A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors

Alessandro Reina
Dept. of Computer Science
Università degli Studi di Milano
ale@security.di.unimi.it

Aristide Fattori
Dept. of Computer Science
Università degli Studi di Milano
aristide@security.di.unimi.it

Lorenzo Cavallaro
Information Security Group
Royal Holloway, University of London
lorenzo.cavallaro@rhul.ac.uk

ABSTRACT

With more than 500 million of activations reported in Q3 2012, Android mobile devices are becoming ubiquitous and trends confirm this is unlikely to slow down. App stores, such as Google Play, drive the entire economy of mobile applications. Unfortunately, high turnovers and access to sensitive data have soon attracted the interests of cybercriminals too with malware now hitting Android devices at an alarmingly rising pace. In this paper we present CopperDroid, an approach built on top of QEMU to automatically perform out-of-the-box dynamic behavioral analysis of Android malware. To this end, CopperDroid presents a unified analysis to characterize low-level OS-specific and high-level Android-specific behaviors. Based on the observation that such behaviors are however achieved through the invocation of system calls, CopperDroid's VM-based dynamic system call-centric analysis is able to faithfully describe the behavior of Android malware whether it is initiated from Java, JNI or native code execution.

We carried out extensive experiments to assess the effectiveness of our analyses on a large Android malware data set of more than 1,200 samples belonging to 49 Android malware families (provided by the Android Malware Genome Project) and about 400 samples over 13 families (collected from the Contagio project). Our experiments show that a proper malware stimulation strategy (e.g., sending SMS, placing calls) successfully discloses additional behaviors on a non-negligible portion of the analyzed malware samples.

1. INTRODUCTION

With more than 500 million of activations reported in Q3 2012, Android mobile devices are becoming ubiquitous and trends show that such a pace is unlikely slowing down [16]. Android devices are extremely appealing: powerful, with a functional and easy-to-use user interface to access sensitive user and enterprise data, they can easily replace traditional computing devices, especially when information is mostly consumed rather than produced.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSec '13, April 14 2013, Prague, Czech Republic
Copyright 2013 ACM 978-1-4503-2120-4/13/04 ...$15.00.

Application marketplaces, such as Google Play and the Apple App Store, drive the entire economy of mobile applications. For instance, with more than 600,000 applications installed, Google Play has generated revenues of about 237M USD per year [9]. Such a wealth and quite unique ecosystem with high turnovers and access to sensitive data have unfortunately also attracted the interests of cybercriminals, with malware now hitting Android devices at an alarmingly rising pace. Users privacy breaches (e.g., access to address book and GPS coordinates) [28], monetization through premium SMS and calls [28], and colluding malware to bypass 2-factor authentication schemes [8] are all real threats rather than a fictional forecasting. Recent studies back easily such statements up, reporting how mobile marketplaces have been abused to host malware or legitimate-resembling applications (usually games) embedding malicious components [26].

Unfortunately, the nature of Android applications makes it hard—if not impossible—to rely on existing VM-based dynamic malware analysis systems as is. In fact, Android applications are generally written in the Java programming language and executed on top of the Dalvik virtual machine [5], but native code invocation is however possible via JNI or Linux ELF binary execution. This mixed environment seems to suggest the need to reconstruct and keep in sync out-of-the-box semantics through virtual machine introspection (VMI) [11] for both the OS and Dalvik views, as very recently shown in [25]. On the one hand, OS-level semantics (e.g., writing to a file, executing a program) would allow to characterize JNI or native ELF-induced behaviors, while Dalvik-level semantics would enable to disclose high-level Android-specific behaviors (e.g., sending an SMS).

While true in principle, we observe that even high-level Android-specific behaviors are indeed achieved via system call invocations, underneath. In fact, as described later, Android applications may interact *with the system* via well-defined system call-initiated IPC and RPC invocations to carry out their tasks. Although Java-related information can undoubtedly further aid malware analysts to understand fine-grained behaviors (e.g., encryption of user data), they seem unnecessary to describe and understand the fundamental actions Android malware perform.

In this paper we present CopperDroid, an approach built on top of QEMU [4] to *automatically* perform out-of-the-box dynamic behavioral analysis of Android malware. To this end, CopperDroid presents a *unified* analysis to characterize low-level OS-specific (e.g., opening and writing to a file, executing a program) and high-level Android-specific

(e.g., accessing personal information, sending an SMS) behaviors. In particular, based on the observation that such behaviors are all 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, JNI or ELF code.

In summary, we make the following contributions:

1. We describe the design and implementation of a unified dynamic analysis technique to characterize the behavior of Android malware. Our analysis is able to automatically describe low-level OS-specific and high-level Android-specific behaviors of Android malware by observing and analyzing system call invocations, including IPC and RPC interactions, carried out as system calls underneath.

2. Based on the observation that Android applications are inherently user-driven and feature a number of implicit but well-defined entry points, we outline the design and implementation of a stimulation approach aimed at disclosing additional malware behaviors.

3. We provide a thorough evaluation of CopperDroid's analysis on more than 1,200 malware samples belonging to 49 Android malware families as provided by the Android Malware Genome Project [27] and about 400 samples over 12 Android malware families from the Contagio project [7]. Our experiments show that CopperDroid is able to *automatically* and *faithfully* describe the behavior of the samples in our data sets. Furthermore, CopperDroid confirms the importance of a proper malware stimulation approach (e.g., sending SMS, placing calls), which allowed us to disclose an average of 28% of unique additional behaviors on more than 60% of the Android Malware Genome Project's samples and 22% unique additional behaviors on more than 70% of the Contagio-provided samples.

4. We developed a web interface[1] through which our users can submit samples to be analyzed by CopperDroid. Results contains behavioral analysis (both in HTML and JSON format, for easy parsing) and many ancillary information (e.g., network traffic).

Although a non-negligible implementation effort, we however consider the framework we developed and briefly describe in Section 3 as a mere yet necessary mechanism to carry out our actual contributions, i.e., CopperDroid's VMI-based system call-centric analysis, malware stimulation approach, and extensive evaluation on large data sets.

2. THE ANDROID SYSTEM

Android applications are typically written in the Java programming language and then deployed as Android Packages archive (APKs). Every APK is considered to be a self-contained application that can be logically decomposed into one or more components. Each component is generally designed to fulfill a specific application task (e.g., GUI-related actions, notification receiver) and it is invoked either by the user or the OS.

According to the Android security model [2], each application runs in a separate userspace process, as an instance

[1] Available at: `http://copperdroid.isg.rhul.ac.uk/`

```
<uses-permission android:name="[...].RECEIVE_SMS" />
<uses-permission android:name="[...].SEND_SMS" />
<uses-permission android:name="[...].INTERNET" />
...
 <receiver android:name=".SMSReceiver">
  <intent-filter>
   <action android:name="..Telephony.SMS_RECEIVED" />
  </intent-filter>
 </receiver>
```

Figure 1: `AndroidSMS` Manifest File.

of the Dalvik virtual machine (DVM) [5], usually with a distinct user and group ID.

Although isolated within their own sandboxed environment, Android applications can interact with other applications, and with the system, through a well-defined API. A number of components can make up an application. In particular, Android defines *activities*, *services*, *content providers*, and *broadcast receivers*.

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 broadcast receivers.

2.1 Manifests

Android manifests are XML files that must be included in every APK. A manifest declares application components as well as 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 application is willing to handle.

Figure 1 reports a stripped-down Android manifest of `AndroidSMS`, a fictional application we developed for explanatory purposes. The manifest clearly shows the application requires permission to receive and send SMS, and to access the Internet. Furthermore, `AndroidSMS` declares a broadcast receiver component (class `SMSReceiver`) that will respond to `SMS_RECEIVED` intents.

Android manifests contain a number of interesting information and their inspection can indeed disclose preliminary insights about an application maliciousness [29].

2.2 Binder: IPC and RPC

The Android OS and applications strongly rely on interprocess communication (IPC) and remote procedure calls (RPCs). To this end, Android uses Binder, a custom implementation of the OpenBinder protocol [20]. As the Binder protocol is quite complex, we highlight next only the information needed to understand CopperDroid's analysis.

Just like any other RPC mechanism, Binder allows a Java process (e.g., an application) to invoke methods of remote objects (e.g., services) as if they were local methods, through synchronous calls. This is transparent to the caller and all the underlying details (e.g., message forwarding to appropriate receivers, start or stop of processes) are handled by the Binder protocol during the remote invocation.

To work properly, the caller application must know the remotely-callable methods with parameters. This is achieved through the Android Interface Definition Language (AIDL), which is leveraged by "server-side" components developers. Once defined, an AIDL file is used to automatically gener-

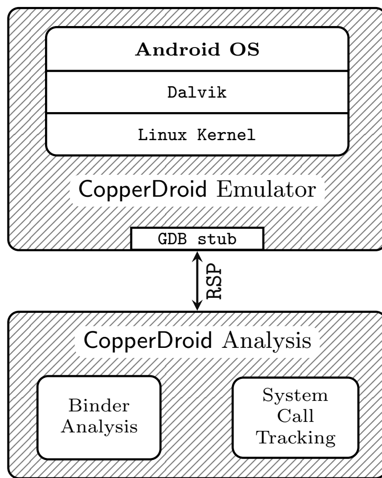

Figure 2: CopperDroid Architecture.

ate client- and server-side code in the form of a proxy class, used by a caller, and a stub class, extended by the callee to implement the logic of the service.

The AIDL files of core Android services are available online. As described later, CopperDroid relies on such interfaces to automatically infer the interactions between applications from low-level events. Although a few AIDL files may be missing (e.g., custom services), CopperDroid has never experienced such an issue in our current experiments. Manual reverse engineering and ad-hoc unmarshalling procedures can be introduced to handle such specific scenarios, even if full reverse engineering IPC-stub generation automation is part of our on-going research effort.

3. COPPERDROID

The architecture of CopperDroid is shown in Figure 2. Our whole Android system runs on top of a modified Android emulator (the CopperDroid emulator), which is built on top of QEMU [4]. 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. As Figure 2 shows, *all our analyses* are executed outside the CopperDroid emulator and we rely on virtual machine introspection (VMI) [11] to fill the semantic gap between the CopperDroid emulator and the whole Android system.

To allow for a flexible host-to-emulator communication and introspection, CopperDroid leverages the remote serial protocol (RSP) of the GNU debugger [12] (see Figure 2). The Android emulator provides GDB support via GDB stubs to developers. A GDB stub is an implementation of RSP, which enables the target machine to communicate with the host machine on which a remote GDB session with a client is established. Therefore, any client that is able to communicate over RSP can debug the target machine. Please note that this *does not* modify *anyhow* the analyzed Android system, nor it can be detected by apps running inside CopperDroid's emulator.

3.1 Tracking System Call Invocations

Tracking system call invocations is at the basis of virtually all the dynamic malware behavioral analysis systems [13,14, 23]. 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 ARM architecture underlying the Android emulator—and therefore CopperDroid—presents a few details that may challenge VMI-based system call invocations tracking and are thus worth describing.

The ARM ISA provides the `swi` instruction for invoking system calls, which causes the well-known user-to-kernel transition by triggering a software interrupt. Once the `swi` instruction is executed, the `cpsr` register is set to `supervisor` mode with the program counter register pointing to the system call handler. To track system call invocations, we instrument QEMU when the `swi` instruction is executed. That instruction is not (dynamically) binary translated and can therefore easily be intercepted when QEMU handles the proper software interrupt. When the `swi` instruction is intercepted, we check if a system call is actually being invoked, if that is in the list of the to-be-tracked system calls, and if the current process is in the list of the to-be-monitored processes. Of course, it is also of paramount importance to detect when a system call is about to return as that allows to save its return value, which enriches the 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`). Therefore, instead of relying on a cumbersome heuristic, the generic approach CopperDroid adopts 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 to uniformly retrieve system call return values, if any.

3.2 Binder Analysis: Dissecting IPC and RPC

As outlined in Section 2.2, the Android system heavily relies on kernel-implemented IPC and RPC channels to carry out tasks and (some) permission-related policy enforcement. Therefore, tracking and dissecting the communications that happen over this media is a key aspect for reconstructing high-level Android-specific behaviors. Although recently explored to enforce user-authorized security policies [24], 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-specific and Android-specific behaviors of malicious Android applications.

Let us consider an application that sends an SMS as our running example. From a high-level perspective (e.g., Java methods), sending an SMS 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` on the instance, with the destination phone number and the text message as the method's arguments. Overall, this corresponds to locating the Binder service `isms` and remotely invoking its `sendText` function with the proper arguments.

Conversely, from a 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 to identify the invoked method and its parameters, enabling

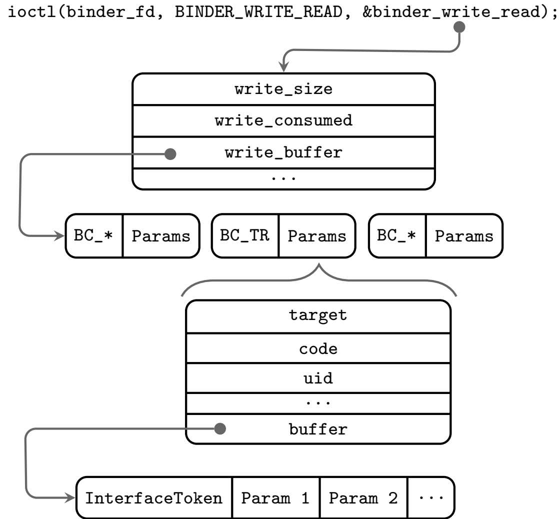

Figure 3: Parameters of a `BINDER_WRITE_READ` `ioctl`.

de-facto to infer the high-level semantic of the operation.

Although the Binder protocol implements other `ioctls`, the `BINDER_WRITE_READ` is the most important one as it allows to transfer data between processes. Figure 3 depicts a few details about the parameter of these `ioctls`. As can be observed, they may embed one or more operations for the Binder protocol. These operations are stored sequentially in the `write_buffer` field of the `ioctl`'s last argument. The Binder protocol supports a number of operations, but CopperDroid focuses only on *transactions*, i.e., IPC operations that actually transfer data. In particular, it focuses on `BC_TRANSACTION` and `BC_REPLY`, operations responsible to initiate and return an answer to IPC transactions, respectively.

Just intercepting transactions may however be of limited use when it comes to understand Android-specific behaviors, which only a thorough analysis can eventually disclose. CopperDroid dynamically parses the structure depicted in Figure 3 and retrieves all the valuable transaction-provided information to describe Android-specific behaviors. In particular, we focus on the `buffer` field that usually contains a string (e.g., *InterfaceToken*), which identifies an interface implemented by the *callee*. The string is followed by the RPC parameters, properly serialized by a custom Android marshalling protocol (*parcel*).

To understand the invoked method and the unmarshalling procedure for its parameters, CopperDroid uses a novel technique. First it identifies the *InterfaceToken* specified in the payload. Then, such information is used to find the AIDL description of the interface CopperDroid needs, to associate the numeric `code` to the invoked method and to understand the types of its parameters. This step is necessary because, even if a parcel includes methods to create easily unmarshallable stream of bytes (including metadata to associate bytes to types), payloads are often marshalled directly as the receiver knows exactly how to unmarshall them.

3.3 Path Coverage

Although effective, a simple install-then-execute dynamic analysis may miss a number of interesting (malicious) behaviors. On the one hand, this problem has long been affecting traditional dynamic analysis approaches as non-exercised paths are simply unanalyzed. If such paths host additional (or the only) malicious behaviors, then any dynamic analysis would fail unless proper, but generally expensive and complex exploration techniques are adopted [6, 17]. On the other hand, 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 with a manifest similar to the one depicted in Figure 1. After installation, the application would only react to the reception of SMS, showing no interesting nor additional behavior otherwise.

Furthermore, traditional executables have a single entry point, while Android applications may have multiple ones. Most applications have a main activity, but ancillary activities may be triggered by the system or by other applications and the execution may reach them *without* flowing through the main. To address such coverage problem, CopperDroid implements a novel approach (based on extracting information from the malware Manifest) to artificially stimulate the analyzed malware with a number of events of interest. For example, injecting events such as phone calls and reception of SMS texts would lead to the execution of the registered application's broadcast receivers. Another example that comes from our experience with Android Malware is the `BOOT_RECEIVED` intent that many samples use to get executed as soon as the victim system is booted (much like `\CurrentVersion\Run` registry keys on Windows systems).

4. EVALUATION

This section presents the experiments and results of a thorough evaluation of CopperDroid on two considerably large sets of Android malware. The first is provided by the Android Malware Genome Project [27] and consists of more than 1,200 malware belonging to 49 different families. The second is made of around 400 malware samples gathered from Contagio [7].

Our experimental setup is as follows. We run an unmodified Android 2.2.3 image on top of our CopperDroid-enhanced emulator. The system is customized to include personal information, such as contacts, SMS texts, call logs, and pictures. Each analyzed malware sample is installed in the emulator and traced until a timeout is reached. At the end of the analysis, a *clean* execution environment is restored to prevent corruptions caused by installing more than one sample in the same system.

4.1 Behaviors under Stimulation

To evaluate the effectiveness of the stimulation approach of CopperDroid we proceed as follows. First, we analyze the whole set of samples without external stimulation. Then, we perform the stimulation-driven analysis of the same malware set, as outlined in Section 3.3.

Once system call traces of all samples in both phases are collected, we extract the behaviors observed during these two different executions. For each sample i, we create two sets T_i and N_i containing the behaviors observed during the analyzed execution of i, respectively with and without stimulation. Our choice of working with high-level representation of behaviors is dictated by the fact that *directly* comparing the sets of collected system calls would be too fine-grained for our purposes. Nonetheless, one more problem still persists. To understand the usefulness of our stimula-

#	Malware Family	Samples w/ Add. Behav.	Behavior w/o Stim.	Inc. Behav. w/ Stimuli
	Android Malware Genome Project			
1	ADRD	17/21	7.24	4.5 (62%)
2	AnserverBot	186/187	31.52	8.2 (26%)
3	Asroot	0/8	5.62	0.0 (0%)
4	BaseBridge	70/122	16.44	5.2 (32%)
5	BeanBot	4/8	0.12	3.8 (3000%)
6	Bgserv	9/9	31.22	6.8 (22%)
7	CoinPirate	1/1	7.00	6.0 (86%)
8	CruseWin	2/2	1.00	2.0 (200%)
9	DogWars	0/1	0.00	0.0 (⊥)
10	DroidCoupon	1/1	4.00	6.0 (150%)
11	DroidDeluxe	1/1	10.00	1.0 (10%)
12	DroidDream	15/16	26.88	5.7 (22%)
13	DroidDreamL.	44/44	7.18	1.7 (25%)
14	DroidKungFu1	4/34	5.03	2.0 (40%)
15	DroidKungFu2	3/30	12.23	4.0 (33%)
16	DroidKungFu3	158/308	16.31	4.5 (28%)
17	DroidKungFu4	31/94	21.69	4.7 (22%)
18	DroidKungFuS.	0/3	3.00	0.0 (0%)
19	DroidKungFuU.	1/1	12.00	1.0 (9%)
20	Endofday	1/1	0.00	1.0 (⊥)
21	FakeNetflix	0/1	0.00	0.0 (⊥)
22	FakePlayer	0/6	3.17	0.0 (0%)
23	GGTracker	1/1	10.00	2.0 (20%)
24	GPSSMSSpy	6/6	0.00	2.3 (⊥)
25	GamblerSMS	1/1	1.00	3.0 (300%)
26	Geinimi	34/63	16.19	3.1 (20%)
27	GingerMaster	0/4	29.00	0.0 (0%)
28	GoldDream	43/47	15.28	6.3 (42%)
29	Gone60	9/9	7.00	1.0 (15%)
30	HippoSMS	4/4	6.25	1.5 (24%)
31	Jifake	1/1	2.00	2.0 (100%)
32	KMin	45/51	66.47	2.2 (4%)
33	LoveTrap	1/1	5.00	2.0 (40%)
34	NickyBot	0/1	14.00	0.0 (0%)
35	NickySpy	0/2	15.50	0.0 (0%)
36	Pjapps	24/40	7.35	4.1 (57%)
37	Plankton	2/11	9.55	1.0 (11%)
38	RogueLem.	2/2	8.50	4.0 (48%)
39	RogueSPP.	3/9	10.00	3.0 (30%)
40	SMSReplic.	1/1	0.00	6.0 (⊥)
41	SndApps	0/10	1.00	0.0 (0%)
42	Spitmo	1/1	0.00	8.0 (⊥)
43	Tapsnake	0/2	1.50	0.0 (0%)
44	Walkinwat	1/1	12.00	1.0 (9%)
45	YZHC	1/22	0.05	6.0 (13200%)
46	Zitmo	1/1	1.00	5.0 (500%)
47	Zsone	12/12	16.67	3.8 (23%)
48	jSMSHider	1/13	18.46	1.0 (6%)
49	zHash	10/11	11.00	3.1 (29%)
–	**Overall**	**752/1226**	**10.3**	**2.9** (28%)
	Contagio			
1	AnserverBot	3/3	11.00	8.0 (73%)
2	BaseBridge	1/4	2.50	2.0 (80%)
3	Beauty	3/3	30.67	6.3 (21%)
4	DroidDream	7/11	26.00	3.3 (13%)
5	DroidKungFu	0/2	12.00	0.0 (0%)
6	Geinimi	27/37	22.70	4.6 (21%)
7	GoldDream	1/2	59.00	11.0 (19%)
8	KMin	45/47	69.87	4.3 (7%)
9	Pjapps	14/15	23.40	5.6 (24%)
10	RootExploit	8/10	14.90	2.2 (16%)
11	Steek	16/17	14.00	10.9 (79%)
12	UNCAT.	157/237	11.77	3.6 (31%)
13	Zitmo	7/7	8.71	5.6 (64%)
–	**Overall**	**289/395**	**23.6**	**5.2** (22%)

Table 1: Results of the stimulation. 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.

tion approach in analyzing one sample, for instance, we have to compare both sets T_i and N_i and observe if any behavior appear in the first set but not in the latter. If this condition holds, then it is likely that our stimulation induced the malware to follow different execution paths, exposing previously unseen behaviors.

However, comparing the two collected sets would not lead to the desired result either. Intuitively, the simplest way to extract stimulation-induced behaviors is to compute $C_i = T_i \setminus N_i$, the *relative complement* of N_i relative to T_i. Then, if $C_i \neq \emptyset$, C_i contains the set of stimulation-only induced behaviors. In addition, as outlined above, every high-level representation of a behavior comes with a set of additional information that allow fine-grained observations and comparisons. Such information are useful to perform a normalization on elements of N_i and T_i, to prevent an overestimation of C_i (e.g., two `HTTP GET` requests to the same host/page with a random parameter that would otherwise be considered distinct). Our future work includes experimenting with more sophisticated invariant-based analysis [18] and clustering techniques [21] to prevent such overestimation.

Table 1 reports the results of applying the analysis just outlined on the whole set of samples counting more than 1,600 malware. The overall results support the effectiveness of our stimulation approach. We can observe an average of 28% additional behaviors on more than 60% of the Android Malware Genome Project's samples, and an average of 22% additional behaviors on roughly 73% of the Contagio's samples.

5. RELATED WORK

In this section we cite and compare against the most relevant work we believe directly relates with CopperDroid.

DroidScope [25] 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 VMI [11] to gather information about the system and exposes hooks and a set of APIs, which enable the development of plugins to perform both fine and coarse-grained analyses (e.g., system call, single instruction trac-

ing, and taint tracking). In principle, CopperDroid could have been built on top of DroidScope, but at the time we implemented it, DroidScope's 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 includes a deep IPC/RPC Binder protocol analysis inspection—and stimulation technique can comprehensively expose Android malware behaviors, as shown by our extensive evaluation.

Andrubis [1] is an extension to the Anubis dynamic malware analysis system to analyze Android malware [3, 13]. According to its web site, it is mainly built on top of both TaintDroid [10] and DroidBox [22] and it thus shares their weaknesses (mainly due to operating "into-the-box"). In addition, Andrubis does not perform any stimulation-based analysis, limiting its effectiveness in discovering interesting Android-specific behaviors.

Aurasium [24] is a technique (and a tool) that enables dynamic and fine-grained policy enforcement of Android applications. To intercept relevant events, Aurasium instruments single applications, rather than adopting system-level hooks. Working at the application level, however, exposes Aurasium to easy detection or evasion attacks by malicious Android applications.

Google Bouncer [15], 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 [19] and it is thus impossible to compare it against our approach.

6. REFERENCES

[1] Andrubis: A tool for analyzing unknown android applications. `http://anubis.iseclab.org/`.

[2] Android. Android developer reference. `http://developer.android.com/reference/packages.html`.

[3] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *Proc. of EICAR*, 2006.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of USENIX ATC*, 2005.

[5] D. Bornstein. Dalvik VM internals. In *Google I/O*, 2008.

[6] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. *Botnet Detection*, 2008.

[7] Contagio Mobile. Mila Parkour. `http://contagiominidump.blogspot.com`.

[8] D. Desai. Malware Analysis Report: Trojan: AndroidOS/Zitmo, Semptember 2011. `http://www.kindsight.net/sites/default/files/android_trojan_zitmo_final_pdf_17585.pdf`.

[9] M. Egele. Invited talk: The state of mobile security. In *DIMVA*, 2012.

[10] 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 *Proc. of USENIX OSDI*, 2010.

[11] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. of NDSS*, 2003.

[12] B. Gatliff. Embedding with gnu: the gdb remote serial protocol. `http://www.huihoo.org/mirrors/pub/embed/document/debugger/ew_GDB_RSP.pdf`, 1999.

[13] Iseclab. Anubis. `http://anubis.iseclab.org`.

[14] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. AccessMiner: Using system-centric models for malware protection. In *Proc. of CCS*, 2010.

[15] H. Lockheimer. Bouncer. `http://googlemobile.blogspot.it/2012/02/android-and-security.html`.

[16] T. Mai. Android Reaches 500 Million Activations Worldwide. http://www.tomshardware.com/news/Google-Android-Activation-half-billion-Sales,17556.html, 2012.

[17] 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.

[18] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. of the IEEE Symposium on Security and Privacy*, 2005.

[19] J. Oberheide and C. Miller. Dissecting the Android's Bouncer. *SummerCon*, 2012. `http://jon.oberheide.org/files/summercon12-bouncer.pdf`.

[20] Palmsource Inc. Open binder documentation. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`.

[21] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. of the USENIX NSDI*, 2010.

[22] The Honeynet Project. Droidbox. `https://code.google.com/p/droidbox/`.

[23] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Proc. of the IEEE Symposium on Security & Privacy*, 2007.

[24] R. Xu, H. Saıdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proc. of USENIX Security*, 2012.

[25] 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.

[26] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of CODASPY*, 2012.

[27] Y. Zhou and X. Jiang. Android Malware Genome Project. `http://www.malgenomeproject.org/`.

[28] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the IEEE Symposium on Security and Privacy*, 2012.

[29] 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.