Securing Enterprise Data on Smartphones using Run Time Information Flow Control

Palanivel Kodeswaran
IBM Research India
Bangalore,India
Email: palani.kodeswaran@in.ibm.com

Vikrant Nandakumar
IBM Research India
Bangalore,India
Email: vikrant.nandakumar@in.ibm.com

Shalini Kapoor
IBM India Software Labs
Bangalore,India
Email: kshalini@in.ibm.com

Pavan Kamaraju
University of Maryland, Baltimore County
Baltimore, USA
Email: pavan4@cs.umbc.edu

Anupam Joshi
University of Maryland, Baltimore County
Baltimore, USA
Email: joshi@cs.umbc.edu

Sougata Mukherjea
IBM Research India
New Delhi,India
Email: smukherj@in.ibm.com

***Abstract*—There is an increasing penetration of smart phones within enterprises. Most smart phone users now run both enterprise as well as personal applications simultaneously on their phones. However, most of the personal apps that are downloaded from public market places are hardly tested for enterprise grade security, and there have been instances of malware appearing in public markets that steal sensitive user information. Smart phone platforms such as Android require users to explicitly provide permissions to applications at install time, yet lack run time monitoring of permission usage by applications. In this paper,we present a framework for the run time enforcement of privacy policies on smart phones, in particular, protecting the privacy of enterprise data on smart phones. Our privacy policies are defined in terms of permissible information flows on the phone during different contexts. This arms users with finer grained control over information access by different applications. In our policy framework, an information flow is defined based on the entities involved in the corresponding inter-process communication(IPC) viz, the caller, callee and the associated IPC data. The information flow policy specifies the conditions under which an IPC flow may be permitted (or denied). Our system tracks information flows at run time and enforces that only flows satisfying all the current policies are permitted on the phone. We describe the design and implementation of our policy based framework in Android, and present performance evaluation results measuring the overhead imposed by our framework.**

I. Introduction

Smart phones are now increasingly being used as gateways to the information infrastructure. Not only do smart phones provide anywhere, anytime access, they come equipped with myriad onboard sensors that can assist in context sensitive retrieval of information such as the nearest restaurant. Smart phones are also being increasingly leveraged by enterprises. However, a fundamental distinction of the smart phone adoption within enterprises compared to the laptop era is that, unlike laptops, the phones are typically owned by the employee. Employee ownership markedly constrains the security capabilities that can be deployed on the phones. While on the one hand enterprises are encouraging employees to “Bring Your Own Device(BYOD)” to interact with the enterprise information infrastructure, on the other hand they are concerned with the confidentiality of data leaving the enterprise servers to reside on the employees device. There is an increasing threat to enterprise data resident on employees’ devices which can now be stolen from the device, either due to theft of the phone or malware [1]. A new challenge in the BYOD space is that the enterprise has no control over applications that are installed on the user device, and cannot prevent those applications from inappropriately accessing (and perhaps sending out) enterprise data. Users install random apps from market places which are not tested for security. Platforms such as Android perform permission checks only at install time and the users generally have no choice but to accept all the permissions sought by an application. There are no run time controls to prevent applications from misusing the permissions granted at application installation time. Furthermore, mobile platforms such as Android have Inter-Process (IPC) and Inter-Component (ICC) communication as part of core application functionality, which opens up possibilities for information leakage via undesirable information flows among apps. For instance, on Android, a (rogue) gaming app installed by the user can look for standard locations (on external storage) where enterprise mail applications store their data, access them, and then transmit them outside. This will bypass any access control restrictions on the transmittal by the mail application to a third party!

In this paper, we propose a policy based framework for securing information access on smart phones through run time monitoring and enforcement of information flow policies. Our system prevents rogue applications from stealing enterprise data at run time by constraining information access based on the type of the requesting process as well as the type of the requested data– enterprise or personal. Furthermore, our system prevents information leakage through inter-process communications by restricting the set of allowable IPCs based on the attributes of the applications involved in the IPC.

Our information flow policies are based on the

1) Inter Process Communication (IPC) call chain defined by the set of processes/apps requesting access

978-0-7695-4713-8/12 $26.00 © 2012 IEEE
DOI 10.1109/MDM.2012.50

IEEE computer society

2) Attributes of the data viz. enterprise or personal data
3) Phone state determined by various attributes such as location, battery status etc. The phone state determines the current policy to be enforced on the phone

The main contributions of our work can be summarized as

1) Policy based framework for system wide run time information flow control on smart phones
2) Run time monitoring of information flows
3) Enforcement of holistic information flow policies based on IPC call chain, data attributes as well as phone state

II. Related Work

There has been considerable research in the recent past with regards to run-time enforcement of privacy policies on smartphones. Bugiel et al. [2] propose TrustDroid that provides domain isolation between enterprise and personal applications. TrustDroid employs Tomoyo linux based Mandatory Access Control(MAC) mechanisms in the kernel to enforce file system as well as UNIX level domain isolation. Furthermore, a firewall manager in the kernel prevents personal applications from communicating to other apps on the phone through sockets. In addition to kernel level mechanisms, the authors modified the Android framework to provide isolation at the ICC as well as content providers and services level as well. However, trustdroid does not secure dynamic enterprise data, as the MAC domains are setup at application installation time, and cannot be modified. Lange et al. [3] propose a virtual machine based approach for securing smart phone applications. While virtualization based approaches provide the desired domain isolation, they tend to be power hungry and require multiple copies of the platform to run simultaneously preventing consolidation of common services and data such as phone contacts.

There has also been a lot of work in protecting user data on phones, and Enck [4] provides a survey of the approaches available to protect smart phone user privacy. The authors in [5] describe TaintDroid, which uses dynamic taint analysis to provide run time monitoring of privacy sensitive information flows. When sensitive data leaves the device through the network interface, TaintDroid raises an alert specifying the type of sensitive data being transmitted as well as the application responsible for the transmission. TaintDroid uses well defined sources of information to tag data, where as in our approach enterprise data could come from any app on the phone. The authors in [6] present a XACML based privacy framework for enforcing user privacy policies with regards to mobile mash-ups accessing device features such as GPS, camera etc. based on a number of parameters such as the URL of the mashup, time of day, number of SMS messages sent by the mashup and so on. Similarly, the authors in [7] allow users to specify their own run time constraints that need to be satisfied before providing access to device capabilities. Jagtap et al. [8] propose using semantic web based technologies to represent higher abstractions of context and specify declarative policies to govern the conditions under which user information may be shared, and at what granularity. However, the above approaches constrain device capability or per application information access and do not consider inter-application control or data flows. SAINT [9] extends Android security mechanisms to allow applications enforce install time and run time policies to protect application interfaces from misuse by other apps. These policies provide finer grained control to application developers to control access to their app from other calling applications. However, the policies do not constrain the data flow among apps.

Davi et al. in [10] describe privilege escalation attacks on Android smartphones exposed through vulnerabilities in the Android Scripting Environment (ASE) that allows malicious apps to send text messages to premium-rate numbers. Dietz et al. in [11] present a light weight call-chain tracking system for Android, that enables the callee to access the entire call chain from the caller, and enforce its policies based on the received call chain. XManDroid (eXtended Monitoring on Android) [12] proposes a security extension to Android's middleware to mitigate privilege escalation attacks at run time. Android's reference monitor is extended to include a graph based model of inter-application communication built by runtime monitoring of communication links between the applications. Policies are specified over the graphical model, and only if all the policies are satisfied, is the inter-application communication permitted. While the above approaches constrain the control flow among apps, they do not consider the corresponding data flows.

III. Problem Statement

In this section, we describe the problem and motivate the need for fine grained information flow control in smartphones based on runtime monitoring. We are primarily concerned with protecting enterprise data on smart phones which can be classified along the following dimensions

1) Static Enterprise Data
2) Dynamic Enterprise Data
3) Concurrent execution of Enterprise and Personal Applications

Static Enterprise Data We define static enterprise data as data that is natively generated by enterprise applications such as email, calendar etc. Email is one of the commonly used enterprise applications and therefore securing emails and the corresponding attachments is critical. Enterprise emails typically contain a lot of sensitive data both in the body as well as attached documents. While most enterprise email clients encrypt and protect email data from arbitrary accesses, clients typically provide a facility for the attached documents to be decrypted and exported out of the app. These attachments are stored in public directories such as the sdcard, and the application no longer has control over the exported data. In most cases, static enterprise data can be secured by simple access control policies that enforce that only enterprise applications can access enterprise data.

Dynamic Enterprise Data refers to data that is typically not generated by enterprise applications, but rather generated by personal or system apps, and become sensitive to the enterprise only at certain times, or broadly, only under certain contexts. One such piece of information is location information. For example, consider a rural mobile banking agent who uses an enterprise app to record transactions as he goes around villages collecting and distributing money. Typically, any personal application that has the required android permissions would have access to the location information. However, when the agent is running the banking app, location information becomes enterprise sensitive, and should be accessible only to the banking app, and not to third party personal apps in order to protect the physical security of the agent (The threat model assumes a malicious app can continuously report location information to a third party server, which can be used for stalking the agent). In other words, the sensitivity of location information dynamically changes, and only privileged enterprise applications should have access to the information.

Securing the concurrent execution of Personal and Enterprise Apps The concurrent execution of enterprise and personal apps on the phone raises yet another serious privacy challenge for enterprise data resident on the phones. Android supports collaboration between applications through Inter-Component(ICC) and Inter-Process Communication (IPC). Application developers are tasked with protecting the interfaces of their applications through permissions, and this is typically error prone as most developers are not security experts, thereby opening the possibilities for privilege escalation attacks. From an enterprise perspective, inter-process communication is a security threat due to the potential of information leakage when an enterprise app calls the services of a personal app along with the associated enterprise data. Existing solutions such as [11] track the provenance of call chains and provide developers an access control primitive to prevent privilege escalation, yet these solutions are independent of the data flow triggered by the ICC. For example, an enterprise app should not be allowed to use facebook to upload enterprise pictures, while it is permissible to use an enterprise service to upload the same pictures. In this case, the enterprise sensitivity of the picture determines which service can be used for uploading. Similarly, privilege escalation attacks can be used to infiltrate enterprise servers with personal and malicious content, when a malicious app uses the unprotected interfaces of the enterprise's uploading service to upload malicious content.

A. Threat Model

The threat model we consider in this paper involves malicious third party apps trying to steal enterprise data, both static and dynamic, resident on the phone, as well as infiltrating enterprise servers with malicious content. We assume that the end user is trusted although they may be benign in installing malicious third party apps. Similarly, we assume that the enterprise apps are trusted although they may be exposing unprotected interfaces that can be exploited by malicious apps.

IV. Policy Based Framework

In this section we describe the policy abstractions that enable the high level specification of permitted information flows to secure enterprise data on phones. Our policies constrain information flows based on context and involve the following entities

- The caller application
- The callee application
- Associated IPC data and its attributes such as provenance
- Phone State such as location,time and the set of currently running apps

A. Application Classification

Our framework requires that apps be classified as enterprise or personal. There are multiple mechanisms available for classifying apps as enterprise based on parameters such as market source, developer signature, or apps pushed directly by the enterprise.

B. Data Classification

Similar to application classification, data on the phone needs to be classified.

Static Enterprise Data Static enterprise data is identified through the writing application, in that all data generated by an enterprise application is classified as enterprise data. Android applications write data to standard locations in the phone memory as well as external storage, and from the application's package name we can locate static enterprise data generated by the application.

Dynamic Enterprise Data: Unlike static data, dynamic enterprise data is determined by context as well as current data flows in the phone. We continuously monitor data flows at run time and classify data that has been opened, and hence tainted, by an enterprise app as enterprise data irrespective of the application that initially created the data. While this approach could potentially lead to a large number of false positives, we choose to be conservative with regards to protecting enterprise data. Similarly, we classify data obtained from an enterprise server as enterprise data e.g. files downloaded by the browser from the URL "http://www.ibm.com".

We will next describe our policies that govern the following inter-process communications

Starting Activities: Android applications can start another application's activity or service to accomplish their task. For example, the contacts application can start the phone app to initiate a phone call. From a security perspective, an enterprise application might want to specify that it can be started only by other enterprise applications, for example, to prevent untrustworthy data from personal apps infiltrating the enterprise servers. While android permissions can be used to protect interfaces, permissions are always requested and granted at install time, and there are no run time controls on permission usage. Further, the app might specify that it can be

started only under a certain device context. Here, we define a device context as a specific device configuration state such as being physically located inside the enterprise, connected to corporate wifi, turning off certain features such as bluetooth, camera, microphone etc. Contexts can be switched on or off either manually by the user or automatically by policy. More generally, our policies can specify which apps can start another app based on the attributes of the source and destination apps such as the app's signature, set of permissions requested etc. [9]. The access control policy to start an enterprise app can be expressed as follows

```
startApp(tgtApp,srcApp):
tgtApp.type=Enterprise ∧
System.DeviceContext=EnterpriseContext
∧ srcApp.type=Enterprise/System →
allow(tgtApp,srcApp)
```

The above policy is interpreted as follows. An application or component, scrApp, is allowed to start/bind to an enterprise app or component, tgtApp, only if srcApp is an enterprise/system app and the device is in Enterprise Context.

Receiver Updates: Android allows applications to register receivers to asynchronously receive updates from the underlying system. These receivers could range from coarse grained Broadcast receivers that receive system wide updates from the Android framework to fine grained application specific receivers such as location receivers that receive specific updates from the underlying location manager. To prevent privacy threats that arise from intent hijacking, it becomes critical to specify and enforce policies that restrict the set of receivers that are called back when an event occurs. In other words, only receivers that satisfy the current policy should be allowed to receive event updates from the underlying system. For example, when an enterprise application takes a picture, only other enterprise apps should be notified of "picture taken" event. Such a policy is expressed as follows

```
recvUpdate(tgtApp,srcApp,event):
tgtApp.type=Enterprise ∧
System.DeviceContext=EnterpriseContext
∧ srcApp.type=Enterprise ∧
event.Action="Picture Taken"
→ allow(tgtApp,srcApp)
```

While it is possible to express more generic policies regarding broadcast receivers, such as those in [9], in this work we focus on policies that govern finer granularity application specific receivers such as location receivers that an application registers with the location manager. Recalling the rural mobile banking scenario, in order to protect the physical security of the agent, the enterprise might require that only an enterprise app have access to location information in the enterprise context. This policy is expressed in our framework as follows

```
locationAccess(app,GPS):
app.type=Enterprise ∧
System.DeviceContext=EnterpriseContext
→ allow(app,GPS)
```

Data Access Policy: Enterprise data resident on the phone must be protected from unauthorized applications. Data access policies specify the conditions under which enterprise data on the phone may be accessed. Our Enterprise policy specifies that enterprise files can be accessed only by enterprise applications with in the Enterprise Context. Such a policy can be expressed as follows

```
dataAccess(app,data):
app.type=Enterprise ∧
System.DeviceContext=EnterpriseContext
∧ app.type=Enterprise → allow(app,Data)

dataAccess(app,data):
data.type=Enterprise ∧ app.type=Personal →
deny(app,Data)
```

IPC Call Chain Data Handling: IPC call chains typically contain the data or, URI's to the data, to act upon. To enforce privacy, the enterprise might want to restrict information flows based on both the IPC call chain as well as data in the IPC. For example, the enterprise might want to enforce that only data from enterprise applications be uploaded to their back end servers. Also the enterprise needs to ensure that data created on the device, and uploaded to the enterprise servers retain their integrity and are not modified by non-enterprise apps. In other words, data tainted by personal apps should not be involved in an IPC that invokes an enterprise app to upload data to the enterprise servers. This policy can be expressed in our framework as follows

```
IPCDataAccess(srcApp,tgtApp,data):
data.taint!=Personal ∧
srcApp.type=Enterprise ∧
tgtApp.type=Enterprise
→ allow(srcApp,tgtApp,data)

IPC DataAccess(srcApp,tgtApp,data):
data.taint=Personal ∧
tgtApp.type=Enterprise →
deny(srcApp,tgtApp,data)
```

V. System Architecture and Implementation

In this section, we describe the design and implementation of our framework. Figure 2 shows the various components of our system architecture. Our framework builds on top of standard Android security mechanisms and consists of the following additional architectural blocks

The Policy Manager: The policy manager is responsible for storing and managing the various policies specified by the user. At system boot, the policy manager reads in policies from a file and stores it in its internal data structures. Figure 1 shows a xml representation of our policies that is used by the

```
<permission = "deny">
<source>
     <application>
          <packagename> any </packagename>
          <type> enterprise </type>
     </application>
</source>
<target>
     <application>
          <packagename> any </packagename>
          <type> personal </type>
     </application>
</target>
<data>
     <url> any </url>
     <type> enterprise </url>
</data>
```

Fig. 1. XML Representation of policy to prevent data leak of enterprise data to personal apps

policy manager. An initialization policy specifies the device configuration to be effected on the phone to place it in the desired start up context. When an ICC occurs (1), the reference monitor in Android's Activity Manager Service calls the Policy manager. The policy manager inspects the intent and extracts the ICC context composed of the caller and callee components, as well as the corresponding data and its attributes such as the taint set, from the appropriate taint tracking service (2). The policy manager retrieves the appropriate policies based on the ICC context, and contacts the package manager (3) and other phone services to retrieve the current system state (4) that is used in policy evaluation. The policy manager evaluates all the retrieved policies, and if all the evaluated policies permit the ICC, the policy manager returns with a permit (5), and Android's standard ICC permission checks are performed (6). If Android's permission checks also evaluate to true, the ICC is permitted to continue (7). If any of the retrieved policies evaluates to false, or the standard Android permission checks fail, the ICC is denied to continue and an error returned to the caller. Similar checks are performed in the location manager service when the underlying hardware generates location change events.

TaintTracker and File System Reference Monitor: The taint-Tracker is responsible for maintaining the provenance of data. We implement the taint tracking service as a linux kernel module in Android's underlying linux kernel. We intercept the file open system call to perform file level taint tracking based on the user id of the calling application. For each file stored on external storage, we maintain a taint set that identifies the set of Android applications, based on their unique uids, that have opened the file. The taint set for a file initially contains the user id of the process that created the file. For taint propagation, at each system call interception, we extract the user id of the caller and store the extracted user id in the file's taint set. Given the taint set of a file, we can contact the package manager service to obtain the set of enterprise and personal applications that have opened the file. If the taint set of a file contains an enterprise application, we classify the corresponding data as enterprise. When an IPC involving file system data occurs, the policy manager retrieves and evaluate the policies corresponding to the file access. Only if all the retrieved data access policies evaluate to true, is the IPC permitted.

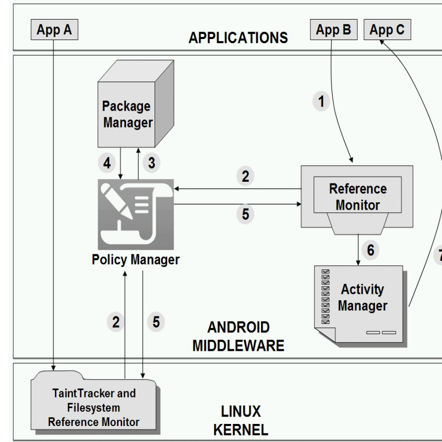

Fig. 2. System Architecture

VI. Evaluation

In this section, we describe our experiments to evaluate the performance overhead posed by our policy framework. All experiments were conducted on a Nexus One running Android 2.2. For our policy evaluation, we modified the android framework to insert our policy hooks in the Android Application Manager Service as well as the Location Manager Service, before calling the standard android security policy evaluation. File-level taint tracking was enabled through a loadable kernel module. Each experiment was conducted five times, and mean values reported.

A. Application Benchmarks

We wrote a set of applications to isolate the performance overhead due to different types of policy checks.

Application Load Time In this experiment, we measure the duration elapsed between starting an activity and when the activity is actually displayed. Our experiment consists of two apps where the first app broadcasts an intent to start the target app's activity. This duration includes the policy evaluation overhead as well as the time taken by Android's Intent resolver.

Figure 3 shows the performance overhead of policy checks. As expected, the overhead experienced (122 ms) with both the middleware and kernel policy checks turned on is much higher compared to the overhead from middleware level only policy checks(35ms). This is due to the fact that, with kernel level policy checks, every file access, including those made by the class loader while loading the target app's dex files, is checked for policy evaluation as well as taint propagation, resulting in increased overhead.

Location Update Policy We wrote a simple app that registers a listener for location updates from the GPS. We also wrote a mock GPS provider, that periodically reads GPS co-ordinates from a file and updates the GPS location. We measure the

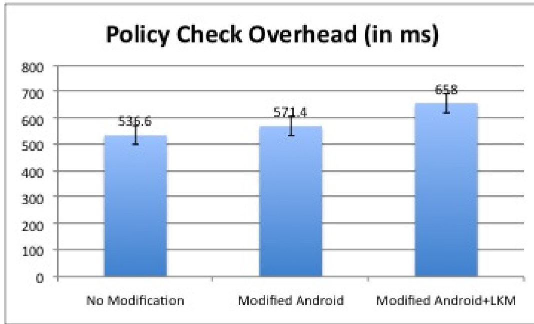

Fig. 3. Performance Overhead of Application Level Policy Checks (in ms)

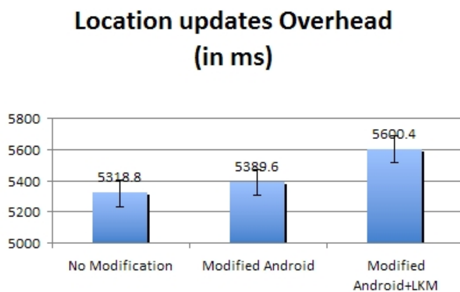

Fig. 4. Performance Overhead of Location Policy Checks (in ms)

duration from the start of the app till the last location update is received. Our location policy evaluation only updates listeners that satisfy the current policy. From figure 4, we find that there is negligible overhead (70ms) imposed due to location policy evaluation in the Android middleware. On the other hand, the high overhead imposed by the file level taint tracking could be attributed to our experimental set up in which the location file is periodically opened to generate location updates.

VII. Future Work and Conclusion

In this paper, we address the problem of securing enterprise data, static and dynamic, on smart phones. To this end, we have presented a policy based system for run time information flow control on smart phones. Our policies are generic and are based on the information flow context viz. caller and callee applications, as well as attributes of the data and phone state. Our system involves tracking information flows at run time and enforcing privacy policies to prevent the leakage of enterprise data from smart phones, as well as the infiltration of personal content onto enterprise servers. In future work, we plan to work on policy management for smart phones. In our current work, the number of policies may grow unwieldly and could potentially result in policy conflicts. We would like to point out there is an inherent trade off between language simplicity and conflict resolution [13], [14], and given the simplicity of our language, conflict resolution of our policies should be decidable. Additionally, we could solicit user input to choose between allowing or denying an information access when there are policy conflicts, which would enable our system to learn from user input. In future work, we plan to set up user studies to understand the impact of our performance overhead on user experience as well as quantify the amount of security our framework provides to enterprise data under real world scenarios.

References

[1] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046618

[2] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046624

[3] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4android: a generic operating system framework for secure smartphones," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 39–50. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046623

[4] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *ICISS*, 2011, pp. 49–70.

[5] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of OSDI*, 2010. [Online]. Available: http://www.usenix.org/events/osdi/tech/full_papers/Enck.pdf

[6] S. Adappa, V. Agarwal, S. Goyal, P. Kumaraguru, and S. Mittal, "User controllable security and privacy for mobile mashups," in *The International Workshop on Mobile Computing Systems and Applications*, Phoenix, 2011.

[7] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332. [Online]. Available: http://portal.acm.org/citation.cfm?id=1755732

[8] P. Jagtap, A. Joshi, T. Finin, and R. L. Z. Gutierrez, "Preserving privacy in context-aware systems." in *ICSC'11*, 2011, pp. 149–153.

[9] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," in *2009 Annual Computer Security Applications Conference*. Ieee, Dec. 2009, pp. 340–349. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5380692

[10] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," *Information Security*, pp. 346–360, 2011.

[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach, "Quire: Lightweight Provenance for Smart Phone Operating Systems," in *USENIX Security Symposium 2011*, 2011. [Online]. Available: http://arxiv.org/abs/1102.2445

[12] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," *Security*, 2011.

[13] L. Kagal, "Rei : A Policy Language for the Me-Centric Project," HP Labs, Tech. Rep., September 2002, http://www.hpl.hp.com/techreports/2002/HPL-2002-270.html.

[14] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken, "Kaos policy management for semantic web services," *IEEE Intelligent Systems*, vol. 19, pp. 32–41, July 2004. [Online]. Available: http://dx.doi.org/10.1109/MIS.2004.31