Runtime Verification meets Android Security

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

[1]NICTA Software Systems Research Group · [2]The Australian National University

Abstract. A dynamic security mechanism for Android-powered devices based on runtime verification is introduced, which lets users monitor the behaviour of installed applications. The general idea and a prototypical implementation are outlined, an application to real-world security threats shown, and the underlying logical foundations, relating to the employed specification formalism, sketched.

1 Introduction

Most mobile platforms, such as Android [8], which is an open-source software stack designed to power tablet PCs and smart phones, offer built-in security mechanisms to protect users from various types of malware, often designed to spy on users or to exert control over (parts of) a mobile device's functionality. An example for the latter consists in the sending of SMS messages secretly, without the user's consent (cf. [11]). However, the existing security mechanisms obviously cannot stop or prevent the rising number of attacks on these platforms: In its Q1/2011 threats report, security firm Kaspersky remarks that "since 2007, the number of new antivirus database records for mobile malware has virtually doubled every year." In case of the Android platform, security firm McAfee asserts in its Q2 threats report that, in fact, "Android OS-based malware became the most popular target for mobile malware developers." In light of these developments various authors have proposed improvements to the built-in security mechanisms of mobile platforms, and in particular to the Android platform which, right now, constitutes the fastest growing platform on the market, and offers researchers the advantage that its source code is freely available.

Arguably, two of the most feature-complete and well-documented security enhancements recently made for Android are TaintDroid [4] and the Saint framework [12]. TaintDroid is an extensive modification of the entire Android stack that tracks the flow of sensitive data through third-party applications at runtime. The modifications allow TaintDroid to detect when sensitive data is leaked in whatever form, e.g., by sending an email or SMS containing the sensitive data, or by uploading a file directly. To cater for all these different scenarios, TaintDroid "taints" sensitive information to keep tracking its use throughout the system. The central components of the Saint framework described in [12] are a modified Android application installer and a so called AppPolicy Provider. The custom installer ensures that at install-time only applications which do not violate policies stored in the AppPolicy Provider can be installed. The authors of Saint have gone to great lengths to check existing applications' permissions for suspicious permission requests and from that derived practically useful policies for that purpose.

While Saint is, more or less, true to Android's own security mechanisms, which are mostly based on assigning permissions statically to third-party applications, thereby deciding which operations an application may or not perform at runtime, TaintDroid goes further, in that it controls what happens with data at runtime. The latter, however, comes at a price, in that such a high level of system instrumentation results in up to 27% runtime overhead [4]. Moreover, an expected downside of such a comprehensive system modification is to try and keep up to date with future releases of Android, which is under active development by a world-wide consortium of OEMs. As long as said consortium does not adapt (and thereby maintain) TaintDroid officially, it will be difficult to install and adapt it for off-the-shelf devices.

Our aim therefore, is to introduce a more light-weight, yet dynamic security extension to Android based on a technique known as *runtime verification*. In a nutshell, runtime verification subsumes techniques that aid in showing that an observed system behaviour satisfies or violates a given specification, often given in terms of automata or logic (cf. [1]). The methods developed in this area typically help to automatically generate a *monitor* from a given specification, such that at runtime one must not consider/store the entire behavioural trace, but merely consume observations in a step-wise and therefore efficient manner. That is, the monitor passively observes the system and raises an alarm if a specification is violated, or switches itself off if a specification has been satisfied. While the complexity of monitor generation, depending on the specification formalism at hand, can be very high (sometimes multiple exponents), the runtime complexity is usually constant-time for each new observation. Runtime verification has been employed in safety-critical contexts, but recently also emerged as a generic monitoring and testing methodology for Java (cf. [7, 3]). Here, we use it specifically to enhance system security, in that we monitor the individual behaviours of third-party applications, installed on Android devices. To this end, users can specify what constitutes a "suspicious behaviour", e.g., an application starts at boot-time, later checks the device's GPS location, and then connects to the internet (possibly to transmit the location). For each such policy, our implementation automatically creates a monitor that, once active, will raise an alarm when such a sequence of events was produced by any installed application. Admittedly, not every application which queries the GPS and then connects to the internet is malicious, but many so called "spywares" are disguised as seemingly harmless toy (e.g., wallpaper) applications, which have no legitimate reason to behave in the aforementioned way.

Our proposed changes to Android are minimal compared to frameworks such as TaintDroid, yet also target the runtime behaviour of applications rather than Android's static permissions. Unlike TaintDroid, however, our goal was not to trace data at runtime, but to use simple behavioural specifications to detect a whole range of malicious applications. In the area of computer security, this is known as *behavioural detection*, where the aim is not to identify malware by comparing the applications in question with signatures stored in a database, but to detect the behaviour of known and future malware, which is expected to be similar to existing malware (for a survey cf. [10]).

2 Android Security Concepts in a Nutshell

Let us briefly discuss the important Android security concepts that are relevant to this paper. Note that the aim of this section is not to give a comprehensive overview of the

Android architecture or its security concepts (cf. [8, 9, 5] for that).

Firstly, Android applications and most of the Android stack are written in Java, whereas a modified Linux kernel serves as the platform's low-level OS. Applications on Android are "sandboxed", meaning that each executes within its own virtual machine, and, from an OS point of view, as unique user; that is, unlike standard Linux processes, which inherit the UID of the user who started them, Android applications all have a unique UID. In other words, each application is treated as an individual user from the low-level OS's point of view[1]. This strict "sandboxing" basically ensures that one application cannot modify (or even read—unless dedicated inter-process API calls are being made) the data of another installed application. Unfortunately, however, it is generally not true that the harm caused by a malicious application, is therefore restricted to its "sandbox." In fact, as also pointed out above, there are countless ways in which a malicious application could exploit the device's capabilities, or spy on its users.

Whether or not an application is allowed to use a certain functionality that an Android device offers is primarily determined at install-time, when the standard Android installer presents to the user a list of required application permissions. Users cannot revoke individual permissions that they may not feel comfortable with or that they do not understand, rather they need to grant all permissions or cannot install the application. In consequence, many users do not review the permissions at install-time [6]. In fact, Android's permission system is predominantly static, meaning that once an application is installed, users have basically no means of controlling that application's runtime behaviour. For example, once an application has been granted permission to send SMS, it may do so in the background without requesting further user confirmation. According to the official documentation [9], the lack of dynamic security mechanisms is a design principle: "Android has no mechanism for granting permissions dynamically (at run-time) because it complicates the user experience to the detriment of security." Note that the situation on other mobile platforms, like Nokia's Symbian OS, is similar (cf. [2]).

3 Modelling Security Policies

We assume a set of predicate symbols $\mathcal{P} = P \cup R$, such that $P \cap R = \emptyset$. Security policies in our framework are based on the grammar $\varphi ::= p(t_1, \dots, t_n) | r(t_1, \dots, t_n) | \neg\varphi | \varphi \wedge \varphi | \mathbf{X}\varphi | \varphi \mathbf{U} \varphi | \forall (x_1, \dots, x_n) : p.\ \varphi$, where $p \in P$, $r \in R$ are n-ary predicate symbols, t_i terms, and x_i variables. The term structure is determined by variables and function symbols of given arities. For a ground term t, $t \downarrow$ denotes its actual value, e.g., $(2 + 3) \downarrow$ yields 5, assuming the usual arithmetic functions to be part of our language and interpreted accordingly. Variables range over specific domains such as strings, integers, or any finite domain. Hence, in a statement $\forall x : p.\ \varphi$, p's arity, $p : \tau \to \mathbb{B}$, uniquely determines the *sort* of variable x to be τ. We model observed application behaviour in terms of *actions* which, in turn, are represented by ground atoms. Sets of actions are called *events*. An application's behaviour, seen over time, is therefore a finite *trace* of events, e.g., $\{sms(1234)\}\{login(\text{“user”})\}\dots$ That is, the occurrence of some ground atom $sms(1234)$ in some trace at position $i \in \mathbb{N}_0$ means that at time i it is the case

[1] There is an exception to this rule, but this is not relevant to this paper: applications which share a developer's signature may run under the same UID.

$w, i \models p(t_1, \ldots, t_n)$ iff $p(t_1 \downarrow, \ldots, t_n \downarrow) \in w(i)$;
$w, i \models r(t_1, \ldots, t_n)$ iff $r(t_1 \downarrow, \ldots, t_n \downarrow)$ is true; $w, i \models \neg\varphi$ iff $w, i \not\models \varphi$;
$w, i \models \varphi \wedge \psi$ iff $w, i \models \varphi$ and $w, i \models \psi$; $w, i \models \mathbf{X}\varphi$ iff $w, i + 1 \models \varphi$;
$w, i \models \varphi\mathbf{U}\psi$ iff there exists $k \geq i$ s.t. $w, k \models \psi$ and $w, j \models \varphi$, for all $i \leq j < k$;
$w, i \models \forall(x_1, \ldots, x_n) : p.\ \varphi$ iff $w, i \models \varphi[c_1/x_1, \ldots, c_n/x_n]$, for all $p(c_1, \ldots, c_n) \in w(i)$.

Fig. 1. Kripke semantics of the language wrt. infinite trace w and position i therein.

that $sms(1234)$ is true. As is standard, the semantics of this language is defined via *infinite* traces (see Fig. 1). Note how, unlike symbols from P, symbols from R do not obtain their interpretations via the trace, but by some computational means assumed to be available in the background when evaluating a policy over some trace.

At runtime, a monitor checking φ, will only see a prefix of an infinite trace, denoted u, and therefore return $\top$ if u is a *good prefix* of φ, $\bot$ if u is a *bad prefix*, and ? otherwise. This is akin to the 3-valued finite-trace semantics for LTL introduced in [1], except that our monitor not necessarily reports *minimal* prefixes. For brevity, we cannot give a detailed, step-wise semantics of our monitor, but refer the reader to Sec. 4 for an outline of our algorithm based on the well-known concept of formula progression. Let us now look at example policies, specified in this language and the usual syntactic "sugar".

Recall the promise of our approach and, more generally, of behavioural detection is that it allows not only the detection of specific, known malware, but of new threats as they appear, so long as their damaging behaviour, exhibited on a device, is sufficiently similar to already known malware. Indeed, the databases by security firms such as McAfee, not only list specific malicious applications for Android, but entire evolutions, classified by abstract IDs, such as Android/NickiSpy, to indicate that there exist multiple incarnations of the same malware, realised in differently branded applications. Android/NickiSpy, for example, represents a family of applications which secretly record a user's phone conversation on SD card in the compressed .amr format (adaptive multi-rate). We can detect this family of malware via a simple policy,

$$\mathbf{G}\forall x : sd\_write.\ \neg regcomp(x, \text{“.*\textbackslash.amr”}),$$

where $regcomp(x, y)$ is true if the string x, in this case representing a file name, is in the language given by the regular expression that is represented by string y. However, should there be legitimate recording of .amr files to SD card, the user is always able to ignore any reported violations of this policy.

As another example, consider the first ever Android Trojan (Trojan-SMS.AndroidOS.FakePlayer.a), disguised as media player, which secretly sent SMS messages to expensive premium numbers [11]. This led us to monitor a more general behaviour, i.e., to be notified if *any* application sends an SMS to a number *not in our contacts*:

$$\mathbf{G}\forall x : sms.\ contact(x).$$

While there may be legitimate violations of this policy, its monitoring at least lets users keep track of which applications exhibit this type of behaviour. It's then up to them to decide to remove an application, if they feel it is not justified.

Finally, a lot of malware is "spyware", meaning that private user data or device details are sent out to remote locations. For example, all applications of type Android/Actrack.A send GPS location, battery and radio status to a central internet server controlled by the vendor at regular intervals. A policy we may want to monitor in regards to that, more generally, could be "no application should request the GPS location, and later connect to the internet (possibly to transmit said location)", which is captured by the following formula, where $connect(x)$ appears in a trace whenever the application under scrutiny triggers the Linux system call `connect` to IP address x, and gps whenever it requests the device's current location:

$$\mathbf{G}(\neg((\mathbf{F}\exists x : connect) \wedge gps)).$$

4 Implementation

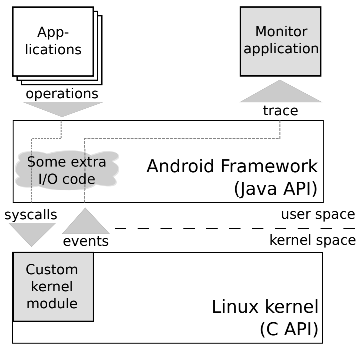

Fig. 2. Architecture.

Currently, our monitors are realised in terms of a stand-alone Android application, written in Java, with a simple GUI that allows users to enter policies. As Android applications are "sandboxed" and therefore unable to monitor each other, we also had to modify the Android stack to facilitate runtime verification in the above sense. To this end, we made some very small, local modifications to exactly two files of the Android system in order to get notified when an application requests permission to perform specific operations or when system events are created, e.g., an application tries to send an SMS message, the system reports low battery status, etc. It is our expectation that this way, our changes will easily carry over to future releases of the platform. Unfortunately, however, it is not possible to obtain all relevant data by intercepting the high-level Android permission checks. In many cases, Android directly consults the underlying, low-level OS if an application is allowed to perform an operation, e.g., based on the application's UID membership in a Linux group. Examples are the opening of a network socket or the writing to an SD card. But also to extract the actual phone number of an outgoing SMS message, we need to monitor the Linux `write` system call (or rather, its arguments) as there are no means to obtain this information in user space without having to modify many additional Android files, but then much to the detriment of portability and maintainability of our solution. For reasons of modularity, we "outsourced" this type of information gathering in our own kernel module that dynamically loads during boot[2]. The architecture is sketched in Fig. 2, where the grey areas are constituents of our system, and arrows indicate relevant information flow.

The actual monitoring of a temporal logic formula is currently realised by means of formula progression; that is, for each formula φ and each application, there is a function $prog$, taking a first-order LTL formula and an event as input and returning a first-order LTL formula, such that $\sigma w \models \varphi$ iff $w \models prog(\varphi, \sigma)$. For example, $prog(\mathbf{G}\psi, \sigma) = prog(\psi, \sigma) \wedge \mathbf{G}\psi$, where $prog(\psi, \sigma)$ may return $\top$ or $\bot$ immediately or after expanding to a more complicated formula. The aforementioned 3-valued finite-trace semantics is obtained by mapping all resulting formulae other than $\top$ or $\bot$ to the $?$-value.

[2] Note that there are numerous Android applications, even on the official Market, that also require the installation of custom kernel modules (e.g., DroidWall requires the netfilter module).

5 Conclusions & Future Work

Although our work is preliminary, arguably, our results show not only that runtime verification is generally feasible on Android devices, but also that it can improve system security by identifying known and *yet unknown* malware. Due to active development, our code is still unreleased, but we have prepared a system demonstration video: http://baueran.multics.org/droid/. Note also that the performance overhead, when executed on an Android emulator as well as on an actual phone, was negligible even in this preliminary, unoptimised version of the code. However, there can be cases, where the runtime performance of our monitoring procedure necessarily deteriorates over time, i.e., the longer the observed trace gets, the longer the formula becomes that needs to be progressed. Although none of our examples triggers this particular problem, there is a need to characterise fragments of our policy language that lead to monitoring algorithms whose complexity at runtime can be guaranteed to depend only on the size of each new event. One such fragment is obtained by discarding the first rule of both the syntax and the semantics, respectively (see Sec. 3), and assuming predicate symbols from R to be rigid. Additionally, the latter must be either at most unary or, if n-ary, their individual use restricted to at most one variable. However, one of the reasons why we have not adopted this fragment here is due to a symbol like $contact$ whose interpretation, arguably, needs to be flexible, i.e., the user can add or delete contacts at any time. Finding useful fragments in the above sense that are also practically relevant is subject of ongoing work.

Acknowledgements: We would like to thank the anonymous referees for their helpful feedback. 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.

References

1. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 20(4):14, 2011.
2. A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proc. 6th Int. Conf. Mobile Systems, Applications, and Services (MobiSys)*, pages 225–238. ACM, 2008.
3. F. Chen and G. Rosu. Java-MOP: A monitoring oriented programming environment for Java. In *Proc. 11th Tools and Alg. for the Construction and Analysis of Systems (TACAS)*, pages 546–550. Springer, 2005.
4. 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 *Proc. 9th USENIX symp. on OS Design and Implementation (OSDI)*. USENIX, 2010.
5. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. 18th ACM conf. Comp. and Comm. Security (CCS)*, pages 627–638. ACM, 2011.
6. A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proc. 2nd USENIX conf. on Web application development*, pages 7–19. USENIX, 2011.
7. A. Goldberg, K. Havelund, and C. Mcgann. Runtime verification for autonomous spacecraft software. In *IEEE 2005 Aerospace Conference (IEEEAC)*, pages 507–516. IEEE, 2005.

8. Google Inc. Android development site. http://developer.android.com/.
9. Google Inc. http://developer.android.com/guide/topics/security/security.html.
10. G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.
11. J. Leyden. First SMS Trojan for Android is in the wild. Web site, The Register, Aug. 2010.
12. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proc. Annual Comp. Sec. Applications Conference (ACSAC)*, pages 340–349. IEEE, 2009.