Certified Lightweight Contextual Policies for Android

Certified Lightweight Contextual Policies for Android

Mohamed Nassim Seghir
University of Edinburgh

David Aspinall
University of Edinburgh

Lenka Marekova
University of Edinburgh

***Abstract*—Security in Android applications is enforced with access control policies implemented via *permissions* giving access to different resources on the phone. These permissions are often too coarse and their attribution is based on an all-or-nothing decision on most of Android distributions. How can we grant permissions and be sure they will not be misused? We propose a policy-based lightweight approach for the verification and certification of Android applications with respect to a given policy. It consists of a verifier running on a conventional computer and a checker residing on an Android mobile device. The verifier applies static analysis to show the conformance between an application and a given policy. It also generates a certificate asserting the validity of the analysis result. The checker, on a mobile device, can then check the validity of the certificate to confirm of refute the fulfilment of the policy by the application before installing it. This scheme represents a potential future model for app stores where apps are equipped with policies and checkable evidence. We have implemented our approach, we report on the preliminary results obtained for a set of popular real-world applications.**

I. Introduction

Android's openness and ubiquity make it an ideal target for malware. Security in Android applications is enhanced with access control policies implemented via *permissions* giving access to different resources on the phone. But the permission model depends on the good judgment of the user, who needs to have some knowledge about the reasonable behavior of the application. For example, Brightest Flashlight Free [1] is an app which was downloaded 50 million times; its purpose is to turn on all the lights on a phone to their maximum level. However, it turned out that this app requested many inappropriate permissions, stealing the user's location and unique ID, and sending them to advertisers [1]. Most users would probably be unaware or surprised by this behaviour.

A straightforward solution to the previous case is to refuse granting permissions (refuse installation) to an app if its natural functionality does not match the requested permissions. But what if an app asks for permissions for some extra functional tasks which are not harmful? On the opposite side, if the required permissions match the logical functionality of the app, can we grant them and be sure they will not be misused? For example, an application for SMS management needs the permission SEND_SMS for sending, but should not use it to send out private data or contact premium rate numbers. Another example concerns a sound recording app. While the RECORD_AUDIO permission is a legitimate requirement for the natural functionality of the app, using it for recording without the user consent is an unwanted and a suspicious behavior.

[1]https://play.google.com/store/apps/details?id=goldenshorestechnologies.brightestflashlight.free

We propose *fine-grained* yet *lightweight* policies to prescribe the reasonable behaviour of applications. They refine the raw permissions model by making permissions bound to specific contexts, similar to the idea used in Pegasus [8]. For example, sound can only be recorded as a response to a user interaction, i.e., responding to a GUI event. We use static analysis to show the conformance between policies and application behaviour. A question that arises: can we trust the soundness (result) of the analysis? Moreover, how do we know that the analysis was indeed carried out? To address these questions, we propose a policy-based scheme, illustrated in Figure 1, which consists of the following ingredients:

- **Policy:** specify a set of rules to which the application must adhere. It can be provided by either a client of the application as a requirement or by the application provider as an advertisement to promote the safety/security features of its application.
- **Verifier:** static analysis that runs on the application provider side. It checks the conformance between the application and a policy, and generates a certificate.
- **Certificate:** audit for the accountability of the static analysis (verifier). It attests the correctness of the verifier outcome.
- **Checker:** static analysis that runs on the client (mobile device) side. It checks the validity of the certificate with respect to the application and the related policy. The checker is much lighter compared to the verifier.

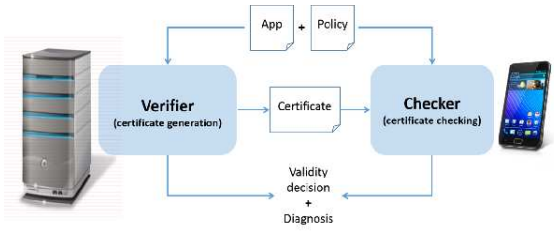

Fig. 1: Contract-based certification scheme

The certificate provides independently verifiable guarantees in concert with cryptographic signatures. It broadens the idea of *Proof-Carrying Code* by Necula [22] by encompassing lightweight forms of evidence specific to particular properties, e.g., program annotations tracking permissions or resource usage. It also goes beyond cryptographic signatures as it allows to certify properties inherent to the functionality of the application, such as the absence of information leakage or

```
1  public class Recorder extends Activity {
2    private MediaRecorder recorder = null;
3    ....
4    public void onCreate(....) {
5          ((Button) findViewById(Start))
6            .setOnClickListener(startClick);
7              ....
8          // startRecording();
9    }
10
11   private void startRecording() {
12         recorder = new MediaRecorder();
13         recorder.setAudioSource
14         (MediaRecorder.AudioSource.MIC);
15         recorder.setOutputFile(/* file name */);
16         ....
17         recorder.start();
18   }
19
20   private View.OnClickListener startClick
21           = new View.OnClickListener() {
22   public void onClick(View v) {
23         ....
24         startRecording();
25   }};
26   ....
27 }
```

Fig. 2: Code snippets and graphical interface of the Recorder app

bugs. The certificate can be independently checked to validate or refute the result of the analysis.

Checking the certificate is efficient compared to its generation. Hence, we are able to directly perform it on mobile devices which are relatively limited in terms of resources. We have extended our tool EviCheck [24] with this new operational scheme (check on device). We report on the results obtained for a set of real-world applications.

II. Example

In this section, we illustrate our approach via possible scenarios of permission misuse.

A. Actions without user consent

Consider the code snippets and the associated graphical interface in Figure 2, which represent the audio recording app Recorder. The access to the recording device is carried out via object recorder (line 2). At the creation phase (onCreate), a callback for a click event is associated with the button Start (line 5). Within the callback onClick, the method startRecording is invoked (line 24) which in turns calls recorder.setAudioSource and recorder.start to set the (on-device) microphone as a source and trigger the recording process. This app requires the permission RECORD_AUDIO which is associated with the API method setAudioSource of the MediaRecorder class. We might ask *how* and *when* is this permission used? In the normal case, the user would expect the recording to begin when the button Start is pushed. A possible malicious behaviour is to trigger the recording without the intervention nor the knowledge of the user. To rule out such a behaviour we provide a policy expressing that the RECORD_AUDIO permission will only be used in the context of the function onClick. An app can have multiple entry points. Hence, in terms of method invocations, we do not want to have a sequence of calls in which the API method associated with RECORD_AUDIO is reachable from an entry point of the app other than onClick. We express this via the following rule:

$$\text{ENTRY\_POINT}, \neg\text{CLICK\_HANDLER} \;:\; \neg\text{RECORD\_AUDIO}$$

The context variable ENTRY_POINT ranges over the set of entry points and CLICK_HANDLER ranges over click event handlers. The notation ¬CLICK_HANDLER means that click event handlers are discarded and ¬RECORD_AUDIO means that the permission for audio recording should not be used. So the rule says: "*in all entry points, apart from click event handlers, the permission* RECORD_AUDIO *must not be used*". This means, setAudioSource should only be reachable from a click event handler. This rule lacks some precision in describing the functionality of the app as the click event handler could be associated with the Start button as well as the Stop button. We can be more precise in our specification, if needed, by directly providing the method identifier instead of using context variables.

To check the validity of the previous rule, we use a simple reachability analysis which computes the transitive closure of the call graph with respect to permission usage. The result of the analysis is a map associating with each method the set of permissions corresponding to API methods which are potentially reachable from it. Starting with the initial map

$$\text{setAudioSource} \;:\; \text{RECORD\_AUDIO}$$

the analysis returns the new map

$$\begin{aligned} \text{setAudioSource} &: \text{RECORD\_AUDIO} \\ \underline{\text{onClick}} &: \text{RECORD\_AUDIO} \\ \underline{\text{onCreate}} &: \\ \text{startRecording} &: \text{RECORD\_AUDIO} \end{aligned}$$

Entry points are underlined. We can see that RECORD_AUDIO is only associated with onClick, thus our policy is valid. If we uncomment the line 8 (Figure 2), the policy is violated as RECORD_AUDIO will be reachable from the entry point onCreate as well.

Certificate. Now the question is how can a client of the analysis trust its claim? The analysis might contain errors or, even worse, an attacker can provide such a result without applying the analysis at all. For this, the computed map will serve as a certificate. To test its validity, we just need to check that for each pair of (caller, callee) methods, the set of permissions associated with the caller includes the ones associated with the callee. An auxiliary implicit condition is that all methods must have entries in the map. Let us try to tamper with the certificate generated for the previous example by omitting RECORD_AUDIO from the entry corresponding to onClick. This will be detected as RECORD_AUDIO is included in startRecording which is called by onClick, so it must be included in the caller as well. Let us have a more extreme scenario where we remove RECORD_AUDIO from all entries. This

```
1  public class uploader extends IntentService {
2    ....
3  protected void onHandleIntent(Intent intent){
4       ...
5       sendFile(/* file name */);
6    }
7
8   private void sendFile(string file_name){
9       // read content of file_name
10      // needs permission READ_EXTERNAL_STORAGE
11
12      // send content of file_name via a socket
13      // needs permission INTERNET
14   }
15 }
```

Fig. 3: Code illustrating file transfer from the phone to a remote destination in the background using an IntentService component

case also is detected as RECORD_AUDIO is associated with setAudioSource by definition (framework implementation), it represents the seeds for the analysis. How about suppressing all the entries? Our analysis will be aware of this case as the first step in the certificate integrity check is to make sure that all the methods used in the program have entries in the map. We would like to emphasize that the call graph is not part of the certificate. It is computed the same way by both the provider and the client of the analysis.

B. Stealing data

The previous scenario illustrates a behaviour which is undesirable, but remains harmless as long as the recorded data do not leave the phone. What if the recorded file is sent out to a remote server via the network connection? This can be done by sending the file just after the user terminates its recording. To make this stealthy, a non-blocking service running in the background is used as shown in Figure 3. The service is called uploader and the file transfer is performed via the method sendFile. We abstract away the implementation details of sendFile as this is not relevant to the presentation. We can specify a policy that rules out such a malicious behaviour as follows:

$$\text{SERVICE} \overset{or}{:} \neg\text{INTERNET} \quad \neg\text{READ\_EXTERNAL\_STORAGE}$$

The context variable SERVICE ranges over the set of methods which are members of service components. The rule says: "*in each method belonging to a service component, either the* INTERNET *or the* READ_EXTERNAL_STORAGE *permission can be used but not both*". The notation $\overset{or}{:}$ stipulates that the right side of the rule is a disjunction (by default, it is a conjunction). If the previous rule is violated, it does not necessarily mean that we have a data leak, but it serves as a lightweight alarm trigger which points out to parts of code to scrutinise more carefully. On the other hand, as the rule represents an over-conservative constraint, its validity rules out the undesirable scenario. We provide a more formal description of the rules semantics in the next section.

III. Policy and Digital Evidence

In this section, we describe the semantics of our policy language and provide an algorithm for checking the satisfiability of a given policy. We also show how to use the result as a certificate.

A. Policy language

Our policy language has the following grammar:

$$\begin{array}{rcl} R & := & H \ (: \mid \overset{or}{:}) \ T \\ H & := & mid \mid (CV|\neg CV)^+ \\ CV & := & \text{ENTRY\_POINT} \mid \text{ACTIVITY} \mid \text{SERVICE} \mid \text{RECEIVER} \\ & & \mid \text{ONCLICK\_HANDLER} \mid \text{ONTOUCH\_HANDLER} \mid LC \\ LC & := & \text{ONCREATE} \mid \text{ONSTART} \mid \text{ONRESUME} \mid \ldots \\ T & := & (\neg id)^* \end{array}$$

In the grammar, mid represents a method identifier which consists of the method name, its signature and the class it belongs to. Also we have CV for context variables, which can be ENTRY_POINT referring to all entry points of the app, ACTIVITY representing methods belonging to activities, SERVICE for methods belonging to service components, RECEIVER for methods belonging to receiver components, in addition to ONCLICK_HANDLER and ONTOUCH_HANDLER respectively referring to the click and touch event handlers. Moreover, CV can also be an activity life cycle callback such as ONCREATE, ONSTART, ONRESUME, etc. Activity callbacks as well as the touch and click event handlers are considered to be entry points. For a context variable CV, we write $S_A(CV)$ to denote the set of methods of the application A represented by CV, e.g., $S_A(\text{ENTRY\_POINT})$ is the set of all entry points of the application and $S_A(*)$ represents the set of all methods of the program. Finally, id simply represents an identifier or a tag such as a permission.

B. Semantics

A policy is given as a set of rules. It is satisfied if all the rules it contains are satisfied. In what follows we show when a rule is satisfied by an application. First, for a rule R we call H the head of the rule and T its tail. A rule can have either an or-semantics ($\overset{or}{:}$) or an and-semantics (:). We define the function $Interpret$ which gives an interpretation for the rule's head within an application; it simply returns a set of method identifiers. If H consists of just one method identifier mid, then $Interpret(H, A) = \{mid\}$. If H is a list of (negated) context variables $CV_1, \ldots, CV_m, \neg CV_{m+1}, \ldots, \neg CV_n$ then

$$Interpret(H, A) = (\bigcap_{i=1}^{m} S_A(CV_i)) \cap (\bigcap_{i=m+1}^{n} (S_A(*) \backslash S_A(CV_i)))$$

Example 1: Let us assume that H is of the form

$$\neg\text{ONTOUCH\_HANDLER ENTRY\_POINT ACTIVITY}$$

In this case $Interpret(H, A)$ represents the set of entry point methods belonging to activity components of the application A, which are not touch event handlers.

Given a rule R of the form $H : T$, we write $Id(T)$ to denote the set of identifiers appearing in the rule's tail T. The

satisfiability of a rule R by an application A is described as follows:

$$A \models R \text{ if } \forall x \in Interpret(H, A). \; Id(T) \cap reach_A(x) = \emptyset \quad (1)$$

The symbol $reach_A$ represents a map associating with each method of the application A a set of tags (E.g., permissions). A tag t belongs to $reach_A(m)$ if t is by definition associated with the method m or if m calls another method m' within the application A such that $t \in reach_A(m')$. The semantics of an or-rule R of the form $H \overset{or}{:} T$ is given by

$$A \models R \text{ if } \forall x \in Interpret(H, A). \; Id(T) \not\subset reach_A(x) \quad (2)$$

A policy P is a mixture of and- and or-rules.

C. Call Graph

The call graph is the key representation on which our analysis relies. It is therefore essential that the generated call graph is as complete as possible, i.e., any pair of (caller, callee) in real executions of the application is present in the call graph. Java and object oriented languages in general have many features, such as method overriding, which makes the construction of an exact call graph (statically) at compile time impossible. Therefore, we over-approximate it using the class hierarchy approach [25] which permits to conservatively estimate the run-time types of the receiver objects. For an object o having a declared type t, its estimated types will be t plus all the subclasses of t. If t is an interface then its estimated types are all classes implementing it or implementing its subinterfaces together with all their subclasses. We write $CG(A)$ to denote the function returning the call graph of an application A.

D. Policy verification

As mentioned previously, our verification technique generates a certificate as an audit for its outcome. This is implemented via Algorithm 1 which takes as input an application and a policy (set of rules) and returns a pair (Boolean, tag map) if the policy is satisfied. The returned map is a certificate for the validity of the analysis. If the policy is violated no certificate is returned. We have previously seen that rules interpretation with respect to an application A (formulae (1) and (2)) depends on the set of tags associated with the different methods in $reach_A$, hence our algorithm proceeds in two phases. First, the tag map $reach_A$ is computed via a simple working list procedure (lines 5-12). Tags are propagated backwards from callees to callers until a fixpoint is reached. In the second phase, we iterate over rules composing the current policy and check their validity (line 14) with respect to the application. This amounts to checking the (non) violation of the formulas (1) and (2) for and-rules and or-rules respectively. If no rule is violated, a map (certificate) accompanying the validity answer is returned (line 17), otherwise the verification process is terminated without providing a certificate (line 16).

E. Certification

To check the validity of the generated certificate (tag map) computed by Algorithm 1, we do not need to re-apply a reachability analysis. In fact, the checking process, which is implemented via Algorithm 2, is lighter than the generation one. It takes an app, a tag map and a policy as parameters and returns true if the certificate is valid and the policy is satisfied or false otherwise. First, we check that all methods belonging to the platform API are present in the certificate together with their predefined tags (lines 1-5). In the next step, it suffices to go through the different methods and locally check if their associated set of tags is equal to the union of all the sets of tags associated with the functions they call (line 6-10). As illustrated by the tests at lines 3 and 8, it suffices to find one inconsistency to invalidate the certificate. If no inconsistency is found then the final step consists of assigning the certificate to $reach_A$ (line 11) and then checking the satisfiability of the policy by the application (lines 12-16), similar to Algorithm 1.

Algorithm 1: VerifyAppForPolicy

```
   Input: application A, poliy P
   Output: (Boolean, tag map)
1  Var list L, set S;
2  Let M be the permission map for API functions;
3  Let reach_A(f) = M(f) if f ∈ API and reach_A(f) = ∅
   otherwise;
4  L := {all functions used in app};
5  while L ≠ ∅ do
6      pick up a function f from L;
7      S := reach_A(f);
8      foreach f' s.t. (f, f') ∈ CG(A) do
9          reach_A(f) := reach_A(f) ⋃ reach_A(f');
10     if S ≠ reach_A(f) then
11         foreach f'' s.t. (f'', f) ∈ CG(A) do
12             add f'' to L;
13 foreach rule r in P do
14     if A ⊭ r then
15         print "policy violated";
16         return (false,-);
17 return (true, reach_A);
```

The procedure CheckCertificate has a linear complexity in the number of methods of the program. It also has a constant space complexity as we are just performing checks without generating any information which needs to be stored. Moreover, we do not require the complete call graph to be present in memory. As we are performing a single (linear) pass, we can get rid of the current entry as soon as we move to the next one.

As the call graph is not part of the certificate, it is computed the same way via function CG in both the verifier (Algorithm 1) and the checker ((Algorithm 2).

F. Discussion

As mentioned previously, generating the call graph by itself is not a trivial task due virtual method dynamic resolution. Reflection is also a known issue for static analysis. A simple and conservative solution for this problem is to associate a tag t_{ref} with methods of the class java/lang/reflect/Method. We then use the tag t_{ref} to make the policy reflection-aware, e.g.,

Algorithm 2: CheckCertificate

Input: application A, policy P, map M
Output: Boolean

Let M_0 be the permission map for API functions;
foreach $f \in API$ **do**
  if $M[f] \neq M_0[f]$ **then**
    print "certificate invalid";
    return false;
foreach $(f, -) \in CG(A)$ **do**
  Let $S = \bigcup \{M[f'] \mid (f, f') \in CG(A)\}$;
  if $M[f] \neq S$ **then**
    print "certificate invalid";
    return false;
$reach_A := M$;
foreach *rule* r *in* P **do**
  if $A \not\models r$ **then**
    print "policy violated";
    return (false);
return (true);

$c : \neg t_{ref}$ to express that reflection should not be used in the context c. A similar solution can be adopted for dynamic code loading by associating a tag t_{dyn} with methods of the class dalvik/system/DexClassLoader.

Another key point is related to the nature of our analysis which is a *may-analysis*. It can show that an application may use a given permission but cannot show that the permission is actually used. This makes it more appropriate for disproving permission usage rather than proving it and explains the occurrence of identifiers in negated form in our policy language.

Finally, a question that needs to be addressed is: who provides policies? Although our tool gives the user the possibility of specifying policies, we do not expect an average user to do it by himself. Security experts could prescribe a bunch of policies based on application categories. What can we do in the absence of expertise? We are currently working on a data-driven automatic approach for policy generation. The preliminary prototype already provides encouraging results.

IV. Implementation and Experiments

a) Implementation: We have implemented the checker, which runs on mobile devices, as part of our tool EviCheck [24][2]. EviCheck accepts apps directly in bytecode (APK) format and uses Androguard [12] as back-end for parsing them. As EviCheck is written in Python, we use kivy[3] to facilitate the deployment of the checker module on Android mobile devices. The verifier module takes an app together with a policy as input, and answers whether the policy is satisfied by the app, and eventually outputs a certificate. The checker takes as input an app, a certificate and a policy, and answers whether the certificate is valid. Both the verifier and checker return diagnostic information pointing to the first violated rule in case of policy violation or to the first inconsistent map entry when checking the certificate. They also generate chain of method calls as witness.

b) Experiments: We have performed experiments on 13 real-world popular applications, from the Google Play store[4], ranging over different domains: banking, multimedia, games, social, etc. We use a typical Linux desktop to host the verifier and a Motorola G3 mobile phone (Qualcom Snapdragon 1.4GHz processor) running Android to host the checker. In our study, we have specified a policy consisting of 6 rules which can potentially match undesirable behaviour. For example, reading contacts and using Internet in the background, which might indicate that private contacts are sent over the Internet. First, we call the verifier to verify the validity of the policy and to generate a certificate. In a second step, the checker is invoked to check the generated certificate. The results are illustrated in Table I. Column #M shows the number of methods per application as an indicator of the application size. Columns V(d) and C(d) respectively represent the verification and checking times on desktop computer. Column C(m) contains the checking times on mobile device. We have included checking times on desktop on purpose to illustrate how checking is more efficient than verification on a similar architecture. This motivated us to carry out the checking directly on mobile device. While the performance of the checker on mobile is not as good as on desktop, it still runs in less than 10 minutes and for one case in less than one minute. This is encouraging given the size of the considered applications and the limitations of mobile devices. To give an idea about the complexity of these apps for static analysis tools, Flowdroid [2] is unable to analyze the *Hsbc* app within a bound of 30 minutes on a desktop computer.

The remaining part of the table concerns the rules forming the policy. The presence of the symbol ✗ indicates that the concerned rule is violated. A description of each rule is given in Figure 4. Policy violation does not necessarily mean malicious behaviour, but it can serve as an alarm to trigger more careful scrutinizing. For example, rule 6 is violated by the *Hsbc* app. This was surprising, knowing that it is a banking app. Why would it use the camera at all? Our investigation revealed that this app offers a mobile check deposit service [5] which uses the camera to take a picture of the check. Further to this, we wanted to know if the app is taking pictures without user consent as rule 6 indicates that the camera is used in a method which is not a click handler. By analysing the bytecode of the application, we found out that there are camera-related methods which are reachable from an onResume callback of an activity. However, they are used for configuration purposes.

1. ENTRY_POINT SERVICE $:^{OR}$ ¬ACCESS_FINE_LOCATION ¬SEND_SMS
2. ENTRY_POINT SERVICE $:^{OR}$ ¬ACCESS_FINE_LOCATION ¬INTERNET
3. ENTRY_POINT SERVICE $:^{OR}$ ¬READ_CONTACTS ¬SEND_SMS
4. ENTRY_POINT SERVICE $:^{OR}$ ¬READ_CONTACTS ¬INTERNET
5. ACTIVITY ENTRY_POINT ¬ONCLICK_HANDLER : ¬RECORD_AUDIO
6. ACTIVITY ENTRY_POINT ¬ONCLICK_HANDLER : ¬CAMERA

Fig. 4: Rules composing the policy used in the study

[2]http://groups.inf.ed.ac.uk/security/appguarden/tools/EviCheck
[3]https://kivy.org
[4]https://play.google.com/store/apps
[5]https://www.us.hsbc.com/1/2/home/personal-banking/pib/mobile/mobile-deposit

App	#M	V(d)	C(d)	C(m)	Policy rules					
					1	2	3	4	5	6
Angrybirds	48324	123.1	50.11	436.27						
CandyCrushSaga	23877	25.38	19.12	182.51						
Facebook	7969	12.16	11.36	67.56						
FacebookMessenger	4201	7.02	6.78	34.34						
FirefoxBrowser	28442	50.52	30.07	287.07						
Hsbc	18365	18.32	13.83	122.56						✗
Instagram	39062	94.55	50.63	427.32						
LinkedIn	50743	191.97	56.88	550.89						
OperaBrowser	28137	34.68	23.61	218.71						
Skyscanner	44374	121.78	55.57	449.75						
Twitter	45700	151.03	47.95	462.65						
Uber	48600	113.84	46.15	426.05						
Viber	50876	153.6	53.69	479.64						

TABLE I: Results of checking a policy composed of 6 rules against 13 popular apps from the Google Play store. The symbol ✗ indicates that a rule (policy) is violated.

V. Related Work

Recently, many tools for analysing different security aspects of Android have emerged. Some of them rely on dynamic analysis [5], [14], [23], [26], [27]. Others are based on static analysis [2], [4], [9], [16], [18]. The last family of tools perform an exhaustive exploration of the application behaviour. This is made possible thanks to abstraction (over-approximation) which also leads to some imprecision. We are interested in this category (static analysis) of tools as our aim is to certify the absence of bad behaviours. Our work is a complement to these tools. In addition to analysing applications, we also return a verifiable evidence attesting the validity of the analysis. The tool Kirin [15] uses lightweight rules which conservatively match undesirable behaviour. Their policy language can refer to permissions but does not refer to their usage context. Our language does not have this limitation. Moreover, our analysis is more faithful to the application behaviour by operating on its code as opposed to Kirin's analysis which is restricted to the manifest file. Chen et al. use temporal logic and model checking to specify and verify API and permission sequences in a given context [8]. While their approach can capture more interesting properties than ours, it does not generate a certificate for the result. A combination of their approach with ours is an interesting track for investigation.

Jeon et al. proposed an approach for inferring and enforcing fine-grained permissions for Android by making them bound to a set of arguments [17]. Our policy language also defines a kind of fine-grained permissions, but they are bound to the usage context. Enforcing fine-grained access control policies was also investigated in the context of runtime monitoring [7], [10], [13], [20] where the policy is checked at runtime. In our case, the policy is statically checked, and we do not have further checks when the application is executed.

The idea of associating proofs with code was initially proposed by Necula under the moniker *Proof-Carrying Code* (PCC) [21], [22]. It was then used to support resource policies for mobile code [3], [6]. Furthermore, Desmet et al. presented an implementation of PCC for the .NET platform [11]. To the best of our knowledge, Cassandra is the only work in the literature about applying PCC to Android [19]. Their approach proposes a type system to precisely track information flows. While precision is an advantage, it is hard to assess the practicability of their approach as no experiments involving real-world applications are reported[6]. Our approach is applicable to real-world large applications.

VI. Conclusion and Further Work

We have presented a policy-based lightweight approach for the verification and certification of Android applications with respect to a given policy. It consists of a simple policy language, a verifier running on a conventional computer and a checker residing on an Android mobile device. We described an implementation of this technique and reported on experimental results obtained on real-world applications. This policy-based scheme represents a potential future model for app stores, where apps are equipped with policies and checkable evidence. Our next step is to increase the efficiency of the checking process on device and to integrate more sophisticated analyses, such as information flow tracking.

References

[1] C. Arthur. Android torch app with over 50m downloads silently sent user location and device data to advertisers. *The Guardian*. 6 December 2013.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, page 29, 2014.

[3] D. Aspinall and K. MacKenzie. Mobile resource guarantees and policies. In *CASSIS*, pages 16–36, 2005.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the Android permission specification. In *ACM Conference on Computer and Communications Security*, pages 217–228, 2012.

[5] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard - enforcing user requirements on Android apps. In *TACAS*, pages 543–548, 2013.

[6] G. Barthe, P. Crégut, B. Grégoire, T. P. Jensen, and D. Pichardie. The mobius proof carrying code infrastructure. In *FMCO*, pages 1–24, 2007.

[7] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium*, pages 131–146, Berkeley, CA, 2013. USENIX.

[8] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. X. Song. Contextual policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.

[9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.

[10] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Crêpe: A system for enforcing fine-grained context-related policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426–1438, 2012.

[11] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .net platform. *Inf. Sec. Techn. Report*, 13(1):25–32, 2008.

[12] A. Desnos. Androguard. http://code.google.com/p/androguard/.

[13] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[6] We have contacted the author with regards to the applicability of Cassandra to real-world apps but so far we have not received a response.

[15] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *ACM Conference on Computer and Communications Security*, pages 235–245, 2009.

[16] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In *ACM Conference on Computer and Communications Security*, pages 50–61, 2012.

[17] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. D. Millstein. Dr. Android and mr. hide: fine-grained permissions in Android applications. In *SPSM@CCS*, pages 3–14, 2012.

[18] J. Kim, Y. Yoon, K. Yi, and J. Shin. Scandal: Static analyzer for detecting privacy leaks in Android applications. In *Mobile Security Technologies (MOST)*, 2012.

[19] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a certifying app store for android. In *SPSM@CCS*, pages 93–104, 2014.

[20] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, pages 328–332, 2010.

[21] G. C. Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[22] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, pages 229–243, 1996.

[23] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC 2013)*, Prague, Czech Republic, April 2013.

[24] M. N. Seghir and D. Aspinall. Evicheck: Digital evidence for android. In *ATVA*, pages 221–227, 2015.

[25] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *OOPSLA*, pages 264–280, 2000.

[26] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *Presented as part of the 21st USENIX Security Symposium*, pages 539–552, Berkeley, CA, 2012. USENIX.

[27] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *ACM Conference on Computer and Communications Security*, pages 611–622, 2013.