On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users

Yanick Fratantonio[1](✉), Antonio Bianchi[1], William Robertson[2],
Manuel Egele[3], Christopher Kruegel[1], Engin Kirda[2], and Giovanni Vigna[1]

[1] University of California, Santa Barbara, USA
{yanick,antoniob,chris,vigna}@cs.ucsb.edu
[2] Northeastern University, Boston, USA
{wkr,ek}@ccs.neu.edu
[3] Boston University, Boston, USA
megele@bu.edu

Abstract. One of the main security mechanisms in Android is the permission system. Previous research has pointed out that this system is too *coarse-grained*. Hence, several mechanisms have been proposed to address this issue. However, to date, the impact of changes in the current permission system on both end users and software developers has not been studied, and no significant work has been done to determine whether adopting a *finer-grained* permission system would be feasible in practice.

In this work, we perform the first study to explore the practicality of the adoption of *finer-grained* system for the Internet permission. In particular, we have developed several analysis tools that we used to perform an empirical study on 1,227 real-world Android applications. The results of this study provide useful insights to answer the following three conceptual questions: (1) Is it practical to apply fine-grained access control mechanisms to real-world Android applications? (2) How can a system for fine-grained permission enforcement be integrated into the application development and distribution life-cycle with minimal additional required effort? (3) What are the incentives and practical benefits for both developers and end users to adopt a fine-grained permission model? Our preliminary results show that, in general, finer-grained permissions could be practical and desirable for Android applications. In addition, we show how the tools we have developed can be used to automatically generate and enforce security policies, and thus could be used to lower the burden of adoption of finer-grained permission systems.

1 Introduction

Smartphones and tablets have become an important part of our everyday lives. We use these devices to make phone calls, read emails, surf the web, make payments, and manage our schedules. As these devices have access to sensitive user information, they have become attractive targets for attackers, as is evident

M. Almgren et al. (Eds.): DIMVA 2015, LNCS 9148, pp. 282–303, 2015.
DOI: 10.1007/978-3-319-20550-2_15

from the continuous increase in the number and sophistication of the malware that targets these mobile devices [22,36].

The primary security mechanism on the Android platform is the permission system. The purpose of the permission system is to provide user-verifiable and OS-enforced constraints upon the runtime behavior of Android applications. All sensitive operations an application may perform (e.g., accessing the Internet, reading or sending text messages, using the NFC interface) are protected by specific permissions. An application must declare the set of permissions that it requires and these permissions must be approved by a user prior to its installation. At runtime, the Android OS acts as a reference monitor and ensures that the application cannot access resources that would require more permissions than those granted at installation time.

While the Android permission model is useful, it is affected by few weaknesses, one of the most important being that it is too *coarse-grained*. For example, Android's permission model for Internet access follows an all-or-nothing approach: Either an application can access the entire Internet (including any possible domain, IP, or port) or nothing at all. In many cases, however, Android applications only need to communicate with a much smaller set of endpoints. As an example, consider an online banking application that requires Internet access to connect to the bank server for checking the user's balance or making transactions. In this case, it would be sufficient for the application to be able to connect to a small set of domains that are under the control of the bank. Another example is constituted by applications that implement games or simple utilities (e.g., a flashlight): In this scenario, the applications would likely require internet access just to contact an online scoreboard, or for advertisement-related reasons. Hence, also in this scenario, they would need to access to only a small number of network endpoints.

In principle, a coarse-grained permission system does not support secure software development, as it prevents the developers from writing code that adheres to the principle of least privilege [28]: This makes an exploit against a vulnerable application more powerful, as it would have permission to access more resources. Moreover, a coarse-grained permission also makes malicious applications stealthier, as these applications would not need to reveal all the network endpoints at installation time. For these reasons, recent research works proposed a variety of approaches through which a finer-grained permission system can be implemented [9,10,18,21,27,31,33,34]. Even if all these works rely on technically-different solutions, they all share the same high-level goal: enabling developers and end users to specify and enforce fine-grained security policies.

While the technical aspects of finer-grained permission systems have been the focus of much research, the impact of such a modified permission system on users and developers has not been thoroughly studied: To the best of our knowledge, no significant work has been conducted to understand whether the adoption of a finer-grained permission model would be feasible in practice. Motivated by this observation, in this work we investigate on the security and engineering implications of finer-grained access control mechanisms and policies in Android.

To this end, we developed several analysis tools and we used them to perform an empirical study on how real-world applications make use of their permissions. In particular, we focus on the Internet permission, as it is the most widely used and studied permission, and it can be easily adapted to allow finer-grained permission specification.

Our study aims to shed light on the following three conceptual aspects:

- Is it practical to apply fine-grained access control mechanisms to real-world Android applications? (Sect. 4)
- Since specifying finer-grained permissions might be more laborious, how can a system for fine-grained permission enforcement be integrated into the application development and distribution life-cycle with minimal additional required effort? (Sect. 5)
- What are the incentives and practical benefits for both developers and end users to adopt a finer-grained permission model? (Sect. 6)

To perform this study, we first developed a *symbolic executor* (Sect. 3.1) to automatically analyze how applications access network resources. Our tool operates directly on Dalvik bytecode and determines, for each resource access (e.g., opening a network connection), the inputs that influence the resource identifier (e.g., domain names). Using the symbolic executor, we analyzed 1,227 Android applications to study how real-world applications access external resources. We find that a large fraction of the applications in our dataset use a limited set of resources, and that the identifiers of these resources are specified in the application code or in configuration files. This suggests that fine-grained permissions could be a practical mechanism, as application bundles contain the necessary information to extract tight security policies.

However, increasing the granularity of the permission system would require developers to invest more effort in permission selection and policy creation, as, currently, the developer is in charge of manually specifying the permissions needed by her application. Additionally, the task of manually generating a security policy is non-trivial, in the general case. For example, if an application contains a closed-source third-party component, it can be prohibitively difficult for a developer to manually specify an accurate, tight security policy. Thus, another aspect that we studied is if automatic tools (such as the one we developed) can be used to assist developers by automatically generating security policies. To this end, we developed a *policy extractor* component (Sect. 3.2), which processes the results from the symbolic executor into a set of fine-grained permissions required by the application. Our preliminary results are encouraging. In fact, we show that, for the applications in our dataset, the generated policies are small and accurate, and this suggests that the adoption of a finer-grained permission system might be practical for both developers and end users.

Finally, to lower the burden of adoption even more, we developed an *application rewriter* component (Sect. 3.3), which allows both developers and end users to rewrite an Android application (even when its source code is not available) to enforce a given security policy.

In summary, this paper makes the following contributions:

- We developed three different components (namely the symbolic executor, the policy extractor, and the application rewriter) to shed light on important security and engineering implications of finer-grained access controls for Android developers and users. Moreover, we make these tools available upon request. (Sect. 3)
- We performed an empirical study on how real-world Android applications make use of the Internet permission. We used our tool to statically analyze 1,227 applications, and, for 67.5% of them, it was possible to extract a non-trivial constraint (i.e., a constraint different from `.*`) for all their network accesses. We also performed several additional experiments that show how it is possible to automatically extract high-quality, initial security policies. In fact, we show that these policies are often precise (in 81.6% of the cases, no refinement is necessary), and small (for 87.8% of the applications, the respective policy is constituted by at most two domain names). (Sect. 4)
- We present how the three components we developed can be used in tandem to assist developers and end users, and to lower the burden of adoption of a finer-grained access control model. (Sect. 5)
- We provide a thorough discussion on the impact and the practical benefits that a finer-grained permission system would bring to all the actors in the Android ecosystem—developers and end users. (Sect. 6)

2 Overview

For this paper, we developed three different components, which we then used to gain insights related to the practicality of adopting a finer-grained permission system for the Internet permission. In this section, we provide an overview of these components and we describe their role within the context of this work. Figure 1 shows how the different components work together, and which insights are provided by each of them.

Symbolic Executor. The first component is a static analysis tool for Android applications that performs symbolic execution on Dalvik bytecode. This component computes, at each call site of an API method of interest (that we will refer

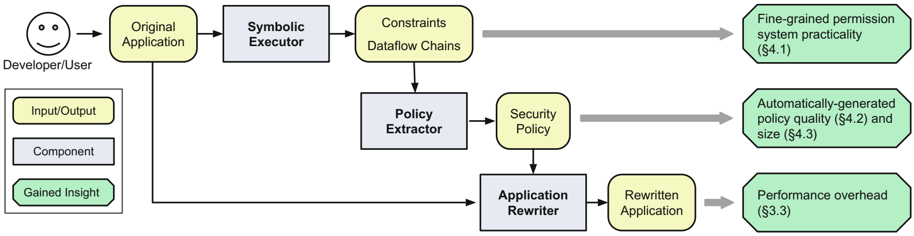

Fig. 1. Overview of the developed components, how they interact, and the insights they provide.

to as a *sink*), an over-approximation of the possible values for this method's arguments. In particular, we explore how Android applications access different sites on the Internet, and we study whether it is feasible to restrict their Internet access without breaking their functionality. We used this tool to perform an empirical study on how real-world applications use network resources. In particular, one of the goals of this experiment is to assess whether network endpoint identifiers are usually statically embedded in the application, or whether they are generated at runtime. This insight directly helps us understanding whether a fine-grained permission system would be practical for real-world deployment. In fact, if in most cases it would be possible to determine these resources only at runtime, it would not be feasible to meaningfully refine a given permission. The technical details of this component are discussed in Sect. 3.1.

Policy Extractor. The second component is a security policy extractor. Intuitively, this component takes as input the information extracted by the symbolic executor, and analyzes them to extract an initial security policy for a given Android application. For the Internet permission, a security policy consists of a set of possible network endpoints associated with their respective call sites. We then performed a series of analyses on these automatically-generated policies to first determine their quality (i.e., how well they "cover" the resources that are accessed at runtime) and their size (i.e., how many entries each policy contains). The technical details of this component are described in Sect. 3.2.

Application Rewriter. The third component we developed is a generic framework to rewrite Android applications. This tool modifies the application's Dalvik bytecode directly, and thus does not need access to source code of the application or any included third party libraries. This component is used to study the feasibility of automatically enforcing a security policy on an Android application by means of bytecode rewriting. In particular, this tool allowed us to understand whether it would be possible to retrofit off-the-shelf Android applications with finer-grained policies. Note how, ideally, the security policies should be enforced by the operating system itself, and hence this component would not be useful. However, in a practical sense, transitioning to finer-grained permissions might be performed incrementally, and hence our tool could be used to lower the burden of adoption. The details of this component are discussed in Sect. 3.3.

3 System Details

In this section, we discuss the technical implementation details of the main components we developed. First, we describe the implementation details of our symbolic executor, and the information it returns. Then, we present the details of the policy extractor component, which automatically extracts (initial) security policies. Finally, we discuss the application rewriter component, which is useful to enforce a security policy on a given Android application.

3.1 Symbolic Executor

Our symbolic execution engine takes an Android APK package as input, unpacks it, and extracts from it a number of artifacts. These include the DEX file with the application code, the application's resources (e.g., localized strings, image files, and UI layouts), and the manifest, which contains information such as the requested permissions as well as the entry points into the application (the main *Activity*, for example).

To disassemble application bytecode, we leverage `dexlib`, a library that is part of the open-source tool `apktool`. Then, the application's Dalvik bytecode is lifted to a custom intermediate representation, on top of which all the subsequent analysis passes operate. Our prototype is entirely written in the Scala language, and consists of a total of 10 K LOC. The next paragraphs describe the technical details of each of the main phase. Note that we do not consider the development of a symbolic execution engine for Dalvik bytecode as a research contribution. In fact, it is not novel (e.g., [20]), and, at least in principle, it would have been possible to re-use existing codebases. However, there are a number of aspects that are peculiar to our analysis. We describe these aspects in the remainder of this section.

Preliminary Steps. As the first step, the tool performs class hierarchy analysis to reconstruct the inheritance relations among the classes defined in the application and in the Android framework. Then, the tool computes the intra-procedural control flow graphs (CFG) of all methods comprising an application, and, finally, it reconstructs the application's super control flow graph (sCFG) by superimposing the inter-procedural call graph over the intra-procedural control flow graphs of each application method.

Forward Symbolic Execution. Our static analyzer performs forward symbolic execution as the basis for the discovery of constraints at privileged API invocations. This analysis step is performed directly on the Dalvik bytecode. Our symbolic execution engine models the semantics of all individual Dalvik virtual machine instructions over an abstract representation of program states, including the current program counter, abstract store, and local environment. Moreover, the invocation of the numerous entry points into the application that are exercised by the Android framework is modeled according to the rules of the various Android components' lifecycles.

Our tool performs a context-sensitive, inter-procedural data flow analysis. In particular, our analysis engine implements a generic framework for generating and merging program contexts. For this work, we opted to implement *2type+1H* type-sensitivity, an approach that has been shown to provide an excellent combination of precision and scalability, especially when analyzing object-oriented programming languages [30].

In addition to characterize which values can reach a specific API method, our static analyzer keeps track of the sources of values. That is, for each argument of an API method of interest, the analysis determines the source method (i.e., a method that reads values from other resources, such as files, network, intents) that "produced" the input value. This kind of information plays a key role when attempting to automatically generate precise security policies.

As an example, consider the snippet of code reported in Fig. 2. In this example, the method `foo` first uses the static string `url` to load a web page, and it stores the content of this web page in `res`. Then, the `extractUrl` method parses

```
public void foo() {
    String url = "http://my.example.com";
    Response res = loadFromUrl(url);
    String newUrl = extractUrl(res);
    Response newRes = loadFromUrl(newUrl);
}
```

Fig. 2. Example of a method that performs multiple "chained" network connections.

`res` to extract a new URL, which is stored in the `newUrl` variable. This new URL is subsequently used as a parameter to the second call to `loadFromUrl`, from which a second web page is downloaded. Intuitively, for the first network access, the application requires some initial information, taken from a string embedded in the program or read from a configuration file. After that, results read from the network are used to construct additional destinations (URLs, domains, etc.).

In this example, the static analysis will easily be able to infer the static value http://my.example.com as the input argument to the first Internet access. However, the analysis will not be able to determine the possible values for the second call to `loadFromUrl`. Nonetheless, the analysis will correctly identify that the result of the first call to `loadFromUrl` is the source that produces this input. In fact, even if the value of the second argument is statically unknown, it is ultimately derived from an access to the http://my.example.com URL. In other words, the information about the sources that are responsible for input values allows us to go back to the "root" of an access, thus providing useful information when characterizing even those sinks that cannot be statically constrained.

Symbolic Expressions. The result of the static analysis is a set of *symbolic expressions* that represent an over-approximation of all the possible concrete values that could be used by Android API sources (e.g., the path of a file read by the application) and sinks (e.g., the network domain names contacted by the application), as well as symbolic expressions that denote the origin and the destination of the corresponding values.

Symbolic expressions are essentially constraints (expressed as regular expressions) that represent sets of possible concrete values. One of the most critical objects to model with high fidelity are strings, as they are pervasively used to identify file system paths, network endpoints, or URLs. Our tool tracks not only concrete string values, but also symbolically models string expressions by recognizing string operations performed using well-known Java classes such as `StringBuffer` and `StringBuilder`. In case our analyzer cannot statically determine a constraint over the possible values that reach a given sink, then it conservatively returns the most generic constraint, i.e., `.*`.

Another object of key importance is the *Intent* object, which is extensively used by Android applications to perform inter-component communication, potentially across application boundaries. For example, the `startActivity` API method specifies an intent argument that contains information necessary for the system to determine which *Activity* should be launched in response to this service request. To correctly model this mechanism, the analysis takes into account how the intent resolution process operates. Moreover, it also models several other *complex* objects, such as the `Bundle` object, a key-value store used to pass auxiliary data along with intent-based service requests. Finally, our analysis models accessory objects such as the `Resource` object, which controls access to static values defined in XML-based resource files packaged with the application.

Android Framework Modeling. Since our tool analyzes applications in isolation from the Android framework codebase, proper models of important Android API methods are required. Mainly, these models allow the static analyzer to identify and track API methods corresponding to information *sources* and *sinks*, and to precisely associate origin and destination information. As a starting point to compile a list of API models, we consulted the resources provided by prior work [3,12,25], and we selected all those API that are relevant to our analysis. In total, our list is constituted by 174 API models.

Another key challenge is the proper modeling of implicit control flow transfers through the Android framework. Callbacks pertaining to the lifecycle of individual components (e.g., Activities) are well documented, and, therefore, it is possible to precisely model them through manual annotations (the authors of FlowDroid [2] followed a similar approach). For the remaining implicit control flow transfers, we used as a starting point the results obtained by EdgeMiner [7] – a system designed to extract a summary of all implicit control flows through the analysis of the Android framework itself.

3.2 Policy Extractor

The *policy extractor* generates an initial security policy based on the information generated by the symbolic execution engine. For the Internet permission, a security policy consists of a set of possible network endpoints, associated with their respective call sites. This section describes how this process works in detail.

The policy extractor first identifies the network endpoints that an application can contact, by analyzing, for all methods that connect to the Internet, the input parameters that specify the destinations of these connections. When the analysis finds that a parameter is one of a set of concrete string values, we extract the domain(s) from these strings, and add them to a set of permissible destinations. When the value is read from a file, we go back to the method that accesses the file, and we try to determine the file name. If a file name can be obtained statically, our component checks whether this file is bundled with the application (and statically exists on disk). If so, the file is parsed for domain names, which are then added to our set. When the destination value used in a network connection is read from the network (as in the example in Fig. 2), we go back to the method that performs the read and try to find the destination for that earlier read (`my.example.com` for the method `foo` in Fig. 2).

We then use this information to generate an initial security policy. In particular, the results from the symbolic execution engine allow us to generate a policy that not only indicates *which* resources are accessed (e.g., a set of network endpoints), but also *where* they are used (e.g., the call site of such sensitive methods). The precision of this information enables the specification of different permission sets for each component of the application. As we discuss in Sect. 6, this has several advantages with respect to the current permission system, which grants the same set of permissions to an application as a whole.

3.3 Application Rewriter

The last component we developed is a tool, called *application rewriter*, that modifies Android applications to enforce fine-grained security policies. The enforcing tool takes as input an APK file and a policy, and it returns as output a modified, self-contained APK where the given policy is enforced. In our context, a security policy specifies a constraint over argument values of a method *sink* at a given program location. Concretely, a policy is specified by a list of tuples in the format `<package_name>:<class_name>:<method_name>:<offset>:<param-idx>:<constraint>`, where the first four parameters uniquely identify a location within the application's code, `<param-idx>` defines the position of the argument for which the constraint is enforced, and `<constraint>` is the value (for example, a domain name) that should be allowed to be contacted by the method invocation at the specified location. Note that all parameters (except `<offset>` and `<param-idx>`) are regular expressions, and hence provide a high-degree of flexibility. Note also that our tool does not need to modify the permissions specified in the application's manifest. Hence, the permission set of the rewritten application is, by design, a strict subset of the original one.

The application rewriter is implemented on top of a generic, low-level API filtering engine. This engine works by disassembling a given application (by using `apktool`), and by modifying its bytecode so that a custom *enforcing method* is invoked just before every invoke instruction that could potentially reach a network sink. In particular, for APIs that accept values representing a URL, the enforcing method matches the actual argument value observed during runtime against any constraints that are specified for this call. If the observed value does not match the constraint, the enforcing method raises an exception, or, if possible, it modifies the values of the parameters that will be used by the subsequent call to the framework API. Moreover, since it is not always possible to precisely model the behavior of specific Android features (e.g., Java classloading, reflection, native code), we provide a choice to selectively disable these functionality at runtime, by placing enforcing code that throws proper Java exceptions when such features are used. We measured the overhead introduced by our instrumentation mechanism, and we determined it to be about 70 µs per API invocation, which is negligible.

We note that the technique we used is similar to the ones proposed in concurrently-developed works [4,9,10]. For this reason, in the interest of space, we omit the technical details related to this component, and we prioritize the description of the truly novel aspects of our work. We invite the interested reader to consult these already-published works for more technical details.

4 Practicality Evaluation

In this section, we describe the results we obtained by running our tool over more than a thousand Android applications. We first evaluate the correctness

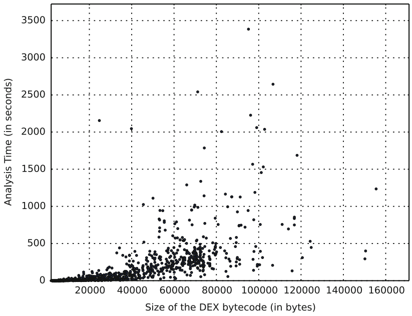

Fig. 3. Relation between DEX bytecode size (in bytes) and analysis time.

Table 1. Constraints for network-related API methods in the analyzed apps.

Category	Frequency
String	1,864
Prefix	209
Complex	620
Low	2,878
Total	5,571

of our approach and its usefulness in supporting the adoption of a finer-grained Internet permission system in Android. Then, we discuss the quality and the correctness of the static analysis results, we show that these results are useful as a starting point for automatically generating security policies, and, finally, we show how, in practice, these policies are often small, suggesting that the adoption of a finer-grained permission system would be practical for both developers and end users.

As a dataset, we selected a corpus of 1,983 applications from a previous random crawl of the official Google Play Store performed in 2012. As our study mainly focuses on the Internet permission, we only considered applications that required such permission. These applications belong to several different categories (e.g., games, entrainment, productivity, shopping, tools), and they are constituted, on average, by hundreds of methods. Their average APK size is about 830 KB, and their bytecode size varies from a few hundreds bytes to about 160 KB, representing small- and medium-sized applications on the market.

We analyzed all these applications with our symbolic execution engine. The experiment was performed on 15 virtual machines with 2 CPUs and 4 GB of memory each, and we set a one hour timeout for the analysis of each application. We successfully completed the analysis for 1,227 applications before the timeout was reached. The average static analysis time was 114.40 s. For these apps, Fig. 3 shows the relation between the size of the application and the analysis time required. As expected, analysis time and size are (loosely) correlated, although there are many outliers.

4.1 Results and Quality of Static Analysis

We first assess the quality of the symbolic expressions that our approach produces for the input arguments of network-related method calls. In the 1,227 apps of our dataset, we identified a total of 5,571 calls to sink methods. Table 1 shows a detailed breakdown of the results. The `String`, `Prefix`, `Complex`, and `Low` labels indicate the quality of the constraint associated to a given sink. In particular, String denotes a set of constant string values; Prefix corresponds to a regular expression that represents a prefix (e.g., a URL without parameters); Complex refers to more complex regular expressions; and Low indicates the most generic (hence less precise) constraint, i.e., .*. Our static analyzer was able to determine a non-trivial constraint for 48.3% of the sinks (2,693 out of 5,571). Moreover, many of the extracted constraints provide useful insights into the application's behavior. For example, our analysis is able to extract complex constraints such as "http://maps.google.com/?lat=\d+&lon=\d+".

As an additional experiment, we computed the number of sinks that are either constrained or whose *source* is meaningfully constrained (the *source* value is retrieved by following backward the data flow chains outputted by the symbolic execution engine, as described in Sect. 3.2). This condition is satisfied for 4,538 of the 5,571 Internet sinks. That is, in 81.4% of the cases, it was possible to characterize the *origin* of the value (either directly or indirectly) that reaches the security-sensitive API. This indicates that even if it is sometimes challenging to statically constrain a network sink, it is often possible to at least statically characterize the value of its *source* (as in the example provided in Fig. 2). This observation highlights an interesting pattern adopted by real-world applications: for the first network access, the application requires some initial information (taken from a string embedded in the program or read from a configuration file), and this information is then used as a starting point for determining the target of the other network sinks.

If we cluster these statistics in terms of number of applications (instead of number of sinks), we obtain similar results: our tool was able to completely constrain 483 applications (39.4%), while this number grows to 828 (67.5%) if the *source* information is taken into account. These results support the idea that it is often feasible to automatically refine the set of permissions requested by a given app, and that the automatic generation of policies is practical (as we will discuss in more detail in the next section).

Comparison with Dynamic Traces. To check the correctness of our static analysis, we first executed all the 1,227 applications in an instrumented emulator (we used the Google Monkey [15] to interact with the application under test, and we set a timeout of two minutes), and we extracted the dynamic traces, which consist in the list of methods that are invoked during the dynamic analysis, accompanied by detailed information about their arguments.

We first assessed the accuracy and the coverage of our static analysis tool by comparing the edges of the call graph generated by the static analysis with the call sequences contained in the dynamic traces. The rationale behind this experiment is that, since the static analysis is designed to produce an over-approximation of the actual program runs, we expect that all the code that is run dynamically was also analyzed statically. We found that ∼97 % of the edges in the dynamic call graph are correctly covered by the static analysis. The sole reason for the missing edges is that some Dalvik instructions can implicitly throw an exception (for example, the check-cast instruction throws an exception at runtime in case of a type mismatch) for which our symbolic execution engine currently lacks support. However, since the static code coverage is high (as discussed in the *Code Coverage* paragraph below) and only very few edges are missing, we believe that these minor imprecisions do not undermine the validity of our analysis.

In a second step, we evaluated the correctness of the constraints that are statically extracted. To this end, we compared the statically-extracted Internet constraints with the actual argument values recorded in the dynamic traces. In total, we observed 6,259 dynamic invocations of 1,311 distinct sinks that are related to the Internet permission. The static analysis produced expressions for 1,307 of these sinks (4 sinks are not reached due to missing exception support). In all 1,307 instances the constraints returned by the static analysis covered all observed values, as expected.

Code Coverage. We estimated the static code coverage by counting the number of methods that are reached by the static analyzer. On average, our approach has a static code coverage of ∼95 %. Similarly, we extracted the dynamic method coverage from the execution traces, and it resulted to be ∼25 %.

Ideally, the dynamic code coverage would have been higher. As a future step, we could use more powerful GUI exploration techniques [26,35] to increase the dynamic code coverage. However, we do not expect significant differences in the results, as state-of-the art dynamic exploration techniques achieve only slightly-higher coverage (e.g., 27 %–33 %). Moreover, to the best of our knowledge, this is the first static analysis work targeting the Android platform that extensively evaluates the static analysis results with dynamically-generated execution traces.

4.2 Quality of the Security Policies

In the next step, we wanted to determine whether the static values extracted by our approach are useful to automatically generate security policies. To this end, we used the output from the static analysis to automatically build (initial) security policies for all of them. For this experiment, the policies are automatically built by our policy extractor component (described in Sect. 3.2), and by considering the set of second-level DNS names for each statically-determined network endpoint. We then used the previously-collected dynamic traces to check what would have happened if we had used our rewriting tool to enforce the policies.

We found that for 1,002 (81.6%) of the applications, our security policies would allow all network accesses that these applications attempt during runtime. In other words, in all these cases, it was possible to statically infer the set of domain names that the application would contact during runtime. In 972 cases, the domain names could be extracted from information contained in the application code base. This indicates that most developers hard-code all domain names into their programs. In 30 cases, it was necessary to get the domains from a configuration file, but the name of this configuration file was specified in the binary. Note that there are a handful of cases in which the dynamic trace shows that a piece of JavaScript is dynamically evaluated by the application. We assume that these scripts do not contact any domain outside our policies, and a quick manual investigation confirmed this assumption.

Table 2. Number of explicit authorizations required when enforcing the automatically-generated security policies.

Number of required explicit authorizations	Frequency	Percentage
No interaction required	1,002	81.66 %
One explicit authorization	180	14.67 %
Two explicit authorizations	42	3.43 %
Three explicit authorizations	2	0.16 %
Four explicit authorizations	1	0.08 %
More than four explicit authorizations	0	0 %
Total	1,227	100 %

Note how this property holds not only for sinks contained in the main core application, but also for third-party libraries such as the popular AdMob advertisement library from Google. This might be surprising at a first glance, but we found that these Ad libraries actually only contact a few Google-related domain names (such as `admob.com` and `googleadservices.com`) to fetch their content. This makes sense because it allows Google to easily control the content that is displayed on millions of Android devices.

Policy Violations. We also investigated the 225 cases where our automatically-extracted policy was violated. We manually analyzed a random sample of 10 % of these applications (23 in total) to have a better understanding of the reasons. We found that in 7 cases, the domains are actually present in the bytecode, but the analysis is not precise enough to extract the proper values. In 12 additional cases, the domains are in configuration files, but again, our approach is not precise enough to extract the values (and related file names) automatically. Only in 4 cases do applications load destination domains from the network, and these domains are not present anywhere in the program or resources. This confirms that, in most cases, the developer seems to have a clear understanding of the Internet resources that should be accessed, and our approach is a useful system to automatically infer initial policies. In the 19 cases above, a developer could easily add the missing domains to the policy manually.

Alternatively, an application could be modified to be paused if a policy violation is detected. During the pause, the user would be prompted whether she wants to permit the violating access. Existing research has established that frequent authorization prompts lead to warning fatigue [6]. That is, users tend to ignore and blindly grant authorization requests if such requests are too frequent.

To evaluate how often the user would be prompted for authorization due to a policy violation, we analyzed, once again, the dynamic execution traces from Sect. 4.1. Table 2 reports the breakdown of the results: user interaction would be required only for 18.4% of the applications; for 96.3% of the applications, the user would be prompted at most once; and no application would require more than four distinct authorizations. Because the user only makes these authorization decisions once per application lifetime, we believe that the number of authorization prompts is well within reasonable bounds.

Table 3. Size of the automatically-extracted security policies.

Policy size	Frequency	Percentage
At most one domain name	956	77.91 %
At most two domain names	1,077	87.78 %
At most three domain names	1,151	93.81 %
At most four domain names	1,175	95,76 %
More than four domain names	52	4.24 %
Total	1,227	100 %

As a final consideration, we note that even if automatically enforcing the security policies extracted by our approach works in a surprisingly-high number of cases, such an approach does not always makes sense. Consider, for example, a web browser application: in this case, the application would need to be able to access arbitrary domain names specified by the user, which means that it would be impossible to extract all the network endpoints through static analysis only.

4.3 Size of the Security Policies

As a final step, we investigated whether the adoption of the proposed fine-grained permission system would be practical. The intuition is that most of the applications access a limited set of resources (hence this approach would make sense), but what if, in practice, these policies are prohibitively large and cause too much overhead for developers to maintain or end users to review?

To answer this question, we analyzed the policies extracted for the 1,227 applications. Table 3, which reports the breakdown of the results, shows how the majority of the applications (87.8%) access at most two domain names. Frequently, one domain name is usually linked to a domain controlled by the owner of the application, while the other one is related to an advertisement framework. A prolific source for additional domains are widgets from popular social networks. This observation becomes apparent when we look at the most frequently accessed domain names. In fact, the domain name that is accessed the most is related to the AdMob advertisement framework, while the other most frequently-accessed domain names are related to well-known social platforms, such as Facebook, Google, and Twitter.

4.4 Discussion and Limitations

In the previous sections, we discussed the results we obtained by using our tools to analyze over a thousand Android applications. We believe that our preliminary results are encouraging, as they suggest that it would be possible to adopt a finer-grained permission system. In fact, the size of the extracted policies is often small, and the domain names that are accessed the most are easily linkable to known companies. As a possible additional step to lower the burden of adoption even further, it might make sense to integrate our system with an approach based on whitelisting, so that *trusted* domain names (such as the ones related to well-known advertisement frameworks and popular social platforms) could be automatically approved.

That being said, we acknowledge that our study suffers from few limitations. For example, while we believe that our dataset contains a sizeable number of applications, the current app store features many more applications (according to [32], more than a million). Thus, it is unknown whether our results generalize to the entire market. Another limitation relates to the fact that, for this study, we did not consider large and complex applications (e.g., the Facebook app) due to the fact that our analysis, at its core, uses symbolic execution, which is known to be affected by scalability issues. Finally, a drastic change in how developers implement Android applications (e.g., tunneling all network traffic through a single network endpoint) could affect the applicability of the fine-grained policies extracted by our analysis. Nonetheless, we believe that our preliminary results already offer useful insights related to an important security aspect of the Android framework. Moreover, the analysis primitives we developed can be used as building blocks to study further security aspects of Android applications that extend beyond fine-grained permissions.

5 Viable Workflows

In this section, we describe how the various actors participating in the Android ecosystem, namely developers and end users, would use a finer-grained permission system, and how they could take advantage of the components we developed. Even if the two categories of actors would use the system for different reasons, they would follow the generic, high-level workflow depicted in Fig. 1: the actor would use our components to statically analyze her application, generate initial security policies, and enforce them.

A developer would take advantage of a fine-grained permission system by writing a security policy to be enforced. Note that writing such a policy might not be a trivial task if external libraries are added, especially if their source code is not available. This is because it can be problematic to understand the full set of resources (e.g., contacted URLs) accessed by these libraries. Moreover, a developer might decide to enforce fine-grained permissions on a third-party component if it is not fully trusted (not only it could be malicious, but, more likely, it could open the main application to security vulnerabilities). In the case that the task of manually compiling a security policy is too burdensome, the developer could take advantage of our symbolic executor and the policy extractor components to automatically extract an initial security policy. Then, the developer could review it and adapt it to her own needs, if possible. At this point, the developer could use our application rewriting tool to rewrite the application so that the security policy is enforced.

The other actor of the system, the user, can then download the rewritten app and review the policy before deciding whether to install the application.

For instance, a user might be comfortable installing an app that only requires the ability to contact a single domain name managed by the application's developer, but she might prefer not to install an application that requires unconstrained Internet permissions. Even in this case, the user could take advantage of the symbolic executor and the policy extractor components to automatically determine an initial security policy, which can be then modified and used as input to the enforcing component.

Note how, in both scenarios, the usage of the enforcing component is superfluous on a modified version of Android that enforces fine-grained security policies. However, on standalone Android versions, our enforcing tool will benefit the community and lower the burden of adoption of fine-grained policies.

6 Security Implications and Benefits

A fine-grained permission system in Android would introduce a variety of significant security improvements that affect all the actors of the Android ecosystem. This section describes these aspects in detail.

Implications for Benign Applications. Several studies showed that Android applications often suffer from confused deputy vulnerabilities [8,11,14,16,23]. While a finer-grained permission system does not eradicate confused deputy vulnerabilities, it significantly reduces the negative impacts of their exploitation. Consider, for example, an application accompanied by a fine-grained policy that identifies all legitimate network endpoints. Even if this application suffers from a confused deputy vulnerability, an attacker could exploit it only to communicate with the endpoints explicitly listed in the policy – a vast improvement over the current unrestricted system.

Another security benefit of a fine-grained permission system is that it allows the specification of a different permission set for each component. This feature is useful whenever the application contains third-party libraries that often require different permissions than the core application. Consider, as an explanatory example, a gaming application that requires the Internet permission for connecting to a game-related scoreboard and to include advertisement. This application could be modified so that the game-specific component would be able to access only the game-related network endpoint, while the advertisement library would be able to access only the advertisement-related one. Note that this use case is currently not supported by the Android permission model. Instead, the current Android permission system grants the same set of permissions to an application as a whole, and all components within the application consequently enjoy the same privileges, thus violating the principle of least privilege.

Implications for Malicious Applications. A widely deployed finer-permission system would place additional constraints on malicious applications that declare excessive permissions. In fact, a malicious application would need to either explicitly specify the domain name of the server under their control, or ask for *unconstrained* network access: both options would make malicious applications less stealthy, as they would need to *reveal* part of their behavior. The security policy declared by an application could also be used as an additional feature to improve the accuracy of existing malware detection tools (e.g., [1,11,17,38]).

Moreover, this explicit mapping between applications and their network endpoints would make information related to the domain names more useful: for example, if a domain name is found to be part of malicious activities, it would then be trivial to identify and flag all those applications that are somehow related to that domain name.

Implications for End Users. A finer-grained permission system would allow a security-conscious user to make more informed decisions about whether it is safe to install a given Android application. These users would be more aware of the risks associated with installing an application that requests *unconstrained* network access, as such permission should be required only by very specific types of applications (e.g., web browsers). Moreover, in case the application would only require access to well-known domain names, the user would be able to install and use the app with greater confidence. We believe that a finer-grained permission system would benefit non-security-conscious users as well. In fact, even if they might not be able to take informed decisions on whether an application is suspicious or not, they would still (indirectly) benefit from the advantages described above. One last aspect to be considered when extending the permission system is that such extension might lead to confusion and, therefore, misuse. However, we note that the new extension would enforce, by design, a set of permissions that is stricter than the original one. Thus, in this context, confusion and misusage would not have negative security repercussions.

As a final consideration, it is worth noting that many of these benefits can be enjoyed only when the finer-grained policies are enforced by the end user or by the system. In fact, in the alternative scenario where the developer herself is in charge of enforcing a given security policy, one would need to assume a trust relationship, which, in most cases, is not realistic.

7 Related Work

Recent research efforts have focused on the analysis and improvement of the Android permission system. For example, Barrera et al. [5] observed that some Android permissions are overly broad (included the Internet permission). Other works aim to understand how Android applications use the current permission system. For example, Felt et al. presented Stowaway [12], and by using it, they found that many Android applications were over-privileged. Based on this observation, several research works [4,9,10,34] developed tools to rewrite applications to enforce finer-grained security policies. Other works proposed Apex [24], AppFence [19], and SEAndroid [29], which provide additional privacy controls for Android through user-defined security policies, data shadowing, and exfiltration blocking. More recently, researchers proposed ASM [18], a framework that provides a programmable interface for defining new reference monitors, CopperDroid [31], which relies on syscall monitoring to enforce finer-grained policies, and DeepDroid [33], which aims to achieve the same goal by dynamic memory instrumentation. Our work is complementary to all these: in fact, the main goal of this work is to shed light on the security and practicality implications of finer-grained access control.

Several recent research works focus on discovering vulnerabilities within benign Android applications [16,23,37]. All these works discover a variety of serious vulnerabilities in real-world applications. These findings indicate that Android developers are often not aware of the many peculiarities of the Android framework, thus leading to severe vulnerabilities in real-world applications that, in most of the cases, lead to a confused deputy problem. As we discussed in Sect. 6, a finer-grained permission system would greatly minimize the threat posed by such confused deputy vulnerabilities.

Felt et al. [13] evaluate two different application permission systems: the Android permission system and the mechanism that is implemented for Chrome extensions. As opposed to the current Internet permission on Android, the permission system for Chrome allows the developer to narrowly define the resources (URLs specified as regular expressions) an extension can communicate with. An analysis of 714 popular Chrome extensions shows that for 60 % of the extensions, the developers explicitly list a narrow set of domains their extension can interact with. This study clearly shows that developers are willing to use a finer-grained permission system, if available.

A work that is close, in principle, to ours is by Jeon et al. [21]. In their work, the authors develop a simple analysis tool that aims to characterize how Android applications use the Internet permission: for their work, the authors developed a best-effort static analysis tool that first consults the application's string pool, it applies pattern matching on all the strings to extract those that *look like* URLs, and it then performs basic constant propagation. On the one hand, their overall goal is aligned with one of ours. In fact, as part of their experiments, the authors perform a study to characterize how many network resources are generally contacted by each application, with the goal of showing that this number is usually small and that, consequently, the adoption of finer-grained security policies is practical. On the other hand, the authors implemented an approach based on a best-effort, simple static analysis that is affected, by design, by false negatives. In particular, the authors specifically state that when their tool cannot statically constrain a network sink, such sink is ignored, instead of conservatively report a `.*` constraint, as our tool does.

This important limitation undermines the validity of their results, and leaves the "Is a finer-grained permission system practical?" question unanswered. In fact, consider, as an example, a browser application, which of course cannot be statically constrained (as the network endpoints to be contacted are chosen at runtime by the user). In this case, their tool would ignore all the unconstrained network sinks, and their results would suggest that it is possible to statically constrain the given application, which is incorrect. Instead, our approach would correctly return a `.*` constraint, the application under analysis would be correctly flagged as non-constrainable, and we would correctly conclude that a finer-grained permission system would not be practical in that case. In summary, while their results provide some interesting data, we believe they cannot be used to understand the practicality of a finer-grained permission system.

Moreover, the authors themselves point out that, to address the limitation of their work, one would need to perform inter-procedural analysis, modeling the heap, and modeling the Android Intent system. The static analysis tool implemented in our symbolic executor component precisely models all such aspects and it is designed to return an over-approximation of all the possible values that reach the *sink* methods.

8 Conclusion and Future Work

In this paper, we studied the security and engineering implications of a finer-grained permission system in Android. In particular, we focused on the Internet permission, and we developed several different analysis tools to shed light on the following three aspects: (1) Is it practical to adopt fine-grained access control mechanisms to real-world Android applications? (2) How can such a system be integrated into the application development and distribution life-cycle with minimal additional required effort? (3) What are the incentives and security benefits in adopting them?

Our preliminary results suggest that a finer-grained Internet permission would be practical. In fact, we found that applications in our sample typically require access to only a small set of resources, and these resources are explicitly referenced in the application's code or configuration files. Finally, our findings suggest that a finer-grained permission model for Android would entail a series of security benefits without overburdening developers or end users.

While our work mainly focuses on the Internet permission, it would be interesting to explore whether the adoption of a finer-grained permission system would be practical to protect other types of resources, such as file-system access, location information, or the user's contact list. We believe this constitutes a very interesting and important direction for future work.

Acknowledgements. We thank the anonymous reviewers and our shepherd Simin Nadjm-Tehrani for their valuable feedback. The work is supported by National Science Foundation (NSF) under grant CNS-1408632, and by Secure Business Austria. This work is also sponsored by DARPA under agreement number FA8750-12-2-0101. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, DARPA, or the U.S. Government.

References

1. Arp, D., Spreitzenbarth, M., Malte, H., Gascon, H., Rieck, K.: Drebin: effective and explainable detection of android malware in your pocket. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2014)
2. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2014)
3. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the android permission specification. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2012)
4. Backes, M., Gerling, S., Hammer, C., Maffei, M., von Styp-Rekowsky, P.: AppGuard – enforcing user requirements on android apps. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 543–548. Springer, Heidelberg (2013)
5. Barrera, D., Kayacik, H.G., Oorschot, P.V., Somayaji, A.: A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2010)
6. Böhme, R., Grossklags, J.: The security cost of cheap user interaction. In: Proceedings of the Workshop on New Security Paradigms Workshop (NSPW) (2011)
7. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: EdgeMiner: automatically detecting implicit control flow transitions through the android framework. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2015)
8. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys) (2011)
9. Davis, B., Chen, H.: RetroSkeleton: retrofitting android apps. In: Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys) (2013)
10. Davis, B., Sanders, B., Khodaverdian, A., Chen, H.: I-ARM-Droid: a rewriting framework for in-app reference monitors for android applications. In: IEEE Mobile Security Technologies (MoST) (2012)
11. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2009)
12. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2011)
13. Felt, A.P., Greenwood, K., Wagner, D.: The effectiveness of application permissions. In: Proceedings of the USENIX Conference on Web Application Development (WebApps) (2011)
14. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: attacks and defenses. In: Proceedings of the USENIX Security Symposium (USENIX Security) (2011)
15. Google: UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html

16. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock android smartphones. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2012)
17. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: RiskRanker: scalable and accurate zero-day android malware detection. In: Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys) (2012)
18. Heuser, S., Nadkarni, A., Enck, W., Sadeghi, A.R.: ASM: a programmable interface for extending android security. In: Proceedings of the USENIX Security Symposium (USENIX Security) (2014)
19. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2011)
20. Jeon, J., Micinski, K.K., Foster, J.S.: SymDroid: symbolic execution for dalvik bytecode. Technical report CS-TR-5022, University of Maryland, College Park (2012)
21. Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S., Millstein, T.: Dr. Android and Mr. Hide: fine-grained permissions in android applications. In: Proceedings of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2012)
22. Lookout: 2014 Mobile Threat Report ((2014)). https://www.lookout.com/resources/reports/mobile-threat-report
23. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS) (2012)
24. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the ACM Symposium on Information, Computer and Communication Security (AsiaCCS) (2010)
25. Rasthofer, S., Arzt, S., Bodden, E.: A machine-learning approach for classifying and categorizing android sources and sinks. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2014)
26. Rastogi, V., Chen, Y., Enck, W.: AppsPlayground: automatic security analysis of smartphone applications. In: Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY) (2013)
27. Russello, G., Jimenez, A.B., Naderi, H., van der Mark, W.: FireDroid: hardening security in almost-stock android. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) (2013)
28. Saltzer, J., Schroeder, M.: The protection of information in computer systems. Proc. IEEE **63**(9), 1278–1308 (1975)
29. Smalley, S., Craig, R.: Security enhanced (SE) android: bringing flexible MAC to android. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2013)
30. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (2011)
31. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of android malware behaviors. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2015)

32. Viennot, N., Garcia, E., Nieh, J.: A measurement study of Google play. In: Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) (2014)
33. Wang, X., Sun, K., Wang, Y., Jing, J.: DeepDroid: dynamically enforcing enterprise policy on android devices. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2015)
34. Xu, R., Saidi, H., Anderson, R.: Aurasium: Practical policy enforcement for android applications. In: Proceedings of the USENIX Security Symposium (USENIX Security) (2012)
35. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X.: SmartDroid: an automatic system for revealing UI-based trigger conditions in android applications. In: Proceedings of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM) (2012)
36. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Proceedings of IEEE Symposium on Security and Privacy (S&P) (2012)
37. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in android applications. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2013)
38. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS) (2012)