Flow Permissions for Android

Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen,
Steven Y. Ko, Lukasz Ziarek
University at Buffalo, The State University of New York, USA
{sholavan, donmanue, vishwasg, bjr24, fengshen, stevko, lziarek}@buffalo.edu

***Abstract*—This paper proposes Flow Permissions, an extension to the Android permission mechanism. Unlike the existing permission mechanism our permission mechanism contains semantic information based on information flows. Flow Permissions allow users to examine and grant explicit information flows within an application (e.g., a permission for reading the phone number and sending it over the network) as well as implicit information flows across multiple applications (e.g., a permission for reading the phone number and sending it to another application already installed on the user's phone). Our goal with Flow Permissions is to provide visibility into the holistic behavior of the applications installed on a user's phone. Our evaluation compares our approach to dynamic flow tracking techniques; our results with 600 popular applications and 1,200 malicious applications show that our approach is practical and effective in deriving Flow Permissions statically.**

I. Introduction

Android is a popular platform for mobile devices. Applications for Android are written mainly in Java and referred to as 'apps.' Unlike other mobile OSes, Android has a unique permission mechanism. At development time, an app writer needs to explicitly request permissions by including them in an app configuration file (`AndroidManifest.xml`). We refer to this configuration file simply as the 'manifest' in the remainder of the paper. During installation, each user needs to review the permissions that the app requests and explicitly grant them for the duration the app is installed.

Currently, there are over 130 permissions which Android apps can request in API level 17. Generally, an application can ask for permissions to use protected APIs for phone resources (e.g, storage, NFC, WiFi, etc.) or information available on the phone (e.g., contacts, location, call logs, etc.). For example, if an application wants to use APIs that control the camera, it needs to request the `android.permission.CAMERA` permission. For perspecuity we will use the shorthand `PERM` when referring to a specific permission of the form `android.permission.PERM`.

Although considered to be robust, the current Android permission mechanism has a number of deficiencies. The burden of deciding to grant permissions is placed on the user, but the permissions themselves provide little contextual information on how sensitive APIs are leveraged by the app. For example, it is unclear if an app with the permission to access the internet, as well as the phone's SIM card, exposes the private telephony data stored on the SIM card to the outside world. Apps can collude with one another to effectively gain permission they were not explicitly given (a danger that is compounded if apps are over privileged), through the many inter-process communication mechanisms Android provides.

To address these issues, we propose a new permission mechanism, called *Flow Permissions*, that extends the existing Android permission mechanism with information on information *flows* between permission domains (*e.g.* reading from the SIM card and sending over the network). We also introduce cross-app Flow Permissions that identify how apps can interact explicitly through IPC mechanisms, and *deployment permissions* – implicit Flow Permissions granted when installing an application based on indirect interactions possible between apps installed on a phone (*i.e.* a deployment) and a newly installed app. To help developers as well as users, we provide an automated tool, called Blue Seal, for synthesizing Flow Permissions. Blue Seal includes a lightweight cross app analysis that can analyze multiple apps to discover cross app flows or can be leveraged at installation time to detect implicit deployment permissions.

In this paper, we make the following contributions:

- **Flow Permissions:** A new permission mechanism based on information flows between permission domains within an app, as well as across multiple apps. Cross app flow detection can be leveraged at installation time to alert the user to implicit deployment permissions.
- **Blue Seal:** A tool, called Blue Seal, for automatically generating Flow Permissions, as well as a primer on how to modify classic program analyses to analyze Android specific constructs statically. Blue Seal generates Flow Permissions for an app statically in order to display the Flow Permissions before a user installs the app.
- **Case studies:** A large validation across 600 popular and 1,200 malicious apps.

The remainder of the paper is organized as follows: we first present a series of motivating examples showing the current problems with the Android permission mechanism in Section II. Our Flow Permission extension to the Android permission mechanism is detailed in Section III along with additional details on the Android platform that make inferring Flow Permissions difficult. Blue Seal is presented in Section IV and results in Section V. Related work and conclusions are given in Section VI and Section VII respectively.

II. Motivation

To motivate the necessity of extending the current Android permission mechanism, we examine four apps in detail: MyCalendar, MySpace, Blackmoon File Browser, and Gmail.

978-1-4799-0215-6/13/$31.00 © 2013 IEEE

ASE 2013, Palo Alto, USA
New Ideas Track

TABLE I
TABLE LISTING ANDROID APPS AND THEIR REQUESTED PERMISSION EXAMPLES.

Android App	Category	Permissions Requested
MyCalendar	Productivity	STORAGE LOCATION NETWORK PHONE CALLS
MySpace	Social	STORAGE NETWORK PHONE CALLS
Blackmoon File Browser	Productivity	STORAGE SYSTEM TOOLS
Gmail	Communication	NETWORK STORAGE

MyCalendar (com.kfactormedia.mycalendarmobile) is a third-party calendar app, MySpace (com.myspace.android) is a social networking app with multimedia support, Blackmoon File Browser (com.blackmoonit.android.FileBrowser) is a popular file manager, and Gmail (com.google.android.gm) is a well-known email app from Google. Although these apps have widely varying functionality, MyCalendar and MySpace request similar permissions.

A partial and stylized set of permissions each app requests is given in Table I. Notice that MyCalendar and MySpace both request PHONE CALLS and NETWORK. The PHONE CALLS permission grants the app a set of more fine grained permissions, which we omit for brevity, including permission to read the phone number, device ID, and the phone state. Similarly, the permission NETWORK allows the app to access the internet, either through wifi or cellular networking.

Savvy users may notice that by granting permissions to read from the phone's log and phone state as well as access to the internet, they are also implicitly granting permission to transmit data stored within the call log and phone state over the internet to an external source. Once the app has permission to read from a given piece of data stored on the phone (*i.e.* a data *source*) as well as permission to send data outside of the app (*i.e.* a data *sink*), the app also *implicitly* has permission to export the source data via the sink. Importantly, the permissions offer no insight if the apps leverage the APIs to ex-filtrate data.

A. Flows as Permissions

The goal of the Flow Permission mechanism is to show whether or not an app contains a *flow* between a source and a sink. The general structure of a Flow Permission is of: *source* $\rightarrow$ *sink*. From Table II, we can see that, even though MyCalendar and MySpace are granted the same permissions (PHONE CALLS and NETWORK), MyCalendar is augmented by our tool to contain the Flow Permission: PHONE NUMBER $\rightarrow$ NETWORK. This Flow Permission indicates that data read from the stored phone number was subsequently exported through the use of the network. Additionally, we can deduce that MySpace does not contain such a flow as it does not report such a permission. The MySpace app does, however, transmit

TABLE II
TABLE LISTING ANDROID APPS AND THEIR REQUESTED PERMISSIONS ALONG WITH OUR PROPOSED FLOW PERMISSION EXTENSIONS.

Android App	Flow Permissions
MyCalendar	PHONE NUMBER $\rightarrow$ NETWORK
MySpace	IMEI NUMBER $\rightarrow$ NETWORK
Blackmoon File Browser	STORAGE $\rightarrow$ NETWORK

the International Mobile Equipment Identity (IMEI) number of the device, which is indicated by the IMEI NUMBER $\rightarrow$ NETWORK Flow Permission.

In this manner, Flow Permissions provide the user additional context on how the standard Android permissions and the resources/data they protect are leveraged by the apps. Nevertheless, it is up to the user to decide if these behaviors should be allowed or not. The existence of a flow does *not* indicate that the app is necessarily malicious. For example, a social networking app might be expected to contain a flow from the IMEI number to the network as this provides the app a mechanism to uniquely identify the device for analytics. However, some users may not be comfortable providing such information to the app developer, as other mechanisms (*e.g.* manual login screens) can be used without exposing such data. In contrast, a calendar app should not have such a flow. We do note, that certain Flow Permissions should never be granted, namely exposure of the user's International Mobile Subscriber Identity [1] (IMSI) number from the SIM card.

B. Interaction Between Apps

Consider a more complicated case that highlights how multiple apps can expose data sources and sinks to one another, thereby acquiring additional implicit permissions [1]. The Blackmoon File Browser app includes functionality to send a file as an email attachment. However, the app cannot access the network to send an email as it does not have the NETWORK permission. Instead, the Blackmoon File Browser leverages Gmail's public interface to send files over the network. In other words, the Blackmoon File Browser is implicitly granted permission, if Gmail is also installed, to use the network without overtly requesting such a permission. Flow Permissions, on the other hand, highlight the flow between the Blackmoon File Browser and the network, accomplished through the RPC mechanism leveraged to transmit the file, as shown in Table II.

III. FLOW PERMISSIONS

Flow Permissions are an extension to the Android permission mechanism that characterizes the *implicit* interactions between data and APIs protected by standard permissions. This interaction is determined by the existence of an information *flow* between the permission *domains*. Although there may be multiple Android permissions dealing with a domain (*i.e.* READ_SMS, WRITE_SMS, RECEIVE_SMS, and SEND_SMS,

[1]This number is used to uniquely identify the user, phone, and subscription plan. Networks use this to establish roaming policies and charges associated with non local network usage.

etc.), we only consider the domains themselves (*e.g.* SMS). Domains are split into three categories: source domains, which can be viewed as sources of data; sink domains, which can be viewed as data export mechanisms; and cross-app domains, domains which act as inputs or outputs between apps.

A. Permission Domain Types

Out of over 130 Android permissions, we have identified thirteen canonical source domains: `NETWORK`, `EMAIL`, `IME`, `SMS`, `MIC`, `CALENDAR`, `ACCOUNTS`, `SDCARD`, `CONTACTS`, `CAMERA`, `CALL LOG`, `SIMCARD` [2], and `LOCATION`. Similarly, we have identified five canonical sink domains: `NETWORK`, `EMAIL`, `SMS`, `SDCARD`, and `LOG`. The third type of domain (cross-app) consists of Android's IPC mechanisms that allow apps to share data or provide services to one another. For example, Gmail exposes its email service to other apps via an IPC mechanism as mentioned in Section II-B. These IPC mechanisms can bridge a source domain in one app to a sink domain in another app.

B. Permission Mechanism

Flow Permissions can be viewed as relations between the three types of permission domains. There are four potential types of Flow Permissions:

- `Source` $\rightarrow$ `Sink`: A flow from a source domain to a sink domain.
- `IPC` $\rightarrow$ `Sink`: A flow from an IPC source domain to a sink domain.
- `Source` $\rightarrow$ `IPC`: A flow from a source domain an IPC sink domain.
- `IPC` $\rightarrow$ `IPC`: A flow from an IPC source domain to an IPC sink domain.

Of the four types of Flow Permissions, those which deal with flows to and from `IPC`, are not reported directly to the user by default [3]. Instead, these Flow Permissions, along with meta-data to disambiguate the IPC, are leveraged at installation time or during cross-app analysis to synthesize cross-app and deployment flows. Abstractly, cross-app and deployment flows are characterized by one app having a Flow Permission of the form: `Source` $\rightarrow$ `IPC`, another app having a Flow Permission of the form: `IPC` $\rightarrow$ `Sink`, and any number of apps having Flow Permissions of the form: `IPC` $\rightarrow$ `IPC`.

IV. System Design

Our system design is built on top of the Soot Java Optimization Framework [2], [3]. Since Soot is originally developed for analyzing Java bytecode, Soot integrated the Dexpler Dex to Java bytecode translator [4] to transform Dex bytecode into Soot's own intermediate represetation (Jimple). In addition, we leverage the PScout Permission Map [5]; abstractly, a permission map is a mapping between Android API calls and the permissions required to enact those calls. Our compiler leverages this precomputed mapping internally within the analyses to associate specific permission to API calls.

[2]We observe that the SIM Card stores two important numbers: IMSI and ICCID and our Flow Permissions distinguish these two cases.

[3]Our tool can be configured to emit these as well.

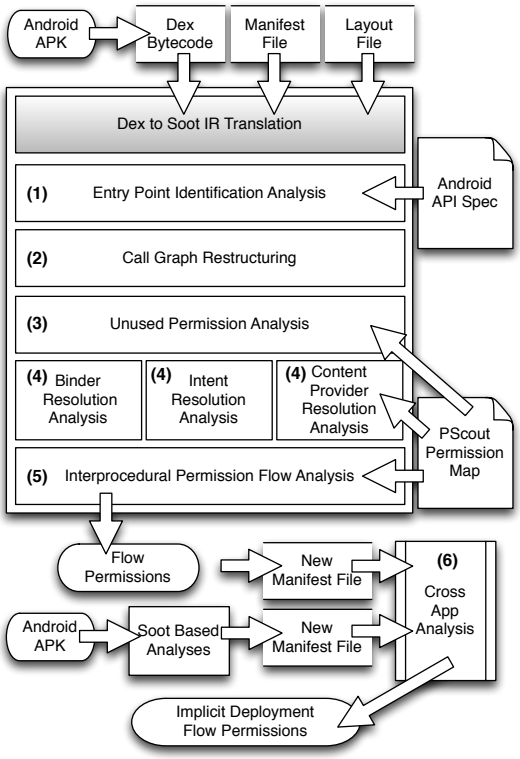

Fig. 1. The Blue Seal Android app analysis framework architecture. Shaded boxes represent components already present in Soot.

At its core, our Blue Seal leverages classic forward and backward intraprocedural dataflow analysis as well as interprocedural dataflow analysis based on graph reachability. As outlined in Fig. 1, Blue Seal leverages six main analysis passes to generate Flow Permissions: 1) entry point discovery, 2) call graph restructuring, 3) unused permission analysis, 4) resolution of intents, content providers, as well as uses of the binder, 5) interprocedural permission flow analysis, and 6) cross-app permission flow analysis. Abstractly, Blue Seal uses analyses 2, 3, and 4 to disambiguate Android specific constructs and identify source and sink points, prior to tracking flows between sources and sinks in analysis 5. Since Blue Seal is built from classic analysis techniques, we tailor our discussion on Android specific linguistic constructs, libraries, and IPC mechanisms and how to modify standard analyses to support them. Currently, Blue Seal is not path or context sensitive. Here, we select several interesting parts to discuss in detail. The complete sytem design description is availiable in our tech report [6] at http://blueseal.cse.buffalo.edu/techreport/techreport.pdf.

A. Entry Point Discovery

The Android platform is event driven and almost all apps have multiple entry points. Prior to static analysis, precise entry point detection should be performed to improve precision.

```
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        new Task().execute("http://www...");
        ...
    }
    ...
    private class Task extends AsyncTask<String, String, Integer> {
        ...
        protected void onPreExecute() {
            ...
        }
        protected Integer doInBackground(String... strs) {
            ...
            publishProgress("intermediate result");
            ...
            return intObj;
        }
        protected void onProgressUpdate(String...strings) {
            ...
        }
        protected void onPostExecute(Integer intObj) {
            ...
        }
    }
}
```

Fig. 2. A code snippet illustrating the methods that comprise the control flow of an async tasks in Android and the implicit flow of arguments provided by the Android framework.

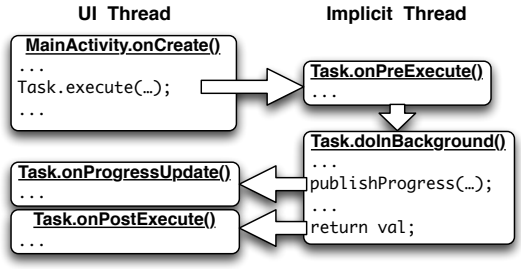

Fig. 3. The execution flow of async task methods in their respective threads at runtime.

B. Call Graph Restructuring

The Android framework is responsible for implicitly invoking methods associated with many of the constructs it provides. To correctly analyze an app, we must infer the association of user-called methods to their corresponding framework-invoked methods. We discuss `AsyncTask` as an example below.

`AsyncTask` is a new threading class introduced in Android. It provides a simple way to write a short lived thread that communicates with the UI thread in an asynchronous fashion. An `AsyncTask` can implement five methods—`onPreExecute`, `doInBackground`, `onProgressUpdate`, `onPostExecute`, and `onCancelled`, which dictate the control flow of the asynchronous task. As an example consider the code snippet in Fig. 2 and the corresponding control flow given in Fig. 3.

The `doInBackground` method performs the actual computation for the async task. The methods `onPreExecute` and `onPostExecute` run before and after `doInBackground` and typically include pre- and post-processing. The `onCancelled` method is called when the async task is cancelled by another thread. Notice that `onPreExecute` will execute in the implicitly created thread backing the asynchronous task, but `onPostExecute` callback will be executed by the UI thread. Similarly, `onProgressUpdate` gets executed as a callback in the UI thread after there is a call to `publishProgress` within `doInBackground`. An app writer can call `AsyncTask`'s `execute` and `executeOnExecutor` to start an `AsyncTask`. Obviously, a typical call graph generation process does not understand this execution flow; hence, we identify all `AsyncTask` instances and augment the call graph to include edges corresponding to the async task control flow. We do this by effectively replacing the invoke of `execute` with invoke calls to `onPreExecute`, `doInBackground`, and `onPostExecute`. Similarly, a call to `publishProgress` is replaced with a `onProgressUpdate` call. Notice that `doInBackground` implicitly passes its return value as an argument to `onPostExecute`. `publishProgress` also passes its arguments as arguments to `onProgressUpdate`. The call graph and method bodies are updated accordingly.

C. Interprocedural Permission Flow Analysis

To synthesize Flow Permissions we leverage an interprocedural forward flow analysis to track flows between sources and sinks. Our analysis is fixed point based, leveraging the standard work list model and method summaries. The flow analysis is parameterized by a listing of sources and sinks. The goal of this analysis is to track data flows originating at sources and terminating at sinks.

1) Computing and Applying Method Summaries: The intraprocedural forward flow analysis, leveraged by our interprocedural analysis, builds a method summary for each reachable method. The intraprocedural analysis is standard and builds in-flow and out-flow sets for each statement in the method body. The method summary constructed during this analysis is a flow graph representing the flows between sources and sinks within the method itself as well as arguments, returns, and class variables the method reads or writes. We add nodes to the graph for every argument, return statement, statement containing a class variable read/write, and statements identified as sources or sinks. Edges between nodes are added when a flow is determined by the intraprocedural forward flow analysis. Argument nodes and source nodes can have only outgoing edges. Sinks and return nodes can have only incoming edges. Nodes which represent class variable reads/writes can have both incoming and outgoing edges. Thus, there are four types of possible flows contained within the flow graph comprising the method summary: 1) generative flows: flows from a source to a return or class variable, 2) terminating flows: flows from an argument or class variables to a sink, 3) local flows: flows from a source to a sink, and 4) transitive flows: flows from arguments or class variables to other class variables or returns. Orphan nodes, nodes with no incoming or outgoing edges, are pruned.

At a method call site the analysis applies the summary for that method. If the method summary contains transitive flows, we add the arguments supplied at the call site to the out-flow set for the call. For both generative and terminating

flows, we add a place holder node into the method summary. This place holder node represents potentially multiple sources and/or sinks, one for each generative and terminating flow. The place holder nodes will be used to synthesize a global flow graph once all method summaries have been computed and the interprocedural analysis reaches a fixed point. Edges between nodes in the flow graph and the place holder node are added as if the place holder node was a source and/or sink node.

2) Synthesizing a Global Flow Graph: Once the interprocedural analysis reaches a fix point, we synthesize a global flow graph from the per method summaries. Place holder nodes that were inserted when method summaries were applied and class variable nodes serve as merge points for combining method summaries. Once all method summaries are merged, paths that do not originate from a source and terminate in a sink are pruned. Flow Permissions can be generated from the graph by enumerating all paths and removing duplicates (*e.g.* an app may send contact data over the network in multiple code blocks). Lastly, we remove any Flow Permissions that correspond to permissions the app does not request. This step is necessary because ad libraries [7] check to see which permissions an app has been granted and perform computation based on these permissions. Thus, an app may contain code that contains flows, but will never be executed at runtime. In general, any flow that requires a permission the app has not been granted cannot be executed at runtime. The Flow Permissions are then added to the manifest file for the app.

Special consideration must be given to apps that leverage Android's shared user ID mechanism. This mechanism allows for multiple apps to execute as a single process. This process is granted the union of the permissions requested by the apps. In this case, we do *not* remove any Flow Permissions, regardless if the app requests the necessary permissions or not.

D. Cross App Permission Flows Analysis

Cross-app permission flow analysis simply enumerates a permutations of pairs of Flow Permissions between apps such that one app has a source to IPC flow and another has a IPC flow to a sink, and the IPC mechanism is the same. In the case where an IPC is not disambiguated statically in either app, our analysis is conservative and assumes *any* IPC mechanism of the same type can be potentially utilized. For file reads/writes to external storage we track the file name(s) if they are deducible statically. Notice that when an app is installed on a phone, its Flow Permissions and associated meta-data (*e.g.* types and identifiers of IPCs) can be compared to the Flow Permissions of installed apps to synthesize implicit deployment Flow Permissions.

V. Results and Discussion

To test the validity of our approach, we tested Blue Seal on 600 of the top rated free apps available on the Google Play Store, as of January 2013, and on 1,200 know malicious apps identified by the MalGenomeProject [4] [8]. We ran Blue Seal on the Amazon EC2 [9] using their 8-core node instance with 7GB of ram.

[4]The full dataset is publicly available at http://www.malgenomeproject.org.

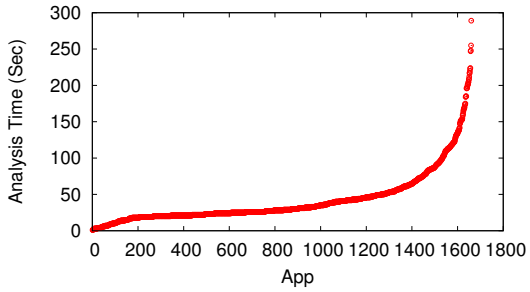

Fig. 4. Scatter plot showing the time taken to analyze all apps in seconds.

A. Performance

Blue Seal is able to analyze and synthesize Flow Permissions for all but the largest apps in under two minutes. Only 79 apps require an analysis time greater than two minutes. Full performance results are given in Fig. 4. However, Soot's front-end Dex bytecode parser, Dexpler, has limitations and generates incorrect intermediate representations for 199 of the apps. Of these apps, 169 are form the Google Play store and 30 are from the MalGenomeProject. We used an alternative tool, dex2jar [10], to translate these 199 apps into Java bytecode. Of the 199 apps, we were able to analyze 127 apps using the dex2jar translation in Blue Seal. We are currently investigating the causes of the mis-translation of the remaining 72 apps.

Cross-app analysis is executed over the Flow Permissions generated by Soot. Effectively, this analysis generates permutations of matches between Flow Permissions that share a common IPC. As the majority of apps have less than 10 Flow Permissions (8.34 Flow Permissions on average), we can calculate implicitly granted deployment permission on a phone that has 400 installed apps in less than 5 seconds. Although these experiments were executed in EC2, these preliminary numbers indicate that with optimizations this analysis is viable to be performed on an actual phone.

B. Limitations

Blue Seal performs static analysis to generate Flow Permissions and thus suffers from the classic limitations of this approach. Although we analyze all of the Android specific constructs, our current implementation does not yet identify all classic Java sources and sinks. We have focused primarily on file, network, and output stream APIs. Wherever possible, Blue Seal leverages the fact that most RPCs, CPs, and Intents are known and enumerated statically by unique integers or unique strings. In most cases Blue Seal is able to disambiguate the components, but in cases where it cannot, Blue Seal necessarily needs to be conservative, leading to potentially many false positives. Our only comparison points were the comparison of synthesized Flow Permissions to TaintDroid [11] and manual introspection of the apps. In the apps we tested, our Flow Permissions correctly identified the same flows as TaintDroid.

This process is unfortunately not guaranteed to generate a flow if one exists. As such, we do not yet have a good metric to quantify false positives.

VI. Related Work

The most closely related work to ours is CHEX [12], which provides a tool for detecting highjack enabling flows with an app. It is the first tool to tackle analysis of Android's constructs such as async tasks and handlers, though it uses a brute force permutation approach for disambiguation. Our call graph restructuring described in Section IV-B can refine CHEX's approach since we identify implicit calls in Android's constructs whenever possible. AndroidLeaks [13] is a static analysis tool implemented in WALA that can find leaks of sensitive information sent over the network from Android apps. It does not support analysis of async tasks, intents, nor content providers and is unable to track cross-app flows. SCanDroid [14] first proposed a methodology for analyzing intents statically, but was never tested on real-world apps. The approach also required the original Java source of the programs. Mann *et al.* created a framework to identify privacy leaks from the Android APIs [15], but the framework has not been evaluated on real-world applications. DroidChecker [16] is a static analysis tool aimed at discovering privilege escalation attacks and thus only analyzes exported interfaces and APIs that are classified as dangerous. ScanDal [17] is an abstract interpretation framework for tracking information flows within apps. Currently, their framework is able to track flows between location information, phone identifiers, camera, and microphone exported to the network and SMS.

Besides static analysis tools, there is a plethora of tools that perform dynamic analyses. TaintDroid [11] is one of the first tools that detects flows within apps that potentially leak private information. Alazab *et al.* [18] provides a dynamic analysis technique that runs apps in a sandbox and can detect malicious apps. MockDroid [19] is a tool that protects users' privacy by supplying mock data instead of sensitive data. Aurasium [20] provides user-level sandboxing and policy enforcement to dynamically monitor an app for security and privacy violations. Notably Aurasium does not require modifications to the underlying OS. We believe that Aurasium is complementary to Blue Seal, as our Flow Permissions can provide a specification of possible malicious leaks.

VII. Conclusion

In this paper, we present a flow based extension to the Android permission mechanisms, called Flow Permissions. We detailed a comprehensive primer on Android specific mechanisms and libraries in our description of Blue Seal, an automated infrastructure for synthesizing Flow Permissions. We provided a comprehensive evaluation of Flow Permissions in a wide variety of Android apps.

References

[1] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, 2011.

[2] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pp. 13–, IBM Press, 1999.

[3] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?," in *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, (London, UK, UK), pp. 18–34, Springer-Verlag, 2000.

[4] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, (New York, NY, USA), pp. 27–38, ACM, 2012.

[5] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[6] S. Holavanalli, D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S. Y. Ko, and L. Ziarek, "Flow permissions for android," Tech. Rep. 2013-04, Department of Computer Science and Engineering, University at Buffalo, The State University of New York, 2013.

[7] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, (New York, NY, USA), pp. 101–112, ACM, 2012.

[8] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (Oakland), 2012 IEEE Symposium on*, 2012.

[9] "Amazon ec2." http://aws.amazon.com/ec2/.

[10] "dex2jar." http://code.google.com/p/dex2jar/.

[11] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[12] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[13] M. S. Ware and C. J. Fox, "Securing java code: heuristics and an evaluation of static analysis tools," in *Proceedings of the 2008 workshop on Static analysis*, SAW '08, (New York, NY, USA), pp. 12–21, ACM, 2008.

[14] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications."

[15] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, (New York, NY, USA), pp. 1457–1462, ACM, 2012.

[16] P. P. Chan, L. C. Hui, and S. M. Yiu, "Droidchecker: analyzing android applications for capability leak," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, (New York, NY, USA), pp. 125–136, ACM, 2012.

[17] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies 2012* (H. Chen, L. Koved, and D. S. Wallach, eds.), (Los Alamitos, CA, USA), IEEE, May 2012.

[18] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian, "Analysis of malicious and benign android applications," in *Proceedings of the 2012 32nd International Conference on Distributed Computing Systems Workshops*, ICDCSW '12, (Washington, DC, USA), pp. 608–616, IEEE Computer Society, 2012.

[19] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, (New York, NY, USA), pp. 49–54, ACM, 2011.

[20] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, 2012.