4     automatically apply correct parcelable read/create
      functions;
5     append string repr. to results;
6   else
7     locate CREATOR field for reflection;
8     use java reflection to get class object;
9     for every class field do
10      if field → primitive then
11        append string repr. to results;
12      else
13        explore field recursively;
14        append string repr. to results;
15      end
16    end
17    if CREATOR → not found and buffer → binder
      reference type then
18      Unmarshall Binder reference
19    end
20  end
21 end

```

Algorithm 1: The Unmarshalling Oracle.

can be seen in Figure 3. This is essentially the buffer minus the `InterfaceToken`, placed at the front of the buffer, which is a length-prefixed UTF-16 string. Serialized parameter values following the token are stored and sent to the Oracle for post-processing (Figure 2 leftmost RSP line and leftmost TCP line). The second extracted piece of information is the numeric code that, when united with the `InterfaceToken`, uniquely identifies the method that is called by a binder transaction.

The `InterfaceToken`, along with the numeric code, is used by CopperDroid to invoke the correct automatically-generated AIDL data extraction routine (as explained above). This routine returns the AIDL description of the interface and enables the Oracle to understand the types of the parameters contained in the `buffer` and unmarshall them. For example, in Figure 4, the `PendingIntent` leads us to an actual Java Intent type, which stores the message “SENT” and can hold other data such as an action to perform after the SMS had been sent (e.g., notify the user or add the sent SMS into a log).

Revisiting the SMS example, after the `ISms.sendText()` method is invoked, CopperDroid intercepts the corresponding binder transaction and uses the `InterfaceToken` and the code to invoke the correct handling function that retrieves information on the called method. For the SMS example, these include the method name (“sendText”), its parameters (`destAddr`, `scAddr`, `text`, `sentIntent`, `deliveryIntent`), and its parameter types (`String`, `String`, `String`, `PendingIntent`, `PendingIntent`). This information is sent to the Oracle along with the marshalled buffer—containing, among other things, the body of the message. The Oracle uses this data to extract the value of parameters passed to `ISms.sendText()`.

C. Oracle Inner Workings and Output

The Oracle relies on Java reflection to unmarshall the

complex serialized Java objects it receives and returns their string representations to CopperDroid to enrich its behavioral profile with Android-specific actions. Therefore, the Oracle must be run with the same Android OS version as the CopperDroid emulator. However, it is worth noting that the unmarshalling Oracle does not require the access to the app, or malware, being analyzed in the separate CopperDroid emulator. All information necessary to unmarshall the Binder data is retrieved from the CopperDroid emulator and sent over to the vanilla Android emulator running the Oracle via CopperDroid’s behavioral reconstruction analysis, as depicted in Figure 2.

Algorithm 1 outlines the working details of the unmarshalling Oracle. It currently unmarshalls Binder communication objects in one of three unique ways depending on whether the type of data is a primitive or basic type (e.g., String, Integer), an Android class object (e.g., Intent), or a Binder reference object (e.g., PendingIntent, Interface). As mentioned previously, the data types are determined by CopperDroid’s AIDL parser and the list of parameter types is sent to the Oracle along with the marshalled parameter values. For example, primitive or basic types are easily unmarshalled by invoking the appropriate unmarshalling function provided by the Parcel class (e.g., `readString`, `readInt`).

In the following, we provide additional details on the automatic unmarshalling process.

Primitives: While iterating through the list of types and class names (e.g., in our SMS example the Oracle would iterate through three String types and two PendingIntent types), if the type is identified as primitive (e.g., String) the corresponding read function provided by the Parcel class is used (e.g., `readString()`). The while loop at line 1 in Algorithm 1 would iterate through those five parameters, while lines 3-5 are responsible for primitive types. In our SMS example, the parameters `destAddr`, `scAddr`, and `text` have primitive String types and would therefore be unmarshalled using the correct `readString()` Parcel function in order to regain the parameter values, such as the SMS text body “Hello”.

Class objects: To unmarshall a class instance, the Oracle application requires Java reflection (see lines 6-7 in Algorithm 1). This method allows the Oracle to dynamically retrieve a reference to the CREATOR field, implementing the Android Parcelable interface. Any of such object must implement this interface, and therefore must have the CREATOR field, in order to be written to and read from a Parcel. Once this has been achieved, the Oracle can begin reading the remaining class data: in our example in Figure 4, the class data of an Android Intent (sent as a PendingIntent) entails the phrase “SENT” [13]. Once we have obtained the CREATOR field, the Oracle can obtain a reference to it, and invoke its `createFromParcel()` method in order to unmarshall the new object and read in its data. Once either a primitive or class type has been unmarshalled, the Oracle creates a string representation of the object by invoking its `toString()` method, if any, or alternatively recursively inspecting each field through Java reflection (see Section IV-D). The string representation is then appended to an output string list, and the marshalled data offset is updated to point to the next item to be unmarshalled next. Additionally, the Oracle iterates to the next parameter primitive or class type on the given list.

IBinder objects: As mentioned previously, certain types of Binder objects are not marshalled and then sent via IPC directly, but instead a reference to the object, stored either in the caller memory address space or in a shared memory space (i.e., `ashmem`), is sent instead. In this case, if the object is neither a primitive nor directly marshalled (see Algorithm 1, line 17), the Oracle verifies whether the data contains a binder reference object. This requires parsing the first four bytes of the marshalled object looking for IBinder reference types. These reference types determine whether the referenced object is a Binder service (i.e., `BINDER_TYPE_(WEAK_) BINDER`—a transaction sending a handle and service name to the `IServiceManager`), a Binder proxy (i.e., `BINDER_TYPE_(WEAK_) HANDLE`—to send objects from clients including `IInterface` classes represented as a binder object), or a file descriptor of an `ashmem` region (`BINDER_TYPE_FD`, although the other types may also send fds with the appropriate flags). Normally the Binder reference keeps the object from being freed; however, if the type is weak, the existence of the reference does not keep the object from being removed, unlike a strong reference.

For instance, referring to our running example in Figure 4, a `PendingIntent` can be used to broadcast whether the SMS was successfully sent or not. By reconstructing this Intent the Oracle can understand whether the SMS was successfully sent. However, this Intent is not actually sent over the IPC channel directly, but is rather sent as a handle reference. Therefore the Oracle unmarshalls the `ioctl` call, just as the receiving process would have in real-time, obtaining a reference. Such a reference may contain an address or, as is the case here, a handle to the `PendingIntent` that can be seen with value “0xa” in our Oracle reconstructed SMS in Figure 4 (c). When sending the IPC communication to the server, Binder passes along the information necessary for the receiving process to seamlessly retrieve the data.

With the referenced-based parceling used by Binder, the Oracle needs to retrieve live ancillary data from the system in order to be able to map the references to the data. To this end, CopperDroid keeps track of these references in real-time, whether they are sent via handle or via a file descriptor. In the latter case, data from `ashmem` regions—created via specific `ioctl` and `mmap` system calls—are retrieved by CopperDroid and unmarshalled by the Oracle. Furthermore, CopperDroid tracks certain calls which register a given allocation (either in the caller space or shared via `ashmem`) and obtains the handle that is assigned for the given allocation. For example, in Figure 4 there is a transaction (`BINDER_TRAN`) for registering the `sentIntent` and corresponding response (`BINDER_REPLY`) with the handle. However, to extract such information, CopperDroid needs to identify the file descriptor or binder handle (e.g., the marshalled binder reference) passed as a reference in the binder transaction. To avoid using ad-hoc extraction procedures and to rely on automatic mechanisms, CopperDroid relies on the support provided by the unmarshalling code generated by the AIDL parser to extract the primitive types and handles/references in the binder transaction.

Thus, whenever CopperDroid intercepts a binder transaction that uses handle-referenced memory (caller allocated or `ashmem`), it can quickly extract the reference at run

time. For instance, as mentioned earlier in Figure 4 (a), when an app creates a `PendingIntent`, it does so by calling the `IActivityManager::getIntentSender` method, which returns a handle specific to that data (the handle is returned in the standard `flat_binder_object` structure for references). In order to extract the references, CopperDroid utilizes two additional fields (i.e., `offsets_size` field and the `offsets` pointer), in the `binder_transaction_data`⁴. The `offsets_size` field is the size (in bytes) of the `offsets` array, comprised of pointer sized values each corresponding to a given reference.

If, when unmarshaling a given binder transaction, the type (based on the code generated by the AIDL parser) is a binder type, then the `offsets` array is used to locate the offset within the transaction data of the specific `flat_binder_object`. The offsets are in the same order as the types in the parcelable object. Referring to the SMS example as shown in Figure 4, the `sentIntent` is the first reference and should thus be the first entry (position 0) in the `offsets` array and, if the `deliveryIntent` were not null, it would be the second entry in the `offsets` array (position 1) and the `offsets_size` field would be 8 (bytes) on a 32-bit ARM system. With this information, CopperDroid identifies and captures the corresponding caller allocated memory region, `ashmem` region, or any other memory region which contains the actual marshalled object. The marshalled data is then sent back to the Oracle for the final unmarshalling procedure⁵.

D. Recursive Printing

The purpose of printing recursively is to thoroughly inspect each field in every object to produce as much information as possible. Normally this means every sub class (or bundle) is explored until a primitive can be found and printed or stored for further analysis by CopperDroid. To do this, the string representation of each primitive is appended to the output string list, and the marshalled data offset is updated to point of the next unmarshalled item. Additionally, the Oracle iterates to the next type, or class name, on the given list. Once the Oracle has completed unmarshalling all the parameters, the final output can be returned to CopperDroid. Figure 4(c) presents an excerpt of a simplified Oracle output corresponding to a reconstructed send SMS behavior. For simplicity, we only include essential details, filtering out less relevant flags and empty fields.

V. OBSERVED BEHAVIORS

We manually examined the results of CopperDroid’s analyses (i.e., system call invocations tracking, Binder analysis, and complex Android objects reconstruction) on a number of randomly selected Android malware extracted from the samples sets at our disposal [11], [30], [49]. Figure 5 shows the insights of our examination, which allowed us to identify six macro classes of behaviors. Each class contains one or more behavioral models, which is defined by a set of *actions*. Actions are traced through CopperDroid and can belong to any

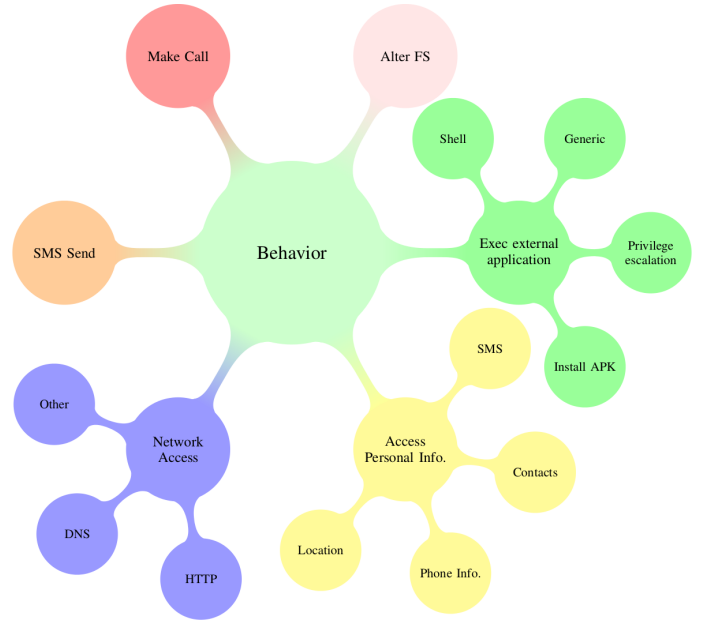


Figure 5: Hierarchical map of behaviors.

level of behavior abstraction (e.g., OS- and Android-specific behaviors). Interestingly, some behaviors are well-known and shared with the world of non-mobile malware. Others, such as those under the “Accessing Personal Info” class, are instead inherently specific to the mobile ecosystem.

Every terminating class in the map corresponds to a behavioral model that can be expressed by an arbitrary number of actions, depending on its specific complexity. The complexity of these elements vary greatly. Some are defined as a single system call, such as `execve`. Others, such as “SMS Send” or those under “Access Personal Info”, are defined as a set of transactions of the Binder protocol. Yet others are defined as multiple consecutive system calls. For instance, outgoing HTTP traffic is modeled as a graph with a `connect` system call, followed by an arbitrary number of `send`-like system calls, whose payload is parsed to detect HTTP messages, possibly interleaved by unrelated non-socket system calls. Terminating classes do not forcibly correspond to just one of the aforementioned models, but may also contain a set of them. To clarify, consider the examples shown in Figure 6 which illustrate how CopperDroid recognizes actions triggered by both these snippets of code as belonging to the class “Install APK”, despite the differences in the manner in which these actions are achieved (an `execve` system call rather than a Binder transaction).

A. App Stimulation


Traditional executables have a single entry point, while Android applications may have multiple. Most apps have a main activity, but ancillary activities may be triggered by the system or by other apps and the execution may reach them *without* flowing through the main. In such a scenario, a simple install-then-execute dynamic analysis may miss a number of interesting behaviors. This problem has long been affecting traditional dynamic analysis approaches as non-exercised paths are simply unanalyzed. If such paths host additional behaviors,

⁴These are not shown in the simplified payload Figure 3.

⁵A limitation of our current implementation is that we can only automatically inspect and unmarshall parameters of methods contained in interfaces of which we have the AIDL files. Bound services that do not use AIDL, e.g., `ActivityManager`, are manually unmarshalled.


```
execve('pm', ['pm', 'install', '-r', 'New.apk'], ... );
```

(a) App installation via direct system call (OS-specific behavior).

```
Intent intent = 
    new Intent(Intent.ACTION_VIEW);
intent.setDataAndType(
    Uri.fromFile(
        new File("/mnt/sdcard/New.apk")),
        "application/vnd.android.pack...");
startActivity(intent);
```

(b) App installation via Binder transaction (e.g., Intent specific).

Figure 6: App installation via direct system call and Binder (Android-level) transaction.

then any dynamic analysis would fail unless proper, but generally expensive and complex exploration techniques are adopted [8], [31]. In addition, this problem is exacerbated by the fact that mobile applications are inherently user driven and interaction with applications is generally necessary to increase coverage. For instance, let us consider an application that operates as a broadcast receiver for SMS_RECEIVED events. After installation, the application would only react to the reception of SMS showing no additional interesting behaviors.

To *qualitatively* address dynamic code coverage issues, CopperDroid implements a technique to artificially stimulate the analyzed malware with a number of valid and interesting events based on the malware’s Manifest. For example, injecting events such as phone calls and reception of SMS texts would lead to the execution of any application registered broadcast receiver. Another example that comes from our experience with Android malware is the BOOT_RECEIVED intent, that many samples use to start execution as soon as the victim system is booted (much like \CurrentVersion\Run registry keys on Windows systems).

The Android emulator offers the possibility to inject a considerable number of artificial events to stimulate a running application. These range from very low-level hardware-related events (e.g., loss of the 3G signal) to higher-level ones (e.g., incoming calls, SMS). CopperDroid could adopt a fuzzing-like stimulation strategy and trigger *all* the events that could be of interest for the analyses. That would unfortunately be of limited effect because of the underlying Android security model and permission system. Instead, CopperDroid leverages static information extracted from the app to carry out a fine-grained targeted stimulation strategy. To this end, CopperDroid examines each APK Manifest to extract events and permission-related information to drive the malware stimulation. Furthermore, an application has the ability to dynamically register a broadcast receiver for custom events at run-time. CopperDroid is able to intercept such operations and add a proper stimulation for the newly registered receiver.

To perform its custom stimulation, CopperDroid leverages the Android emulator’s capabilities to inject a number of artificial events into the emulated system. In particular, it leverages MonkeyRunner, a tool that provides an out-of-the-box API to control an Android device or emulator, through the Python programming language [3]. A summary of the main

#	Stimulation	Parameters
1	Received SMS	<i>Text, from number</i>
2	Incoming call	<i>From number, duration</i>
3	Location update	<i>Geospatial coordinates</i>
4	Battery status	<i>Amount of battery</i>
5	Phone Reboot	
6	Keyboard input	<i>Typed text</i>

Table I: Main stimulations and parameters.

```
OutputStreamWriter out =
    new OutputStreamWriter(
        openFileOutput("samplefile.txt",
            MODE_WORLD_READABLE));
out.write("Data write", 0, 10);
```

(a) File access behavior at Java level.

```
open("files/samplefile.txt", 0x20241, 0x180) = 0x1c
read(0x3f, 0x470bec04, 0xf) = 0xf
...
write(0x1c, "Data write", 0xa) = 0xa
```

(b) File access behavior at system call level.

```
FS_ACCESS::Creation of "sampefile.txt"
(link, ancillary info: 10 (bytes))
```

(c) Reconstructed file access behavior.

Figure 7: CopperDroid behavior reconstruction of a file access.

events CopperDroid handles is reported in Table I, which also shows the parameters that can be customized for each event.

B. Value-based Data Flow Analysis

We extended CopperDroid with the ability to abstract a stream of related low-level events to a more meaningful high-level behaviors and to recreate resources (e.g., files, network communications) associated with an application to enrich CopperDroid’s behavioral reconstruction analyses (see Figure 7(c), e.g., FS_ACCESS and NET_ACCESS).

To this end, CopperDroid performs a value-based data flow analysis by building a system call-related data dependency graph and def-use chains. In particular, each observed system call is initially considered as an unconnected node. A forward slicing algorithm then inserts edges for every inferred dependence between two calls. As the slicing proceeds, both nodes and edges are annotated with the system call argument constraints; these annotations are essential in the creation of our def-use chains. Def-use chains, where each call is linked by def-use dependencies, are formed when the output value by one system call (the *definition*, e.g., open, dup, dup2) is the input value to a following (non-necessarily adjacent) system call (the *use*, e.g., write, writev). Therefore, by building a data dependency graph over the set of observed system calls, and performing forward slicing, we can recreate file system-related events and the actual resources involved. Our analysis retains deleted files (unlink) and multiple versions of the resources with identical file names. Although we focus this discussion on file system-related system calls, a similar process holds and has been implemented for network-related calls.

VI. EVALUATION

Our experimental setup is as follows. We ran unmodified Android images on top of the CopperDroid-enhanced emulator. Each clean image was customized to include personal information, such as contacts, SMS texts, call logs, and pictures to mimic, as closely as possible, a real device. Each analyzed malware sample was installed in the image and traced via CopperDroid until a timeout was reached (10 minutes by default). At the end of the analysis, a clean execution environment is restored to prevent corruptions and side-effects caused by installing multiple malware samples in the same system. To limit noisy results, each sample was executed and analyzed six times: three times *without* stimulation and three times *with* stimulation; single execution results were then merged.

We evaluated CopperDroid on three datasets: two publicly available [11], [49] and one provided by McAfee [30]. These datasets are composed of 1,226, 395 and 1,365 samples, respectively, counting more than 2,900 samples overall.

A. Effectiveness

To evaluate the effectiveness of CopperDroid’s stimulation approach, we proceeded as follows. First, we analyzed all the samples without external stimulation. Then, we performed the stimulation-driven analysis of the same malware sets, as outlined in Section V-A. A summary of the effects of the stimulation on the three datasets is presented in Table II; details of the analysis on the first two datasets were presented in our preliminary workshop work [36], while results on the McAfee dataset are reported in Table VI (see Appendix A). As Table II shows, the results of our analysis on the new McAfee dataset are consistent with our previous and preliminary results (see Table II and [36]): 836 out of 1365 (61%) McAfee samples exhibited additional behaviors (see Section V for what we consider to be a behavior) and, on average, the number of additional behaviors was roughly 6.5, out of an average number of exhibited behaviors of 22.8, observed during non-stimulated executions⁶. Of course, it is important to understand whether an observed behavior is new or if it refers to a similar, previously-observed action (e.g., same network communication with a different timestamp). Our current approach is simple: We consider pseudorandom or ephemeral values observed in specific behaviors (e.g., the above mentioned timestamp parameter, or a pseudorandom ID characterized by high a byte entropy) to refer to an already observed behavior and therefore to not contribute to the percentage of additional behaviors observed due to app stimulation. All the other behaviors are considered to be new or additional and therefore contribute to the aforementioned percentage. Future work includes building on recent and promising hierarchical similarities [26] to better discern among new or additional behaviors.

Table III reports the overall breakdown of the observed behaviors (see Figure 5) on McAfee dataset. Each row identifies the class of behavior and how many samples over the total exhibited at least one occurrence of such behavior, *without* and *with* stimulation, respectively. As can be observed the two most influenced behavioral class are *Access Personal Information* and *Make/Alter Call*. The first is triggered by a non-negligible

⁶Solutions to *quantitatively* improve code coverage may be built on top of symbolic execution (e.g., [1], [9]), but unfortunately they do not scale well.

Malware Dataset	Incr. Behav. (Samples)	Avg. Increment	Std. Dev
Genome	752/1226 (60%)	2.9/10.3 (28.1%)	2.4/11.8
Contagio	289/395 (73%)	5.2/23.6 (22.0%)	3.3/19.8
McAfee	836/1365 (61%)	6.5/22.8 (28.5%)	9.5/30.1

Table II: Summary of stimulation results, per dataset.

Behavior Class	No Stimulation	Stimulation
FS Access	889/1365 (65.13%)	912/1365 (66.81%)
Access Pers. Info.	558/1365 (40.88%)	903/1365 (66.15%)
Network Access	457/1365 (33.48%)	461/1365 (33.77%)
Exec. Ext. App.	171/1365 (12.52%)	171/1365 (12.52%)
Send SMS	38/1365 (2.78%)	42/1365 (3.08%)
Make/Alter Call	1/1365 (0.07%)	55/1365 (4.03%)

Table III: Overall behavior breakdown of McAfee dataset.

number of samples that receive an incoming message sent by CopperDroid stimulation technique (and exhibits an access to the user’s personal information, otherwise hidden). The latter is mostly due to a set of malware that, whenever a call is received, hide its notification to the user. Table V, instead gives a more detailed overview of single behavioral subclasses (defined in Section V) and if—and how—they are influenced by stimulation.

Lastly, we also ran a number of malware samples with no, selective, and full stimulation. This experiment aimed at qualitatively gathering which individual stimulus induced what amount of incremental behavior, and whether combinations of stimulation are more effective than individual triggers. Again, these stimulations are tailored to each malware by analyzing their Manifest to determine what triggers were possible due to the permission scheme. We deliberately chose Android malware samples that in our experiments had the highest, average and lowest incremental behavior both percentage wise and amount wise. If several families had the same maximum amount of incremental behavior we choose the one with the highest percentage incremental behavior and vice versa. Lastly we determined the best representative sample from each family based on the amount and diversity of behaviors. The results of various stimulations on these malware samples can be seen in Table IV. With the table we can begin to see correlations between different stimuli and behaviors. As Table IV shows, our selective stimulations was able to disclose a number of *additional* previously-unseen (e.g., YZHC SMS stimulation showed access of personal account information) or already-observed (e.g., SHBreak showed 113 additional generic execution) behaviors.

B. Performance

In this section we present an evaluation of CopperDroid overhead through a number of different experiments conducted on a GNU/Linux Debian 64-bit system equipped with an Intel 3.30GHz core (i5) and 3GB of RAM. Benchmarking a multi-layered system, such as Android, in conjunction with a complex technique, such as CopperDroid (and in an emulated environment), can be a rather complicated task. For example, traditional benchmarking suites based on measuring I/O

Sample Family	Behavior Class	Behavior Subclass	Behaviors No Stim.	Incr. Behavior Type Stim.	Incr. Behavior SMS Stim.	Incr. Behavior Loc. Stim.
YZHC	Network Access	HTTP	4	-	-	N/A
		DNS	1	-	-	N/A
	Exec External App	Generic	3	+10 (+433%)	-	N/A
		Shell	1	+3(+400%)	-	N/A
		Priv. Escalation	2	-	+2(+100%)	N/A
		Install APK	4	-	-	N/A
	Access Personal Info	Account	-	-	+1(↓)	N/A
	FS Access	Write	414	-	-	N/A
zHash	Network Access	HTTP	2	+2 (+100%)	+5 (+350%)	N/A
		DNS	-	-	+1 (↓)	N/A
	Exec External App	Generic	1	+12 (+1300%)	+3 (+400%)	N/A
		Shell	1	+3 (+400%)	-	N/A
		Priv. Escalation	4	-	-	N/A
		Install APK	4	-	-	N/A
	Access Personal Info	Account	2	-	-	N/A
	FS Access	Write	163	-	+255 (+257%)	N/A
SHBreak	Network Access	HTTP	3	-	N/A	N/A
	Exec External App	Generic	2	+113 (+5750%)	N/A	N/A
		Shell	1	+22 (+2500%)	N/A	N/A
		Install APK	4	+4 (+100%)	N/A	N/A
	FS Access	Write	195	+353 (+281%)	N/A	N/A
Droid KungFu	Network Access	HTTP	13	-	N/A	-
	Exec External App	Generic	1	+2 (+300%)	N/A	+1 (+200%)
		Shell	1	-	N/A	-
		Install APK	4	-	N/A	-
	FS Access	Write	3	+197 (+6667%)	N/A	+144 (+4800%)
Fladstep	Network Access	HTTP	15	-	N/A	N/A
	Exec External App	Generic	3	+17 (+633%)	N/A	N/A
		Shell	1	+5 (+500%)	N/A	N/A
		Install APK	4	-	N/A	N/A
	FS Access	Write	171	+80 (+47%)	N/A	N/A

Table IV: Incremental behavior induced by various stimuli, N/A means stimulus not possible based on Manifest.

operations are affected by caching mechanisms of emulated environments. On the other hand, CPU-intensive benchmarks are meaningless against the overhead of CopperDroid, as it mainly operates on system calls.

To address such issues, we performed two different benchmarking experiments. The first is a *macrobenchmark* that tests the overhead introduced by CopperDroid on common Android-specific actions, such as accessing contacts and sending SMS texts. Because such actions are performed via the Binder protocol, these tests give a good evaluation of the overhead caused by CopperDroid’s Binder analysis infrastructure. The second set of experiments is a *microbenchmark* that measures the computational time CopperDroid needs to analyze a subset of interesting system calls.

To execute the first set of benchmarks, we created a fictional Android application that performs generic tasks, such as sending (SEND_SMS) and reading (SMS) texts, accessing local account information (GET_ACC), and reading all contacts (CONTACTS). We then ran the test application for 100 iterations and collected the average time required to perform these operations under three different settings: on an unmodified Android emulator (i.e., without CopperDroid—baseline), on a CopperDroid-enhanced emulator with CopperDroid configured to monitor the test application (the common setting when analyzing a piece of malware—CD (targeted)), and on a CopperDroid-enhanced emulator with CopperDroid configured to track system-wide events (CD (full)). Results are reported in Figure 8 (A). As can be observed, the overhead introduced by the targeted analysis is relatively low, respectively $\approx 26\%$, $\approx 32\%$, $\approx 24\%$ and $\approx 20\%$. On the other hand, system-

wide analyses increase the overhead considerably ($>2x$). This is due to the of the number of Android components that are concurrently analyzed.

The second set of experiments measure the average time CopperDroid requires to inspect a subset of interesting system calls. This experiment collected more than 150,000 system calls obtained by running apps and subjecting them to arbitrary (and artificial) workloads. As tracking a system call requires to intercept entry and exit execution points, we report such measures separately in Figure 8 (B) (the average times are $0.092ms$ for entry and $0.091ms$ for exit).

VII. RELATED WORK

In this section we cite and compare similar works related to CopperDroid. First off, DroidScope [43] is a framework to create dynamic analysis tools for Android malware that trades off simplicity and efficiency for transparency: as an out-of-the-box approach, it instruments the Android emulator, but it may incur high overhead (for instance, when taint-tracking is enabled). DroidScope leverages a 2-level VMI [20] to gather information about the system and exposes hooks and a set of APIs, which enabled the development of plugins to perform both fine and coarse-grained analyses (e.g., system call, single instruction tracing, and taint tracking). In contrast with CopperDroid, DroidScope offers a set of instrumentation points that analyses can build upon to intercept interesting events and does not perform any behavioral analysis *per-se*. For example, a tool leveraging DroidScope can intercept every system call executed on an Android system, but would still

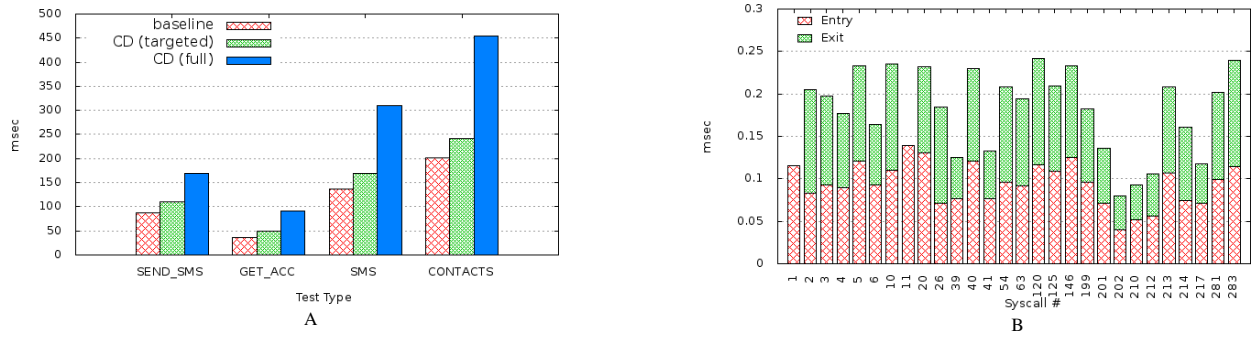


Figure 8: Binder Macrobenchmark (A) and System Call Microbenchmark (B).

Behavior Class	Subclass	No Stim.	Stim
Network Access	Generic	483	489
	HTTP	309	318
	DNS	416	416
FS Access	Write	889	912
Access Personal Info.	SMS	32	266
	Phone	510	559
	Accounts	51	672
	Location	143	147
Exec. External App.	Generic	132	132
	Priv. Esc.	103	103
	Shell	73	73
	Inst. APK	8	8
Send SMS	—	38	42
Make/Alter Call	—	1	55

Table V: Behavior breakdown on the Android malware samples provided by McAfee.

need to do its own VMI to inspect the parameters of each call. Following this principle, CopperDroid could have been built on top of DroidScope, but at the time we implemented it, the DroidScope framework was not publicly available. Moreover, the main focus of our research is *not* to illustrate how to build a framework or a clever VMI technique for Android systems, but rather to point out how a proper system call-centric analysis—which still includes a thorough IPC and RPC Binder-related protocol analysis as well as automatic and seamlessly complex Android object reconstructions—and stimulation technique can comprehensively expose Android malware behaviors.

Enck *et al.* presents TaintDroid [14], a framework to enable dynamic taint analysis for Android applications. TaintDroid’s main goal is to track how sensitive information flow between the system and applications, or between applications, to automatically identify leaks. Because of the complexity of the Android system, TaintDroid relies on different levels of instrumentation to perform its analyses. For example, to propagate taint information through native methods and IPC, TaintDroid patches JNI call bridges and the Binder IPC library. TaintDroid is both extremely effective, as it allows it to propagate tainting between many different levels, and efficient, as it does that with a very low overhead. Unfortunately, the price to pay is low resiliency and transparency: modifying internal components of Android inevitably exposes TaintDroid to a series of detection

and evasions techniques [10], [37], [38]. For instance, even applications with standard privileges can detect TaintDroid’s presence by calculating checksums over instrumented and readable components. Moreover, TaintDroid cannot track the taintedness of native code. Conversely, applications that can escalate their privileges can go even further: identifying and disabling TaintDroid’s hooks and analysis. Furthermore, the decision to modify internal components exposes TaintDroid to the issues deriving from constantly adapting the analysis code to highly-mutable architecture as the Android OS.

AppsPlayground [35] performs a much granular stimulation than CopperDroid, but its full capabilities require non-negligible modifications to the Android framework (e.g., to capture image identifiers in GUI elements). It also does not analyze native code (with the exception of specific and known low-level signatures, such as known root exploits), and integrates a number of well-known techniques (e.g., TaintDroid), inheriting their limitations, e.g. non-negligible modification of the Android runtime and limited taint-tracking when analyzing malicious apps [10]. PuppetDroid [21] (as AppsPlaygrounds) is an interesting approach to stimulate Android apps with stimulation traces gathered from crowdsourcing. It is more effective than CopperDroid with respect to stimulation, but limited to the subset of apps for which there exists a similar recorded stimulation trace. The downside, however, is that the overhead of PuppetDroid is significantly high whereas CopperDroid does not require any modification to the Android OS (nor runtime). This avoids dealing with the ever-evolving Android runtime system.

DroidBox is a dynamic, in-the-box, Android malware analyzer [40] that leverages custom instrumentation of the Android system and kernel to track a sample’s behavior through TaintDroid’s taint-tracking of sensitive information [14]. Using TaintDroid and instrumenting Android’s internal components makes DroidBox prone to the problems of in-the-box analyses: malware can detect, evade, or even disable them.

Andrabis [28] is an extension to the Anubis dynamic malware analysis system to analyze Android malware [5], [24]. According to its web site, it is mainly built on top of both TaintDroid [14] and DroidBox [40] and it thus shares their weaknesses (mainly due to operating “into-the-box”). In addition, Andrabis does not perform any stimulation-based analysis, limiting its effectiveness in discovering interesting Android-specific behaviors.

DroidMOSS [46] relies on signatures to detect malware in

app markets. Similarly, DroidRanger [48] and JuxtApp [22] identify known mobile malware repackaged into other apps. Although successful, signature-based techniques does limit the detection effectiveness to only known malware. In [15], Enck et al. studied Android permissions found in a large dataset of Google Play apps to understand their security characteristics. Such an understanding is an interesting starting point for designing techniques to enforce security policies [42] and avoid the installation of apps requesting a dangerous combination [16] or an overprivileged set of permissions [18], [33]. Although promising, the peculiarity of Android apps (e.g., a potential combination of Java and native code) can easily elude policy enforcement or collude to perform malicious actions while maintaining a seemingly legitimate appearance. This clearly calls for continuous research in this direction.

Aurasium [42] is a technique (and tool) that enables fine-grained dynamic policy enforcement of Android apps. To intercept relevant events, Aurasium instruments single apps, rather than adopting system-level hooks. Working at the application level, however, exposes Aurasium to easy detection or evasion attacks by malicious Android applications. For example, regular applications can rely on native code to detect and disable hooks in the global offset table, even without privilege escalation exploits. Aurasium’s authors state that their approach can prevent such attacks by intercepting `dlopen` invocations needed to load native libraries. It is however unclear how benign and malicious code can be distinguished, as this policy cannot be lightheartedly delegated to Aurasium’s end-users. Conversely, CopperDroid’s VMI-based system call-centric analysis is resilient to such evasions.

Google Bouncer [29], as its name suggests, is a service that “bounces” malicious applications off from the official Google Play (market). Little is known about it, except that it is a QEMU-based dynamic analysis framework. All the other information come from reverse-engineering attempts [32] but they are too scarce to properly compare it against our approach.

SmartDroid [45] leverages a hybrid analysis that statically identifies paths that lead to suspicious actions (e.g., accessing sensitive data) and dynamically determines UI elements to take the execution flow down those identified paths. To this end, the authors instrument both the Android emulator and Android’s internal components to infer which UI elements trigger suspicious behaviors. They also evaluated SmartDroid on a testbed of seven different malware samples and found it vulnerable to obfuscation and reflection, which make it hard—if not impossible—to statically determine every possible execution path. Conversely, CopperDroid’s dynamic analysis is resilient to static obfuscation and reflection.

To overcome the limits of dynamic analysis (e.g., code or path coverage), Anand *et al.* proposed a concolic-based solution [1] to automatically identify events an application reacts to by generating input events for smartphone applications. While no learning phase is required, such a solution has two main drawbacks: it is based on instrumentation (i.e., easy to detect) and is extremely time-consuming (i.e., up to *hours* to exercise a single application). Although an interesting direction to explore further, that approach is ill-suited to perform large-scale malware analysis. As described in Section V-A, CopperDroid relies on a simple-yet-effective stimulation technique that is

able to improve basic dynamic analysis coverage and discover additional behaviors with low overheads.

VetDroid is a dynamic analysis platform for reconstructing sensitive behaviors in Android apps from a permissions use perspective [44]. Zhang et al. points out that traditional system call analysis is not appropriate for characterizing the behaviors of Android apps as it misses high-level Android-specific semantics and fails at reconstructing IPC and RPC interactions. Contrary to this, we have shown that CopperDroid’s unified system call-centric analysis is able to automatically and seamlessly reconstruct IPC and RPC interactions as well as complex Android objects, generating insightful behavioral profiles.

Finally, one study recently used the insights of the CopperDroid workshop paper to manually discover vulnerabilities in the Android IPC [4].

VIII. CONCLUSION

We proposed CopperDroid, a VM-based dynamic system call-centric analysis and stimulation technique to both uniformly, and automatically, reconstruct behaviors of Android malware. In particular, we show how a careful dissection of system calls coupled with the ability to automatically track and deserialize IPC and RPC interactions, typically contextualized through complex Android objects, is key to the reconstruction of both OS- and Android-specific behaviors from a unique, well-known, (system level) observation point. Not only is this simplicity more transparent to changes in the Android runtime system and its inner details, but it also makes the approach agnostic to the underlying action invocation mechanisms (e.g., Java or native code). We evaluated the effectiveness and performance of CopperDroid on more than 2,900 real world Android malware, showing that a simple, external, stimulation contributes to the discovery of additional behaviors.

We believe the novelty of CopperDroid’s analyses opens the possibility to reconsider rich and unified system call-based approaches as effective techniques to build upon to mitigate Android malware threats.

AVAILABILITY

CopperDroid and information about our ongoing project-related research are available at:

<http://s2lab.isg.rhul.ac.uk/projects/mobsec/>

where users can reach out to a publicly-available version of CopperDroid and submit APK files to be analyzed. Analysis results include behavioral profiles available in a number of different formats (e.g., HTML and JSON, for easy parsing) and additional ancillary information (e.g., network traffic, reconstructed files and executables, complete system call traces with behavior reconstruction).

ACKNOWLEDGMENTS

This research has been partially supported by the UK EPSRC grant EP/L022710/1 and by a generous donation from Intel Security (McAfee Labs). We are equally thankful to the anonymous reviewers’ comments and Timothy Leek (and Andrew Davis), our shepherd, for their invaluable comments and suggestions to improve the paper. We also thank Alessandro

Reina, Santanu Dash, Johannes Kinder, Igor Muttik, and Alex Hinchliffe for their valuable suggestions and discussions on the work.

REFERENCES

- [1] S. Anand, M. Naik, H. Yang, and M. Harrold, “Automated concolic testing of smartphone apps,” in *Proc. of FSE*, 2012.
- [2] Android, “Android developer reference,” <http://developer.android.com/reference/packages.html>.
- [3] —, “Monkeyrunner,” http://developer.android.com/tools/help/monkeyrunner_concepts.html.
- [4] N. Artenstein and I. Revivo, “Man in the binder: He who controls ipc, controls the droid,” 2014.
- [5] U. Bayer, C. Kruegel, and E. Kirda, “Ttanalyze: A tool for analyzing malware,” in *Proc. of EICAR*, 2006.
- [6] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proc. of USENIX ATC*, 2005.
- [7] D. Bornstein, “Dalvik VM internals,” in *Google I/O*, 2008.
- [8] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, “Automatically identifying trigger-based behavior in malware,” *Botnet Detection*, 2008.
- [9] C. Cadar, D. Dunbar, and D. R. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI*, 2008.
- [10] L. Cavallaro, P. Saxena, and R. Sekar, “On the limits of information flow techniques for malware analysis and containment,” in *DIMVA*, 2008.
- [11] Contagio Mobile, “Mila Parkour,” <http://contagiominiidump.blogspot.com>.
- [12] D. Desai, “Malware Analysis Report: Trojan: AndroidOS/Zitmo,” September 2011, http://www.kindsight.net/sites/default/files/android_trojan_zitmo_final_pdf_17585.pdf.
- [13] A. Developers, “Parcelable,” <http://developer.android.com/reference/android/os/Parcelable.html>.
- [14] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” in *USENIX OSDI*, 2010.
- [15] W. Enck, D. Ocateau, P. McDaniel, and S. Chaudhuri, “A study of android application security,” in *USENIX Security*, 2011.
- [16] W. Enck, M. Ongtang, and P. McDaniel, “On lightweight mobile phone application certification,” in *ACM CCS*, 2009.
- [17] —, “Understanding android security,” *IEEE Security and Privacy*, vol. 7, no. 1, pp. 50–57, Jan. 2009. [Online]. Available: <http://dx.doi.org/10.1109/MSP.2009.26>
- [18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” in *Proc. of CCS*, 2011.
- [19] S. Fiegeman, “Android now has 1 billion active users,” Jun 2014.
- [20] T. Garfinkel and M. Rosenblum, “A Virtual Machine Introspection Based Architecture for Intrusion Detection,” in *Proc. of NDSS*, 2003.
- [21] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, “Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications,” *CoRR*, vol. abs/1402.4826, 2014.
- [22] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, “Juxtapp: A scalable system for detecting code reuse among android applications,” in *Proc. of DIMVA*, 2012.
- [23] M. Hibben, “Apple ios vs. android: The wealth of ecosystems,” Jun 2014.
- [24] Iseclab, “Anubis,” <http://anubis.iseclab.org>.
- [25] R. King, “Google readies android ‘kitkat’ amid 1 billion device activations milestone,” Sep 2013.
- [26] D. Kirat, G. Vigna, and C. Kruegel, “BareCloud: Bare-metal Analysis-based Evasive Malware Detection,” in *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, 2014.
- [27] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “AccessMiner: Using system-centric models for malware protection,” in *Proc. of CCS*, 2010.
- [28] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, “Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors,” in *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.
- [29] H. Lockheimer, “Bouncer,” <http://googlemobile.blogspot.it/2012/02/android-and-security.html>.
- [30] McAfee, “Mcafee,” <http://www.mcafee.com>.
- [31] A. Moser, C. Kruegel, and E. Kirda, “Exploring multiple execution paths for malware analysis,” in *Proc. of the IEEE Symposium on Security and Privacy*, 2007.
- [32] J. Oberheide and C. Miller, “Dissecting the Android’s Bouncer,” *SummerCon*, 2012, <http://jon.oberheide.org/files/summercon12-bouncer.pdf>.
- [33] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Using probabilistic generative models for ranking risks of android apps,” in *ACM CCS*, 2012.
- [34] B. PETROVAN, “Xposed framework creator weighs in on lollipop, art, and xposed,” <http://www.androidauthority.com/xposed-framework-lollipop-540928/>.
- [35] V. Rastogi, Y. Chen, and W. Enck, “Appsgplayground: Automatic security analysis of smartphone applications,” in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’13. New York, NY, USA: ACM, 2013.
- [36] A. Reina, A. Fattori, and L. Cavallaro, “A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors,” in *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.
- [37] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, “On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices,” in *SECURITY*, Jul. 2013.
- [38] A. Slowinska and H. Bos, “Pointless tainting?: evaluating the practicality of pointer tainting,” in *EuroSys*, W. Schröder-Preikschat, J. Wilkes, and R. Isaacs, Eds. ACM, 2009, pp. 61–74.
- [39] Statista, “Cumulative number of apps downloaded from the google play android app store as of july 2013,” Jul 2013.
- [40] The Honeynet Project, “Droidbox,” <https://code.google.com/p/droidbox/>.
- [41] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE S&P*, 2007.
- [42] R. Xu, H. Sardi, and R. Anderson, “Aurasium: Practical policy enforcement for android applications,” in *Proc. of USENIX Security*, 2012.
- [43] L.-K. Yan and H. Yin, “DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis,” in *Proc. of USENIX Security*, 2012.
- [44] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, “Vetting undesirable behaviors in android apps with permission use analysis,” in *ACM CCS*, 2013.
- [45] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, “Smart-Droid: an automatic system for revealing UI-based trigger conditions in Android applications,” in *Proc. of SPSM*, 2012.
- [46] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party android marketplaces,” in *Proc. of CODASPY*, 2012.
- [47] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *Proc. of the IEEE Symposium on Security and Privacy*, 2012.
- [48] 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 *Proc. of NDSS*, 2012.
- [49] Y. Zhou and X. Jiang, “Android Malware Genome Project,” <http://www.malgenomeproject.org/>.

APPENDIX

Malware Family	Samples w/ Add. Behaviors	Behavior w/o Stim.	Incr. Behavior w/ Stim.	Malware Family	Samples w/ Add. Behaviors	Behavior w/o Stim.	Incr. Behavior w/ Stim.	Malware Family	Samples w/ Add. Behaviors	Behavior w/o Stim.	Incr. Behavior w/ Stim.
Ackposts	1/1	4	+3 (+75%)	GameX Dropper	1/1	8	+1 (+13%)	RuFraud	4/6	4.5	+5 (+111%)
AcTrack	1/1	4	+1 (+25%)	Geinimi	11/19	23.68	+12.4 (+52%)	SGSpy	1/1	60	+39 (+65%)
AndroidSMS	2/2	0	+1 (L)	GGeeGame	1/1	62	+6 (+10%)	SGSpyAct	0/1	0	+0 (L)
Anserver	13/21	16.48	+5.2 (+32%)	GoldenDream	7/8	31.12	+9.9 (+32%)	ShdBreak	0/1	28	+0 (+0%)
ApkMon	1/2	49	+1 (+2%)	GoldenEagle	1/1	0	+7 (L)	SilentWap	3/3	2	+5 (+250%)
AppHnd	4/4	37.25	+16.8 (+45%)	GoneSixty	11/11	16.64	+5.5 (+33%)	SMS.*	16/21	4.77	+8.59 (+180%)
AreSpy	1/1	11	+6 (+55%)	GpsNake	0/1	1	+0 (+0%)	Sngo	1/1	65	+2 (+3%)
Arspam	1/1	3	+2 (+67%)	HippoSMS	1/1	16	+4 (+25%)	Spitmo	2/2	0	+9 (L)
BackReg	1/1	78	+12 (+15%)	Hnway	0/1	49	+0 (+0%)	SpyBubb	2/2	25.5	+20 (+78%)
Backscript	2/6	9.67	+19.5 (+202%)	Imlog	5/6	19	+9.2 (+48%)	Spytrack	1/1	20	+8 (+40%)
BaseBridge	10/12	4.5	+3.3 (+73%)	IMWebViewer	1/1	94	+11 (+12%)	Stamper	1/1	63	+7 (+11%)
Bgyoulu	3/5	17.6	+4 (+23%)	InstBBridge	0/1	9	+0 (+0%)	SteamyScr	2/2	25.5	+8.5 (+33%)
BookFri	1/1	15	+4 (+27%)	J	7/13	30.96	+3.65 (+12%)	Steek	15/15	8.4	+2.1 (+25%)
Carotap	2/2	4	+3 (+75%)	Jifake	1/5	1	+4 (+400%)	Stiniter	0/1	3	+0 (+0%)
Coolpaperleek	1/1	55	+4 (+7%)	Jmsonez	2/2	11.5	+12 (+104%)	Sunzand	0/3	7	+0 (+0%)
Crusewin	4/4	6.25	+8.5 (+136%)	LdBolt	8/8	46.62	+7.8 (+17%)	SusetupTool	0/1	0	+0 (L)
Dialer	0/1	1	+0 (+0%)	LoggerKid	4/4	4.5	+2 (+44%)	Sxjspy	1/1	24	+4 (+17%)
DitesEx	23/43	26.58	+8.9 (+33%)	Logkare	0/1	0	+0 (L)	TattooHack	1/2	6	+1 (+17%)
DIYAds	18/18	163.72	+37.6 (+23%)	LoveTrp	1/1	5	+6 (+120%)	Tcent	1/1	0	+17 (L)
DougaLeaker	16/16	4	+1.6 (+40%)	Lvedu	33/56	26.93	+5.2 (+19%)	ToorKing	1/1	37	+6 (+16%)
Drd.*	5/5	10.6	+6 (+57%)	Maistrealer	1/1	8	+1 (+13%)	ToorSatp	3/8	7.5	+1.3 (+17%)
DroidDeluxe	30/32	24.74	+7.55 (+31%)	Malebook	1/1	94	+14 (+15%)	Toplank	6/9	37.44	+6 (+16%)
DroidKungFu	3/85	9	+1 (+11%)	Mania	1/2	0.5	+2 (+400%)	Twikabot	1/1	0	+12 (L)
DropDialer	2/11	0	+6.1 (+20%)	MarketPay	1/1	98	+7 (+7%)	TypStu	4/6	0.83	+1 (+120%)
Ecobatr	1/1	25	+1.5 (L)	Mob.*	11/11	43.67	+9.75 (+22%)	UranaiCall	1/1	51	+13 (+25%)
EICAR	0/2	1.5	+1 (4%)	Moghava	1/1	0	+2 (L)	VDLoader	10/10	43.7	+8.8 (+20%)
Enesoluty	1/1	11	+0 (+0%)	MoneyFone	1/1	0	+3 (L)	Vidro	1/1	58	+16 (+28%)
EvoRoot	0/1	0	+2 (+18%)	Nandrobox	1/1	0	+4 (L)	Voldbrk	9/17	48.82	+1.2 (+2%)
Fake.*	314/677	6.39	+5.69 (+89%)	Netisend	1/1	8	+4 (+50%)	WalkTxt	1/1	14	+2 (+14%)
FlashStep	1/1	176	+80 (+45%)	NotCompatible	2/2	71	+10.5 (+15%)	Wapaxy	2/2	0	+9 (L)
FlashRec	1/2	8	+3 (+38%)	Nyearleaker	0/1	7	+0 (+0%)	Wooboolleaker	1/1	5	+2 (+40%)
FndNCII	1/1	36	+2 (+6%)	OneClickFraud	22/22	23	+5 (+22%)	XanireSpy	9/9	27.11	+5.9 (+22%)
Foney	2/2	1	+4 (+400%)	PdaSpy	1/4	0	+17.2 (+106%)	XobSms	1/1	28	+15 (+54%)
FoneyDropper	1/1	23	+1 (+4%)	PIApps	36/39	27.41	+6.1 (+22%)	YiCha	10/10	21.5	+4.6 (+21%)
FriectSpy	8/9	7.56	+10 (+132%)	Qicsomos	0/1	15	+0 (+0%)	Zitmo	3/3	2.67	+5.7 (+213%)
Frogonal	2/2	27.5	+2.5 (+9%)	QueTing	1/1	0	+4 (L)				
Frunk	1/1	73	+17 (+23%)	QuoteDoor	0/1	6	+0 (+0%)				
FunsBot	2/2	5	+2 (+40%)	ResCaller	1/1	2	+4 (+200%)				
GameX	1/1	11	+2 (+18%)	RootSmart	2/2	17	+9 (+53%)				
Overall								Overall	836/1365	22.78	+6.54 (+28.7%)

Table VI: Results of the stimulation shown in three adjacent tables. First column reports the malware family, second column reports the number of samples that exhibited additional behaviors over the total number of samples belonging to the same family, third column report the average number of observed behaviors without stimulation and last column reports the average number of additional behaviors exhibited by stimulated samples and their percentage over non-stimulated behaviors.