Monitoring Real Android Malware

Jan-Christoph Küster[1,2] and Andreas Bauer[3]

[1]NICTA*, [2]Australian National University, [3]TU München

Abstract. In the most comprehensive study on Android attacks so far (undertaken by the Android Malware Genome Project), the behaviour of more than 1,200 malwares was analysed and categorised into common, recurring groups of attacks. Based on this work (and the corresponding actual malware files), we present an approach for specifying and identifying these (and similar) attacks using runtime verification.
While formally, our approach is based on a first-order logic abstraction of malware behaviour, it practically relies on our Android event interception tool, MonitorMe, which lets us capture almost any system event that can be triggered by apps on a user's Android device.
This paper details on MonitorMe, our formal specification of malware behaviour and practical experiments, undertaken with various different Android devices and versions on a wide range of actual malware incarnations from the above study. In a nutshell, we were able to detect real malwares from 46 out of 49 different malware families, which strengthen the idea that runtime verification may, indeed, be a good choice for mobile security in the future.

1 Introduction

The landmark work undertaken by the Android Malware Genome Project (AMGP, [15]) is the first that comprehensively collected and systematically analysed more than 1,200 Android malware samples. Despite the high total amount of unique samples, their study reveals that those can be divided into only 49 families and described by even fewer recurring attack patterns, which fall into the following categories: information stealing, financial charges, privilege escalation and malicious payload activation.

Inspired by those patterns, we formalise in this paper common malicious behaviour in our own specification language (published prior in [2]) to dynamically identify real malware on a user's Android device; that is, by checking its runtime footprint against our specifications. As our approach has access to the actual executed behaviour of apps, it can complement static analysis techniques, whose malware detection often faces difficulties in face of code obfuscation (cf. [11]). Dynamic analysis techniques on the other hand, which are often used on an emulator in order to detect malware, face difficulties with samples that employ recent emulator-detection techniques (cf. [13]). Naturally, this is not a problem either when working directly on the device.

We were able to detect suspicious behaviour of 46 out of 49 malware families from the AMGP, while generating 28% positive alerts when monitoring a representative set of 61 benign apps with our specifications.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

For conducting our experiments we have developed a standalone monitoring app, called MonitorMe. Compared to other approaches, which are usually either *app- or platform-centric* (see discussion in §5), its major advantage is that it combines the strengths of both "worlds", i.e., while it is easy to install on a user's off-the-shelf device, it is also capable of gathering all system events necessary for our analysis, without requiring the modification of either the Android platform or apps themselves (cf. [6, 4, 1, 14]). Hence, for a future, stable version of our prototype, we have the average phone user in mind, assuming that specifications are centrally created by security experts and that the app receives regular updates to them over the internet. Currently, MonitorMe runs on various devices of the Google Nexus family (tested for Nexus S, 7 and 5), and is portable to older and very recent Android versions (2.3.6, 4.3 and 5.0.1). However, we require devices to be rooted to load a Linux kernel module. This may seem restrictive, but one should keep in mind that it has become common practice by now and does not disrupt the user experience by reinstalling the system on the device at hand.

Outline. In the next section, we give a technical overview of MonitorMe. Our specification language and its use for monitoring malware is introduced in §3, followed by experiments (§4.1) demonstrating that our policies help identify most AMGP-families. The experiments further demonstrate (§4.2) that only few false positives for benign apps are generated and that performance- and portability-wise our approach does, indeed, lend itself to be executed on almost arbitrary end-user Android devices.

2 Event Interception with MonitorMe

To enable our modular way of malware detection on a user's device, we have developed a monitoring app,[1] which has two main components (depicted in Fig. 1): a framework for collecting system events on the Android platform (grey area, named *DroidTracer*[2]), and an analysis component (on top of DroidTracer), which receives those events in chronological order and incrementally "feeds" them into monitors generated by Ltlfo2mon.[3] We create a monitor for each of our policies that specifies a certain malware behaviour and run a copy of them per app under inspection. Ltlfo2mon is written in Scala, but is compatible to run as part of an Android app in Java. In short, the monitoring algorithm creates an automaton for each LTL-like subformula in a policy. These are then spawned with concrete values based on observed system events at runtime (for details of the algorithm see [2]). It is worth

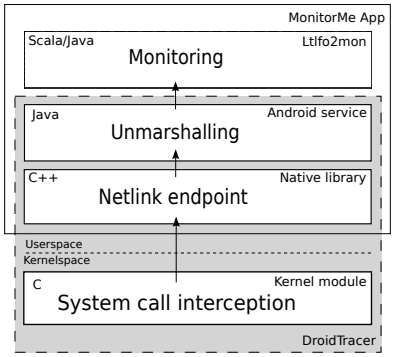

Fig. 1: Architecture of MonitorMe.

[1] http://kuester.multics.org/MonitorMe/
[2] http://kuester.multics.org/DroidTracer/
[3] https://github.com/jckuester/ltlfo2mon

pointing out that DroidTracer works without polling for events, i.e., a Java callback method is triggered whenever a new system event occurs. Furthermore, DroidTracer is implemented as a *standalone library* so that it can be integrated in third-party apps for other analyses.

DroidTracer. In the following we explain the inner workings of DroidTracer (three sub-components marked as white inside the grey area); that is, the novel way on how we intercept interactions between apps and the Android platform without requiring platform or app modifications. As there is no public API for this task, not even on a rooted device, nor any complete documentation about Android's internal communication mechanism, our approach is mainly based on insights gained from reverse engineering.

System call interception. We exploit the fact of Android's security design that the control flow of all apps' actions that require permission, such as requesting sensitive information (GPS coordinates, device id, etc.) or connecting with the outside world (via SMS, internet, etc.), *must* eventually pass one of the system calls in the Linux kernel; for example, to be delegated to a more privileged system process that handles the request. In other words, intercepting the control flow at a central point in kernel space does not allow apps to bypass our approach. Furthermore, system calls are unlikely to change so that hooking into them is fairly robust against implementation details on different Android versions.

Hence, our idea is to use kprobes[4] (the kernel's internal debugging mechanism) to intercept system calls in a non-intrusive way. More specifically, we built a custom kernel module (bottom sub-component in Fig. 1), which contains handler methods that get invoked by small bits of trampoline code (so called probes). We add those with kprobes dynamically to the following system calls:

- sys_open(const char __user *filename, ...), opens files with filename for read/write.
- sys_connect(int sockfd, const struct sockaddr *addr, ...), where addr contains the IPv4 or IPv6 address of an established internet connection.
- do_execve(char *filename, char __user *__user *argv, ...), where filename is a program or shell script being executed with the arguments argv.
- ioctl(...), is used to control kernel drivers, such as Android's Binder driver.

From the function arguments of ioctl we cannot directly retrieve relevant information (unlike for the other system calls shown above, which provide to us IP addresses, opened files, or executed program names). The reason is that information is compactly *encoded* (for efficiency reasons), when sent through ioctl by Android's own *inter process communication* (IPC) mechanism, called Binder.[5] As Binder handles the majority of interesting interactions between apps and the Android platform, its decoding is crucial for our analysis. Hence, for a deeper understanding of Binder's control flow, we look at the following Java code snippet of a method call that an app developer might write to send an SMS.

```
SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage("12345", null, "Hello!", null, null);
```

[4] https://www.kernel.org/doc/Documentation/kprobes.txt
[5] http://developer.android.com/reference/android/os/Binder.html

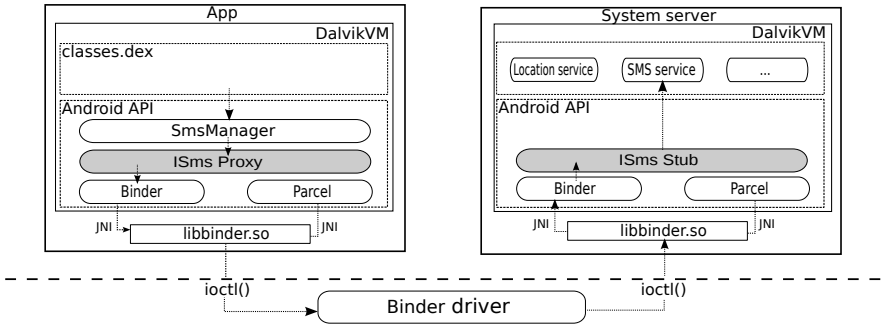

Fig. 2: An app (left) requesting the Android platform (right) to send an SMS.

Figure 2 illustrates the control flow of the Binder communication when this code is executed. All Java code of an app is compiled into a single Dalvik executable classes.dex (upper left box), which runs on its own Dalvik VM. The called method sendTextMessage() is part of the Android API (lower left box), which is linked into every app as a JAR file. But instead of implementing the functionality of sending an SMS itself, it rather hides away the technical details of a *remote procedure call* (RPC); that is, a call of a Java method that lives in another Dalvik VM (right box). What further happens is that SmsManager calls the method sendText() of the class Proxy, which has been automatically generated for the ISms interface. The Proxy then uses the class Parcel to *marshall* the method arguments of sendText() into a byte stream, which is sent (together with other method call details) via the Binder driver to the matching Stub of ISms (lower right box). There, the arguments are *unmarshalled* and the final implementation of sendText() in the SMS service is executed. As the SMS service is running on a Dalvik VM privileged to talk to the radio hardware, it can send the SMS.

Unmarshalling. The main challenge in reconstructing method calls was to reverse engineer how Binder encodes them to send them through the kernel, such that the task of unmarshalling for our analysis can be automated. Like for the code snippet of sending an SMS, we aim at reconstructing every method call in its original human readable format (including its Java method arguments and types). In what follows, we describe how we achieved it and what the implementation of this feature looks like on a technical level. All we can intercept in the kernel is the following C structure, which wraps the information copied by Binder driver from the sender into the address space of the receiving process.

```
struct binder_transaction_data {
 unsigned int code;                          // value 5 for our SMS example
 uid_t sender_euid;               // UID of the app initiating the request
 const void *buffer;           // Fig. 4 shows its content for our example
};
```

A comprehensive technical report contains more details on the implementation [10] and shows how we intercept binder_transaction_data during a certain stage of the Binder driver communication, which follows a strict protocol. The integer sender_euid provides us with the UID to unambiguously identify the sender app of a request. However, the method name the integer code translates to, and which

```
01: public void sendText(String destAddr, ...,
02:                      String text, ...) ...
03: {
04:  android.os.Parcel _data =
05:              android.os.Parcel.obtain();
06:  ...
07:  _data.writeInterfaceToken(DESCRIPTOR);
08:  _data.writeString(destAddr);
09:  ...
10:  _data.writeString(text);
11:  ...
12:  mRemote.transact(Stub.TRANSACTION_sendText,
13:                  _data, ...);
14:  ...
15: }
```

```
01: private ... String DESCRIPTOR =
02:   "com.android.internal.telephony.ISms";
03: ...
04: switch (code) { ...
05:  case TRANSACTION_sendText: {
06:    data.enforceInterface(DESCRIPTOR);
07:    ...
08:    _arg0 = data.readString();
09:    ...
10:    _arg2 = data.readString(); ...
11:    this.sendText(_arg0,..., _arg2,...);
12: }}
13: ...
14: static final int TRANSACTION_sendText =
15:   (IBinder.FIRST_CALL_TRANSACTION + 4);
```

Fig. 3: Auto-generated Proxy (left) and Stub class (right) for the ISms interface.

arguments are encoded in the byte stream of buffer, is not transmitted, mainly for efficiency reasons. We have a closer look at some code of the Stub and Proxy (shown in Fig. 3), which are *automatically* generated for the interface ISms, to better understand why there is no need for Binder to send this information.

When the Proxy makes the actual RPC for sendText() via Binder (left, line 12), it includes the integer TRANSACTION_sendText defined in its corresponding Stub (right, lines 14-15). We discovered that this is the value of code we find in binder_transaction_data. The second argument _data is an instance of the class Parcel and relates on a lower level to the buffer we intercept. More specifically, the Proxy takes a Parcel object (reused from a pool for efficiency) and then writes the DESCRIPTOR (left, line 7), which is the name of the interface ISms (right, lines 1-2), followed by the method arguments into it (left, lines 8-10). We can observe that this is done in the *order* of the arguments appearing in the method signature. Furthermore, dedicated write methods are used provided by Parcel, such as writeString(). When the Stub receives the call, it executes the TRANSACTION_sendText part of a switch construct (right, line 5), which reconstructs the arguments from the byte stream of the Parcel object; that is, using the equivalent read methods in the exact same order as the write methods have been used. Based on those key observations, we designed the following (three-step) algorithm to automate unmarshalling for arbitrary method calls with DroidTracer (top sub-component in Fig. 1):

1. **Unmarshall interface name (e.g., ISms)**
   (a) Take a Parcel object and fill it with the byte stream buffer. This is possible, as the class Parcel is public and provides an according method.
   (b) Read the DESCRIPTOR from the Parcel object via readString(), as it is always the first argument in buffer (see Fig. 4).
2. **Unmarshall method name (e.g., sendText)**
   (a) Use Java reflection to find the variable name with prefix TRANSACTION_ and code assigned to in the Stub of the unmarshalled interface name. This works, as every app, and so MonitorMe, has access to the JAR of the Android API.
3. **Unmarshall method arguments (e.g., "12345", null, "Hello!", null, null)**
   (a) Determine the order and types of method arguments by accessing the signature of the unmarshalled method name via reflection.
   (b) Apply Parcel's read methods according to the type and order of arguments appearing in the method signature. This works for Java primitives, but we also reverse engineered how complex objects are composed into Java primitives.

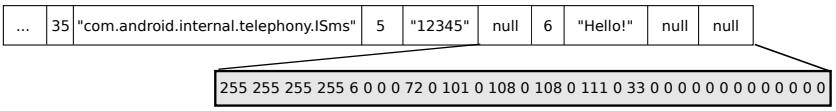

Fig. 4: The buffer sent via Binder containing the arguments of sendTextMessage().

It is worth pointing out that our unmarshalling algorithm does not rely on low-level Binder implementations, which might vary for different Android versions, as we are able to use the exact Java read methods of the class Parcel that are also used by the Android framework itself on a specific device.

Netlink endpoint. As event interception takes place solely inside the kernel space and unmarshalling relies on access to the Android API, we need a mechanism that allows us to pass data from inside the kernel module up to DroidTracer in user space. Moreover, we need to send data also in the other direction so that the user can control the kernel module for even the most basic tasks, for example, to switch event interception on and off. Android has no built-in way to serve as a solution, but we were able to use netlink[6] (a socket based mechanism of the Linux kernel) to bidirectionally communicate with user space. As only the kernel but not the Android API offers netlink support, we had to build a *custom* endpoint for our app (middle sub-component in Fig. 1), using the *Netlink Protocol Library Suite* (libnl[7]). Note that netlink implements a callback principle so that rather than polling the kernel module for new occurring system events, DroidTracer can push them all the way to our analysis component.

3 Specifying Malware Behaviour

Whenever an app causes a system event on the Android platform that we can intercept with DroidTracer, we capture it as an *action*. We represent actions in our internal, logical model by ground terms $p(c_1, \ldots, c_n)$, where p is a predicate symbol and c_i is a constant. Typically, p denotes the method and interface name of an intercepted method call, and c_i (if any) its ith unmarshalled method argument. That is, we write $sendText@ISms(\text{“12345”}, null, \text{“}Hello!\text{”}, null, null)$ for a ground term representing the sending of an SMS, given as an example in §2, where we conventionally delimit method and interface name by the @-symbol. Let us refer to finite sets of actions as *worlds*. An app's behaviour, observed over time, is therefore a finite trace of worlds. Note that in the case of our analysis, worlds contain only one action and are ordered by the position at which the corresponding system event has been sent from our kernel module via netlink. That is, there is no predefined delay between worlds as the trace is only extended by one world whenever a new system event occurs. Table 1 shows a selection of events collected for a sample of the malware family Walkinwat. Distinct malware samples (i.e., with different hash values) are usually grouped under the same family name if they share the same behaviour and manner in which they spread.

[6] http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto

[7] http://www.carisma.slowglass.com/~tgr/libnl/

Table 1: Trace of system events for malware Walkinwat collected by DroidTracer.

ID	Interface	Method	Arguments
334	IPhoneSubInfo	getDeviceId	
386	IContentProvider	QUERY	content://contacts/phones, null, null, null, display_name ASC
392	syscall	sys_connect	wringe.ispgateway.de
397	ISms	sendText	451-518-646, null, "Hey,just downlaoded a pirated App off the Internet, Walk and Text for Android. Im stupid and cheap,it costed only 1 buck.Don't steal like I did!", null, null
407	IActivityManager	getIntentSender	1, com.incorporateapps.walktext, null, null, 0, Intent { act=SMS_SENT }, null, 0
408	IActivityManager	getIntentSender	1, com.incorporateapps.walktext, null, null, 0, Intent { act=SMS_DELIVERED }, null, 0
414	ISurfaceComposer	N/A, code: 10	
578	IActivityManager	startActivity	null, Intent { act=android.intent.action.VIEW dat=market://details?id=com.incorporateapps.walktext }, null, N/A, N/A, N/A, N/A, N/A, N/A, N/A

Each row contains a system event, where the ID indicates its position in the trace and the remaining columns show the outcome of the three steps in our unmarshalling algorithm (see §2). For all other system calls, our algorithm returns "syscall" as the interface name, and beyond that the actual syscall function name and its intercepted arguments in the kernel. For example, look at the third row, which means an internet connection has been established via the system call sys_connect to URL *wringe.ispgateway.de*. An action at position $i \in \mathbb{N}$ in some trace means that at time i this action holds (or, from a practical point of view for the 397th world in Table 1 that Walkinwat has requested the Android framework to send an SMS to number "451-518-646" with text "Hey, just ...").

We specify undesired malware behaviour in terms of formulae (or policies) in a first-order linear temporal logic, called LTL^{FO}. From a formal point of view, we merely used safety formulae and our tool to detect finite counterexamples. For brevity, we recall here only the key concepts of LTL^{FO} by explaining an example policy, whereas the full syntax and semantics as well as our monitoring algorithm can be found in [2]. LTL^{FO} is an extension of propositional future LTL with quantifiers that are restricted to reason over those actions that appear in a trace, and not arbitrary elements from a (possibly infinite) domain (i.e., we can't express "for all numbers x of SMS messages that an app has not sent"). Let us consider the example that apps must not send SMS messages to numbers not in a user's contact book. Assuming there exists a predicate $sendText@ISms$, which is true (i.e., appears in the trace), whenever an app sends an SMS message to phone number $dest$, we could formalise said behaviour in terms of of a policy $\mathsf{G}\forall(dest, \_) : sendText@ISms.\ inContactBook(dest)$, shown as φ_{18} in Table 2. Note how in this formula the meaning of $dest$ is given implicitly by the first argument of $sendText@ISms$ and must match the definition of $inContactBook$ in each world. We use the "_"-symbol simply as placeholder for one or more remaining program variables that are not used in the formula. Also note that we call $sendText@ISms$ an uninterpreted predicate as it is interpreted indirectly via its occurrence in the trace, whereas $inContactBook$ never appears in the trace, even if true. $inContactBook$ can be thought of as interpreted via a program that queries a user's contact database, whose contents may change over time. An interpreted predicate can also be rigid; that is, its truth value never changes for the same arguments. For example, look at $regex(uri, ".*\ \text{calls}.*")$ in policy ψ_6 in Table 3, which is true if uri (an identifier for Android's content providers),

Table 2: Key characteristics of Android malware behaviour specified in LTL^{FO}.

Information stealing					
$\varphi_{i\in[1,14]}$	$\mathsf{G}\neg\psi_i$			$[\varphi_{i\in[1,14]}]$	$\mathsf{G}(\psi_i \rightarrow \neg\mathsf{F}\psi')$
$[[\varphi_{i\in[1,14]}]]$	$\mathsf{G}(\psi_i \wedge \neg\psi' \rightarrow (\neg\psi'\mathsf{W}(\text{N/A@ISurfaceComposer} \wedge \neg\psi')))$			$[[\varphi_{i\in[1,4]}]]'$	$\mathsf{G}(\psi_i \rightarrow \neg\mathsf{F}\psi'')$
Privilege escalation					
φ_{15}	$\mathsf{G}\neg\exists(args)$: do_execv@syscall. $regex(args,$ ".*su|pm (un)?install|am start.*")				
Launching malicious payloads					
$\varphi_{i\in[16,17]}$	$\mathsf{G}\neg\psi_i$	$[\varphi_{17}]$	$\mathsf{G}(\psi_{17} \rightarrow \neg\mathsf{F}\psi')$	$[[\varphi_{17}]]$	$\mathsf{G}(\psi_{17} \rightarrow \neg\mathsf{F}\psi'')$
Financial charges					
φ_{18}	$\mathsf{G}\forall(dest, \_)$: sendText@ISms. $inContactBook(dest)$				
φ_{19}	$\mathsf{G}(\psi_{17} \rightarrow \neg\mathsf{F}\exists(w, x, y, z, abort)$: finishReceiver@IActivityManager. $regex(abort,$ "true"))				

Table 3: Auxiliary formulae for Table 2.

ψ_1	getDeviceId@IPhoneSubInfo	ψ_6	$\exists(uri, \_)$: QUERY@IContentProvider. $regex(uri,$ ".*calls.*")
ψ_2	getSubscriberId@IPhoneSubInfo	ψ_7	$\exists(uri, \_)$: QUERY@IContentProvider. $regex(uri,$ ".*contacts.*")
ψ_3	getIccSerialNumber@IPhoneSubInfo	ψ_8	$\exists(uri, \_)$: QUERY@IContentProvider. $regex(uri,$ ".*phones.*")
ψ_4	getLine1Number@IPhoneSubInfo	ψ_9	$\exists(uri, \_)$: QUERY@IContentProvider. $regex(uri,$ ".*bookmarks.*")
ψ_5	getDeviceSvn@IPhoneSubInfo	ψ_{10}	$\exists(uri, \_)$: QUERY@IContentProvider. $regex(uri,$ ".*preferapn.*")
c_{16}	".*BOOT_COMPLETED.*"	ψ_{11}	$\exists(uri, \_)$: QUERY@IContentProvider. $regex(uri,$ ".*sms.*")
c_{17}	".*SMS_RECEIVED.*"	ψ_{13}	$\exists(args)$: do_execv@syscall. $regex(args,$ ".*logcat.*")
ψ_{12}	$(\exists(\_)$: getInstalledPackages@IPackageManager. $true)\vee$ $(\exists(\_)$: getInstalledApplications@IPackageManager. $true)$		
ψ_{14}	$(\exists(\_)$: notifyCellLocation@ITelephonyRegistry. $true)\vee$ $(\exists x$: getLastLocation@ILocationManager. $regex(x,$ ".*gps.*")		
$\psi_{i\in[16,17]}$	$\exists(intent, txt, \_)$: system#scheduleReceiver@IApplicationThread. $(regex(intent, c_i) \wedge regex(txt,$ ".*<pkg>.*"))		
ψ'	$(\exists(\_)$: sys_connect@syscall. $true) \vee (\exists(\_)$: sendText@ISms. $true)\vee$ $(\exists(x, intent, \_)$: startActivity@IActivityManager. $regex(intent,$ "action.SEND")		
ψ''	$(\exists(dest, x, msg, \_)$: sendText@ISms. $regex(msg,$ ".*<sensitiveInfo>.*"))$\vee$ $(\exists(x, intent, \_)$: startActivity@IActivityManager. $regex(intent,$ ".*<sensitiveInfo>.*")		

matches the regular expression ".* calls.*". This way, we check whether the database that stores the call history on a phone is accessed.

As we can't anticipate when or if an app stops, a policy φ normally specifies behaviour in terms of an *infinite* trace. But the monitor we build to check φ will only see a prefix (observed system events so far), denoted u, and therefore return $\bot$ if u is a *bad* prefix, and ? otherwise. This means that a monitor for φ_{18}, when it processes the 397th event, will return $\bot$ and terminate, as the number "451-518-646" is not in the contact book. Furthermore, there exists a third case, that the monitor returns $\top$ if u is a good prefix, but as we only monitor safety formulae, this case is not relevant for our study. Our monitoring semantics is akin to the 3-valued finite-trace semantics for LTL introduced in [3].

Based on the patterns from four different categories identified by the AMGP [15], we have formally specified various key characteristics of malware behaviour in LTL^{FO}. The results are the policies listed in Table 2. For readability, we write $\varphi_{i\in[a,b]}$, grouping policies of the same pattern together, where ψ_i, ψ' or ψ'' are auxiliary formulae listed in Table 3. We surround a policy φ_i with n square brackets (calling it the nth refinement of φ_i), if its bad prefixes are a strict subset of the bad prefixes of φ_i surrounded with $n-1$ square brackets. For example a bad prefix of $[\varphi_1]$, the first refinement of φ_1, has to contain after accessing the device id as well an event of establishing a connection to the internet, describing from a practical point of view a more severe malware behaviour for the user.

Information stealing. The AMGP discovered that malware is often actively harvesting various sensitive information on infected devices. Thus, our policies $\varphi_{i\in[1,11]}$ specify that an app should neither request any permission secured sensitive data, such as the device or subscriber id, SIM serial or telephone number, or device software version, nor query any of the content providers that contain the call history, contact list, phone numbers, browser bookmarks, carrier settings or SMS messages. Policy φ_{12} covers the harvesting of installed app or package names on a device, and φ_{13} the reading of system logs via the Android logging system, called logcat. Note that before Android 4.1, an app could read other apps' logcat logs, which might contain sensitive messages. Policy φ_{14} specifies that neither the coarse grain location based on cell towers nor the more precise GPS location should be accessed. The policies $[\varphi_{i\in[1,14]}]$ refine the policies above towards the more suspicious behaviour that an app should not, after requesting the sensitive information, eventually connect to the internet, send an SMS or exchange data with another app. Even though a detected bad prefix for those polices does not guarantee that information has been leaked, the usage of above sinks bears at least its potential. However, the data could have been encrypted or in other ways obscured, which makes it hard to prove leakage in general based on the trace we collect. Furthermore, $[[\varphi_{i\in[1,14]}]]$ expresses the absence of any screen rendering (via *N/A@ISurfaceComposer*) in between information request and potential leakage. This excludes the case that the sending of data was caused by some user interaction with the app, but rather by some app's malicious background service. Note that we represent with "N/A" methods which we could not unmarshall. Also note that we used the well-known specification patterns [5] to specify our policies, where the first and second refinements are based on the "absence after", and "exists between" patterns, respectively. $[[\varphi_{i\in[1,4]}]]'$ are further refinements as they only trigger if we find the device id, etc. cleartext (represented by the placeholder "<sensitiveInfo>") in the trace.

Privilege escalation. The attack of exploiting bugs or design flaws in software to gain elevated access to resources that are normally protected from an application, is called privilege escalation. From the samples in [15], 36.7% exploit a version-specific known software vulnerability of either the Linux kernel or some open-source libraries running on the Android platform, such as WebKit or SQLite, to gain root access (e.g., to replace real banking apps with a fake one, for phishing attacks). Therefore, policy φ_{15} lets us detect when an app opens a root shell, secretly starts, installs or removes other packages via the *activity manager* (am) or *the package manager* (pm). Monitoring of this behaviour is possible, because the do_execv() system call is exclusively used for the execution of any binary, shell command or script on the underlying Linux OS.

Launching malicious payloads. Apps' background services, which don't have any UI, can't only be actively started when clicking on an app's launch icon, but also by registering for Android system-wide events (called broadcasts). The AMGP discovered that 29 of the 49 malware families contain a malicious service that is launched after the system was booted, or for 21 families when an SMS was received (i.e., they registered for the BOOT_COMPLETED or SMS_RECEIVED

broadcast, respectively). Therefore, we consider it as suspicious if services are activated by the broadcasts mentioned above; which we specify in form of $\varphi_{i\in[16,17]}$, where we replace "<pkg>" for each app specifically with its package name. Note that, to monitor this behaviour, we need to intercept system events of the Android system (UID 1000) as it starts the services that have registered for a certain broadcast (via *scheduleReceiver@IApplicationThread*). We prefix those predicates with "system#"; that is, to distinguish them from an app's events in a trace. Since malware, after registering for SMS_RECEIVED, gets access to the sender and content of an incoming SMS, we check with the refinements $[\varphi_{17}]$ and $[[\varphi_{17}]]$ for information stealing. This means, similar as specified by the refinements of $\varphi_{i\in[1,14]}$, the internet should not be accessed, and so on, after the broadcast was received.

Financial charges. The AMGP discovered apps, such as the first Android malware FakePlayer, that secretly call or register for premium services via an SMS. As this behaviour can result in high financial charges for the user, Google labels the permissions that allow to call or send an SMS with "services that cost you money". Instead of defining policies that check outgoing messages against a fixed list of potential premium numbers, φ_{18} more generically specifies that an SMS should not be sent to a number not in the user's contact book. Since Android 4.2, Google added a similar security check, where a notification is provided to the user if an app attempts to send messages to short codes as those could be premium numbers. Note that we could have specified that apps should not make phone calls to numbers not in the phone book as well, but as we have not observed this behaviour during our experiments, we neglect the policy for it.

Before Android 4.4, apps could block incoming SMS messages, which was used by malware to suppress received confirmations from premium services or mobile banking *transaction authentication numbers* (TANs). The latter were then forwarded to a malicious user. Thus, policy φ_{19} checks if apps *abort* a broadcast after receiving SMS_RECEIVED, in which case the SMS would not be delivered further to appear in the usual messaging app on a device.

4 Identifying Malware Behaviour

We installed MonitorMe, provided with the polices introduced in §3, on our test device Nexus S running Android 2.3.6. We then monitored separately one malware sample for each of the AMGP-families. That is, we first installed its *application package* (APK), and before starting it (i.e., clicking on its launch icon if it had any), test using and finally uninstalling it, we tried to activate potential background services by sending the broadcast BOOT_COMPLETE via the *Android Debug Bridge* (adb) and an SMS to our phone. Even though MonitorMe performed *online* monitoring, which means that monitors processed events incrementally when they occurred, we also persisted the trace for each malware in an SQLite database on the phone;[8] both, for repeatability of our own experiments, and to provide them to other researchers.

[8] Traces are available at http://kuester.multics.org/DroidTracer/malware/traces

4.1 Experiments' Results

Table 4 summarises for which malware families (49 in total) and policies MonitorMe detected bad prefixes during our experiments. The second column indicates, whether a malware or one of its services crashed during our experiments; e.g., due to incompatibility with the Android version on our test device. Thus, we might have missed observing some critical behaviour. The third column tells us if a malware had no launch icon, which is usually intended to stay hidden and spy on the user. SMSReplicator, for example, is used by parents to secretly forward all SMS messages received on their childrens' phones. As we monitored in general all UIDs above 10000,[9] apps without an icon could not bypass our analysis unnoticed. The fourth column shows the number of system events we have recorded for each malware. Between the double lines are the individual monitor results, where the single lines separate results from the four categories introduced in Table 2. The cell containing $\overset{[[\varphi_{18}]]}{397}$ in the row for Walkinwat denotes that the monitor for $[[\varphi_{18}]]$ found a bad prefix for the Walkinwat sample after 397 worlds. As this implies that the same prefix is also a bad one for lower refinements of φ_{18}, we neglect showing this information. The last column shows the number of bad prefixes found in total per malware. In summary, for 46 (93.9%) out of 49 families, we detected suspicious behaviour. This is under the assumption that we consider bad prefixes for φ_{16} alone as *not* critical. Note that our results take into account that, according to [15], we would have observed additional malicious behaviours guarded by φ_{15}, φ_{18} and φ_{19} (indicated by an $\lightning$ in Table 4). The reason for φ_{15} is that the nine marked families targeted Android versions below 2.3.6. Thus, their exploits were not attempted in the first place or unsuccessful. We missed bad prefixes for φ_{18} as malware often waited to receive instructions from a remote server, which wasn't active anymore. The servers are needed for malware to send an SMS, as they provide premium numbers dynamically. Regarding φ_{19}, we could rarely observe the blocking of incoming SMS messages as most malware was designed to only suppress the received confirmation from specific premium numbers. Out of 46 detected families, 34 can be associated with potential information stealing, as they use the internet or other sinks after accessing sensitive information. For NickySpy and SMSReplicator we discovered that the device id was leaked *cleartext* via an SMS, and an SMS received was forwarded to a malicious user, respectively. To discuss limits of our malware detection, which are by no means unique to our approach, consider the FakeNetflix family. It uses a phishing attack for which is no observable behaviour in the trace; that is, it shows a fake login screen to the user and then sends the entered credentials to a malicious server.

False positives. Finally, we checked if our policies are suitable to distinguish malware from benign apps. Therefore, we ran MonitorMe on a Nexus 5 with Android 5.0.1 that had more than 60 apps from common app categories installed: social (Facebook, Twitter, LinkedIn, etc.), communication (Whatsapp, etc.), transportation (Uber, etc.), travel & local (Yelp, TripAdvisor, etc.), and games (Cut the rope, etc.) to name a few. We discovered suspicious behaviour

[9] UIDs below 10000 are reserved for system apps with higher privileges.

Table 4: Monitor results for malware of the Android Malware Genome Project.

Malware family	Crash	Hide	Worlds	Monitor results for $\varphi_1 - \varphi_{19}$	Total $\perp$
ADRD			983	$[\varphi_1]$ 39 $[\varphi_2]$ 39 $[\varphi_3]$ 458 $[\varphi_4]$ 458 $[\varphi_{11}]$ 153 $[\varphi_{12}]$ 115 φ_{14} 2057 ↯ φ_{16} 194 φ_{18} 247	3
AnserverBot			2201	$[\varphi_1]$ 458 $[\varphi_2]$ 458 $[\varphi_3]$ 458 $[\varphi_4]$ 458 $[\varphi_{12}]$ 115 φ_{14} 2057 ↯ φ_{16} 194 φ_{18} 247	7
Asroot			192	$[\varphi_2]$ 126 $[\varphi_{11}]$ 126 ↯ φ_{16} 194 ↯ φ_{18} 238	3
BaseBridge		✓	754	$[\varphi_1]$ 824 $[\varphi_2]$ 824 $[\varphi_4]$ 824 $[\varphi_{10}]$ 303 $[\varphi_{14}]$ 853 ↯ φ_{16} 194 ↯ φ_{18} 238	5
BeanBot			2030	$[\varphi_1]$ 824 $[\varphi_2]$ 81 $[\varphi_4]$ 824 $[\varphi_{10}]$ 303 $[\varphi_{14}]$ 853 ↯ φ_{16} 194 ↯ φ_{18} 238	5
Bgserv		✓	2248	$[\varphi_1]$ 824 $[\varphi_2]$ 81 $[\varphi_4]$ 824 $[\varphi_{10}]$ 303 $[\varphi_{14}]$ 853 ↯ φ_{16} 194 ↯ φ_{18} 238	5
CoinPirate			895	$[\varphi_1]$ 41 $[\varphi_2]$ 41 $[\varphi_4]$ 41 ↯ φ_{16} 194 ↯ φ_{18} 238	2
CruseWin			120	$[\varphi_1]$ 41 $[\varphi_2]$ 41 $[\varphi_4]$ 41 ↯ φ_{16} 194 ↯ φ_{18} 238	1
DogWars			550	↯ φ_{16} 194 ↯ φ_{18} 238	1
DroidCoupon			841	$[\varphi_1]$ 810 $[\varphi_2]$ 810 $[\varphi_3]$ 810 $[\varphi_4]$ 810 $[\varphi_{12}]$ 810 $[\varphi_{13}]$ 610 ↯ φ_{16} 87	7
DroidDeluxe			1167	↯ φ_{16} 87	2
DroidDream			480	$[\varphi_1]$ 360 $[\varphi_2]$ 360 ↯ φ_{16} 87	2
DroidDreamLight			536	↯ φ_{15} 231	1
DroidKungFu1			892	↯ φ_{15} 231	1
DroidKungFu2			2012	$[\varphi_1]$ 132 $[\varphi_2]$ 132 $[\varphi_3]$ 132 $[\varphi_4]$ 132 ↯ φ_{15} 231	3
DroidKungFu3			370	$[\varphi_1]$ 383 $[\varphi_2]$ 383 $[\varphi_3]$ 383 $[\varphi_4]$ 383 ↯ φ_{16} 124	5
DroidKungFu4			974	$[\varphi_1]$ 383 $[\varphi_2]$ 383 $[\varphi_3]$ 383 $[\varphi_4]$ 383	1
DroidKungFuSapp		✓	1392	$[\varphi_1]$ 267 $[\varphi_2]$ 267 $[\varphi_3]$ 267 $[\varphi_4]$ 267 $[\varphi_5]$ 267 ↯ φ_{16} 129 φ_{17} 267 ↯ φ_{18} 247	6
DroidKungFuUpdate			506	$[\varphi_1]$ 256 $[\varphi_2]$ 256 $[\varphi_3]$ 256 $[\varphi_4]$ 418 $[\varphi_{12}]$ 304 $[\varphi_{14}]$ 598 ↯ φ_{16} 136 φ_{17} 267 ↯ φ_{18} 247	5
Endofday			594	$[\varphi_4]$ 418 $[\varphi_{12}]$ 304 $[\varphi_{14}]$ 598 ↯ φ_{16} 136 φ_{17} 267 ↯ φ_{18} 247	4
FakeNetflix			731	$[\varphi_4]$ 321 ↯ φ_{16} 136 φ_{17} 267 ↯ φ_{18} 247	1
FakePlayer		✓	16	↯ φ_{16} 136 φ_{17} 267 ↯ φ_{18} 247	1
GamblerSMS		✓	564	$[\varphi_1]$ 267 $[\varphi_2]$ 267 $[\varphi_3]$ 267 $[\varphi_4]$ 267 $[\varphi_5]$ 267 φ_{11} 176 φ_{16} 129 φ_{17} 267 ↯ φ_{18} 247	6
Geinimi			659	$[\varphi_1]$ 267 $[\varphi_2]$ 267 $[\varphi_3]$ 267 $[\varphi_4]$ 267 $[\varphi_5]$ 267 $[\varphi_{11}]$ 193 $[\varphi_{14}]$ 598 ↯ φ_{16} 136 φ_{17} 267 ↯ φ_{18} 247	6
GGTracker			508	$[\varphi_1]$ 173 $[\varphi_2]$ 173 $[\varphi_3]$ 173 $[\varphi_4]$ 173 $[\varphi_{12}]$ 1061 ↯ φ_{16} 124 ↯ φ_{18} 236	5
GingerMaster			627	$[\varphi_1]$ 173 $[\varphi_2]$ 173 $[\varphi_3]$ 173 $[\varphi_4]$ 173 $[\varphi_{12}]$ 1061 ↯ φ_{16} 124 ↯ φ_{18} 236	6
GoldDream			1517	$[\varphi_{12}]$ 1061 ↯ φ_{16} 124 ↯ φ_{18} 236	1
Gone60			555	$[\varphi_6]$ 618 $[\varphi_7]$ 350 $[\varphi_8]$ 350 $[\varphi_9]$ 350 $[\varphi_{11}]$ 350 ↯ φ_{16} 135 ↯ φ_{18} 218	5
GPSSMSSpy		✓	23	↯ φ_{16} 135 ↯ φ_{18} 218	1
HippoSMS			888	$[\varphi_3]$ 352 $[\varphi_{11}]$ 329 φ_{17} 253 ↯ φ_{18} 218	4
Jifake		✓	584	$[\varphi_1]$ 558 $[\varphi_2]$ 558 $[\varphi_{10}]$ 679 φ_{16} 129 ↯ φ_{18} 414	4
jSMSHider			1283	φ_{15} 250 ↯ φ_{16} 135 ↯ φ_{18} 218	4
KMin			850	$[\varphi_2]$ 378 $[\varphi_3]$ 175 $[\varphi_4]$ 175 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	3
LoveTrap		✓	1791	$[\varphi_1]$ 175 $[\varphi_2]$ 175 $[\varphi_3]$ 175 $[\varphi_4]$ 175 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	4
NickyBot			619	$[\varphi_1]$ 175 $[\varphi_2]$ 175 $[\varphi_3]$ 175 $[\varphi_4]$ 175 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	3
NickySpy		✓	441	$[\varphi_1]$ 175 $[\varphi_2]$ 175 $[\varphi_3]$ 175 $[\varphi_4]$ 175 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	3
Pjapps			1808	$[\varphi_1]$ 58 $[\varphi_2]$ 58 $[\varphi_3]$ 58 $[\varphi_4]$ 58 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	1
Plankton			561	$[\varphi_1]$ 58 $[\varphi_2]$ 58 $[\varphi_3]$ 58 $[\varphi_4]$ 58 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	2
RogueLemon			2077	$[\varphi_1]$ 652 $[\varphi_2]$ 652 $[\varphi_3]$ 652 $[\varphi_4]$ 652 $[\varphi_{14}]$ 1030 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	4
RogueSPPush			1653	$[\varphi_1]$ 175 $[\varphi_2]$ 175 $[\varphi_3]$ 175 $[\varphi_4]$ 175 $[\varphi_7]$ 781 $[\varphi_8]$ 781 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	3
SMSReplicator			316	$[\varphi_7]$ 236 $[\varphi_8]$ 236 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	2
SndApps		✓	559	φ_1 168 φ_4 170 $[\varphi_4]$ 242 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	3
Spitmo		✓	771	↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	1
Tapsnake		✓	355	$[\varphi_7]$ 492 $[\varphi_8]$ 492 ↯ φ_{16} 202 φ_{17} 386 ↯ φ_{18} 218	4
Walkinwat			848	$[\varphi_1]$ 492 $[\varphi_2]$ 492 $[\varphi_3]$ 492 $[\varphi_4]$ 492 $[\varphi_8]$ 492 φ_{11} 507 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	5
YZHC			631	$[\varphi_1]$ 387 $[\varphi_2]$ 387 $[\varphi_3]$ 387 $[\varphi_4]$ 387 $[\varphi_8]$ 1293 $[\varphi_{10}]$ 1293 $[\varphi_{11}]$ 618 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	7
zHash			865	$[\varphi_1]$ 618 $[\varphi_2]$ 618 $[\varphi_3]$ 618 $[\varphi_4]$ 618 $[\varphi_8]$ 1293 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	7
Zitmo			2027	$[\varphi_1]$ 1045 $[\varphi_2]$ 1045 $[\varphi_3]$ 1045 $[\varphi_4]$ 1045 $[\varphi_8]$ 1293 $[\varphi_{10}]$ 1293 $[\varphi_{11}]$ 618 ↯ φ_{16} 135 φ_{17} 386 ↯ φ_{18} 218	7
Zsone		✓	553	$[\varphi_1]$ 344 $[\varphi_2]$ 344 $[\varphi_3]$ 344 $[\varphi_4]$ 344 $[\varphi_8]$ 1293 $[\varphi_{10}]$ 1293 $[\varphi_{11}]$ 618 ↯ φ_{16} 135 φ_{17} 344 ↯ φ_{18} 384	2
Total $\perp$				24 | 13 | 10 | 15 | 2 | 2 | 4 | 3 | 2 | 3 | 8 | 5 | 1 | 4 | 2 | 13 | 9 | 8 | 1	

for 17 (28%) out of 61 apps, using the same assumption as above that φ_{16} alone is not critical. The false positive rate seems high at first glance, but a closer look reveals that some benign apps bear unwanted behaviour for the user so that a warning of MonitorMe seems reasonable. For example, eleven apps surprisingly requested the device or subscriber id, which is explicitly not recommended by the Google developer guidelines.[10] Under those apps were a soccer news and Yoga app, which in our opinion both do not require this data for its functionality, but rather collect sensitive data from the user. Another app was the private taxi app Uber, which has been criticised in the past due to collecting personal data without the user's permission.[11] Only five apps started after boot, such as dropbox, which we consider as necessary regarding its purpose, and only two apps after an SMS was received, which were a secure SMS app and Twitter.

[10] http://developer.android.com/training/articles/security-tips.html#UserData

[11] http://thehackernews.com/2014/11/ubers-android-app-is-literally-malware_28.html

Table 5: Execution of Android method calls (each up to 10,000 times) with and without DroidTracer. The margin of error is given for the 95% confidence interval.

Interface	Method	Android (in ms)	Kprobes (in ms)	DroidTracer (in ms)	Kprobes Overhead	DroidTracer Overhead
IPhoneSubInfo	getDeviceId	5309 ± 15	5517 ± 18	5811 ± 11	3.92%	9.46%
IPhoneSubInfo	getIccSerialNumber	5346 ± 16	5524 ± 16	5817 ± 7		8.81%
LocationManager	getLastKnownLocation	3516 ± 13	3562 ± 13	4126 ± 5		17.35%
ISms	sendText	9166 ± 13	9396 ± 13	10216 ± 10	2.51%	11.46%
IPackageManager	getInstalledApplications	15730 ± 204	15514 ± 202	15422 ± 172		
IConnectivityManager	getAllNetworkInfo	5769 ± 53	5841 ± 60	5671 ± 7		
syscall	sys_open	15360 ± 72	15531 ± 67	15455 ± 38		

4.2 Performance and Portability

We evaluated (1) the performance of DroidTracer and MonitorMe, i.e., the bare system event interception including unmarshalling as well as when running our monitors on top. Moreover, we (2) demonstrate that our automated approach to unmarshalling is portable to different Android devices and versions.

Performance. We wrote seven test apps, where each was designed to generate 100 runs of up to 10,000 system events named by the interface and method names in Table 5. When MonitorMe is being executed with the policies in §3 and monitors our test apps individually, the *highest* performance overhead is 38.6% for the system event $sendText@Isms$. This was determined on a Nexus 7 (quad-core CPU, 1 GB RAM) with Android 4.3. Note that the overhead involves the monitor for φ_{18} checking the contact book each time an SMS was sent.

Furthermore, as the results of these test runs are specific only to our implementation of runtime verification, we also need to measure the performance overhead of DroidTracer when no further analysis is undertaken. Table 5, shows the execution time when intercepting the method calls of the above seven system events in three different modes of operation: (1) without DroidTracer enabled to get a reference execution time for the unmodified system, (2) with only the event interception part of our kernel module enabled, and (3) with unmarshalling and netlink communication added. During the experiments, we ran all four cores of the Nexus 7 on a fixed frequency rate, which allowed us to reduce the margin of error dramatically. Note that we left cells empty, where overhead could not significantly be determined wrt. the t-test. As the results show, the actual performance overhead of using just our kernel module with kprobes is only 2.51–3.92%, whereas the complete performance overhead of DroidTracer is 8.81–17.35%. What is noteworthy is that getDeviceId() and getIccSerialNumber() have significant lower overhead than getLastKnownLocation() and sendText(), as both former method signatures have no arguments that require unmarshalling. The call getLastKnownLocation() has the highest overhead, probably because its arguments contain several complex objects, for example, one of type LocationRequest, which unmarshalling involves additional reflection calls. As sendText() contains only Java primitives as arguments, its unmarshalling overhead is slightly lower.

Portability. We ran DroidTracer on three different devices and Android versions, including 5.0.1, which is, at the time of writing, the most recent one. Table 6 demonstrates the success of unmarshalling events we have intercepted. While we could unmarshall the interface name of all method calls, we could unmarshall

Table 6: Unmarshable parts of observed system events.

Device	Android version	Events	Interfaces Unique	Methods Unique	Unmarsh.	Succ. rate	Events with arguments Total	Unmarsh. (Totally / Partially)	Succ. rate
Nexus S	2.3.6	102,545	58	804	368	45.77%	54,596	43,318 / 47,923	79.34% / 87.78%
Nexus 7	4.3	107,977	89	378	236	62.43%	70,746	67,866 / 69,474	95.93% / 98.20%
Nexus 5	5.0.1	449,429	108	474	326	68.78%	264,058	227,708 / 255,263	86.23% / 96.67%

45.77%-68.78% of unique method names; that is, we were able discover for an integer code its according method name in the Android API via reflection. The number of unmarshalled method names seem low, but missing ones are mainly specific to Android internals, for example to render the screen. As such, they have no Stub or Proxy in the Android API, but only in some native C library. This is not accessible to the developer and therefore usually contain no relevant events for our analysis. If method calls had arguments, we could unmarshall for 79.34%-95.93% all and for 87.78%-98.20% at least some arguments. Note that if we failed to unmarshall one argument of a call, we also failed for all the remaining of that call, as Parcel's read methods have to be applied in the correct order.

5 Conclusions and Related Work

To the best of our knowledge, our work is the first runtime verification approach to comprehensively monitor the collected malwares by the AMGP. Arguably, detection rates are promising and help substantiate the claim that methods developed in the area of runtime verification are, in fact, suitable not only for safety-critical systems, but also when security is critical. Indeed, at the time of writing, the samples of the AMGP are ca. three years old, which in the rapid development of new attacks seems like a long time. However, the database has grown over a number of years and the underlying patterns emerged as a result of that. While there are always innovative, hard to detect malwares, it is not unreasonable to expect the bulk of new malwares to also fall into the existing categories, identified by the AMGP, and therefore detectable by our approach. Validation of this hypothesis, however, must be subject to further work.

Besides MonitorMe, one corner stone of our approach is the ability to specify policies over traces that contain parameters. Other runtime verification works that haven't been applied to Android, but also allow monitoring parametric traces are, for example, Hallé and Villemaire's [7], who use a logic with quantification identical to ours, but without arbitrary computable predicates. Furthermore, JavaMOP [9] is by now a quasi-standard when dealing with parametric-traces, although it is not based on first-order logic, but on "trace-slicing".

Most monitoring approaches for Android can be divided into two categories. *App-centric* ones (cf. [1, 12, 14]) intercept events inside the apps by rewriting and repackaging them, so that neither root access nor modifying the Android platform is necessary. As a consequence, they are portable to most phones and Android versions, and are easy to install even for non-experts. Examples are AppGuard [1] and Aurasium [14], which is even able to enforce security polices for apps' native code as it rewrites Android's own libc.so that is natively linked into every app. However, the ease in portability comes at the expense of inherent vulnerabilities, namely that security controls run inside the apps under scrutiny

and thus could be bypassed; e.g., by dynamically loading code after rewriting. Also, as apps have to be actively selected for rewriting, hidden malware, such as the ones without launch icon that we came across in §4.1, might be overlooked.

Platform-centric approaches (cf. [6, 4, 8]) usually tie deep into the source code of the Android platform and are therefore less portable. TaintDroid [6], a pioneering platform-centric tool for taint flow analysis, requires modifications from the Linux kernel all the way up to the Dalvik VM. Although it is being actively ported, users have to be sufficiently experienced to not only compile their own version of Android, including the TaintDroid changes, but also to make it work on a hardware of their choice. Our approach is, conceptually, a combination of the advantages of app- and platform-centric monitoring; that is, MonitorMe can be loaded even into a currently running Android system, yet is able to trace app (even preinstalled Google apps that can't be rewritten) and Android system interactions all the way down to the OS kernel level.

References

1. M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. AppGuard - enforcing user requirements on Android apps. In *TACAS*, volume 7795 of *LNCS*, pages 543–548. Springer, 2013.
2. A. Bauer, J.-C. Küster, and G. Vegliach. The ins and outs of first-order runtime verification. *To appear in: Formal Methods in System Design (FMSD)*, 2015.
3. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14, 2011.
4. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012.
5. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. IEEE, 1999.
6. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*. USENIX, 2010.
7. S. Halle and R. Villemaire. Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72. IEEE, 2008.
8. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *CCS*, pages 639–652. ACM, 2011.
9. D. Jin, P. O. Meredith, C. Lee, and G. Rosu. JavaMOP: Efficient parametric runtime monitoring framework. In *ICSE*, pages 1427–1430. IEEE, 2012.
10. J.-C. Küster and A. Bauer. Platform-centric Android monitoring—modular and efficient. Comp. Research Repository (CoRR) arXiv:1406.2041, ACM, June 2014.
11. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *ACSAC*, pages 421–430. IEEE, 2007.
12. S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. DroidForce: Enforcing complex, data-centric, system-wide policies in Android. In *ARES*, pages 40–49. IEEE, 2014.
13. T. Vidas and N. Christin. Evading Android runtime analysis via sandbox detection. In *ASIACCS*, pages 447–458. ACM, 2014.
14. R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for Android applications. In *USENIX Security Symp.*, pages 27–27. USENIX, 2012.
15. Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *S&P*, pages 95–109. IEEE, 2012.