OASIS: Operational Access Sandboxes for Information Security

Mauro Conti *
Università di Padova
Padova, Italy
conti@math.unipd.it

Earlence Fernandes
University of Michigan
Ann Arbor, Michigan, USA
earlence@umich.edu

Justin Paupore
University of Michigan
Ann Arbor, Michigan, USA
jpaupore@umich.edu

Atul Prakash
University of Michigan
Ann Arbor, Michigan, USA
aprakash@umich.edu

Daniel Simionato
Università di Padova
Padova, Italy
daniel.simionato@gmail.com

ABSTRACT

Android's permission system follows an "all or nothing" approach when installing an application. The end user has no way to know how the permissions are actually used by the application, and how the sensitive data flows during its execution. With this work we present OASIS (Operational Access Sandboxes for Information Security), a trusted component that allows developers to execute operations on sensitive data while keeping that data confidential. OASIS allows the end user to have full control over the data available to applications, and also grants policy based regulation of sensitive data flows. Moreover, our system can be deployed via a simple application installation, and does not require any modification to the stock Android OS.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and protection

Keywords

Android permissions; operational access; trusted execution environment; sensitive data

1. INTRODUCTION

Android requires applications to specify a set of permissions in order to access sensors (e.g., GPS, camera), sensitive operations (e.g., SMS), or sensitive data (e.g., contacts, phone IMEI). When installing an application package (*app*), the package manager displays the required permissions to the user. To install the app, all of the requested permissions must be accepted by the user – there is no way, at least in the stock Android OS, to install an app while denying access to one of its required permissions. This permission system forces a draconian choice: to be able to use the app, the user must grant all of the permissions requested and, furthermore, give up any ability to monitor or control whether the granted permissions are used properly.

Let us consider, for example, a barcode scanner app. This app obviously needs the `CAMERA` permission to scan barcodes and QR codes. It also needs the `READ_CONTACTS` permission to generate a QR code from a contact, and the `INTERNET` permission to search information on a code. These are all legitimate uses, and thus permissions are required, but a hypothetical malicious barcode app could also misuse the granted permissions to send all the user contacts to a data-collecting server without user consent.

OASIS provides a framework for an alternative permissions system for Android, in the form of a trusted service that helps apps operate on sensitive data safely, in a manner specified by the developer, while allowing users to monitor or prevent the flow of sensitive information out of the device.

To the best of our knowledge, there is no solution capable of completely addressing this problem. Kirin [5] was the first effort to identify potentially dangerous combinations of Android permissions. However, it has no concept of "operations" in an application, and thus it cannot distinguish safe and unsafe operations within the same app. We will examine Kirin and other Android security solutions in Section 2.

In this paper we present OASIS, our solution that allows an app to completely isolate itself from the sensitive data it uses, while allowing operational access to that data. We will discuss our solution in Section 3. OASIS allows the app developer to define his own operations on private data, and provides a safe and isolated environment to execute those operations, while never disclosing actual sensitive data to the application. The results of the operations on sensitive data are provided to the app in the form of a *token*, without disclosing the sensitive data itself. Tokens can be used for subsequent operations within OASIS's safe execution environment. OASIS tracks the permissions used in generating each token at runtime. Users can define a policy that restricts dangerous combinations of permissions, without hindering safer combinations. For example, contact data could be restricted from being sent over the network by the app, while the app could still display contact data on the screen, or retrieve other information from the network.

OASIS is designed to work as a service on the stock Android OS, meaning it can be installed like a normal app. The

*The names of the authors are in alphabetical order.

SPSM'14, November 07, 2014, Scottsdale, AZ, USA.

978-1-4503-2955-5/14/11$15.00.
http://dx.doi.org/10.1145/2666620.2666629.

main advantage of OASIS is that it provides app developers with a safer mechanism to operate on sensitive data than standard Android. Additionally, users can also better track the flows of sensitive information from the app to the outside world. OASIS does not change or restrict the behavior of existing Android apps. We present the architecture and an implementation of OASIS in sections 3.3 and 3.4.

In short, OASIS allows:

- an app developer to execute code on sensitive data safely, with more assurance that the data remains protected, by preventing or detecting flows of sensitive data with harmful combinations of permissions.
- the user to monitor the flow of sensitive data to off-device destinations.
- the user or an app developer to define a policy that specifies which combinations of permissions can and can't be used.

2. RELATED WORK

In the state of the art, there are various approaches for protecting sensitive data. We will briefly discuss solutions focused on Android permissions, taint tracking, non-sensitive data, virtualization, app rewriting and OS hardening.

Android Permissions. Kirin [5] identifies dangerous applications at install time, by evaluating the permissions required and checking with a user-defined policy. Apex [10] defines a "permission extension framework" that allows the user to define policies regulating the usage of system permissions at runtime.

These solutions extend the Android permission system, but they do not distinguish between the different operations of an application. Instead, our proposal focuses on the interaction between the permissions in a data flow, which allows us to express a policy in terms of operations on sensitive data rather than permissions used.

Taint tracking. Taint tracking solutions monitor sources of sensitive data and use taints to track its usage in an app. This tracking can be done either in a static or dynamic way. Flowdroid [1] is a static analysis tool used to identify data leaks in Android applications. TaintDroid [4] is an efficient dynamic taint tracking tool able to track multiple data flows, by modifying Android's Dalvik VM to track explicit data flows. Appfence [8] is a solution built on top of TaintDroid that can shadow sensitive data and block exfiltration (i.e., unauthorized data leakage via network access).

The main limit of these solutions, as shown in [13], is that it is possible to use control flows (i.e., implicit flows) to remove the taint from sensitive data. OASIS does not have this problem since it does not rely on taint analysis.

Non-sensitive data. Another approach to securing sensitive data is to simply return non-sensitive information to untrusted apps. MockDroid [2] protects sensitive data by allowing the user to return fake data to all requests involving a specific permission. Similarly, TISSA [15] allows the user to choose at runtime which permissions are granted to an app, and which permissions should be "emulated" by returning fake or no data.

Neither of these solutions allow the user to set a policy for the behavior of app, nor consider the interaction between sensitive sources in a data flow. OASIS instead allows the user to set a policy that regulates these interactions.

RePriv [6] allows in-browser data collectors to access sensitive data via opaque handles, and allows operations on the handles via side-effect-free functions. While this is similar to our approach in OASIS, RePriv requires the data collectors to be written in a special language, while OASIS SODAs can be written in ordinary Java.

Virtualization. Virtualizing grants isolation, but requires computational power, and although smartphones are becoming more and more powerful, energy consumption severely hinders the adoption of resource-demanding solutions. While projects like L4Android [9] aim to bring paravirtualization to smartphones, MOSES [12] uses a different approach with "light virtualization". The main feature of this system is the usage of *security profiles* to achieve separated virtual environments for the application to run on.

Restricting the data available through a user-defined policy is a good approach, especially in the use case of a corporate environment with a BYOD (Bring Your Own Device) policy. However, even in MOSES there is no concept of interaction between permissions, and since MOSES uses the TaintDroid framework to taint the data, its controls can be avoided using control flows.

Application rewriting. Repackaging apps allows to insert the necessary security checks without modifying the OS. Aurasium [14] rewrites the code of an application, yielding a hardened version of the same app, in which every access to sensitive data is intercepted and checked against pre-defined policies. However, rewriting an application raises problems with copyright laws, and, in practice, any tampering with application packages is forbidden by the Google Play Store Developer agreement.[1]

With OASIS, we do no try to secure all apps; instead we offer a trusted environment where applications that use our system are guaranteed to comply to user policies and sensitive data is kept undisclosed.

OS hardening. Instead of building solutions on top of the OS, a more effective approach would be to modify the OS directly – we discuss a few examples in the following. Layercake [11] allows secure embedding of widgets inside Android applications (via Access Control Gadgets). The Android Security Modules framework (ASM) [7] offers a *programmable* interface to define reference monitors in Android.

This type of approach is surely promising, but since it requires the modification of the OS internals, without support from Google or a major Android OEM, it is unlikely to achieve a widespread adoption.

3. OUR SOLUTION: OASIS

In this section, we first provide an overview of OASIS (Section 3.1) together with related challenges (Section 3.2). Then, we discuss its architecture, feasibility, and further issues in sections 3.3, 3.4, and 3.5, respectively.

3.1 Overview

Our solution, OASIS, offers a trusted component that manages sensitive data, preventing apps from accessing the data directly, while allowing them to use the data in their code. One way to do this would be to use a simple delegation pattern, where the trusted service directly exposes the

[1]https://play.google.com/about/developer-distribution-agreement.html#prohibited

functionalities needed. However, this solution would imply having a public API on the trusted component able to satisfy all possible usages of private data in an Android application, which seems a requirement very difficult to satisfy.

Instead of designing a "good enough" API, OASIS allows app developers to define their own operations on private data, in the form of a *SODA* (Sensitive Operation Defined by the App) – a self-contained, stateless function that produces an output from specified inputs. Our system would then execute this SODA within a trusted and isolated environment, while returning the result of the computation as an *opaque handle* to the app. The app cannot access the results directly, since it only gets back a handle, but it can pass the handle to subsequent SODAs for further operations on sensitive data. Inputs to a SODA can be ordinary data or handles, and the result returned is arbitrary, but is always returned as a handle. The OASIS service keeps tracks of the real values associated with the handles, as well as the permissions used in building up the handle. Only the disclosure of the real data that is embodied in the handle is subject to policy, not the operations on the data itself.

This achieves four key goals:

1. access to private data is kept separate from the application itself;
2. the transformations over private data are defined by developer;
3. since access to private data is done in a safe and controlled environment (via a trusted service), we can precisely track the combination of permissions used by an app when private data is externalized.
4. permissions from the user are not needed for sensitive data accessed via OASIS, except when sensitive data is desired to be sent out of the trusted service, e.g., to the network, in unencrypted form.

The OASIS service offers functionality to declassify handles and output raw values to destinations such as the device's display or the network (sink); since it is the trusted component itself that does this declassification, and since a handle carries all the permissions used in its computation, OASIS can control and monitor these flows via policy. Such control and monitoring is simply not possible with the standard Android permissions system that only has knowledge about the data (and sinks) that will be available to the app.

3.2 Challenges

The design of OASIS presents some interesting challenges. One challenge is that it must work on top on the stock Android OS. We believe that the requirement to install a modified OS would negatively influence the adoption of OASIS, since only advanced users (with unlocked phones) would benefit from the system. We think that designing the solution on top of stock Android will facilitate a more widespread adoption. Another benefit of an unmodified OS is that it is compatible with other solutions, past and future, that address the same or related problems. By using Android APIs we can also better maintain OASIS as future releases of the OS occur. This does have the downside that OASIS is an opt-in service for apps; apps can still request normal Android permissions and bypass OASIS. Our hope is that OASIS-compliant apps (that only access sensitive data via OASIS) will be more trusted by end users.

The second challenge is that SODAs should be easy to implement by Android developers. A very desirable requirement is to be able to easily modify an existing app to use our service, without significant architectural changes. The goal is to have a system with clear functionality that does not interfere with applications that do not use it, but that gives the apps that wish to use OASIS an easy way to help secure the user's private data.

A third challenge is that the execution of a SODA must be isolated from normal app code, and the SODA must run in a monitored environment where it can not cause side effects. On the other hand, since we want to access private data in the SODA (under the trusted component's supervision), we must devise a way to let the SODA easily access sensitive information while preventing it from leaking the information to untrusted components.

Finally, we must track the permissions used by a SODA at run-time efficiently. Ideally, run-time taint-tracking should not be required for efficiency reasons. Analyzing the untrusted code beforehand would avoid the execution of the code if malicious or suspicious instructions were detected. However a static analysis precise enough might add a much greater overhead than monitoring the SODA execution itself. OASIS avoids the complexity of instruction-level analysis, instead treating SODAs as black-boxes and tracking permissions associated with each handle based on their inputs and outputs. Isolating the execution of SODAs in a trusted service also offers the advantage that the entire app does not have to be analyzed or taint-tracked.

3.3 Architecture

OASIS uses a client-server model, where the client is an Android app and the server is a service that exposes functionality to execute SODAs on sensitive data. An app requests the execution of a SODA via a blocking inter-process communication (IPC) call to the service. The app sends the SODA and any required additional parameters as arguments of the call to the service, and then waits for the result from the service. The OASIS service tracks usage of sensitive data during the SODA's execution, checking against a policy whether those uses are allowed or not.

The architecture of OASIS is illustrated in Figure 1. In particular, the service is composed of three main Android services:

- Sandbox service (number 1 in Figure 1), which realizes the isolated environment for the SODA execution;
- DataGateway service (number 2), used to access private data from the Sandbox;
- OasisService (number 3), the service which encapsulates the other two and is called by the application.

We use the DataGateway service as a way to keep the Sandbox isolated and prevent an app from accessing sensitive data directly. Apps need to use the DataGateway service to retrieve or output the sensitive information. Thus, at the DataGateway, we can monitor and track the sensitive data ("sources") accessed by a SODA as well as where this data flows to ("sinks").

OasisService receives the calls from an app (number 4 in Figure 1) to execute a SODA. It selects a Sandbox from a pool of identical sanitized sandboxes. The architecture uses a pool of sandboxes to scale better in case of multiple concurrent requests and for resiliency. The parameters given

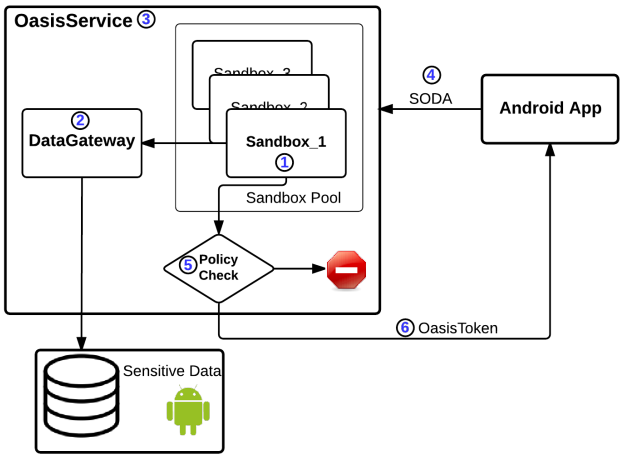

Figure 1: High view architecture of our solution.

to OasisService for the SODA call are passed by value into the Sandbox - this prevents SODAs from modifying parameters to leak data from the sandbox. When a SODA has finished executing, its result is then returned to OasisService, which checks (with a call to the DataGateway service) the sources and the sinks accessed by the SODA. OasisService stores the result and returns an opaque handle to the result back to the app. OasisService also calculates the taint of the SODA, starting it at the union of the taints on the inputs, and adding any inputs used directly by the SODA as it executes. The final taint of the SODA is also associated with the returned handle.

If the SODA accesses a sink (e.g., the network), policy is checked to see if the access should be logged or prevented (number 5 in Figure 1). For example, if the taint set when the network is accessed includes the camera, that implies that there is a potential flow from an image acquired by the camera to the network.

The results from the SODAs are returned in the form of an opaque handle, called OasisToken (number 6 in Figure 1). An OasisToken is composed of two fields: the encrypted result of a SODA execution, and a taint field that tracks the cumulative set of permissions used (i.e., sensitive sources used) to obtain the token. OasisService also adds a digest (generated with a cryptographic hash function) to the token, in order to verify the integrity of the token if used as input in subsequent computations. By encrypting information and returning it to the caller application, we can offload the data management to the application itself, which can use this OasisToken as input for other SODAs or simply store it. In the future, we will avoid the cost of encrypting computation results by keeping them in the OasisService (up to a maximum amount of time) and only returning a symbolic token to the application. Encrypted values will be available to applications only with an explicit request.

To prevent data leakage between SODAs with different taints, SODAs cannot store data between executions, except by returning that data at the end of the execution.

OasisService also offers functions to display information on the screen (e.g., via dialogs or toast notifications). Depending on whether one trusts Android to display sensitive information securely, these may or not be subject to policy. OasisService also includes operations to declassify information and send it through a sink (e.g., sending GPS coordinates to the network), subject to policy. These are needed to keep the app functional while being able to track or prevent risky flows of sensitive data.

Apps can get operational access to sensitive data without requesting access to the raw sensitive data. This potentially reduces the number of permissions that apps need to get approved from users.

To use OASIS, a developer needs to write a SODA for every operation he wants to do on sensitive data, except displaying operations. Inside the SODA, the developer can retrieve sensitive data with calls to the DataGateway service. The OasisToken resulting from a SODA execution can be displayed by calling OasisService. The end user would simply need to install the OASIS application (and optionally define a policy) to use the system.

3.4 Feasibility Evaluation

We implemented a preliminary prototype of the architecture to establish feasibility. Our solution is composed of a service package (OasisService) and a common library (OASISCommon). The common library is meant to be imported in the apps that want to use OASIS, and contains the type definition of OasisToken, a java interface for the SODAs (IOASISoda), and the AIDL (Android Interface Definition Language) interfaces of OasisService and DataGateway services.

The application developer that wants to use our service has to write a set of SODAs that will handle the operations on private data. Each SODA is defined as a separate Java class that implements the interface IOASISoda included in the common library. In Figure 2 we can see the main interactions in a sample execution. When a developer needs to execute a SODA, he needs to bind to OasisService and then call `OasisToken runSODA(String sClass, Map inputTokens, Bundle args)` (number 1 in Figure 2) – a method with a standard signature in the common library. The first argument is a String specifying the name of the class of the SODA, in order to allow OasisService to load the class from a different process. The last two arguments are used as parameters of the SODA to send in input data. The formal parameter `inputTokens` is an optional dictionary with OasisTokens as values and Strings as keys, in order to help the developer retrieve the right token during the execution. The parameter `args` is an optional Bundle that contains all other non-sensitive parameters that a SODA may need. The result of this invocation will be an OasisToken containing the encrypted information resulting from the SODA execution.

OASIS library wraps the runSODA method into the following method that contains two additional parameters, `authToken` and `dataGW`, whose values are supplied by the library and also also available to user SODAs:

```
String execSODA(Map<String, OasisToken> inputTkns,
Bundle args, IBinder authToken, IBinder dataGW).
```

The two parameters, `authToken` and `dataGW` are objects of type IBinder and created by the infrastructure for the developer: they are used to identify a SODA instance and the DataGateway service, respectively, since multiple SODAs could be executing in different sandboxes at any given time. We use an IBinder type for the last two parameters because the semantics of IBinder objects on Android ensure

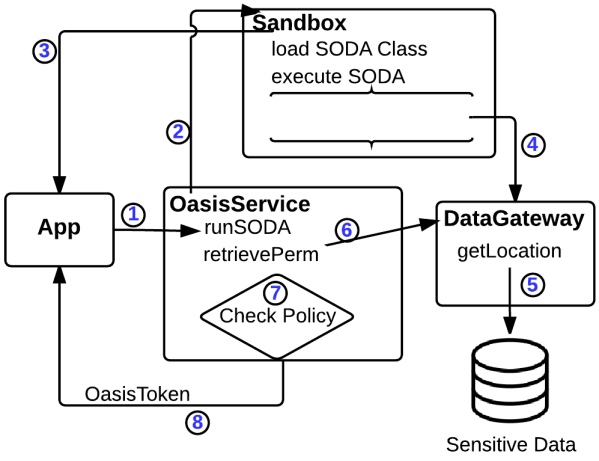

Figure 2: Sample execution of a SODA.

they will maintain a unique identity across all processes. Finally, the result expected from the execution is always a String. It represents the string value of an OasisToken that encapsulates the results of executing a SODA.

Sandboxing SODAs: When the service executes the `runSODA` method, it retrieves the calling application package name. The service then generates an `IBinder` object to be used to identify this SODA. Subsequently, it chooses a clean Sandbox from the pool and calls the Sandbox method `execSODA(String sourceAPK, String sClass, Map inputTokens, Bundle args, IBinder authToken, IBinder DataGW)` (number 2 in Figure 2). Inside this method, the Sandbox binds to the DataGateway service, after retrieving a proper binder from the IBinder it receives as argument. The Sandbox is implemented as an Android service with the property *isolatedProcess* set to true. Such a service runs in a separate process with no permissions of its own, and the only communication possible is through the service API. Since the only binder object that is passed to the Sandbox is the DataGateway binder, the only way to retrieve or send out sensitive information is by calling the DataGateway service. The isolation of the process prevents any unapproved I/O communication, even if the SODA is written poorly.

The Sandbox loads the class implementing a SODA from the caller's APK (number 3 in Figure 2). We then use Java reflection to instantiate a new object of the SODA class, and execute the method that runs the SODA. The result of this execution will be returned to OasisService as a String.

The DataGateway service is a local service that can only be called inside OasisService application. This service acts as interface for the SODA to gather sensitive data, e.g., a developer would call `datagw.getLastLocation(IBinder authToken)` to get the current location in a SODA (number 4 in Figure 2). The DataGateway will call the Android LocationManager (number 5) and retrieve the desired data, which will be returned to the SODA. The authToken is necessary to identify the SODA. Every call made by a SODA sets one or more permission flags in the DataGateway. This allows to track every data access during the SODA execution and apply security policy. The DataGateway service also offers a registration method to OasisService. During the registration, the DataGateway acquires a token that is used to authenticate to OasisService for subsequent calls.

At the end of the execution of a SODA, OasisService will make a call to the DataGateway method `int retrievePermissionsUsed(IBinder authToken)` to retrieve the permissions used by that SODA, using the previously generated IBinder as an identifier (number 6 in Figure 2). OasisService encrypts the result of the SODA execution and creates a new OasisToken. The retrieved permissions are added to the permissions required by all the OasisTokens used as input to the SODA, and result in the final set of permissions associated with the SODA and the returned OasisToken. This OasisToken object contains the encrypted result, the permissions used by the SODAs and a digest of the contents added by OasisService to verify the token integrity. The token is returned (number 8) to the calling app, where it can be stored, used as input of another SODA or be displayed by calling an OasisService output method.

OASIS uses notifications and dialogs as a way to display information on top of another application without disclosing sensitive data. The ability to display information on top of the calling apps without disclosing sensitive data will be added in the future, either by using Access Control Gadgets (like the ones described in [11]) or by drawing on top of the app window using available Android APIs.

Usage example. We use a barcode scanner like the one presented in the introduction app as an example. The app operations on sensitive data must be moved inside SODAs. These operations include using the camera to take a picture and computing the code from that picture (SODA 1), and looking up information on the computed value via the network (SODA 2). These SODAs are independent and can be executed at different times (e.g., if we already have an OasisToken from a previous execution of SODA 1, we will not need to execute the first SODA). Without loss of generality, we also note that the two SODAs can be combined into a single SODA. The first SODA to get a camera picture and convert to a barcode value has the following code outline:

```
BufferedImage codeImg =
  datagw.getCamPic(authTkn);
//Camera permission is added
//process the image
String code = processQRImg(codeImg);
return code; //uses Camera permission.
```

An OasisToken carrying the encrypted code will be returned to the app. The second SODA look up information (having `OasisToken codeTkn` as input and carrying a taint of Camera permission) has the following outline:

```
String code =
  datagw.decodeToken(codeTkn,authTkn);
if (isContact(code))
  //add as contact activity
  datagw.addContact(..,authTkn);
  //Contact permission added
else
  //search code on Google
  datagw.openURL(..,authTkn);
  //Internet permission added
...
return someResult;
//uses Camera+Contact or Camera+Internet
```

Whenever unencrypted sensitive data is externalized, OasisService checks if the permissions associated with the SODA so far comply with the externalization policy. Note that because we use the DataGateway to track the permissions used

by the SODA, only the permissions effectively used will be added as required by the SODA. This allows OASIS to selectively block only the operations (SODA executions) that do not comply with the policies.

3.5 Discussion and Future Work

The end user as well as legitimate app developers are the main beneficiaries of OASIS. The user can precisely control the flow of sensitive data in the application by defining his own policy. App developers can often request fewer permissions, since they get access to sensitive data via OASIS, but still operate on sensitive data. The permissions that they do request will be in clearer terms to the user (e.g., "use the Camera and send the data to the Internet"). For permissions that are granted, the OASIS framework can log when the permissions are used, the combination of permissions used, and the number of bytes sent out. This can help users of the app to develop information on real-world usage of permissions and better judge whether the app's use of permissions is reasonable.

The application developer, by using OASIS, can also gain the trust of end-users by delegating sensitive data manipulation to OasisService. We find an interesting use case in a corporate environment with a BYOD policy, where the system administrator can define a mandatory policy for its users, and only allow apps that use the OASIS framework exclusively for permissions. OASIS will ensure that all the applications using the service will comply with the policy, effectively creating a trusted environment for the apps.

Methods for expressing policy for externalizing sensitive data in OASIS still need to be defined and are not yet implemented. It could be similar to the current Android model, except that combinations of permissions can be expressed (e.g., allow network access, allow access to the contacts database, but do not allow contacts to go out on the network). These policies will need to be distinct from standard Android permissions since they are enforced by OASIS.

Since OASIS checks the policies at run time, it would be possible to change active policies in relation to the context to obtain a behavior similar to CRêPE [3].

We believe it is possible, at least to some degree, to automatically inspect application code and identify the operations on sensitive data that could be executed by OASIS in the form of a SODA. We plan to build a tool to help developers adapt their current application to use OASIS.

Finally, we note that even though our implementation of OASIS is Android-specific, the general architecture and design principles are applicable in other contexts where sensitive data is accessed.

4. CONCLUSIONS

In this work we presented OASIS, our solution to the problem of sensitive data leaking in Android. OASIS allows an app developer to run code that uses sensitive data in a trusted, monitored, and sandboxed environment. The apps that use our system do not have access to sensitive data, but can operate on it, and thus do not need any Android permissions in many cases (unless they access sensitive resources directly outside OASIS). Only externalization of sensitive data, not use, is subject to policy and monitoring.

5. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1318722. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Mauro Conti is supported by a EU Marie Curie Fellowship (grant No. PCIG11-GA-2012-321980), by the Italian MIUR PRIN Project Tenace (grant No. 20103P34XC), and by the Project "Tackling Mobile Malware with Innovative Machine Learning Techniques" funded by the University of Padua.

6. REFERENCES

[1] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings PLDI*. ACM, 2014.

[2] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Hotmobile*. ACM, 2011.

[3] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Crêpe: A system for enforcing fine-grained context-related policies on android. *TIFS*, 7(5):1426–1438, 2012.

[4] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*. USENIX, 2010.

[5] W. Enck, M. Ongtang, and P. McDaniel. Mitigating android software misuse before it happens. Technical report, 2008.

[6] M. Fredrikson and B. Livshits. Repriv: Re-imagining content personalization and in-browser privacy. In *Oakland*, pages 131–146. IEEE, 2011.

[7] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending Android security. Technical Report TUD-CS-2014-0063, CASED / TU Darmstadt, 2014.

[8] 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 *CCS*, 2011.

[9] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: a generic operating system framework for secure smartphones. In *SPSM*. ACM, 2011.

[10] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *CCS*. ACM, 2010.

[11] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security*, 2013.

[12] G. Russello, M. Conti, B. Crispo, and E. Fernandes. Moses: supporting operation modes on smartphones. In *SACMAT*. ACM, 2012.

[13] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, 2013.

[14] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, 2012.

[15] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on Android). In *TRUST*. Springer, 2011.