JOSEPH HALLETT

USING AUTHORIZATION LOGICS TO MODEL SECURITY DECISIONS IN MOBILE SYSTEMS

THESIS PROPOSAL

Contents

Introduction

Aim

When an app is installed on Android it prompts the user to accept a list of privileges granted to the app. The user makes the decision based on what they know about the app and their own security policies. In practice a large number of users accept the permissions whatever. This is problematic because some apps are over privileged[1] and some are malicious[2]. Other apps are considered to be potentially unwanted software (PUS) because though they are not malicious in themselves they handle data in a way that is not in the user's interests.

[1] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *the 18th ACM conference*, pages 627–638, October 2011

[2] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, 2012

More generally users and computers make decisions, whether it update an app; whether to connect to a website, based on security policies and trust relationships. These security policies may include the use of tools or experts to decide whether something is malicious. For instance a user may trust a firewall program to enforce their network policy; and they may trust a tool like *Shorewall* to generate the actual policy for them. Alternately a user might wish to be able to install apps but only trust apps *Amazon* have vetted to be installed on their device. *The aim of this project is to formalize these security policies so they can be studied and enforced automatically.*

Mobile operating systems are similar to existing systems but come with a different trust model and are used in a different manner. Software is downloaded from app stores, Apps run within sandboxes and must collaborate and collude to share data with other apps. These devices contain more personal data than ever before: sensors tracking users' locations, gyroscopes measuring how users move, and microphones listening to users calls. The bring your own device (BYOD) movement empowers users to take the devices they have at home and bring them into work. This creates a tension between how the corporate IT department may require employees to use their devices and the user's policies on how they want to use their devices. These features add a novel challenge to modelling these devices and the stores and users surrounding them.

Formalizing the policies allows comparisons to be made between different systems and users. When comparisons are made between the two biggest mobile OSs, iOS and Android, the comparisons is informal: we say iOS is more closed, more of a *walled garden*, Android is more permissive. With a formal language to describe system policy we can make a precise comparison. It allows us to

There is decades of research on analysis tools. These tools can infer complex security properties about the code and systems they

analyze. What there is missing is a glue layer between the assurances these tools can give and the policies users are trying to enforce. By using an *authorization logic* as the glue layer we can enforce the policy by building on the work on access control in distributed systems. Our static analysis tools can be trusted to give statements about code, with other principles, that can be combined to implement a security policy.

Objectives

1. *To instantiate a logic of authorization that allows us to model the trust relationships between the components of an operating system and the users.* This will include analysis tools as principals and allow making decisions based on signed statements from them. The logic must be able to model what happens when apps can collude. The logic may be based off of earlier work on the *SecPAL language*[3] that was used for distributed access control decisions.
2. *To model the decisions made by different operating systems (including Android) using the logic and compare them.*
3. *To implement an app store that serves users only the apps that meet their security policies.* This will include a user-study where we evaluate how well users comprehend their policies and the decisions made for them.
4. *To evaluate something with a user study.*

[3] M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Proc IEEE Computer Security . . .*, 2006

Project description

Introduction and motivation

Related work

Dissertation outline

Review of Android Security

Android is a Linux-based operating system designed for mobile phones and increasingly used in consumer electronics. It comes with a large software market that distributes apps. These apps are built on top of a virtual machine, called Dalvik, and run within a sandbox provided by the OS that is based on Linux's permissions model[4].

[4] Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, March 2014

Permissions and Apps

On top of the traditional permissions level Android has API permissions that apps must request at install time. API permissions

control access to functionality such as the internet, reading or writing external storage, or to learn about the state of the phone. These permissions are displayed to users at install time and must be accepted if the app is to be installed. Most users do not pay attention to these permissions and okay them no matter what is asked for[5]. This has led to malware and PUS that requests excess permissions; which leads to apps sending premium text messages (a common monetization strategy[6]) or stealing private information.

There are tools, however, which can detect when an app is over privileged. The *Stowaway* tool[7] mapped Android permissions onto the API calls to access the functionality they enabled. This allowed them to detect when apps were over privileged by looking for apps which had the permissions but none of the associated API calls. The *PScout* tool[8] improved upon Stowaway by increasing the accuracy of the permissions map and by deriving the map from the Android source code, rather than the fuzzing based methods Stowaway used.

One criticism of the API permissions is that they are quite broad. For instance the *internet* permission allows an app to send or receive anything on the internet. Several people have proposed a *fine grained permissions model* and developed tools to support it. The *RefineDroid, Dr. Android & Mr. Hyde* tools[9] are a suite designed to discover which permissions can be made fine-grained, rewrite apps to use these permissions and then enforce them at runtime; they do this on a stock Android system without requiring rooting or kernel modification. Alternately the *AppFence* tool[10] works without modifying apps. It allows users to write fine grained policies for what data an app can receive: if an app exceeds its bounds then the request is either denied or fake data supplied in its place. This does require modifications to the Android OS however. The *AppGuard* tool[11] offers something in between: it rewrites apps to use a security monitor to enforce security policies at runtime.

[5] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. *Android permissions: user attention, comprehension, and behavior.* ACM, July 2012

[6] E Chien. Motivations of recent android malware. *Symantec Security Response*, 2011

[7] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *the 18th ACM conference*, pages 627–638, October 2011

[8] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. *the 2012 ACM conference*, pages 217–228, October 2012

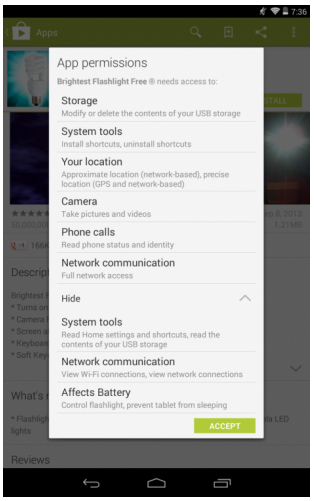

Figure 1: The *Brightest Flashlight Free* app prompting for it's permissions at install time. This app is over privileged as a flashlight app should have no need for GPS or phone data, or network access.

[9] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. *Dr. Android and Mr. Hide: fine-grained permissions in android applications.* ACM, October 2012

[10] P Hornyack, S Han, J Jung, and S Schechter. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th . . .*, 2011

[11] M Backes, S Gerling, C Hammer, and M Maffei. AppGuard–Enforcing User Requirements on Android Apps. *Tools and Algorithms for . . .*, 2013

Intents and Collusion

Android uses a novel IPC mechanism called *Binder*. Apps use *intents* to share data and handle events. For instance if an app wishes to handle an `SMS_RECEIVED` action it can declare itself a *broadcast receiver* for it and the app will be started when the event occurs. Alternatively if an app wants to open a URL in the browser it can send an `ACTION_VIEW` intent and the user's chosen browser will take open the URL. Apps can create their own intents and can restrict usage of them (if they wish) to only a limited number of other apps (or those signed with a specific key).

These mechanisms also allow apps to collude to increase their

privilege levels, or share data inappropriately. To do this consider the case where there are two apps communicating: one which can use the network and another which cannot. If the unprivileged app asks the privileged app to send data on its behalf, and the privileged app forwards the network responses back to it; then the unprivileged app has the network permission without needing to declare it to the user. Alternately if a privileged app does not secure its intents then they may break the protections offered by permissions. An example of this was the *Kies* app on Samsung Galaxy S3 phones that could be exploited to allow any app to install other apps[12].

Several tools have been developed to detect these privilege escalation attacks such as *Quire*[13] that added origin tracing to intents. Other tools such as *ScanDroid*[14] statically analyze apps to find data-flows across components and produce a series of constraints that should be satisfied to guarantee information security. The *Kirin*[15] tool certifies apps at install time against a policy based on the potential data-flows introduced by the permissions they request.

The *TaintDroid*[16] and *FlowDroid*[17] have both had a large impact. Both tools both use taint analysis to track the data passed between apps and detect when sensitive data is being leaked to an app that should not have access to it through intents, however others have shown that the approach is not perfect[18] and can be defeated by malicious apps.

[12] Andre shka Moulu. From 0 perm app to INSTALL_PACKAGES on Samsung Galaxy S3, July 2012. URL `http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html`

[13] S Bugiel, L Davi, and A Dmitrienko. Towards taming privilege-escalation attacks on Android. *19th Annual . . .*, 2012

[14] A P Fuchs, A Chaudhuri, and J S Foster. SCanDroid: Automated security certification of Android applications. . . . , 2009

[15] William Enck, Machigar Ongtang, and Patrick McDaniel. *On lightweight mobile phone application certification.* ACM, New York, New York, USA, November 2009

[16] 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

[17] C Fritz, S Arzt, and S Rasthofer. Highly precise taint analysis for android applications. *EC SPRIDE*, 2013

[18] G Sarwar, O Mehani, and R Boreli. On the effectiveness of dynamic Taint analysis for protecting against private information leaks on android-based devices. *International Conference on Security and Cryptography*, 2013

Policy Languages

When an action is performed, such as reading a file or installing an app, there are a set of conditions that must be met for the action to go ahead. These conditions form the *authorization policy* for that decision. When these policies describe what actions are permissible in order to maintain a secure system then we call it the *security policy*.

The policies often contain a notion of *trust* where certain principals may be trusted to make statements about other principals and what is permissible. To model the relationships many logics have been proposed that can be implemented to decide whether an action can be authorized automatically.

Early authorization logics, such as *PolicyMaker*[19] grew out of the logics of authentication proposed by Wobber, Abadi, Burrows, and Lampson[20,21]. PolicyMaker allowed authorities to declare trust in other principles (identified through asymmetric keys) for certain actions or to declare further trust relationships. The language was designed to be minimal and did not specify how the policies should be checked: they suggested by using regular expressions, or checking programs written in a sandboxed version of AWK however any lan-

[19] M Blaze, J Feigenbaum, and J Lacy. Decentralized trust management. *Security and Privacy*, 1996

[20] B Lampson, M Abadi, and M Burrows. Authentication in distributed systems: Theory and practice. *ACM Transactions on . . .*, 1992

[21] E Wobber, M Abadi, M Burrows, and B Lampson. Authentication in the Taos operating system. *ACM Transactions on . . .*, 12(1):3–32, 1994

guage could have been used. The author suggested that the language might be good as a model for the public-key infrastructure. Later work introduced *KeyNote*[22] which they claimed was a simplified version of PolicyMaker designed to support public-key infrastructure.

Later other languages such as *RT*[23] were introduced. RT allowed principals to be given roles (in a manner similar to an role based access control (RBAC) access-control system) and for decisions to be made based on which roles an entity held. This meant that RT could express general statements that were not expressible in the PolicyMaker languages such as:

> "Anyone who is a preferred customer and a student can get a discount."

Several different versions of RT were described: the simplest being RT_0[24] and with RT_1 and RT_2 adding support for parameterized-roles and logical-objects respectively; each with extensions to provide other features.

By providing a translation into *Datalog* (specifically *Datalog with constraints* or *Datalog*^{C}[25]), the RT family of languages was shown to be tractable, unlike earlier languages. Datalog is a query language similar to *Prolog* but that doesn't support nested sub-queries or functions and has a safety condition[26]. Datalog is a subset of first-order logic and is known to be tractable: i.e. all queries can be done in polynomial time.

Influences form the RT family of languages and Datalog^C can be seen in the *Cassandra* policy rule language[27]. Cassandra was a trust management system that could be used to model large complicated systems. In his doctoral thesis Becker showed how the NHS Spine (a complex and informally defined system concerning access control and roles in health care) could be formally modelled in the Cassandra language.

In Cassandra principals activate and deactivate roles. Actions can only be completed if the principal holds the required roles. Delegation is allowed through an appointment mechanism where one principal can activate a role on another principal. Like the RT languages Cassandra is tractable as it can be translated to Datalog^C.

The *Binder* language[28] was designed for authorization decisions[29] and implemented as an extension of Datalog. Properties are given to entities by creating arbitrary predicates for them, and a special *says* modality allows statements to be imported from third parties.

Authorisation is granted by checking to see if a predicate can be deduced from the knowledge base, however because Binder does not add any special predicates, and Datalog does not allow functions there can be no notion of state.

[22] Matt Blaze, Joan Feigenbaum, and Angelos D Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Security Protocols*, pages 59–63. Springer Berlin Heidelberg, Berlin, Heidelberg, January 1999

[23] N Li and J C Mitchell. Design of a role-based trust-management framework. *Security and Privacy*, 2002

[24] N Li, W H Winsborough, and J C Mitchell. Distributed credential chain discovery in trust management. *Journal of computer security*, 2003

[25] Ninghui Li and John C Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Practical Aspects of Declarative Languages*, pages 58–73. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2003

[26] All variables in the head must occur in the body.

[27] M Y Becker and P Sewell. Cassandra: flexible trust management, applied to electronic health records. *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 139–154, 2004

$$\texttt{canActivate}(mgr, \texttt{AppointEmployee}(emp)) \leftarrow \texttt{hasActivated}(mgr, \texttt{Manager}())$$
$$\texttt{canActivate}(emp, \texttt{Employee}(app)) \leftarrow \texttt{hasActivated}(app, \texttt{AppointEmpolyee}(emp))$$

Figure 2: Role delegation in the *Cassandra* policy language. A manager is allowed to activate the employee role for an arbitrary entity by appointing them.

[28] J DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113. IEEE Comput. Soc, 2002

[29] M Abadi. Logic in access control. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 228–233. IEEE Comput. Soc, 2003

```
can(X, read, file) :- employee(X, company).
employee(X, company) :- hr says empolyee(X, company).
hr says employee(john, company).
```

Figure 3: Statements in *Binder* to say that in the current context only employees can read a file, and that an employee they must have

SecPAL

The *SecPAL* authorization language[30] is an authorization logic for decentralized systems. Early experiments indicate that it is highly suitable for modeling the distributed nature of software installation, app stores and mobile devices so we will describe it in more detail than other authorization languages.

Syntactically SecPAL appears similar to Binder, however it has a richer syntax that allows for constraints and decisions to be made based on state (such as the time). SecPAL was designed to be readable and has a more verbose, English like, language than other authorization logics.

Like Binder it contains a *says* statement however unlike Binder it requires that all statements are said by a principal explicitly rather than relying on a default context. SecPAL also allows arbitrary predicates to be created, and also adds two additional special modalities to the logic. The *can-say* statement allows for explicit delegation and has two varieties. The *can-say*$_\infty$ phrase allows for nested delegation, whereas the *can-say*$_0$ statement does not. SecPAL also adds a *can-act-as* phrase that allows for aliasing entities.

Later extensions of SecPAL[31] add support for guarded universal quantification and remove the *can-act-as* statement. Other languages such as *DKAL*[32] built and eventually split from SecPAL. DKAL was designed to express distributed knowledge between principals by adding to the trust delegation mechanisms already in SecPAL. They also showed how any SecPAL statement could be translated into DKAL. The *SecPAL4P* language[33] was an instantiation of (the extended version of) SecPAL designed to specify how users' wished their personally identifiable information (PII) to be handled.

The inference rules for SecPAL are shown in Figure 4. Queries are evaluated against a set of known statements (the assertion context (AC)) and an initially infinite delegation level (D). If the rules show that the query is valid then SecPAL says the statement is okay else it is rejected.

[30] M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Proc IEEE Computer Security . . .*, 2006

[31] M Y Becker. SecPAL: Formalization and Extensions. 2009

[32] Yuri Gurevich and Itay Neeman. DKAL: Distributed-Knowledge Authorization Language. *. . . Symposium*, pages 149–162, 2008

[33] M Y Becker, A Malkis, and L Bussard. A framework for privacy preferences and data-handling policies. *. . . Cambridge Technical Report*, 2009

Proposal

Work Done In First Year

During the first year of my studies we have focussed on developing an authorization logic that can express the security policies a user might have for their smart phone; in particular the policies when the user is installing apps. We have considered what kinds of policies and trust relationships a user might wish to express and shown how they can be expressed in the language.

$$\frac{(A \text{ says } fact \text{ if } fact_1, \ldots, fact_k, c) \in AC \quad AC, D \models A \text{ says } fact_i\theta \ \forall i \in \{1 \cdots k\} \quad \models c\theta \quad \text{vars}(fact\theta) = \emptyset)}{AC, D \models A \text{ says } fact\theta}\ \text{cond}$$

$$\frac{AC, \infty \models A \text{ says } B \text{ can say}_D fact \quad AC, D \models B \text{ says } fact}{AC, \infty \models A \text{ says } fact}\ \text{can say}$$

$$\frac{AC, D \models A \text{ says } B \text{ can act as } C \quad AC, D \models A \text{ says } C \ verbphrase}{AC, D \models A \text{ says } B \ verbphrase}\ \text{can act as}$$

Figure 4: The inference rules used to evaluate SecPAL. All SecPAL rules are evaluated in the context of a set of other assertions AC as well as an allowed level of delegation D which may be 0 or ∞.

To do this we initially looked at a variety of authorization logics including BLF[34] and Binder[35] before settling on SecPAL as it was both simple, extensible and readable. SecPAL's decentralized nature was felt to be ideal for describing a mobile-device and app-store ecosystem as there isn't a single authority making decisions about what can and cannot be installed onto a device.

When considering the policies we wanted to allow users to delegate decisions to experts who might be third party certification services or static analysis services running on a remote server or on the device itself. There should be the ability to use digital evidence[36] as a means of increasing trust in an external tool as this might allow proof checking to be done with less strain on a mobile's battery. We also wanted to have a clear separation between the checking of the user's security policy for the device (the *device policy*) and the policies any tool was checking for an app (the *application policy*). This means that any analysis tool needn't use the same logic for checking the application policy, as the device uses for checking it's own policy. In the security policy static analysis tools are treated as oracles: they can utter statements about their inputs but we do not know (or care) how they came to these conclusions.

To do this we extended SecPAL with two new predicates. The *meets* predicate is used to state that some entity believes an app meets an application policy. For instance if Alice believed that the *Angry-Birds* app met her policy to not leak information about her then we would have the statement:

```
Alice says AngryBirds meets NoInfoLeaks.
```

To express the notions of proof carrying code[37] and digital evidence we want to say that some evidence *shows* a policy is met. To do this we introduce the *shows-meets* predicate (whose notation we sugar somewhat). As an example consider again Alice who this time has managed to get some digital evidence to show Angry-Birds won't

[34] 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

[35] J DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113. IEEE Comput. Soc, 2002

[36] I Stark. Reasons to Believe: Digital Evidence to Guarantee Trustworthy Mobile Code. In *The European FET Conference*, pages 1–17, September 2009

[37] G C Necula and P Lee. Proof-carrying Code. Carniegie Mellon University, 1996

leak her information.

```
Alice says Evidence shows AngryBirds meets NoInfoLeaks.
```

Example

To provide a better idea of the logic might be used we will describe a story where a user is trying to install an app on their mobile phone. This example is built from our work that was presented as a paper at the ESSoS Doctoral Symposium[38] and as a poster at the FMATS workshop.

[38] Joseph Hallett and David Aspinall. Towards an authorization framework for app security checking. In *ESSoS Doctoral Symposium*, pages 1–6. University of Edinburgh, February 2014

Suppose Alice has a smart phone. Alice has a security policy that says:

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

Alice trusts Google to decide whether something is malware or not (or at at least recommend an anti-virus vendor who can be trusted), and she trusts the *NLLTool* to decide whether an app will leak her location data. Alice has heard about digital evidence and is happy that if an app can come with a proof of it meeting a policy then she will believe it.

She translates her policy into SecPAL thus:

```
Alice says app is-installable
  if app meets NotMalware,
     app meets NoLocationLeaks.

Alice says Google can-say inf app meets NotMalware.
Alice says NLLTool can-say 0 app meets NoLocationLeaks.

anyone says app meets policy
  if evidence shows app meets policy.
```

Alice wishes to install Angry Birds. To do so she downloads the app from a modified app store where apps come with statements about their security. Alice takes the statements from the store and builds her assertion context. These statements include a delegation from Google to say McAfee can be trusted to decide whether an app is malware or not, as well as some statements from McAfee and the NLLTool about the app itself. The full assertion context is shown in Figure .

Alice then uses SecPAL to decide whether or not the assertion context supports the idea that `Alice says app is-installable.`

```
Alice says app is-installable
  if app meets NotMalware,
  app meets NoLocationLeaks.
anyone says app meets policy
  if evidence shows app meets policy.
Alice says Google can-say inf
  app meets NotMalware.
Alice says NLLTool can-say 0
  app meets NoLocationLeaks.
Google says McAfee can-say 0
  app meets NotMalware.
McAfee says
  AngryBirds meets NotMalware.
NLLTool says ABProof shows
  AngryBirds meets NoLocationLeaks.
```

Figure 5: The full assertion context used to evaluate Alice's query.

Implementation

To evaluate her security policy we implemented the SecPAL logic. The implementation was done in the Haskell programming language and is around a thousand lines of code plus five hundred lines of test cases (including comments).

Whilst in the original SecPAL paper[39] Becker, Fournet, and Gordon describe an efficient implementation using Datalog; this implementation uses a simpler goal-oriented *brute-force* approach. It was written in this way to quickly evaluate whether SecPAL is a good fit for the problem, rather than to be an efficient production ready inference engine[40] That said it supports command history, dynamically loaded constraint-functions, comes with syntax highlighting plug-ins for Vim, and has handled simple assertion contexts with over a thousand statements: whilst it is not ideal it can happily serve as a reference for a later efficient implementation.

[39] M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of computer security*, 2010

[40] In fact it could never be used for this as most Android devices run using ARM processors which are poorly supported by Haskell compilers.

```
AC, inf [app\AngryBirds] |= Alice says AngryBirds is-installable.
 AC, inf [app\AngryBirds] |= Alice says AngryBirds meets NotMalware.
  AC, inf [app\AngryBirds] |= Alice says Google can-say inf app meets NotMalware.
  ---------------------------------------------------------------------------
  AC, inf [app\AngryBirds] |= Google says AngryBirds meets NotMalware.
   AC, inf [app\AngryBirds] |= Google says McAfee can-say 0 app meets NotMalware.
   ---------------------------------------------------------------------------
   AC, 0 |= McAfee says AngryBirds meets NotMalware.
    AC, 0 |= True
    -------------
 AC, inf [app\AngryBirds] |= Alice says AngryBirds meets NoLocationLeaks.
  AC, inf [app\AngryBirds] |= Alice says NLLTool can-say 0 app meets NoLocationLeaks.
  ---------------------------------------------------------------------------
  AC, 0 [anyone\NLLTool, ...] |= NLLTool says AngryBirds meets NoLocationLeaks.
   AC, 0 [evidence\ABProof] |= NLLTool says ABProof shows AngryBirds meets NoLocationLeaks.
    AC, 0 |= True
    -------------
   AC, 0 |= True
   -------------
 AC, inf |= True
 --------------
```

Figure 6: Proof output by the SecPAL tool when evaluating Alice's query. The proof is presented as an inverted $infty$erence tree where indented statements are the proofs for each condition of the unindented line above. Underlining indicates something is known to be true as it either exists in the assertion context or is true in itself. Variable substitutions are shown in brackets to aid debugging

Second year

Third year

Bibliography

M Abadi. Logic in access control. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 228–233. IEEE Comput. Soc, 2003.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: analyzing the Android permission specification. *the 2012 ACM conference*, pages 217–228, October 2012.

M Backes, S Gerling, C Hammer, and M Maffei. AppGuard–Enforcing User Requirements on Android Apps. *Tools and Algorithms for . . .*, 2013.

M Y Becker. SecPAL: Formalization and Extensions. 2009.

M Y Becker and P Sewell. Cassandra: flexible trust management, applied to electronic health records. *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 139–154, 2004.

M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Proc IEEE Computer Security . . .*, 2006.

M Y Becker, A Malkis, and L Bussard. A framework for privacy preferences and data-handling policies. *. . . Cambridge Technical Report*, 2009.

M Y Becker, C Fournet, and A D Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of computer security*, 2010.

M Blaze, J Feigenbaum, and J Lacy. Decentralized trust management. *Security and Privacy*, 1996.

Matt Blaze, Joan Feigenbaum, and Angelos D Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Security Protocols*, pages 59–63. Springer Berlin Heidelberg, Berlin, Heidelberg, January 1999.

S Bugiel, L Davi, and A Dmitrienko. Towards taming privilege-escalation attacks on Android. *19th Annual . . .*, 2012.

E Chien. Motivations of recent android malware. *Symantec Security Response*, 2011.

J DeTreville. Binder, a logic-based security language. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 105–113. IEEE Comput. Soc, 2002.

Joshua J Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A Ridley, and Georg Wicherski. *Android Hacker's Handbook*. John Wiley & Sons, March 2014.

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.

William Enck, Machigar Ongtang, and Patrick McDaniel. *On lightweight mobile phone application certification*. ACM, New York, New York, USA, November 2009.

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *the 18th ACM conference*, pages 627–638, October 2011.

Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. *Android permissions: user attention, comprehension, and behavior*. ACM, July 2012.

C Fritz, S Arzt, and S Rasthofer. Highly precise taint analysis for android applications. *EC SPRIDE*, 2013.

A P Fuchs, A Chaudhuri, and J S Foster. SCanDroid: Automated security certification of Android applications. *. . .*, 2009.

Yuri Gurevich and Itay Neeman. DKAL: Distributed-Knowledge Authorization Language. *. . . Symposium*, pages 149–162, 2008.

Joseph Hallett and David Aspinall. Towards an authorization framework for app security checking. In *ESSoS Doctoral Symposium*, pages 1–6. University of Edinburgh, February 2014.

P Hornyack, S Han, J Jung, and S Schechter. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th . . .*, 2011.

Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. *Dr. Android and Mr. Hide: fine-grained permissions in android applications*. ACM, October 2012.

B Lampson, M Abadi, and M Burrows. Authentication in distributed systems: Theory and practice. *ACM Transactions on . . .*, 1992.

N Li and J C Mitchell. Design of a role-based trust-management framework. *Security and Privacy*, 2002.

N Li, W H Winsborough, and J C Mitchell. Distributed credential chain discovery in trust management. *Journal of computer security*, 2003.

Ninghui Li and John C Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Practical Aspects of Declarative Languages*, pages 58–73. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2003.

Andre shka Moulu. From 0 perm app to INSTALL_PACKAGES on Samsung Galaxy S3, July 2012. URL `http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html`.

G C Necula and P Lee. Proof-carrying Code. Carniegie Mellon University, 1996.

G Sarwar, O Mehani, and R Boreli. On the effectiveness of dynamic Taint analysis for protecting against private information leaks on android-based devices. *International Conference on Security and Cryptography*, 2013.

I Stark. Reasons to Believe: Digital Evidence to Guarantee Trustworthy Mobile Code. In *The European FET Conference*, pages 1–17, September 2009.

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.

E Wobber, M Abadi, M Burrows, and B Lampson. Authentication in the Taos operating system. *ACM Transactions on . . .*, 12(1):3–32, 1994.

Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, 2012.