Saarland University
Information Security & Cryptography Group
Prof. Dr. Michael Backes

Android Security Lab WS13/14
Project Proposals

M.Sc. Sven Bugiel

Version 1.0 (October 15, 2014)

Contents

Organizational Matters

General Topic

In this lab, you will work on selected projects related to contemporary issues of Smartphone Security, where the focus is on the open-source and popular Android OS.

Teams

Each team consists of 2 students and will work on one project. Register your team before the corresponding deadline via email to the lab supervisor, stating name, student ID and email address of each team member as well as the preferred project and one to two backup projects.

If you do not have a group, please contact the lab supervisor by email stating your name, student ID, preferred project and backup projects, so that he can assign you to another group accordingly.

Custom Topics

It is generally possible for you to propose a custom project idea or deviations from the proposed projects. However, this has to be discussed with and approved by the lab supervisor **before** the project assignment deadline!

Android Version

In general, the teams are free to chose the Android version on which they implement their architecture. A minimum of version 2.2 is recommended to target more common versions[1]. However, it should be noted that some projects (like KeyChain Extension and Integration) are only available on Android 4.x which has extensive hardware requirements in terms of storage, memory and processing power for building! If you do not have a *good* workstation or desktop machine at your disposal, please reconsider taking another project! If an actual root exploit is required (instead of simulating it with the *su* tool), the team is advised to use a vulnerable Android 2.x version.

Reporting

Although the work is to a big extent conducted autonomously by the teams, you are advised to keep in contact with the lab supervisor to ensure your work is on track and satisfies the requirements for a (very) good grade.

In addition, each team has to submit a final report (approx. 10 pages), that is mandatory for successful participation. The final report also has to include the source code (or, in case of system modifications, a patch file) of the team's solution. The teams should clearly state which authors were responsible for which part of the work and reports. Language of the report can be either German or English (the choice does not affect the grade, however, you are encouraged to write in English). The style of the report can be freely chosen, but should be reasonable w.r.t. margin width, font size, etc. to ensure readability.

The final report should be written for an audience that is familiar with the topic (i.e., no extensive background sections, but focus on the project related facts). The final report should contain the following content:

1. a short introduction to the chosen topic which explains the addressed problem/vulnerability, motivation of the work, and proves that you understood the problem set
2. a design section, explaining the architecture/attack and design decisions made w.r.t. the Android design as well as *briefly* discussing possible alternative approaches
3. an implementation section briefly describing technical aspects (e.g., in which class which function was added/modified) in order to realize the design and to ease the code review process
4. a *short* conclusion summarizing the results, lessons learned, and potential open problems

At the end of the course, a concluding meeting is held, at which each team has to shortly present its results to the other teams and an audience familiar with Android security research.

[1] `http://developer.android.com/about/dashboards/index.html`

Deadlines

Registration of teams and topics	Friday, October 17, 2014
Project assignment	Friday, October 17, 2014
Final report and code submission	Friday, November 14, 2014

Project 1: Capability Leakage Detection

Description

A *confused deputy* [15, 6, 10, 23] (on Android) is a privileged app that unintentionally performs privileged actions on behalf of an unprivileged app. For instance, Enck et al. [10] have shown that it was possible on (an obsolete version of) Android for an unprivileged app to trigger the default Phone app to start a call on behalf of this app. Felt et al. [23] have reported more confused deputies among the system apps. Furthermore, different frameworks [31, 20, 13, 12] for detection of privacy leaks in Android applications have been presented, e.g., a privileged application that accesses and locally stores (e.g., in custom, local ContentProviders) security/privacy sensitive information (such as Contacts information) is unintentionally exposing this information over unprotected interfaces (e.g., querying of the local ContentProviders).

This project deals specifically with detection of such privacy or capability leaks in *3rd party apps.* That means, a privileged app from, e.g., the Google Play market, stores security and privacy sensitive data such that another 3rd party app can retrieve this information without holding the corresponding permissions or that the app leaks these information (unintentionally) to data sinks such as the network or the SD card. For instance, an app with *READ_ CONTACTS* permissions might store the results from its queries to the ContactsProvider in an unprotected local ContentProvider, on unprotected storage (e.g., SD card or a private file but with *0777* file permission), or send them to the system log.

Project Goal

Develop a *simple* analysis tool for APK files, which can determine based on blackbox testing whether a 3rd app leaks its capabilities for a set of selected privileged operations (e.g., access to user's contacts). In essence, this tool should be used by a human tester and it should check whether the unprotected interfaces of privileged apps can be exploited for retrieving sensitive information or if the app leaks sensitive data, e.g., via the file-system [26] or the log sub-sytem [18, 28].

The approach could be based on an analysis app that tests other installed apps or based on static analysis of APK files.

Proposed Work Items

- First, potentially vulnerable apps must be identified. One metric to do so, would be to analyse the Manifest of apps to check for exported Service or ContentProvider interfaces that are not protected by a (custom) permission, or for permissions that allow access to certain sinks such as the external storage or log.
- Blackbox testing:
  - Blackbox testing relies on accessing the unprotected interfaces and checking whether one can retrieve sensitive data such as contacts or SMS information.
  - Additionally, the log entries on the platform should be investigated for sensitive information, that the vulnerable app logged [18, 28]
  - Moreover, recent incidents further show that apps might store files in the private data directory, however, with system-wide read/write permission [26]. Thus, the tool should also investigate the data directory of the app.
  - For blackbox testing, a semi-automated[2] testing tool *is* expected, with which a human tester can check an app for vulnerabilities. When performing blackbox testing at runtime against other installed apps, this most likely requires that the target apps are used for some time so that they actually retrieve and store privacy-sensitive data.
  - The functionality of the tool should be tested with a synthesized example app, but also other 3rd party apps that can be provided to you by the TA.

[2]Automating user-input is a hard task and does not have to be solved.

- Alternatively to blackbox testing, static analysis could be applied. Different tools, like *androguard*, exist to help static analysis.
- Although beneficial, the solution does *not* have to be fully automated! Rather, the focus of this work should lie on solving the following challenges:
  - Data flow sinks and sources must be identified and a subset of them considered in the implementation. For instance, sources are calls to security and privacy sensitive APIs like location manager, contacts provider, telephony manager, etc. Sinks are API calls which store data from the sources in unprotected locations (e.g., exported content provider not protected by a permission, storage on disk with insufficient file permissions, or sending the data to the log subsystem of Android). In particular identifying a sink like an unprotected content provider requires pre-processing (e.g., analysis of the app's manifest file) in order to detect vulnerable interfaces.
  - Depending on the identified sources and sinks an exemplary analysis should be performed for one source-sink pair. The analysis can be performed on 3^{rd} party apps or a synthesized example app.
  - Android apps have several entry points due to the event driven model (e.g., broadcast receivers, event listeners, user input, etc). Thus, the analysis has to take this into account when performing data flow analysis. For instance, data flowing from a sensitive source such as the ContactsProvider via a local ContentProvider to an unprivileged app consists of two separate data flows that have to be connected during analysis: 1) ContactsProvider $\longrightarrow$ Local ContentProvider; and 2) Local ContentProvider $\longrightarrow$ Unprivileged app.

Project 2: Data Shadowing

Description

The high popularity of smartphones and the increasing amount of private data stored and processed on these devices has made them an appealing target of spyware and malware attacks. These privacy violating attacks range from overprivileged apps [19] to nosy advertisement libraries [14] and also relate to controversial discussions about the popular WhatsApp [4, 3] Path [2], and Facebook [1] apps, which have been alleged to clearly overstep the necessary boundaries of their access to user's private data, violating the user's privacy.

Project Goal

The goal of this project is to enable *data shadowing* [17, 32, 7] on Android. Data shadowing means, that an app receives empty, fake, or filtered data on queries to sensitive APIs. For instance, a query to the ContactsProvider might return an empty Cursor or calling the LocationManager for the current location could return a fake location. Thus, data shadowing protects the user's privacy from nosy apps or spyware. In comparison to throwing a (security) exception, returning empty/fake/filtered data when an app should not access certain sensitive data is more "gracefully", i.e., the chance that the app crashes unexpectedly is decreased.

Proposed Work Items

- For implementing data shadowing both ContentProviders (e.g., Contacts) and Services (e.g., Location) have to be considered. For simplicity, the focus should lie on the *system* providers and services, which, in contrast to 3^{rd} party apps, can be modified to achieve data shadowing features.
- The students should consider at least one example each for shadowing a system Service and a system ContentProvider.
  - System Services usually require that the API functions of the service are instrumented to perform the shadowing (e.g., *getLastKnownLocation* of the LocationManager).
  - System ContentProvider are more flexible: 1) The filtering could be specific and inherent to the provider (e.g., using SQLViews) or 2) more generic at the interface level (e.g., modifying the result Cursor; Android 4.x provides corresponding Cursor classes).
  - The resolution of the access control should be considered carefully. Returning empty data or fake data is technically easy, in contrast, providing fine-grained access control might be more difficult depending on the location of the access control enforcement (inherent vs. at interface) but allows the user to adjust the access rights of apps according to his needs to share and protect private data. For example, while returning only fake or no contacts information to a messenger app like WhatsApp would protect the user's privacy, it also prevents this app from functioning correctly. Being able to adjust the data filtering such that this app only receives names and phone numbers of contacts but no further information would enable the user to chose the middleground between protecting and sharing his data. For ContentProvider, which represent structured data in table-format, the access control can be as fine as per-cell or per-row/per-column. For instance, filtering contacts can be based on the group (i.e., an app can only access "Friends" but not "Workmates") or based on the data mimetype (i.e., an app can retrieve name and phone number of all contacts, but not email, nickname, IM contact, etc.) or a combination of these two.
- A particular additional challenges is how the policies are managed, i.e., who configures to which data an app has access and how does he configure it. For instance, the user could configure it via an extra app or an external party could configure it by deploying policy files.

Project 3: KeyChain Extension and Integration

Description

Since Android API level 14 (Android 4.0), Android offers a *KeyChain*[3] service that provides storage of and access to private keys and their corresponding certificate chains. Thus, the KeyChain implements a system-wide credential storage to which 3^{rd} party apps can store their credentials and later request access to keys, usually by means of an *alias*.

Project Goal

The nature of this project is rather of a *functional* nature and aims at improving the applicability and integration of KeyChain into certain services such as SMS or Contacts management. In particular, KeyChain lacks some of the features that 3^{rd} party apps such as AGP[4] provide (e.g., integration with K9 Mail for encrypted emails), but, on the other hand, has as a *system* app the potential for tighter integration into other system services such as the SMS service. This integration and extended functionality of KeyChain should be explored in this project.

Warning: KeyChain is only available on Android 4.x which has extensive hardware requirements in terms of storage, memory and processing power for building! If you do not have a *good* workstation or desktop machine at your disposal, please reconsider taking another project!

Proposed Work Items

More precisely, the following two ideas should be analyzed and (a subset of them) implemented:

1. Integration into other system services and system apps
   - One obvious integration is the system Contacts app and the system ContactsProvider. Similar to a key ring, Contacts entries could be extended with pointers to the keys that are assigned to the ID of the corresponding contact. Thus, services which use the Contacts data, e.g., communication services, can request that sent/received data to/from this contact can be explicitly encrypted.
   - This integration could be extended to the SMS/MMS app, which would now offer communication with encrypted SMS (group SMS do not have to be implemented) with Contacts for which a key has been assigned.
   - A point that should be considered (but not necessarily implemented) is how to roll out these keys. For instance, users could perform a "key signing party" with their smartphones to exchange keys (e.g., using Bluetooth or NFC) or the keys could be delivered together with the Contacts data, e.g., from an exchange or LDAP server.
2. Extended functionality
   - Currently, KeyChain supports only asymmetric keys. One drawback of asymmetric keys in the above described scenario could be their size, meaning SMS messages may be unnecessarily large due to the key size.
   - To this end, KeyChain should in a second step be extended beyond its current functionality by providing (simple) support for some other credential formats such as symmetric keys like AES keys.
   - Topics related to this task, which should be considered but not necessarily implemented, are the required access control to such "de-/encryption oracle" functions, the role of the user in this process, and different backends ("providers") for KeyChain such as SmartCards.

[3]`http://developer.android.com/reference/android/security/KeyChain.html`
[4]`http://de.appbrain.com/app/apg/org.thialfihar.android.apg`

Project 4: Enhanced Installer

Description

The package installer of Android is part of its *PackageManagerService*. By default, all new apps that shall be installed are handled by this component[5], which parses the APK files, inserts relevant information, e.g., from the Manifest file, into its databases, assigns a (shared) UID, and moves the app's files into the corresponding locations on disk. Moreover, before the installation actually starts, the user is prompted to allow or deny the installation based on his assessment of the requested permissions.

Project Goal

The goal of this project is to extend the installation process with further security-related features. In particular, the installer should enforce security policies which determine if an app can be installed or not before the user is prompted. This project targets rather corporate security related issues, i.e., business smartphones on which the company that issues the phone to its employees is the primary security principal on the phone and decides on the security policies. Security policies in this project can be based on 1) static properties of the app (e.g., which permissions the app requests [11], which intent-filters it registers (e.g., is it substituting the default Launcher), which developer signature the app has [22], or if it was signed by the company [7]); 2) dynamic properties of the system (e.g., if AppA is installed, AppB cannot be installed); 3) additional security mechanisms like a Virusscan.

Proposed Work Items

- In essence, three additional steps should be added to the install process by modifying the Package-ManagerService at the corresponding locations:
    1. Enforcement of policies based on the static properties of the new app (i.e., developer signature, permissions requested).
    2. Enforcement of policies based on the system state, i.e., which applications are already installed. In particular shared UIDs should be considered, since an app inherits all permissions previously granted to this shared UID, i.e., granted to all installed apps with this UID!
    3. Implementation of a *simple* virus scanner for native code contained in APKs, which denies installation if it detects malicious code.
- Consider how the installation policies are managed. For instance, they could be managed through Android device management API or you could design a custom app/service to securely manage the install policies.
- Additionally, different alternative approaches should be discussed and their drawbacks and benefits be evaluated. For instance, security apps such as LookOut do not require modifications to the operating system, but instead monitor system events for newly installed apps, scan those apps, and trigger uninstallation if necessary.

[5]Exceptions exist on rooted phones, where apps can be installed via an *adb push* command.

Project 5: Context-aware Malware and KeyLogger

Description

One design principle of Android OS is that all applications are equal. As such, (most) pre-installed system applications' functionality can be substituted by a 3^{rd} party app. For instance, the user can install an alternative browser, launcher (i.e., the default home screen), or keyboard. When an Activity should be started for which different applications are registered (e.g., opening a URL or going back to the home screen), the user is prompted to chose which application should be used to execute this action.

Additionally, smartphone devices are equipped with an increasing number of on-board sensors to which applications can request access. For instance, GPS to determine the geolocation of the the device or accelerometer and gyroscope to determine the device's orientation and rotation. Besides those sensors, a number of other device states can be retrieved by applications, such as the state of the screen (on/off), the Wi-Fi network to which the device is connected, discovered/connected Bluetooth devices, etc. In sum these information allow an application and also malware to operate more intelligently based on the current device context/state.

Project Goal

Goal of this project is to implement and evaluate two exemplary attacks in the spirit of ethical hacking. The two selected scenarios for this attack are 1) context-aware malware, which performs malicious operations only when a certain context is met (e.g., particular app in foreground on screen, connection to particular network, state of the screen, etc); and 2) a malicious keyboard app which contains keylogging functionality and sends logged input (in particular usernames and passwords) to a remote attacker. Specifically, the project should consider how easy the user can be victim of such attacks and which security mechanisms Android provides to counter this class of attacks.

Proposed Work Items

- The basis for the KeyLogger implementations can be the source code of the original keyboard app available in the Android sources. For the implementation of context-aware malware, the implementation can be freely chosen.
- Different approaches from research to context-aware malware exist and can be used for inspiration. For instance, *(sp)iPhone* [21], *PlaceRaider* [27], *SoundComber* [24], *Stealthy Video Capturer* [29], or *TouchLogger* [9]/*TapLogger* [30].
- Special consideration and discussion should be the deployment and prevention of such attacks.

Project 6: Subverting Binder

Description

At the heart of Android's inter-process communication is *Binder IPC*. In particular, it is the basis for communication between application components. For instance, starting an Activity, sending Intent messages, or calling the application framework API (e.g., reading the device location, contacts, sending SMS, etc.) is implemented on top of Binder IPC. Thus, in many situations privacy- and security-sensitive data is sent via Binder IPC packages between application processes. In some situations, even intuitively intra-app communication (such as switching Activities) involves in fact Binder IPC. Generally, the Binder IPC communication is not encrypted, thus susceptible to monitoring.

Project Goal

The goal of this project is to subvert the Binder communication of apps. You should use library injection[6] of a modified `libbinder.so` to mount different attacks, such as keylogging or stealing sensitive data like SMS messages. The modified `libbinder.so` will act as a man-in-the-middle in the target app's IPC and extract privacy- and security-sensitive information from Binder IPC packages and then forward this information to an attacker app installed on the device.

Proposed Work Items

- Develop a malicious version of `libbinder.so`. You can test it based on modifying an Android build or exchanging the `libbinder.so` on an AOSP debug build.
- Develop an attacker app that will 1) inject the finished library into other apps and 2) receive the stolen data and present it to the attacker. For simplicity, it is sufficient to just log this information instead of sending it to a remote server, etc.
- Library injection requires root privileges. For simplicity you can use the `su` tool to emulate a successful and persistent root exploit of the device.

[6]For example `https://code.google.com/p/libandroidinjector/`

Project 7: Intent Reference Monitor

Description

Intent messages are the most important abstraction of inter-process communication on Android and are used in a plethora of situations, such as starting Activities, starting Services, or Broadcasting information. Responsible for the life-cycle management of Intents are the applications. That means, an Intent object is created and modified within the application process and only handed to the application framework when the Intent is being sent to another application or system service. As a consequence, system-centric access control architectures can only enforce policies on Intents when these are being sent. For instance, type enforcement approaches [25, 8] or more generic access control frameworks [16, 5] have to label (or classify) every Intent when it is being sent. Moreover, once the Intent object has reached its receiver, control is lost over how the receiver modifies the Intent or whether it should access all data payload of the Intent.

Project Goal

Goal of this project is to implement a reference monitoring system service for Intent objects. The Intent life-cycle management should be centralized into a new system service and applications hold only references to Intent objects managed by this system service. Thus, sending an Intent is reduced to forwarding a reference to an Intent object from the sender application to the receiver application(s).

The benefit of this approach is, that the new Intent service acts as a reference monitor that can control to which Intent or even which attribute of an Intent object every app has access to. Certain attributes that are important for policy models like type enforcement can be protected from modification by apps and hence allow for the first time a persistent and secure labeling of Intent objects. Conceptually, this can be compared to known, established reference monitors like the virtual file system: Intent takes the role of files, the Intent service takes the role of the virtual file system, and application processes are the subjects in both cases.

Proposed Work Items

- A new system service for managing (i.e., creating, modifying, etc.) Intent objects should be created.
- A new Manager class (i.e., Proxy) for this service must be created. This manager should support the same interface as the current `Intent.java`. Moreover, the manager must manage a reference (e.g., UUID) to the Intent object it represents.
- Sending of this new kind of Intents requires modification of the `ActivityManagerService` and related subsystems like the `BroadcastQueue`.
- For simplicity, a set of test apps should be created to test this new service.
  - **Important notice:** For simplicity, the new Intent service should only be deployed in parallel to the existing Intents and no integration into system apps/services is required!
- Particular technical challenges that should be considered are:
  - Access to the same Intent object by multiple applications. Since every app holds a reference to a centrally stored Intent object, multiple apps could hold the same reference (e.g., after sending the Intent to another app), and hence this access must synchronized. One possible approach could a *copy-on-write* scheme, where modifications to an Intent that is shared by multiple apps must be cause the object to be copied before modification.
  - Garbage collection of Intents. When the Intent service manager is cleaned by the garbage collector, the Intent service must be informed about this event, so that it can delete the Intent that the manager represented and thus avoid a memory leak. This could be implemented based on the Java object's finalization routine (`finalize()`) of the manager object.[7]

[7]For example `https://www.science.uva.nl/ict/ossdocs/java/tutorial/java/javaOO/finalize.html`

References

[1] Facebook Caught Reading User SMS Messages? | TalkAndroid.com. http://www.talkandroid.com/94623-facebook-caught-reading-user-sms-messages/.

[2] Path uploads your entire iPhone address book to its servers. http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html.

[3] WhatsApp storing messages of users up to 30 days | Your Daily Mac. http://www.yourdailymac.net/2012/02/whatsapp-storing-messages-of-users-up-to-30-days/.

[4] WhatsApp took all my contacts and sent to their servers without asking me - BlackBerry Forums at CrackBerry.com. http://forums.crackberry.com/blackberry-apps-f35/whatsapp-took-all-my-contacts-sent-their-servers-without-asking-me-649363/.

[5] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android Security Framework: Extensible multi-layered access control on Android. In *Proc. 30th Annual Computer Security Applications Conference (ACSAC'14)*. ACM, 2014.

[6] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)*, 2012.

[7] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

[8] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. 22nd USENIX Security Symposium (SEC'13)*. USENIX Association, 2013.

[9] L. Cai and H. Chen. Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *6th USENIX conference on Hot topics in security (HotSec'11)*. USENIX Association, 2011.

[10] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, 2008.

[11] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS'09: Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[12] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *5th international conference on Trust and Trustworthy Computing (TRUST'12)*. Springer-Verlag, 2012.

[13] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[14] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *5th ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)*. ACM, 2012.

[15] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988.

[16] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC'14)*, Mar. 2014.

[17] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *18th ACM conference on Computer and communications security (CCS'11)*. ACM, 2011.

[18] A. Lineberry, D. L. Richardson, and T. Wyatt. These aren't the permissions you're looking for. BlackHat USA 2010. http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf, 2010.

[19] Lookout Mobile Security. Security alert: Geinimi, sophisticated new Android Trojan found in wild. `http://blog.mylookout.com/2010/12/geinimi_trojan/`, 2010.

[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM conference on Computer and communications security (CCS '12)*. ACM, 2012.

[21] P. Marquardt, A. Verma, H. Carter, and P. Traynor. (sp)iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, 2011.

[22] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *25th Annual Computer Security Applications Conference (ACSAC'09)*, 2009.

[23] A. Porter Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, 2011.

[24] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS'11: Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[25] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. 20th Annual Network & Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[26] C. Smith. Privacy flaw in skype android app exposed. `http://www.t3.com/news/privacy-flaw-in-skype-android-app-exposed/`.

[27] R. Templeman, Z. Rahman, D. Crandall, and A. Kapadia. Placeraider: Virtual theft in physical spaces with smartphones. In *20th Annual Network & Distributed System Security Symposium (NDSS'13)*. Internet Society, 2013.

[28] C. von Eitzen. Cyanogenmod logged lockscreen swipe gestures. `http://www.h-online.com/open/news/item/CyanogenMod-logged-lockscreen-swipe-gestures-1734701.html`, 2012.

[29] N. Xu, F. Zhang, Y. Luo, W. Jia, D. Xuan, and J. Teng. Stealthy video capturer: A new video-based spyware in 3g smartphones. In *2nd ACM Conference on Wireless Network Security (WiSec '09)*. ACM, 2009.

[30] Z. Xu, K. Bai, and S. Zhu. Taplogger: inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Fifth ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC'12)*. ACM, 2012.

[31] W. Zhou, X. Zhang, and X. Jiang. Detecting passive content leaks and pollution in android applications. In *20th Annual Network & Distributed System Security Symposium (NDSS'13)*, 2013.

[32] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *4th international conference on Trust and trustworthy computing (TRUST'11)*. Springer-Verlag, 2011.