Towards an authorization framework for app security checking

Joseph Hallett and David Aspinall *

University of Edinburgh

Abstract. Apps don't come with any guarantees that they are not malicious. This paper introduces a PhD project designing the authorization framework used for App Guarden. App Guarden is a new project that uses a flexible assurance framework based on distribution of evidence, attestation and checking algorithms to make explicit why an app isn't dangerous and to allow users to describe how they want apps on their devices to behave. We use the SecPAL policy language to implement a device policy and give a brief example of a policy being used. Finally we use SecPAL to describe some of the differences between current app markets.

1 Introduction

App stores allow users to buy and install software on their devices with a minimum of fuss. Users trust the app stores not to supply them with malware but sometimes this trust is misplaced. A survey of the various different Android markets revealed some malware in all of them—including Google's own Play Store [13]. Malware on mobile platforms typically steals user information and monetises itself by sending premium rate text messages [10]. Sometimes malware on mobile platforms isn't intentional: developers are not security specialists and sometimes make poor security decisions such as signing apps with the same key [2] which can lead to permission escalation on Android or by requesting excessive permissions [7].

Both Google and Apple (who operate the two biggest app markets) check apps submitted to them for malicious code, however neither states exactly what they check for. There is expectation amongst users that these markets check for and are free from malware, but this trust is misplaced [6]. Attempts to reverse engineer the checking procedures for Google's store suggest that they incorporate static and dynamic analysis as well as manual checking by a human being [9]. They offer no guarantees however that any check was actually run.

Digital evidence [11] can be used to confirm the results from static analyses; it is similar to older techniques [8] and allows us to split the *inferring* of a static analysis result (which may be computationally expensive) from the *checking* the evidence (which should be efficient). This allows us to verify the results of static analyses for security properties without having to fully run the analysis.

* With thanks to Martin Hofmann and Andy Gordon for their time and advice.

Static analysis checks used to create digital evidence could include taint analysis tools to check where sensitive information is being leaked between apps [5], or a security proof based on the architecture of the machine [3].

The App Guarden project aims to use digital evidence to provide apps with guarantees that the app cannot act maliciously. This will allow us to make explicit the checks made on an app. With explicit checks users will be able to write policies which describe what kinds of apps they will allow on their devices, and what level of trust they require in these checks for an app to be considered *installable*. We aim to create an enhanced app store that uses inference services, and assurance logics on a modified Android to increase trust in devices.

Figure 1 shows an App Guarden store. It provides apps with evidence generated by an inference service. Devices can check the evidence using a trusted checking service, or rely on their trust that the store believed the app to be safe. The SecPAL Engine on the phone is used to check the assertions made by the store and the checker service and enforce a device policy.

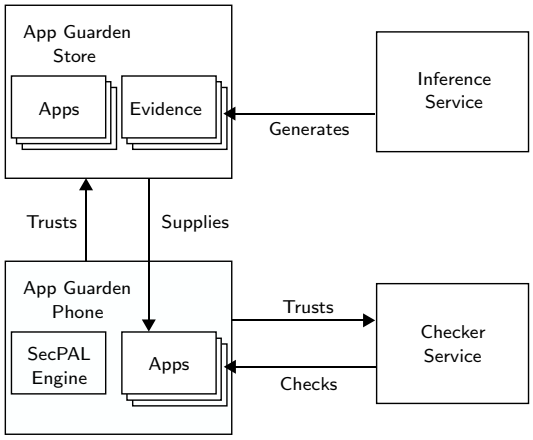

Fig. 1: The architecture of App Guarden.

2 Aims and Objectives

Our aim is to create a modified Android ecosystem where apps come with digital evidence for security and where users can say how they want apps to behave on their device. To do this we will:

- Show how an authorization logic can specify a policy for how a device behaves. This will use ideas such as proof checking and inference, and show how trust can be distributed between principals.
- Show how to use the logic to specify a device policy. We will use digital evidence, and assertions between trusted principals and show how the trust

relationships can be propagated between devices. We show how we can use these statements to build confidence that an app is secure.
– Describe the differences between current app markets using the logic and show how they manage access to special resources (such as the address book or a user's photos). This will allow us to compare App Guarden to other systems and allow comparison between other different schemes.
– Put the logic in a real device and app store and see how the system behaves with real malware and how they interact with device policies. We will look for ways to attack devices with our framework and explore it's limitations.

3 SecPAL and App Guarden

Suppose a user has a mobile device. They might wish that:

> "No app installed on my phone will send my location to an advertiser, and I wont install anything that Google says is malware."

We call this their *device policy*. It says what the user will allow on their device for an app to be installable. To formally write the device policies we use the SecPAL authorization language [4]. The device policy is show in (3). SecPAL is designed for distributed access control decisions. It was designed to be human readable which makes it ideal for our application as we would like users to be able to understand (if perhaps not write) their own device policies. Another advantage is that it lets us separate the constraint solving (the digital evidence checking) from the authorization logic. This means there is no restriction on having the digital evidence be in the same format and logic as the assurance framework (as is the case with other authorization logics such as BLF [12]); this increases our flexibility for implementation as we can re-purpose existing logic solvers.

SecPAL is designed to be extensible. Authorization statements take the form of assertions made by principals. Statements can include predicates which are defined by the SecPAL program, as well as *constraints* (as seen in (9)) that can be checked as sub-procedures for the main query engine. We add the following statements:

$$e_{\text{evidence}} \ \mathsf{shows} \ e_{\text{app}} \ \mathsf{meets} \ e_{\text{policy}} \tag{1}$$

$$e_{\text{app}} \ \mathsf{meets} \ e_{\text{policy}} \tag{2}$$

The shows ... meets predicate in (1) tells us that some evidence could be checked to show that an app satisfies a security policy; an example of which can be seen in (9) bellow (and which also uses a constraint). The plain meets in (2) is an assertion that an app meets a policy whilst offering no actual evidence that the app is secure. The shows statement is stronger than the meets statement as anyone who says that some evidence shows an app meets a policy suggests

they would also say the app meets policy. We can write this in SecPAL with the assertion: "Phone says *app* meets *policy* if *evidence* shows *app* meets *policy*."

We also add an is installable statement to indicate an app is installable on a device and a requires statement to say an app needs a permission to run.

SecPAL allows for attestation of said statements with the *can say* statement: these come with a delegation depth. A delegation depth of zero (as shown in (5)) means the statement must be spoken by the principal and given to us directly in the assertion context. A delegation depth of infinity (as in (6)) means that the statement might not be given to us directly but rather inferred from a statement by different principal but with several *can say* phrases linking it back to the required principal.

To evaluate the device policy and decide whether a specific app is installable SecPAL requires a set of assertions called the *assertion-context*. An example assertion-context and the proof, using two of SecPAL's evaluation rules from the original paper [4]. To illustrate a usage of SecPAL we give an example where a user wishes to decide whether an app Game can be installed. To do this we use the assertion context bellow:

$$
\begin{aligned}
&AC := \{ \\
&\text{Phone says } app \text{ is installable} \\
&\quad \text{if } app \text{ meets NotMalware}, \\
&\quad\ app \text{ meets NoInfoLeaks}. && (3)\\
&\text{Phone says } app \text{ meets } policy \text{ if } evidence \text{ shows } app \text{ meets } policy. && (4)\\
&\text{Phone says NILInferer can say}_0\ app \text{ meets NoInfoLeaks}. && (5)\\
&\text{Phone says Google can say}_\infty\ app \text{ meets NotMalware}. && (6)\\
&\text{Google says AVChecker can say}_0\ app \text{ meets NotMalware}. && (7)\\
&\text{AVChecker says Game meets NotMalware}. && (8)\\
&\text{NILInferer says Evidence shows Game meets NoInfoLeaks} \\
&\quad \text{if LocalNILCheck}(\text{Evidence}, \text{Game}) = \text{true}. && (9)\\
&\}
\end{aligned}
$$

We evaluate it using the SecPAL rules shown in the COND (10) and CAN SAY (11) rules. The COND rule is SecPAL's equivalent to the *modus ponens* rule. Whereas CAN SAY is similar to *speaks for* relationships [1]: if a first principal says a second principal can say something and the second does say it then the first says it too. AC represents the assertion context, D is the current delegation level. From these rules and the assertion context we can prove an assertion that "Phone says Game is installable."

$$
\frac{\begin{array}{r}\{A \text{ says } fact \text{ if } fact_1, \ldots, fact_n, c\} \cup AC, D \models A \text{ says } fact_1, \ldots, fact_n \\ \models c \\ vars(fact) = \emptyset\end{array}}{\{A \text{ says } fact \text{ if } fact_1, \ldots fact_n, c\} \cup AC, D \models A \text{ says } fact}\ \text{COND} \qquad (10)
$$

$$\frac{AC, \infty \models A \text{ says } B \text{ can say}_D \; fact \quad AC, D \models B \text{ says } fact}{AC, \infty \models A \text{ says } fact} \text{ CAN SAY} \tag{11}$$

4 Comparing different devices

We can use the SecPAL language to describe the differences between various app markets and devices. The two most popular operating systems are Google's Android, and Apple's iOS. Apple's offering is usually considered to be the more restrictive system as it doesn't allow the use of alternate market places, whereas an Android user can choose to relax the restrictions and install from anywhere.

For example on iOS an app is only installable on an iPhone if Apple has said the app can be installed (12). In contrast on Android any app can be installed if the user says so (14). The user also trusts any app available on Google's Play Store (15), but the zero delegation-depth means that the user will only believe the app is installable if Google provides the assertion directly whereas the user in (14) is free to delegate the decision because of the infinite delegation depth.

$$\text{iPhone says Apple can say}_\infty \; app \text{ is installable.} \tag{12}$$

$$\tag{13}$$

$$\text{AndroidPhone says AndroidUser can say}_\infty \; app \text{ is installable.} \tag{14}$$

$$\text{AndroidUser says Google can say}_0 \; app \text{ is installable.} \tag{15}$$

Similarly comparisons can be made when apps try to access resources. When accessing components like the address book or photos; on iOS this is only allowed when the user has explicitly okayed it. Other operating systems such as Windows Mobile also follow this pattern. On Android, however, an app can access any resource it declared itself as *requiring* it when it was installed.

$$\text{iPhone says User can say}_0 \; app \text{ can access resource.} \tag{16}$$

$$\tag{17}$$

$$\begin{aligned} &\text{AndroidPhone says } app \text{ can access resource} \\ &\qquad \text{if } app \text{ is installable,} \\ &\qquad\quad app \text{ requires resource.} \end{aligned} \tag{18}$$

This is a brief comparison of the implicit device policies in current markets. It shows however that SecPAL is capable of describing the various difference. This could be extended in future to allow proper comparisons of different markets.

5 Conclusion and future directions

We have given a brief introduction to App Guarden and the SecPAL language used to describe policies. Using the SecPAL language we described some of the

differences between current systems. The PhD work here is at an early stage. Next steps will include porting the SecPAL language to Android and using it to find what kinds of device policies are most effective at stopping dangerous apps, without overly inconveniencing the user. From here we will build an *enhanced app market* with its own set of policies where apps come with digital evidence to allow users to trust apps further.

Mobile devices are a growing target for malware, and with app stores appearing in conventional computers the differences between mobile and traditional computing environments are blurring. The App Guarden will help give users greater assurance their software is secure; and provide a new security mechanism to compliment access control and anti-malware techniques.

References

1. M. Abadi. Logic in access control. *18th Annual IEEE Symposium on Logic in Computer Science*, pages 228–233, 2003.
2. D. Barrera, J. Clark, D. McCarney, and P.C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Security and Privacy in Smartphones and Mobile Devices*, pages 81–12, New York, New York, USA, 2012. ACM Press.
3. G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E.ric Vétillard. MOBIUS: Mobility, Ubiquity, Security. *Trustworthy Global Computing*, 4661(Chapter 2):10–29, 2007.
4. M.Y. Becker, C. Fournet, and A.D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. Technical report, Microsoft Research, 2006. MSR-TR-2006-120.
5. W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, and J. Jung. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *OSDI*, 2010.
6. W. Enck and P. McDaniel. Not So Great Expectations. *Secure Systems*, pages 1–3, September 2010.
7. A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. *Computer and Communications Security*, pages 627–638, October 2011.
8. G.C. Necula and P. Lee. Proof-Carrying Code. Technical report, Carnegie Mellon University, January 1996.
9. J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon*, 2012.
10. A. Porter Felt, M. Finifter, E. Chin, and S. Hanna. A survey of mobile malware in the wild. *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices, CCS-SPSM11*, 2011.
11. I. Stark. Reasons to Believe: Digital Evidence to Guarantee Trustworthy Mobile Code. In *The European FET Conference*, pages 1–17, September 2009.
12. N. Whitehead, M. Abadi, and G. Necula. By reason and authority: a system for authorization of proof-carrying code. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 236–250. IEEE, 2004.
13. 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. *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.