A *SEALANT* for Inter-App Security Holes in Android

Youn Kyu Lee*, Jae young Bang†, Gholamreza Safi*, Arman Shahbazian*, Yixue Zhao*, and Nenad Medvidovic*
*Computer Science Department, University of Southern California
941 Bloom Walk, Los Angeles, California, USA 90089
{younkyul, gsafi, armansha, yixuezha, neno}@usc.edu
†Kakao Corporation
Seongnam, Gyeonggi, Korea 13494
jae.bang@kakaocorp.com

***Abstract*—Android's communication model has a major security weakness: malicious apps can manipulate other apps into performing unintended operations and can steal end-user data, while appearing ordinary and harmless. This paper presents *SEALANT*, a technique that combines static analysis of app code, which infers vulnerable communication channels, with runtime monitoring of inter-app communication through those channels, which helps to prevent attacks. *SEALANT*'s extensive evaluation demonstrates that (1) it detects and blocks inter-app attacks with high accuracy in a corpus of over 1,100 real-world apps, (2) it suffers from fewer false alarms than existing techniques in several representative scenarios, (3) its performance overhead is negligible, and (4) end-users do not find it challenging to adopt.**

I. Introduction

This paper targets a known vulnerability in the design of Android's communication model [1], in which components in a single app or across multiple apps communicate by exchanging messages called *intents*. Inter-component communication (ICC) via intent exchange can expose a vulnerable surface to several security attacks, including *intent spoofing* [2], *unauthorized intent receipt* [2], and *privilege escalation* [3]. In these attacks, a malicious app sends and receives intents in a way that appears as if those are ordinary message exchanges.

A large volume of research has focused on ICC vulnerabilities in Android [2], [4]–[14]. However, existing *detection* techniques target only certain types of inter-app attacks [4]–[6], [15] and/or do not support compositional analysis of multiple apps [2], [12], [14]. The state-of-the-art techniques [4], [5], [15] employ data-flow analyses that rely on lists of frequently used Android API methods [16], but tend to overlook ICC vulnerabilities caused by custom methods. Moreover, these analyses [4]–[6] have been shown to experience scalability problems when applied on large numbers of apps [17]. Meanwhile, the runtime *protection* techniques suffer from acknowledged frequent "false alarms" [11], [15] because of the coarse granularity at which they capture ICC information. Additionally, these techniques assume a degree of expertise in Android security [8], [9], [11], [13]. While certain techniques [15], [17] combine vulnerability detection with runtime protection to aid ordinary end-users, they also suffer from potentially large numbers of false alarms.

We present *SEALANT* (Security for End-users of Android via Light-weight ANalysis Techniques), a technique that aims to enable ordinary end-users to protect against inter-app attacks. *SEALANT* identifies vulnerable ICC paths between a given set of apps, inspects each intent sent via those paths at runtime to detect potential attacks, and enables end-users to block the intent on-the-fly. *SEALANT* is distinguished from the existing research because (1) it simultaneously prevents multiple types of Android inter-app attacks—with the current implementation focusing on *intent spoofing, unauthorized intent receipt,* and *privilege escalation*, (2) it extends the detection coverage via a novel combination of static data-flow analysis and compositional ICC pattern matching, (3) it causes fewer false alarms than existing techniques through a finer-grained characterization of ICCs, (4) it supports compositional analysis scaling to a number of apps, and (5) it integrates static detection with runtime monitoring and control of vulnerable ICC paths.

SEALANT comprises two tools: (1) *Analyzer* identifies vulnerable ICC paths by performing static analysis on app bytecode; (2) *Interceptor* is an extension to the Android framework that manages inter-app intent exchanges. We elected to modify Android over two other alternatives—instrumenting the installed apps' bytecode and acquiring administrator privileges, i.e., "rooting"—because (1) once our approach is applied to a device, it does not require altering any of the installed apps, and (2) rooting itself introduces serious vulnerabilities [18].

We have evaluated *SEALANT* in four different ways. (1) We assessed its effectiveness via comparative analysis against existing techniques. *SEALANT* suffered from fewer false alarms while blocking the same or greater number of vulnerable ICC paths. (2) We performed a case study targeting *Analyzer*'s ability to identify vulnerable ICC paths between a set of apps, and *Interceptor*'s ability to selectively block those paths. To this end, we used a test suite comprising 1,150 apps. The test suite includes apps previously identified as vulnerable [17], an open-source testing ground [19], externally developed real-world apps that implement inter-app attacks, and real-world apps randomly selected from publicly available sources [20], [21]. *Analyzer* was able to identify vulnerable ICC paths with high accuracy, while *Interceptor* was able to capture and block each identified path. (3) We evaluated *SEALANT*'s performance by measuring the analysis time of *Analyzer* on different numbers of apps, and the resource overhead imposed by *Interceptor*'s runtime intent inspections. *Analyzer* is scalable to a large number of apps, while *Interceptor* requires nominal additional resources. (4) We performed a user study and survey involving 189 Android end-users in employing *SEALANT*. Overall, the users were able to effectively use *SEALANT* to block vulnerable inter-app intent exchanges and did not find it burdensome to use.

The research we present in this paper is based on our prior work on inter-component communication in event-based

systems (EBS) [22], [23]. While this paper focuses explicitly on Android, *SEALANT* can be expanded to other EBS (e.g., [24]–[28]) with certain modifications.

This paper makes four contributions: (1) *SEALANT*, a technique that enables Android users to protect their devices from multiple ICC vulnerabilities, with a proof-of-concept implementation focusing on intent spoofing, unauthorized intent receipt, and privilege escalation; (2) *Analyzer*, a tool that accurately finds vulnerable ICC paths between apps through a novel combination of data-flow analysis and compositional ICC pattern matching; (3) *Interceptor*, an Android framework extension that automatically detects malicious intents at runtime and enables users to block them; and (4) extensive evaluations of *SEALANT* that involve 1,150 Android apps, compare *SEALANT* to existing alternatives, and engage real end-users.

Section II illustrates inter-app attacks. Section III describes *SEALANT*'s architecture, Section IV its implementation, and Section V its evaluations. Related work is discussed in Section VI, and conclusions are presented in Section VII.

II. Motivating Examples

In this section, we present simplified examples of the three inter-app attack types that *SEALANT* targets: (1) *intent spoofing*, (2) *unauthorized intent receipt*, and (3) *privilege escalation*.

Figure 1(a) and Listings 1 and 2 depict *intent spoofing*. Figure 1(a) shows component `M1` from malicious app `MalApp1` that may send an intent to component `V2` from victim app `VicApp1`. Listing 1 shows where `VicApp1`'s vulnerability resides: `V2` is designed to transfer money to a recipient specified by an incoming intent. Listing 2 illustrates how `M1` of `MalApp1` sends an *explicit intent* that specifies `V2` as its destination component, along with the attacker's account number as the recipient. This is an example of a vulnerable ICC path, from `M1` to `V2`.

Figure 1(b) and Listing 3 illustrate *unauthorized intent receipt*. In Android, if an intent is broadcast without proper permission restrictions, a malicious component can receive it by declaring attributes matching those of the intent. Component `V3` of `VicApp2` from Figure 1(b) is designed to broadcast intents to components in the same app such as `V4`. Listing 3 shows `V3`'s code that broadcasts an implicit intent on a click event, with the action attribute `ShowLocation` and the location information. Although not an intended receiver, malicious component `M2` of `MalApp2` is able to eavesdrop by listening to `ShowLocation` intents and to obtain the user's current location. This is another example of a vulnerable ICC path, from `V3` to `M2`.

Figure 1(c) depicts *privilege escalation*. Component `V6` of `VicApp3` provides a sensitive API that is protected with permission `P1`. While component `V8` of `VicApp4` is granted `P1`, `M3` of `MalApp3` is not, which means that `M3` is restricted to directly access the API of `V6`. Nonetheless, `M3` can still invoke the API in an indirect way, via `V8` which is not protected by any permissions and can be triggered by any component via an explicit intent. By triggering `V8`, `M3` is able to access the sensitive API of `V6` without acquiring `P1`. This is an example of a *transitive* vulnerable ICC path, from `M3`, via `V8`, to `V6`.

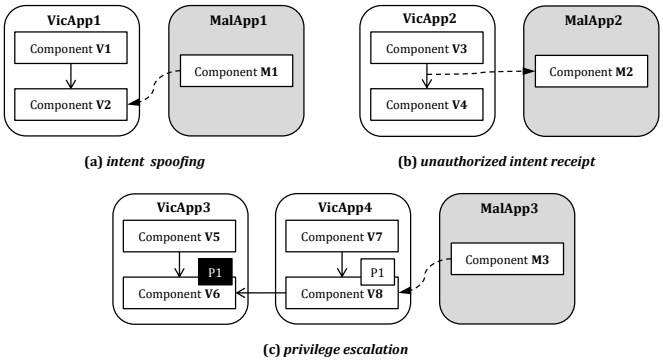

Fig. 1: Inter-App Attacks

Listing 1: Component `V2` of `VicApp1`

```
1 public class V2 extends Activity {
2   public void onStart() {
3    Intent i = getIntent();
4    String recipient = i.getStringExtra("Recipient");
5    String amount = i.getStringExtra("Amount_USD");
6    sendMoneyToRecipient(recipient, amount); }}
```

Listing 2: Component `M1` of `MalApp1`

```
1 public class M1 extends Activity {
2   public void onCreate ( Bundle savedInstanceState ) {
3    Intent i = new Intent();
4    i.setClassName("com.VicApp1", "com.VicApp1.V2");
5    i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
6    i.putExtra("Recipient", ATTACKERS_ACCOUNT_NUMBER});
7    i.putExtra("Amount_USD", 1000000);
8    startActivity(i); }}
```

Listing 3: Component `V3` of `VicApp2`

```
1 public class V3 extends Activity {
2   public void onClick(View v) {
3     LocationManager m = (LocationManager)
          getSystemService(Context.LOCATION_SERVICE);
4     Location location =
5     m.getLastKnownLocation(LocationManager.GPS_PROVIDER);
6     Intent i = new Intent() ;
7     i.setAction ("ShowLocation");
8     i.putExtra ("Location", location.toString());
9     sendBroadcast(i); }}
```

The above examples demonstrate that the attacks are administered in a way that does not differ from ordinary intent exchanges between apps. This makes the identification and restriction of inter-app attacks especially challenging. Moreover, since an ICC can be performed in an essentially invisible way (e.g., via `sendBroadcast()` or through transitive paths), it is difficult for end-users to recognize when the attacks are actually committed. An app developer's caution may minimize the risk of the attacks, but it requires error-prone manual effort, while end-users may still download other unsafe apps.

Although security violations such as these have been studied in computer networks and distributed systems [24]–[29], those techniques cannot be directly applied to Android due to the specifics of its communication mechanism and features. For example, role-based access control [24], [25] has been applied in Android as a form of permission grants; however, it can be violated by *privilege escalation* attacks. Encryption [26],

[27], another popular technique, is not a good fit for Android due to encryption-key distribution issues and limited mobile resources. Meanwhile, techniques specifically targeting Android have either not focused on these issues or have been unable to adequately resolve them, as detailed in Sections V and VI.

III. SEALANT

This section introduces *SEALANT*, a technique that automatically identifies vulnerable ICC paths between Android apps, and enables users to control the ICCs on those paths at runtime. *SEALANT* recognizes each instance of ICC as a relation between a sender, a receiver, and an intent. When an intent from a sender component matches an intent that can be received by a receiver component (either explicitly or through an intent filter), *SEALANT* reports an ICC relation. *SEALANT* builds an ICC graph in which vertices are components and edges are the ICC relations. It then extracts all possible vulnerable ICC paths in the ICC graph and monitors them at runtime. When an instance of ICC matches one of the extracted vulnerable paths, *SEALANT* may block it based on the user's choice.

Figure 2 shows two key components that comprise *SEALANT*: (1) *Analyzer* uses static analysis to generate a list of vulnerable ICC paths between apps, and runs on a user's computer or as an online service; (2) *Interceptor* extends Android to perform runtime monitoring and enable advanced ICC control such as blocking of specific ICCs identified by *Analyzer*. *SEALANT*'s overall process is as follows:

1) *Analyzer* processes the APK[1] files of the installed apps and identifies the vulnerable ICC paths between them.
2) *Analyzer* can optionally contact expert users to confirm specific vulnerable paths that should be monitored.
3) *Analyzer* feeds the highlighted vulnerable ICC paths to the *Interceptor* in a pre-defined format (*SEALANT List*).
4) At runtime, whenever an intent is sent, *Interceptor* captures the information of the corresponding ICC path (e.g., sender's name) from Android's *ActivityManager*.[2]
5) If the captured path information matches one of the vulnerable paths in the *SEALANT List*, *Interceptor* contacts the end-user to determine whether to propagate the intent.
6) Based on the end-user's choice, *Interceptor* will instruct the *ActivityManager* either to block or to route the intent.

We discuss *Analyzer* and *Interceptor* in more detail next.

A. Analyzer

Analyzer performs static analysis on APK files in four phases: (1) analyze target apps, (2) build ICC graph, (3) find vulnerable paths, and (4) generate *SEALANT List*. *Analyzer* is novel in that it returns multiple types of vulnerable ICC paths in a single pass and distinguishes different types of threats, which enables tailor-made countermeasures. It does so by focusing, both, on the data-flow between components and on compositional patterns of ICCs derived from published literature [2]. This enables *Analyzer* to identify a larger number of vulnerable paths and path types than existing techniques (e.g., paths involving

[1]APK is an archive file format that distributes and installs Android apps.
[2]*ActivityManager* is the Android component that governs ICC.

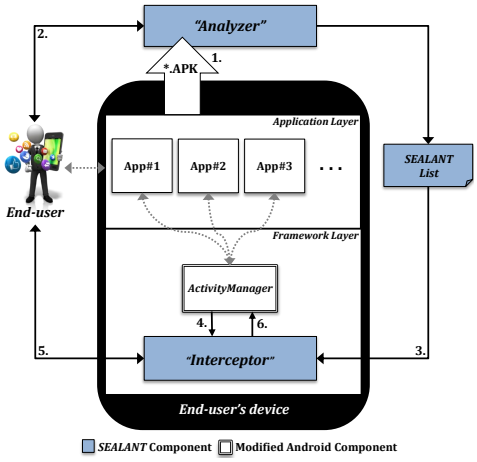

Fig. 2: Overview of *SEALANT*'s Operation

custom methods). Its summary-based model enables analyzing a number of apps at a time, as well as reusing prior analysis results when apps are installed, updated, or removed.

*1) **Analyze Target Apps**: Analyzer* extracts and summarizes each app's architectural information by analyzing the APK file. The summary includes components, intents, intent filters, and permissions. *Analyzer* extracts each component's name, package, permissions held or required, and exported status. To communicate across apps, an Android component must have its exported status set to `true` or contain an intent filter. *Analyzer* only considers exported components in creating ICC graphs. *Analyzer* extracts each intent's attributes (i.e., target component, action, categories, and data) using string constant propagation [22]. If an attribute's value cannot be determined, *Analyzer* conservatively assumes it to be any string.

Once extraction is completed, *Analyzer* examines each component's vulnerability. A vulnerable component is the one containing an intra-component path between an ICC call method and a sensitive method. An *ICC call method* is a standard Android method for sending or receiving intents (e.g., `startActivity()`) [30]. A *sensitive method* is an Android API method that can access sensitive user information (e.g., `getLastKnownLocation()`) or trigger a sensitive operation (e.g., `sendTextMessage()`) [16], [31]. *Analyzer* identifies the relevant paths by employing a static taint analysis that tracks data-flows between methods [32]. If the direction of an intra-component path is from an ICC call method to a sensitive method, *Analyzer* sets the component's vulnerability type to *Active*, because the component is vulnerable to attacks such as *intent spoofing* and *privilege escalation*. If the intra-component path is from a sensitive method to an ICC call method, the vulnerability type is *Passive*, representing attacks such as *unauthorized intent receipt*. In Figure 1(b), for example, the vulnerability type of `V3` is *Passive*, because the location data directs from `getLastKnownLocation()` to `sendBroadcast()`. If a component has multiple intra-component paths, *Analyzer* creates separate component models for each path.

By managing a summary-based model of each app, *Analyzer* is scalable to inspecting a number of apps in a single pass, as evaluated in Section V-C1. Furthermore, when apps are

TABLE I: Attributes of *component* and *intent* in ICC graph

Entity	Attributes	Description
component	*Name*	*The name of the component*
	Pkg	*The name of the package to which the component belongs*
	VulType	*The type of vulnerability (i.e., Active, Passive, or null)*
	VulMeth	*The name of the sensitive method (e.g., sendTextMessage())*
	PermUse	*The name of the permission that the component holds*
	PermReq	*The name of the required permission to access the component*
intent	*Target*	*The name of the component to which the intent is directed*
	IntentAttr	*The intent's attributes: action, category, data type, and data scheme*
	Sender	*The name of the component which sends the intent*
	SenderPkg	*The name of the package to which the sender component belongs*

installed or updated subsequent to running *Analyzer*, *Analyzer* extracts only the architectural information from the newly updated apps and reuses the prior analysis results on the rest.

2) ***Build ICC Graph****:* With the extracted information, *Analyzer* builds an ICC graph based on the rules from Android's API reference documentation [33]. It defines an edge as a tuple $< s, r, i >$, where s is a sender and r a receiver component, and i is an intent between them. *Component* and *intent* are entities that manage summarized information as shown in Table I.

Since the extraction of architectural information is performed in a conservative way (specifically, relying on attributes of intents), the set of edges may include false positives. However, this will not affect *SEALANT*'s runtime accuracy because no ICC instances to be routed via those edges will ever be initiated.

3) ***Find Vulnerable Paths****: Analyzer* implements Algorithm 1 on the ICC graph to identify vulnerable paths. *Analyzer* marks an edge as vulnerable (1) if it has a vulnerable component at one or both ends, or (2) if it forms a particular compositional pattern. To find vulnerable transitive ICC paths, *Analyzer* recursively identifies a set of connected edges that can access a vulnerable component by calling the *PathFinder* method (Algorithm 2). x_y indicates attribute y of entity x (depicted in Table I), and $x.y$ represents element y in edge x.

Analyzer first parses edges into two sets: inter-app (*IAC*) for edges between components belonging to different apps, and inter-component (*ICC*) otherwise. Algorithm 1 iterates over each edge e in $IAC \cup ICC$ (lines 5-19) and considers four different cases that cover all types of vulnerable paths we target in this paper: the first two cases identify paths that involve vulnerable components; the latter two cases identify paths based on previously identified compositional patterns [2].

Case 1 (line 8) occurs when e directs to a receiver vertex whose vulnerability type is "*Active*". If e is an IAC edge, Algorithm 1 determines the type of attack by calling $PermCompare(c_1, c_2, m)$ (line 10), a method that returns the type of attack by comparing the permissions of components c_1 and c_2, where m is a sensitive method that forms an intra-component path with an ICC call method within c_2. If c_2 holds a permission that c_1 does not, and the permission is required to use m [31], *PermCompare* returns *"privilege escalation"*; otherwise, it returns *"intent spoofing"*. Once the type of attack is determined, Algorithm 1 adds $\{e\}$ to the set *VulPaths* that contains all detected vulnerable ICC paths (line 10), and then calls *PathFinder* to identify transitive ICC paths (line 11).

Algorithm 1: *Identifying vulnerable ICC paths*

Input: $G \Leftarrow$ an ICC graph
Output: $VulPaths \Leftarrow$ a set of vulnerable paths

```
1  Let IAC be a set of IAC edges in G
2  Let ICC be a set of ICC edges in G
3  Let s be a sender component
4  Let r be a receiver component
5  foreach e ∈ IAC ∪ ICC do
6      s ⇐ e.sender
7      r ⇐ e.receiver
8      if (r_VulType = "Active") then
9          if e ∈ IAC then
10             add ({e}, PermCompare(s, r, r_VulMeth)) to VulPaths
11         PathFinder(s, r, {e})
12     else if (s_VulType = "Passive") and (e ∈ IAC) then
13         add ({e}, "unauthorized intent receipt") to VulPaths
14     else if (e ∈ IAC) then
15         foreach (g ∈ ICC) do
16             if (r = g.receiver) then
17                 add ({e}, "intent spoofing") to VulPaths
18             else if (s = g.sender) ∧ (e.intent = g.intent) then
19                 add ({e}, "unauthorized intent receipt") to VulPaths
```

Algorithm 2: *PathFinder*

Input: $s, r \Leftarrow component$, $E \Leftarrow$ a list of distinct edges
Output: updated *VulPaths*

```
1  foreach f ∈ IAC ∪ ICC do
2      if (f.receiver = s) and (∀e ∈ E, e.receiver ≠ f.sender) then
3          append f to E
4          if (∃e ∈ E, e ∈ IAC) then
5              add (E, PermCompare(f.sender, r, r_VulMeth)) to VulPaths
6          PathFinder(f.sender, r, E)
7  remove the last element of E
```

As depicted in Algorithm 2, *PathFinder* iterates over each edge $f \in IAC \cup ICC$, to check if f connects to the previously identified edge's sender component s, and if f's own sender is a newly visited component (line 2). If so, *PathFinder* appends f to the list of distinct connected edges E (line 3). If E contains an inter-app edge ($e \in IAC$), *PathFinder* determines the type of attack by calling *PermCompare*, and adds E to *VulPaths* (line 5). *PathFinder* recursively identifies other components that are connected to a vulnerable component through edges in the ICC graph. It stops its processing when it visits all transitively connected components to the original edge's receiver r or reaches an already visited component. When it finishes iterating, *PathFinder* removes the last element from E to enable correct exploration of additional transitive paths.

Case 2 (lines 12-13 in Algorithm 1) deals with the situation when the vulnerability of *e.sender* is *Passive* and $e \in IAC$, which may result in leaking sensitive information between apps through e. If so, the type of attack is set to *"unauthorized intent receipt"* and $\{e\}$ is added to *VulPaths* (line 13).

Case 3 (lines 14-17) occurs when edges $e \in IAC$ and $g \in ICC$ ($e \neq g$) both lead to the same receiver vertex. It represents a pattern of attack in which g is an intended access to r within an

app, but e may be a spoofed access from a malicious component across apps. In this case, the type of attack is set to *"intent spoofing"* and the edge $\{e\}$ is added to *VulPaths* (line 17).

Case 4 (lines 18-19) occurs when edges e and g share the same sender and intent. If g represents an originally intended receipt within the app and e an unintended receipt across apps, Algorithm 1 will set the type of attack to *"unauthorized intent receipt"* and append $\{e\}$ to *VulPaths* (line 19).

*4) **Generate SEALANT List**:* As the last step, *Analyzer* generates the *SEALANT List* based on *VulPaths*, the output from the previous phase. *Analyzer* first normalizes the output by checking for redundant paths. It then transforms the information about identified paths into a pre-defined format that is compatible with *SEALANT*'s *Interceptor* component.

B. Interceptor

Interceptor monitors and analyzes each instance of ICC. Whenever an ICC is requested, *Interceptor* checks whether it is specified in the *SEALANT List*. *Interceptor*'s ICC control strategy is distinguished from competing techniques due to its finer-grained characterization of ICC paths based on (1) sender, (2) receiver, and (3) intent. As evaluated in Sections V-A and V-B, this increases the accuracy in blocking ICCs.

Interceptor resolves two challenging issues: (1) extracting each component's information at runtime to effectively prevent malicious ICCs, while (2) minimizing the impact on Android's functionality. *Interceptor* captures a sender component's information by instrumenting the framework-level class of each type of component (e.g., *Activity*) in the Android framework, while it captures an intent's and a receiver's information by extending a core component that governs intent routing (i.e., *ActivityManager*). *Interceptor* minimizes the impact on Android by avoiding removal of standard components or modification of core methods (further discussed in Sections IV and V-C).

*1) **Interceptor's Architecture**:* *Interceptor* extends the Android framework with four components, as depicted in Figure 3. Three components—*Blocker*, *ChoiceDatabase*, and *ListProvider*—are newly added, while one—*ActivityManager*—is a modification of an existing Android component.

Blocker interacts with end-users and performs *Interceptor*'s core functionalities: monitoring, matching, and blocking.

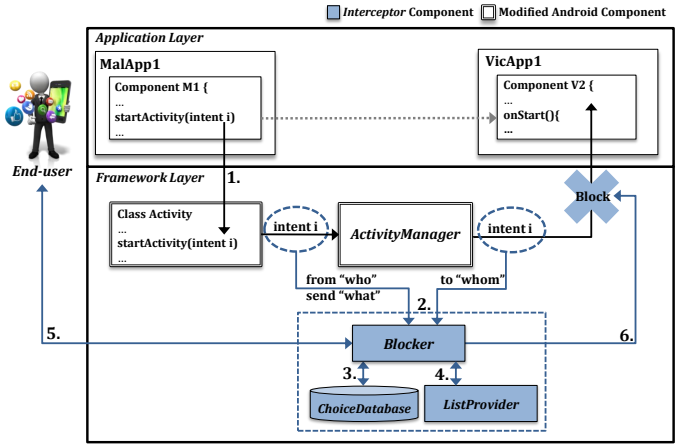

Fig. 3: The Architecture and Operation of *Interceptor*

Blocker directly communicates with *ActivityManager* to obtain the detailed information of each instance of ICC, and to possibly induce *ActivityManager* to block a particular instance of ICC. *Blocker* imports the *SEALANT List* from *ListProvider*, and refers to the previously made choices from *ChoiceDatabase*.

ActivityManager controls every instance of ICC processed through the Android framework, by collaborating with other Android components (e.g., *PackageManager*). We extended *ActivityManager* to capture the information of each ICC instance (sender and receiver components, and intent's attributes), share the information with *Blocker*, and block a particular instance of ICC upon *Blocker*'s request.

ChoiceDatabase stores end-user choices (to block or route) for each vulnerable ICC path. Stored choices are automatically applied when the same ICC is requested, and can be removed upon end-user's request. When a new *SEALANT List* is imported, *ChoiceDatabase* expunges only the choices that correspond to the updated or removed apps.

Finally, *ListProvider* imports and maintains the *SEALANT List*. When a *SEALANT List* is installed in the pre-defined space of the user device (e.g., external SD card), *ListProvider* imports it and maintains the specified information as a permanent condition until a new *SEALANT List* is introduced.

*2) **Interceptor's Operation**:* Figure 3 illustrates the interaction among *Interceptor*'s four components. For clarity, the depicted six-step scenario is based on the example from Listings 1 and 2, but it is reflective of *Interceptor*'s operation in general.

1) When `M1` of `MalApp1` tries to send intent `i` by calling `startActivity()`, request is routed to *ActivityManager*.
2) *ActivityManager* extracts sender's (i.e., `M1`'s) information and searches for components permitted to receive intent `i`. If a receiver is identified (i.e., `V2` of `VicApp1`), *ActivityManager* passes the ICC information to *Blocker*.
3) After receiving information about the ICC, *Blocker* first examines *ChoiceDatabase*. If a choice for the ICC already exists, *Blocker* induces *ActivityManager* to act (block or route the ICC) without engaging the end-user.
4) In case no corresponding choice exists in *ChoiceDatabase*, *Blocker* scans the *SEALANT List* provided by *ListProvider*.
5) If the information about the requested ICC matches that in the *SEALANT List*, *Blocker* will give the user four options: (1) allow the ICC once, (2) block it once, (3) allow it always, and (4) block it always. If the user selects options (3) or (4), her choice will be stored in *ChoiceDatabase*.
6) If the end-user chooses to allow (resp. block) the requested ICC, *Blocker* will instruct *ActivityManager* to send intent `i` to `V2` (resp. trap it).

*3) **Interceptor's Strategy for Blocking ICCs**:* *Interceptor* is engaged between the times when an intent is first requested and when it is actually dispatched to its destination. *Interceptor*'s operation may thus cause a delay in processing intents, which may be exacerbated by the number of vulnerable ICC paths in the *SEALANT List*. However, since Android's ICC is performed via asynchronous API calls, we hypothesize that this delay will not significantly impact the system's operation. In Section V-C, we empirically evaluate *Interceptor*'s performance overhead.

In case when an end-user has blocked a requested ICC, the apps that are involved in the ICC will not get any response to their request back from the framework. Since Android implements ICCs asynchronously, those apps will simply "skip" the corresponding operation without causing runtime crashes.

To block a vulnerable transitive ICC, *Interceptor* begins by matching the first path of the vulnerable transitive ICC path and setting its `transitive_flag` to `true`. This flag is managed per each vulnerable transitive ICC path and remains `true` as long as the subsequently requested ICCs match the subsequent paths in the vulnerable transitive path. Once the last path of the vulnerable transitive ICC path is reached, *Interceptor* alerts the end-user and resets `transitive_flag` to `false`. In the example from Figure 1(c), let us assume that the vulnerable transitive ICC path `M3` → `V8` → `V6` is in the *SEALANT List*. If `M3` launches `V8` via an intent, *Interceptor* will set `transitive_flag` to `true`. Then, if `V8` launches `V6` via an intent, *Interceptor* will alert the user and reset the flag.

IV. Implementation

We have implemented *SEALANT*'s *Analyzer* as a stand-alone Java application that receives as input a set of Android apps in APK files, and exports a *SEALANT List* in the pre-defined XML format. *Analyzer*'s implementation combines approximately 3,000 newly written LOC with three off-the-shelf tools. The tools are used in the first of *Analyzer*'s four phases (recall Section III-A). *Analyzer* integrates two static analysis tools, IC3 [34] and COVERT [7], to extract architectural objects from apps. We employed both tools because neither of them alone discovers all of the needed information: IC3 misses outbound intents in certain scenarios [34], while COVERT only returns coarse-grained intent information that excludes certain attributes (e.g., data type) [7]. *Analyzer* orchestrates the two tools together and combines their outputs in order to generate a more complete list of architectural objects. In identifying intra-component paths between ICC call methods and sensitive methods, *Analyzer* uses FlowDroid [32], a highly precise intra-component taint analysis tool for Android.

We implemented *SEALANT*'s *Interceptor* on top of Android Open Source Project (AOSP) 4.4.4 KitKat [35], which is the most popular version of Android [36] today. We directly modified the source code of several standard Android components including *ActivityManagerService*, *ActivityManagerNative*, and *IntentFirewall*. In total, we introduced about 600 LOC spread over 10 classes. To minimize the impact on the original functionality of Android, we did not remove any standard components or methods. Our modification was limited to parts of Android that are usually a layer beneath manufacturers' customizations, and can easily be applied to Android versions 4.4 and later without significant changes. We were able to successfully run *Interceptor*'s system image, both, on the Android emulator [37] and on a Google Nexus 7 device.

Since framework-level components in Android do not provide a user interface (UI), we also implemented an Android app that provides a UI to perform (1) pushing the *SEALANT List* from an external SD card to *Interceptor*'s *ListProvider*, (2) removing the list from *ListProvider*, and (3) removing previous choices from *Interceptor*'s *ChoiceDatabase*.

Running *SEALANT* requires compiling *Interceptor*'s source code with the provided drivers, and installing the image files using the freely available Android debug bridge [38] and Fastboot [39]. This cost can be minimized by bundling *SEALANT* with Android. *SEALANT*'s code, required drivers, and compiled tools are available at `http://softarch.usc.edu/sealant/`.

V. Evaluation

We evaluate *SEALANT* for effectiveness (Section V-A), accuracy (V-B), performance (V-C), and usability (V-D).

A. Effectiveness

To the best of our knowledge, two existing works share *SEALANT*'s goal of providing *protection* of end-users from inter-app attacks: SEPAR [15] (previously named DroidGuard [17]) and XmanDroid [3], [11]. SEPAR identifies vulnerable surfaces of a set of apps via static analysis and uses dynamic memory instrumentation that hooks the method calls of target apps at runtime. For example, in the scenario from Figure 1(a), SEPAR would identify the vulnerability of `V2` and hook the `startActivity()` method that sends an intent to `V2`. XmanDroid is a technique that only targets *privilege escalation* attacks by leveraging an extension to Android. XmanDroid enables a user to pre-define a list of ICC restriction policies, and automatically blocks ICCs that match any of those policies.

An ideal comparison of *SEALANT* against these two techniques would have included executing their implementations in a controlled setting and/or on a set of real-world Android apps. However, the implementation of XmanDroid we obtained from its authors only runs on a prior version of Android (2.2.1), while the current prototype implementation of SEPAR is missing certain features covered by the underlying technique (e.g., the policy enforcement module). In Section V-B, we do evaluate *SEALANT* directly against one of the implemented features of SEPAR. We tried unsuccessfully to build an installation of XmanDroid on several recent versions of Android. Given the changes in Android since 2.2.1, continuing with this strategy proved impractical. For these reasons, we decided to analytically compare the three techniques, relying on the published algorithms of SEPAR [15] and XmanDroid [3], [11].

*1) **Comparison with SEPAR**:* A detailed comparison of *SEALANT* and SEPAR using probabilistic random variables to capture their respective operations can be found at `http://softarch.usc.edu/sealant/`. Here we provide a summary of that analysis. *SEALANT* raises fewer false inter-app attack alarms compared to SEPAR, because SEPAR does not support a finer-grained characterization of ICC paths (i.e., sender, receiver, and intent). For example, in the scenario depicted in Figure 1(a), whenever an explicit intent is routed to `V2`, SEPAR would raise an alarm, even if the intent was sent from the component in the same app (i.e., `V1`).

*2) **Comparison with XmanDroid**: SEALANT* suffers from fewer false negatives than XmanDroid. The detection mechanism of XmanDroid requires a user to explicitly specify policies

TABLE II: Applying *SEALANT* on the 135 Apps in Our Core Test Suite

Attack Type of Test Suite	Number of Apps			Vulnerable ICC Paths		Identified ICC Paths (Precision / Recall)			Blocked ICC Paths (Precision / Recall)
	Vulnerable	Malicious	"Trick"	Direct	Transitive	SEPAR	IccTA	*Analyzer*	*Interceptor*
IS	27	26	4	25	1	0.00 / 0.00	1.00 / 0.04	1.00 / 0.96	1.00 / 1.00
UIR	26	25	4	25	0	0.50 / 0.20	1.00 / 0.16	1.00 / 0.92	1.00 / 1.00
PE	8	11	4	4	4	0.00 / 0.00	0.00 / 0.00	1.00 / 1.00	1.00 / 1.00
Total	61	62	12	54	5	0.50 / 0.08	1.00 / 0.08	1.00 / 0.95	1.00 / 1.00

IS = *intent spoofing*; UIR = *unauthorized intent receipt*; PE = *privilege escalation*.

indicating the types of inter-app attacks she wishes to detect and ICC paths to monitor at runtime. This may omit critical inter-app attacks. Recall the *privilege escalation* attack scenario from Figure 1(c). When component `M3` in `MalApp3` requests an ICC to access `V8` in `VicApp4`, XmanDroid inspects the permissions of `MalApp3` and `VicApp4` based on the pre-defined policies. Although a few general policies for XmanDroid have been proposed [11], they do not cover all vulnerability scenarios. In the above scenario, if a user-specified policy does not prohibit an ICC between an app with permission `P1` and another app without it, XmanDroid will not raise an alarm. Since *SEALANT* inspects all ICC paths via static analysis to identify vulnerable paths, it does not suffer from this type of false negative.

SEALANT also suffers from fewer false positives than XmanDroid. XmanDroid finds ICCs that match policies specifying the sender and receiver permission combinations. However, this would also block safe ICCs initiated by a benign app with an identical set of permissions as a malicious app. Suppose that XmanDroid has a policy that would block ICCs between `MalApp3` and `VicApp4` in the scenario depicted in Figure 1(c), and the device had another app, `BenignApp`, which is confirmed as reliable and holds identical permissions to `MalApp3`. In that case, even if `BenignApp` initiated an ICC to a method of `VicApp4` that does not require `P1`, XmanDroid would block that ICC. *SEALANT* would not trigger such a false alarm.

B. Applicability and Accuracy

We evaluated *Analyzer*'s accuracy in identifying vulnerable ICC paths by comparing its results against those of SEPAR [15], [17] and IccTA [4], [40], state-of-the-art tools for ICC vulnerability detection [5], [6], [32]. We evaluated *Interceptor*'s ability to block vulnerable ICC paths at runtime. We used a test suite of 1,150 Android apps in total.

*1) **Experimental Setup**:* To build our test suite, we first selected several real-world apps that are vulnerable to inter-app attacks. Among the apps that were previously identified [15] from repositories such as Google Play [20], F-Droid [41], MalGenome [21], and Bazaar [42], we selected 13 that are exposed to the three types of attacks *SEALANT* targets. We also included six apps from DroidBench 2.0 [19], an app collection for benchmarking ICC-based data leaks. Since several of the 19 vulnerable apps did not have readily available malicious apps targeting them, we built 25 malicious apps, each of which performed one inter-app attack. To mitigate internal threats to the validity of our results, we also asked 39 graduate students at University of Southern California (USC) to build sets of apps that implement inter-app attacks based on published literature [2], [11]. Each of those sets was either a pair of apps forming a simple path, or a trio of apps forming a transitive path. Each set consisted of at least one vulnerable app and at least one malicious app that exploits the vulnerable app. Without any intervention by the authors, the students built 41 distinct sets. This yielded 91 apps in total, of which 47 were new, while 42 were modified and 2 unmodified apps obtained from public sources [43], [44].

In total, this yielded 65 sets containing 135 apps, with 54 vulnerable ICC paths and 5 vulnerable transitive ICC paths. To ensure that inter-app attacks can be actually launched, we manually inspected the code of each set, and installed and ran the set on a Google Nexus 7. We confirmed that the attacks from the malicious apps were successfully launched and exploited the vulnerable apps by observing the apps' behavior via the device's UI and via *logcat*, a native Android tool for monitoring system debug outputs [45]. Our test suite also includes 12 "trick" apps containing vulnerable but unreachable components, whose identification would be a false warning. We divided this *core* test suite into three different groups, based on the type of attack to which a vulnerable app is exposed, as shown in Table II. Subsequently, we created an *expanded* test suite totaling 1,150 apps, by including another 1,015 apps randomly selected from Google Play [20] and MalGenome [21].

*2) **Evaluation of Analyzer**:* We evaluated *SEALANT*'s *Analyzer* for accuracy in identifying vulnerable ICC paths as compared to SEPAR and IccTA. We used our core test suite to measure all three approaches' (1) *precision*, i.e., identified ICC paths that were actually vulnerable, and (2) *recall*, i.e., the ratio of identified to all vulnerable ICC paths. As depicted in Table II, *Analyzer* detected vulnerable ICC paths with 100% precision and 95% (56 of 59) recall. It was unable to correctly extract intent information in three cases due to the inaccuracies inherited from IC3 [34] and COVERT [7] (recall Section IV). *Analyzer* correctly ignored all "trick" cases with unreachable vulnerable paths. SEPAR had 50% precision and 8% recall. This is primarily because SEPAR was designed (1) to identify vulnerable components or interfaces rather than specific ICC paths between them and (2) to return an ICC path only when both sender and receiver contain sensitive Android API methods [16], hampering its applicability in cases such as *privilege escalation* via a transitive ICC. IccTA had 100% precision and 8% recall. Since it targets a single type of attack (privacy leaks), IccTA also returned an ICC path only when it involved sensitive API methods [16]. Although *SEALANT* outperformed SEPAR and IccTA in our evaluation, it is important to note that SEPAR and IccTA support both intra- and inter-app analysis and may detect additional vulnerabilities that *SEALANT* does not.

We then used our expanded test suite of 1,150 apps (9,964 components, 20,787 ICC paths). We created 23 non-overlapping bundles, each comprising 50 apps randomly selected from

TABLE III: *Analyzer*'s Performance on Different Num. of Apps

Number of Apps	25	50	75	100
Avg. Number of Components	237	553.5	761	1200
Avg. Number of ICCs	218	701.5	1110.5	1690.5
Avg. Analysis Time (Sec.)	22.17	42.24	107.27	118.43

the suite. We created 50-app bundles because this number is higher than the recently cited number of apps an average smartphone user regularly uses each month [46]. We ran all three tools on each bundle and manually checked if each identified ICC path is indeed vulnerable. *Analyzer* flagged 86 ICC paths, with 93% precision. The six false-positives were caused by IC3's inaccuracy in identifying intents and COVERT's omission of intent attributes in certain scenarios. SEPAR and IccTA were unable to analyze the bundles on four different hardware configurations. SEPAR's logs indicated that it was unable to generate flow-analysis results in some cases, while it did not return any vulnerabilities in other cases. IccTA invariably crashed; it was unable to analyze more than one app at a time in more than 75% of our attempts.

3) ***Evaluation of Interceptor****:* We evaluated *Interceptor*'s accuracy in detecting and blocking malicious ICCs at runtime. To monitor all ICCs exchanged on a device, we integrated a logging module that outputs information of each ICC instance via *logcat* [45] into *ActivityManager* (recall Section III). We installed the 135 apps in our core test suite on a Google Nexus 7 with *Interceptor* set up, ran *Analyzer* on the device, and provided the resulting *SEALANT List* to *Interceptor*.

To run test scripts that trigger ICCs, we used monkeyrunner [47], an Android tool for running test suites. We designed each script to trigger one type of vulnerable ICC in the *SEALANT List* as well as various benign ICCs. We configured the scripts to choose to block an ICC when *Interceptor* prompts for a blocking choice. We repeated executing each script until we accumulated 30 blocked ICCs. At the end of each test script execution, we manually inspected the logs in order to measure (1) *precision*, i.e., if all blocked ICCs corresponded to vulnerable paths specified in the *SEALANT List*, and (2) *recall*, i.e., if *Interceptor* allowed any ICC attempts over the vulnerable paths. *Interceptor* was able to block vulnerable ICCs in the core test suite with perfect precision and recall (see Table II).

C. Performance

1) ***Evaluation of Analyzer****:* To evaluate the performance of *Analyzer*, we used a PC with an Intel dual-core i5 2.7GHz CPU and 4GB of RAM. We divided our expanded test suite into four categories with different numbers of apps (25, 50, 75, and 100). For each category, we created ten different bundles randomly selected from the 1,150 apps, and ran *Analyzer* on each bundle. On average, extracting architectural information from each app took 77.95s and identifying vulnerable ICC paths took 1.08s per app. While the extraction is relatively time-consuming, in scenarios where an app is newly installed or updated, *Analyzer* reuses the previously extracted app models to minimize the execution time. It performs the extraction only on the new app, and then runs the vulnerable path identification over all apps.

TABLE IV: Differences in Execution Times (in milliseconds)

	Mean	Min	Max	Std Dev
Interceptor	25.51	11.31	81.12	10.22
AOSP	25.20	10.09	45.85	7.18
Difference	0.31	1.22	35.27	3.04

Table III shows the average numbers of components and ICCs in each category. Since our approach manages an individual summary-based model of each app, the analysis time scales linearly with the number of apps.

2) ***Evaluation of Interceptor****:* To evaluate *Interceptor*'s impact on performance, we measured the differences in execution times between Android with *Interceptor* and without it (in the remainder of this section, referred to as "*Interceptor*" and "AOSP" [35], respectively). We configured the two environments to be highly similar and to reasonably reflect the real-world. We employed the Google Nexus 7 in both environments and configured both to use Android KitKat 4.4.4. We installed the 50 most popular third-party apps [48] on the devices.

To observe *Interceptor*'s worst-case performance overhead, we manually created a *SEALANT List* that would induce the longest execution time. The list contained 10 paths (amounting to 20% of the installed apps), none of which matched the actual ICC paths between the 50 installed apps. This maximized the overhead of *Interceptor*'s detection operation which sequentially matches an ICC to each path in its list. The above numbers were selected because they reflect (in fact, surpass) those found in the real-world: an average user regularly uses about 30 apps per month [46], and around 10% of Android apps are vulnerable to inter-app attacks [17]. To trigger a large number of ICCs on the test devices, we used Monkey [49], which generates pseudo-random streams of user- and system-level events on a target device. We used the same seed value in *Interceptor* and AOSP so that Monkey would generate identical event sequences in both environments. We injected 5,000 events in each environment and measured the time it took to process each event. We repeated this five times to mitigate the impact of conditions such as battery-status changes.

Table IV describes the results we obtained. The difference in mean execution times was less than 1ms, and in maximum execution times under 40ms. Differences of this degree are negligible because the threshold at which an end-user begins noticing slowdown in mobile app response is 100-200ms [50]. *Interceptor* introduces low overhead because it simply extends an existing operation that AOSP already regularly performs to match a requested ICC with the list of paths on the device [35].

D. Usability

When an intent exchange matches a vulnerable ICC path, *SEALANT* requires the end-user to either block or allow the exchange in order to secure her device. To assess how challenging such choices are for end-users, we conducted a user study and a survey, guided by two hypotheses:

- **H1:** The intent-exchange control choices *SEALANT* requires an end-user to make are *not more difficult* than the choices required of end-users by "stock" Android.

TABLE V: Difficulty, Confidence, and Response Time per Dialog Type

Dialog Type	User Study										Survey					
	Difficulty			Confidence			Response Time				Difficulty			Confidence		
	n	$\bar{x}$	s	n	$\bar{x}$	s	n	$\bar{x}$	M	s	n	$\bar{x}$	s	n	$\bar{x}$	s
`Type 1`	34	5.26	1.62	34	5.65	0.95	34	6.99	4.06	8.29	155	4.14	1.29	155	4.16	1.30
`Type 2`	34	4.68	1.70	34	5.35	1.20	34	12.32	9.92	7.92	155	4.29	1.25	155	4.15	1.33
`Type 3`	34	4.79	1.68	34	5.56	1.02	34	9.02	6.51	8.58	155	4.65	1.38	155	4.35	1.36
`Type 4`	34	4.97	1.40	34	5.50	1.21	34	6.24	5.10	4.32	155	4.35	1.33	155	4.17	1.26
`Type 1-3`	102	4.91	1.67	102	5.52	1.06	102	9.44	7.24	8.53	465	4.36	1.32	465	4.22	1.33

n = num participants. $\bar{x}$ = mean. M = median. s = std deviation. Difficulty and confidence values are on 7-point Likert scales (1 = very difficult, 7 = very easy; 1 = not confident at all, 7 = fully confident). Response time values are in seconds. The `Type 1-3` row presents merged data from `Type 1` through `Type 3` rows.

- **H2:** A non-expert user can make intent-exchange control choices that prevent an inter-app attack *most of the time*.

*1) **Experimental Setup**:* Our user study and survey were designed to simulate situations in which users make choices reflective of daily Android use (e.g., whether to install an app after being shown the list of permissions it requires). Among those choices, we also inserted choices required by *SEALANT*. We asked the participants how difficult it was to make each choice and how confident they were in making the choice.

The study included 34 participants, all graduate students at USC, recruited via an e-mail list. The students' majors spanned engineering, communication, business, and social work. The background survey showed that the participants had used a mobile device for 59 months on average. 25 of the participants (74%) reported Android as their primary mobile platform or one they had experience using; 9 (26%) had not used Android previously. 5 participants (14%) were aged between 18 and 24, and the remaining 29 (86%) between 25 and 34.

We provided each user study participant a Google Nexus 7 with *SEALANT* pre-installed. They were presented with a series of 20 common scenarios of four different types:

- `Type 1` – A dialog asks the user whether to install an app randomly selected from a credible source (Google Play [20]) given the list of permissions the app requires.
- `Type 2` – Same as `Type 1`, but with apps randomly selected from an unreliable source.
- `Type 3` – Intent matches multiple filters. Android displays a dialog so the user can choose which app to use.
- `Type 4` – A dialog prompts the end-user to make a choice to block or allow a vulnerable inter-app access.

We used native-Android dialogs in 12 of the 20 scenarios (`Type 1-3`), and in the remaining 8, we used *SEALANT*'s customized dialog (`Type 4`) that presents (1) the sender/receiver apps' names, (2) the identified attack type, and (3) block, allow, and always buttons among which the end-user must choose. Half of the apps used in `Type 4` scenarios were selected from apps used in `Type 1` and the other half from `Type 2` scenarios.

During the study, we logged every interaction between a participant and the device via logcat [45]. At the end of each scenario, we asked participants to assess the scenario's difficulty and confidence in their choices, using a 7-point Likert scale.

In order to expand our dataset, we designed the online survey in the same manner as the user study. We took screenshots of what a user would see on her device as she went through the 20 scenarios, presented the screenshots to the survey respondents, and prompted them to make the corresponding choices.

We sent out 200 survey requests and received 155 valid responses (78%); 45 people did not respond or only partially completed the survey. We sent requests to known e-mail lists and contacts, and allowed them to self-select. The respondents had used a mobile device for 51 months on average. 138 (89%) named Android as their primary mobile platform or had experience using it. The survey covered a range of age groups and occupations. 11 respondents (7%) were aged 18-24, 46 (30%) were 25-34, 37 (24%) were 35-44, 35 (22%) were 45-54, and 26 (17%) were 55+. Respondents included 46 students (30%), 27 medical doctors (17%), 20 business people (13%), 11 housewives (7%), 10 software engineers (7%), 9 professors (6%), 5 retailers (3%), 5 lawyers (3%), and 22 others (14%).

More detailed information about the user study and survey is available at `http://softarch.usc.edu/sealant/`.

*2) **Results**:* We evaluate hypotheses **H1** and **H2** using the user study and survey data. For simplicity, we refer to the user study participants and survey respondents as "participants".

H1 – We compared (1) the difficulty perceived by participants in making their choices, (2) the confidence participants had in their choices, and (3) the time it took to make choices for native-Android dialogs (`Type 1-3`) and *SEALANT* dialogs (`Type 4`). Table V presents the data we obtained. A comparison of the mean degrees of difficulty showed that they did not differ significantly between the two groups of scenarios (Student's t-test; p-value 0.928 for user study and 0.972 for survey). A comparison of the mean degrees of confidence yielded the same conclusion (Student's t-test; p-value 0.853 for user study and 0.646 for survey). Finally, the median response time was significantly lower for `Type 4` than for `Type 1-3` scenarios (the Mann-Whitney-Wilcoxon test; p-value 0.000). These results support the conclusion that *SEALANT*'s intent-exchange control choices are not more difficult than those of stock Android.

H2 – We measured the proportion of instances in which a participant elected to block an intent exchange and prevent an attack in a `Type 4` scenario. In general, users may deliberately allow vulnerable intent exchanges (e.g., a user trusts both apps). However, in our study, unbeknownst to the users, we only included paths actually leading to exploits, allowing us to know the correct behavior. Recall that one half of the apps in the `Type 4` scenarios came from reliable and the other half from unreliable sources. In the combined `Type 1` (credible apps) and `Type 2` (unreliable apps) scenarios, participants chose to cancel installation 51% of the time. That tendency, halting an on-going activity to avoid security threats, was much higher for `Type 4` scenarios. The 34 user study participants chose intent blocking 70% of the time, while 155 survey participants

chose blocking 68% of the time. Participants were thus able to make intent-exchange choices that did not lead to inter-app attacks at a much higher rate than their "average" behavior.

E. Threats to Validity

Our user study participants were students. To address any resulting bias, we additionally conducted the survey whose respondents spanned a variety of ages and occupations. The survey merely emulated a mobile environment, possibly influencing the participants' choices. As a mitigation, we carefully described each scenario to provide the participants with the context they would have had if they had used an actual device. We also separately analyzed the user study and survey results, and both support our conclusions. Lastly, the participants elected to allow a fair portion (≈30%) of the vulnerable ICCs in cases we designed blocking to be the appropriate choice. While we consider the users' choices to block the rest ≈70% of ICCs that would otherwise have remained uncaught without *SEALANT* as a positive result, this indicates that improvements may be possible with regards to how *SEALANT* presents the vulnerable ICCs to end-users.

VI. Related Work

Approaches that target Android's vulnerabilities use program analysis, ICC analysis, and/or policy enforcement.

Program analysis is employed by several approaches [2], [23], [32], [51]–[63]. ComDroid [2] categorizes vulnerabilities in inter-app communication and detects vulnerabilities in target apps via static analysis. FlowDroid [32] provides intra-component taint-flow analysis. CHEX [12] leverages data-flow analysis to discover component hijacking vulnerabilities. Unlike *SEALANT*, these techniques mainly focus on individual apps.

ICC analysis is the focus of another body of research [4]–[7], [14], [34], [64]–[74]. Epicc [14] and IC3 [34] statically extract information from Android apps for ICC-aware analyses. DidFail [5] uses taint-flow analysis to locate sensitive inter-app data-flows, but targets only *Activity* components and neglects intents' data scheme. AmanDroid [6] identifies privacy leaks by tracking components interactions, but has been shown to work incorrectly on *Content Provider* components and certain ICC methods. IccTA [4] is a taint-flow analysis targeting privacy leaks. While instrumenting source code to resolve the connections between components does improve its precision, it does not target other types of inter-app attacks. COVERT [7] introduces a compositional analysis of inter-app vulnerabilities, especially against permission leakage. It does not target other types of inter-app attacks or handle intents' data scheme. These approaches detect but do not protect against ICC vulnerabilities.

Policy enforcement in Android is explored via (1) app code instrumentation [8], [17], [75]–[81], (2) Android framework extension [9]–[11], [13], [54], [82]–[86], and (3) dynamic memory instrumentation [15], [87]. Aurasium [76] enforces arbitrary policies by interposing code into the target app. DroidForce [8] enforces custom data-centric policies by instrumenting an app's bytecode. While rewriting apps can be effective, incomplete implementations of bytecode rewriting results in a number of potential attacks [88]. Since repackaging assigns a different signature to a target app, it can also no longer be updated by the original issuer. Saint [9] extends Android to enable control of an app's behavior via app provider's policies. XmanDroid [11] also extends the monitoring mechanism of Android to prevent app-level *privilege escalation* attacks based on permission-based policies. ASM [89] provides an API that enables enforcement of app-specific security requirements. End-users typically lack expertise in devising policies and have to rely on general policies written by experts. By contrast, *SEALANT* provides finer-grain protection by automatically generating and enforcing what amounts to target-specific policies for a set of apps. DeepDroid [87] provides enterprise policy enforcement by applying dynamic memory instrumentation (i.e., rooting) to Android's runtime environment. SEPAR [15] automatically synthesizes security policies, which it also enforces through dynamic memory instrumentation. Rooting may introduce vulnerabilities and compatibility issues on custom ROM [18].

VII. Concluding Remarks

SEALANT is an integrated technique that monitors and protects ICC paths through which Android inter-app attacks can take place. *SEALANT*'s combination of static and dynamic analysis improves upon existing techniques in automatically identifying the vulnerable ICC paths between a set of apps, monitoring each instance of ICC to detect potential attacks, and empowering end-users to stop the attacks. Our evaluation demonstrates *SEALANT*'s effectiveness, efficiency, accuracy, scalability, and usability. Notably, we have shown that *SEALANT* outperforms existing alternatives in blocking inter-app attacks and can be applied in real-world scenarios, with a negligible performance overhead and a minor adoption barrier.

Several avenues of future work remain. *Analyzer* shares two limitations of static-analysis tools it leverages (i.e., IC3, COVERT, and FlowDroid). First, reflective calls are resolved only when their arguments are string constants. To this end, we will explore reflection analysis techniques [90]. Second, incomplete models of native methods and dynamically loaded code can cause unsoundness in our results. This can be remedied by leveraging additional sources of vulnerabilities [91] and dynamic analysis techniques [54], [92]. Inter-app attacks can also be launched via covert channels in the Android core system components and via kernel-controlled channels (e.g., *confused deputy* attacks over a local socket connection or *collusion* attacks over the file system). We can counter such attacks by combining our solution with kernel-level solutions (e.g, SELinux [93] and FlaskDroid [10]). Another direction for our work is to feed end-users' choices into a statistical model, to provide more specific guidance. Eventually, we can incorporate these techniques in designing applications [94], [95].

VIII. Acknowledgments

We appreciate the reviewers' helpful comments and the contribution of Ruhollah Shemirani in evaluating an earlier prototype of *SEALANT*. This work is supported by the U.S. National Science Foundation under award number 1618231.

REFERENCES

[1] "2012 Norton Cybercrime Report," http://www.norton.com/2012cybercrimereport, Symantec Corporation, 2012.

[2] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks," Technische Universität Darmstadt, Tech. Rep. TR-2011-04, 2011.

[4] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android App," in *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[5] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android Taint Flow Analysis for App Sets," in *Proceedings of the 3rd International Workshop on the State of the Art in Java Program Analysis*, 2014.

[6] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[7] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "COVERT: Compositional Analysis of Android Inter-App Permission Leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

[8] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android," in *Proceedings of the 9th International Conference on Availability, Reliability, and Security (ARES)*, 2014.

[9] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[10] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," in *Proceedings of the 22nd USENIX Conference on Security*.

[11] S. Bugiel, L. Davi, R. Dmitrienko, and T. Fischer, "Towards Taming Privilege-Escalation Attacks on Android," in *NDSS*, 2012.

[12] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *Proceedings of the Conference on Computer and Communications Security*, 2012.

[13] M. Dietz, S. Shekhar, D. S. Wallach, Y. Pisetsky, and A. Shu, "QUIRE: Lightweight Provenance for Smart Phone Operating Systems," in *Proceedings of the 20th USENIX Conference on Security*, 2011.

[14] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," in *Proceedings of the 22nd USENIX Conference on Security*.

[15] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Practical, Formal Synthesis and Automatic Enforcement of Security Policies for Android," in *Proceedings of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[16] S. Rasthofer, S. Arzt, and E. Bodden, "A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks," in *NDSS*, 2014.

[17] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek, "Automated Dynamic Enforcement of Synthesized Security Policies in Android," George Mason University, Tech. Rep. GMU-CS-TR-2015-5, 2015.

[18] "Is Rooting Your Phone Safe? The Security Risks of Rooting Devices," https://insights.samsung.com/2015/10/12/is-rooting-your-phone-safe-the-security-risks-of-rooting-devices, Samsung Electronics America.

[19] "DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android," https://github.com/secure-software-engineering/DroidBench, 2015.

[20] "Google Play," http://play.google.com/store/apps, Google, 2015.

[21] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, 2012.

[22] J. Garcia, D. Popescu, G. Safi, W. G. J. Halfond, and N. Medvidovic, "Identifying Message Flow in Distributed Event-Based Systems," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[23] G. Safi, A. Shahbazian, W. G. Halfond, and N. Medvidovic, "Detecting Event Anomalies in Event-Based Systems," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[24] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody, "Role-Based Access Control for Publish/Subscribe Middleware Architectures," in *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS)*, 2003.

[25] B. Shand, P. Pietzuch, I. Papagiannis, K. Moody, M. Migliavacca, D. M. Eyers, and J. Bacon, *Reasoning in Event-Based Distributed Systems*, 2011, ch. Security Policy and Information Sharing in Distributed Event-Based Systems.

[26] L. I. W. Pesonen, D. M. Eyers, and J. Bacon, "Encryption-Enforced Access Control in Dynamic Multi-Domain Publish/Subscribe Networks," in *Proceedings of the Inaugural International Conference on Distributed Event-based Systems (DEBS)*, 2007.

[27] M. Srivatsa, L. Liu, and A. Iyengar, "EventGuard: A System Architecture for Securing Publish-Subscribe Networks," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, pp. 10:1–10:40, 2011.

[28] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann, "Engineering Event-Based Systems with Scopes," in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[29] S. J. Templeton and K. E. Levitt, "Detecting Spoofed Packets," in *DARPA Information Survivability Conference and Exposition Proceedings*, 2003.

[30] "android.app | Android Developers." [Online]. Available: http://developer.android.com/reference/android/app/package-summary.html

[31] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android Permission Specification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[32] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps," in *Proceedings of the 35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2014.

[33] "Intents and Intent Filters | Android Developers." [Online]. Available: https://developer.android.com/guide/components/intents-filters.html

[34] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite Constant Propagation: Application to Android Inter-Component Communication Analysis," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[35] "Android Open Source Project," https://source.android.com, 2016.

[36] "Dashboards | Android Developers," https://developer.android.com/about/dashboards, 2016.

[37] "Run Apps on the Android Emulator | Android Studio," https://developer.android.com/studio/run/emulator.html, 2016.

[38] "Android Debug Bridge," http://developer.android.com/tools/help/adb.html, 2016.

[39] "Running Builds–Booting into Fastboot Mode." [Online]. Available: https://source.android.com/source/running.html#booting-into-fastboot-mode

[40] "IccTA," http://sites.google.com/site/icctawebpage, 2016.

[41] "F-Droid – Free and Open Source Android App Repository," https://f-droid.org, 2016.

[42] "Bazaar," http://cafebazaar.ir, 2016.

[43] "apps-for-android — Google Code Archive," https://code.google.com/archive/p/apps-for-android/, 2016.

[44] "Sourcecodester.com," http://www.sourcecodester.com/android, 2016.

[45] "logcat Command-line Tool | Android Studio," https://developer.android.com/studio/command-line/logcat.html, 2015.

[46] "So Many Apps, So Much Time For Entertainment," http://www.nielsen.com/us/en/insights/news/2015/so-many-apps-so-much-more-time-for-entertainment.html, 2015.

[47] "monkeyrunner | Android Studio," https://developer.android.com/studio/test/monkeyrunner, 2016.

[48] "Download APK Android Apps and Games," http://www.appsapk.com, 2016.

[49] "UI/Application Exerciser Monkey | Android Studio," http://developer.android.com/tools/help/monkey.html, 2016.

[50] "Keeping Your App Responsive | Android Developers," http://developer.android.com/training/articles/perf-anr.html, 2016.

[51] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android Applications," Dept. of Computer Science, University of Maryland, Tech. Rep., 2009.

[52] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST)*, 2012.

[53] Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," in *NDSS*, 2013.

[54] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[55] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *NDSS*, 2012.

[56] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," in *NDSS*, 2015.

[57] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection," in *Proceedings of the ACM SIGSAC Conference on Computer Communications Security (CCS)*, 2013.

[58] P. P. Chan, L. C. Hui, and S. M. Yiu, "DroidChecker: Analyzing Android Applications for Capability Leak," in *Proceedings of the 5th Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[59] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[60] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *Proceedings of the Mobile Security Technologies (MoST)*, 2012.

[61] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static Control-Flow Analysis of User-driven Callbacks in Android Applications," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[62] C. Mann and A. Starostin, "A Framework for Static Detection of Privacy Leaks in Android Applications," in *Proceedings of the 27th Symposium on Applied Computing (SAC)*, 2012.

[63] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[64] L. Li, A. Bartel, J. Klein, and Y. L. Traon, "Automatically Exploiting Potential Component Leaks in Android Applications," in *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM)*, 2014.

[65] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-App Security Analysis with FUSE: Statically Detecting Android App Collusion," in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[66] F. Shen, N. Vishnubhotla, C. Todarka, M. Arora, B. Dhandapani, E. J. Lehner, S. Y. Ko, and L. Ziarek, "Information Flows As a Permission Mechanism," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014.

[67] K. Choi and B.-M. Chang, "A Type and Effect System for Activation Flow of Components in Android Programs," *Information Processing Letters*, vol. 114, no. 11, pp. 620–627, November 2014.

[68] S. Bartsch, B. Berger, M. Bunke, and K. Sohr, "The Transitivity-of-Trust Problem in Android Application Interaction," in *Proceedings of the 8th International Conference on Availability, Reliability and Security*, 2013.

[69] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "DroidAlarm: An All-Sided Static Analysis Tool for Android Privilege-Escalation Malware," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*, 2013.

[70] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative Verification of Information Flow for a High-Assurance App Store," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[71] J. Wu, T. Cui, T. Ban, S. Guo, and L. Cui, "PaddyFrog: Systematically Detecting Confused Deputy Vulnerability in Android Applications," *Security and Communication Network*, vol. 8, no. 13, September 2015.

[72] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. L. Traon, "Combining Static Analysis with Probabilistic Models to Enable Market-Scale Android Inter-Component Analysis," in *Proceedings of the 43rd Symposium on Principles of Programming Languages*, 2016.

[73] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The Impact of Vendor Customizations on Android Security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, 2013.

[74] K. O. Elish, D. Yao, and B. G. Ryder, "On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions," in *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2015.

[75] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard: Enforcing User Requirements on Android Apps," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[76] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[77] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications," in *Proceedings of the Mobile Security Technologies*, 2012.

[78] B. Davis and H. Chen, "RetroSkeleton: Retrofitting Android Apps," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2013.

[79] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[80] K. Z. Chen, N. Johnson, S. Dai, K. Macnamara, T. Magrino, E. Wu, M. Rinard, and D. Song, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *NDSS*, 2013.

[81] M. Zhang and H. Yin, "Appsealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications," in *NDSS*, 2014.

[82] Z. Zhao and F. C. Colon Osono, ""TrustDroid™": Preventing the Use of Smartphones for Information Leaking in Corporate Networks Through the Used of Static Analysis Taint Tracking," in *Proceedings of the 7th International Conference on Malicious and Unwanted Software*, 2012.

[83] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[84] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff, "Kynoid: Real-Time Enforcement of Fine-Grained, User-Defined, and Data-Centric Security Policies for Android," in *Proceedings of the 6th IFIP International Conference on Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent*, 2012.

[85] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proceedings of the 20th USENIX Conference on Security (SEC)*, 2011.

[86] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These Aren't the Droids You're Looking for: Retrofitting Android to Protect Data from Imperious Applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[87] X. Wang, K. Sun, Y. Wang, and J. Jing, "DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices," in *NDSS*, 2015.

[88] H. Hao, V. Singh, and W. Du, "On the Effectiveness of API-Level Access Control Using Bytecode Rewriting in Android," in *Proceedings of the 8th Symposium on Information, Computer and Communications Security*.

[89] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A Programmable Interface for Extending Android Security," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[90] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[91] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets," in *NDSS*, 2012.

[92] Y. K. Lee, J. Bang, J. Garcia, and N. Medvidovic, "ViVA: A Visualization and Analysis Tool for Distributed Event-based Systems," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[93] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing Android-Powered Mobile Devices Using SELinux," *IEEE Security & Privacy*, vol. 8, no. 3, 2010.

[94] A. Shahbazian, G. Edwards, and N. Medvidovic, "An End-to-End Domain Specific Modeling and Analysis Platform," in *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, 2016.

[95] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner, "Automated Extraction of Rich Software Models from Limited System Information," in *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016.