SEALANT: Preventing Inter-Application Attacks in Android


Youn Kyu Lee, Ruhollah Shemirani, Jae young Bang, Arman Shahbazian, Gholamreza Safi, and Nenad Medvidovic
Computer Science Department, University of Southern California
941 Bloom Walk, Los Angeles, California, USA 90089
{younkyul, rshemira, jaeyounb, armansha, gsafi, neno}@usc.edu

ABSTRACT
Android is the most widely adopted mobile operating system today. However, Android has a major security weakness that resides in its communication model via which apps interact with each other. This weakness can expose the end-users’ personal, sensitive information to attackers. Specifically, malicious apps may exploit Android’s communication model to manipulate other apps into performing unintended operations and to steal end-users’ data. Even the latest version of Android does not provide framework-level protection from those attacks. As a result, the end-users must defend their device and data, which is difficult since the apps interact in the background, while the behaviors of the malicious apps often appear ordinary and harmless. This paper presents SEALANT, an integrated technique that leverages static analysis of app code to automatically infer vulnerable communication channels via which attacks may take place between a group of apps installed on the same device; then, at runtime, SEALANT monitors all inter-app communications through those channels to prevent the attacks. We evaluated SEALANT in four ways and observed that (1) it blocks inter-app attacks with 100% accuracy in a controlled experiment using real-world applications, (2) it suffers from fewer false alarms than existing techniques in several representative scenarios, (3) the performance overhead added by SEALANT’s runtime monitoring is negligible, and (4) adopting SEALANT does not present a significant challenge to end-users.

1. INTRODUCTION
The modern mobile computing devices store their users’ private information, from phone numbers and addresses to highly sensitive information such as banking credentials. Mobile operating systems (OSs) implement various security measures to protect the sensitive information from attackers, yet many of the OSs still have known security vulnerabilities. Google’s Android shares these problems [23]. Android is particularly important for two reasons. First, it is the most widely used mobile OS today [13]. Second, it provides a native development framework in addition to the base OS services, in the process also raising the OS-level vulnerabilities to the level of application (“app”) development.

This paper targets a vulnerability in the design of Android’s communication model in which app components communicate by exchanging messages called intents. An intent is similar to a remote procedure invocation with optional arguments [33]. Android allows intent exchanges between components in a single app (intra-app) or across different apps (inter-app). Unless app developers carefully control the incoming and outgoing intents, inter-component communication (ICC) via intent exchange can expose a vulnerable surface to several well-known security attacks: intent spoofing [33], unauthorized intent receipt [33], and privilege escalation [30]. The different versions of Android have not provided framework-level protection from those attacks to date, but have expected developers to build protective measures in each app. This exposes the end-users to the chance of becoming victims of the attacks.

It is challenging for a user of Android to distinguish a malicious app from the safe, benign apps. This is especially the case with non-expert end-users because, in many cases, a malicious app disguises itself as benign, and will send and receive intents in a way that appears as if those are ordinary message exchanges. For example, a malicious app can silently eavesdrop on the data carried in a broadcast intent (unauthorized intent receipt) without interrupting the broadcast. It would be hard for an end-user to detect such a passive attack and realize that the malicious app captured her data. Such vulnerabilities are already known, but the threats still persist.

A large volume of research has been conducted with the aim of assisting the detection of malicious apps and protecting the end-users’ personal data. However, the existing work has primarily targeted users who have expertise in Android systems security (e.g., app developers) [27, 31–33, 42, 46, 49, 56], making the resulting techniques overly complicated for average end-users to adopt. A more recent body of research has proposed tools that assist non-expert end-users of Android devices. However, those tools either only target limited types of inter-app attacks [31] or suffer from frequent “false alarms” [28, 31].

In this paper, we present SEALANT (Security for End-users of Android via Light-weight ANalysis Techniques), an integrated technique that protects vulnerable ICC paths on which malicious inter-app attacks can take place at runtime. SEALANT automatically identifies vulnerable ICC paths between a given set of apps. It then inspects each intent sent via those paths to detect potential attacks, and enables end-users to block the attacks before they actually take place. SEALANT is distinguished from the existing research because (1) it prevents all three well-known Android inter-app attacks—intent spoofing, unauthorized intent receipt, and privilege escalation—and (2) it causes fewer false alarms than the existing techniques.

We have developed two tools that together implement SEALANT: (1) Analyzer identifies vulnerable ICC paths between a group of Android apps by performing static analysis on the app bytecode; (2) Interceptor is an extension to the Android framework that manages inter-app intent exchanges. Together, these tools enable an end-user to either allow or block vulnerable inter-app intents on-the-fly.

In developing SEALANT (in particular, its Interceptor), we elected to modify the Android framework over two other alternatives—instrumenting the bytecode of the apps installed on a device and acquiring administrator privileges, i.e., “rooting” the device. We made this choice for two reasons. First, once our approach is applied to a device, it does not require altering any of the installed apps. Second, rooting is undesirable as it introduces vulnerabilities into the system that may be more serious than the ICC vulnerabilities [15].
We have evaluated SEALANT in four different ways. First, we assessed SEALANT’s effectiveness via a comparative analysis against existing techniques. Our solution suffered from fewer false alarms while blocking the same or greater number of vulnerable ICC paths. Second, we performed a case study targeting Analyzer’s ability to identify all vulnerable ICC paths between a set of apps, and Interceptor’s ability to selectively block those paths. To this end, we created a malware test suite that simulates inter-app attacks on real-world apps. The test suite includes 19 apps that we randomly selected from (1) a pool of apps previously identified as vulnerable to inter-app attacks [28] and (2) an open-source testing ground that contains apps with data leaks [9], designed to benchmark techniques like SEALANT. Analyzer was able to identify the vulnerable ICC paths with perfect accuracy, while Interceptor was able to capture and block all of the identified paths. Third, we evaluated SEALANT’s runtime performance by measuring the resource overhead imposed by Interceptor’s intent inspections. The additional amount of resources Interceptor requires is nominal. Finally, we performed a user study and survey involving 189 Android end-users in employing SEALANT to block or allow vulnerable inter-app intent exchanges. Users were able to make appropriate choices in general and did not find the choices required SEALANT burdensome.

This paper makes four contributions:

1. SEALANT, a technique that enables Android end-users to protect their devices from three types of inter-app attacks, namely, intent spoofing, unauthorized intent receipt, and privilege escalation;
2. Analyzer, a tool that finds vulnerable ICC paths between a set of apps along which malicious intents can be sent;
3. Interceptor, an Android framework extension that automatically detects potentially malicious intents at runtime and enables end-users to block them if considered malicious; and
4. the results of four different evaluations of SEALANT that involve real-world Android apps, compare SEALANT to existing alternatives, and engage a number of end-users.

It is important to note that, in this paper, we make two assumptions in order to manage the scope of our research. First, the bundled apps that come with a factory-setting Android device are not malicious. Second, we do not consider other unorthodox channels of inter-app communication that do not use intents such as file locks (as in [31]), sockets, or the file system (as in [32]).

The rest of the paper is organized as follows. In Section 2, we describe the architectures of SEALANT’s Analyzer and Interceptor in Section 3, and their implementation in Section 4. We present the four evaluations of SEALANT in Section 5. A discussion of related work is provided in Section 6, and our conclusions in Section 7.

2. MOTIVATING EXAMPLE

In this section, we present examples of the three types of inter-app attacks that exploit inter-component communication (ICC) between Android apps and that are targeted by SEALANT: (1) intent spoofing, (2) unauthorized intent receipt, and (3) privilege escalation.

Figure 1(a) and Listings 1 and 2 depict intent spoofing. As depicted in Figure 1(a), Component C belonging to the malicious app MalApp1 can send an intent to Component B belonging to the victim app VicApp1 in order to exploit the functionality of VicApp1 (e.g., sending an SMS). Listing 1 shows where VicApp1’s vulnerability resides. Whenever Component B is initiated, sendTextMessage() will be subsequently executed, and it will then dispatch an SMS to the number provided by an intent. Even if Component B may have been designed to be triggered by an intent from another component belonging to the same app (e.g., Component A), malicious apps can exploit this vulnerable surface in order to send an SMS to unintended numbers. Listing 2 illustrates how MalApp1 performs an attack by sending an intent containing an unsafe number to Component B. In Android, a component can trigger another component by sending an explicit intent.1 As depicted in Listing 2, Component C sends an explicit intent that designates Component B as a destination and contains an unsafe number. In this case, we consider that a vulnerable ICC path exists from Component C to Component B.

Figure 1(b) and Listing 3 illustrate unauthorized intent receipt. As depicted in Figure 1(b), although Component A of VicApp2 has been designed to broadcast an intent to deliver information to other components belonging to the same app (e.g., Component B), a malicious Component D belonging to MalApp2 is able to eavesdrop on the intent. Listing 3 shows that, when a user clicks a button, Component A broadcasts an implicit intent that contains its action attribute as showID, and a device ID which is considered to be sensitive data. As long as an intent is broadcast without any permission restrictions, any component that declares the attributes matching those of intent can receive that intent. Hence, by declaring an action attribute showID, Component C is able to eavesdrop on the intent and obtain VicApp2’s device ID. In this case, we consider that a vulnerable ICC path exists from Component A to Component C.

Figure 1(c) depicts the privilege escalation attack. Component B belonging to VicApp3 contains a sensitive API that is protected with permission P1. While Component D belonging to VicApp4 has been granted the permission P1, Component E belonging to MalApp3 has not, which means that Component E is not able to directly access Component B. Nevertheless, Component E can reach Component B in an indirect way, via Component D which is not protected by any permissions. Even if Component D has been designed to be triggered by an intent from another component belonging to the same app (e.g., Component C), Component E is also able to trigger D via an explicit intent. By triggering Component B, Component E is able to access Component B without acquiring the necessary permission P1. In this case, we consider that a transitive vulnerable ICC path exists from Component E to D to B.

Listing 1: Component B of VicApp1

```java
public class Component_B extends Activity {
    public void onStart() {
        Intent i = getIntent();
        String smsNum = i.getStringExtra("SmsNum");
        String smsText = i.getStringExtra("SmsText");
        SmsManager mSmsManager = SmsManager.getDefault();
        mSmsManager.sendTextMessage(smsNum, null, smsText, null, null);
    }
}
```

1 An explicit intent specifies its destination component, while an implicit intent does not name a destination component, but instead declares attributes, such as action, category, and data type [14].
### Listing 2: Component C of MalApp1
```java
public class Component_C extends Activity {
    public void onCreate (Bundle savedInstanceState) {
        Intent i = new Intent();
        i.setComponent(new ComponentName("com.VicApp1", 
        i.addCategory("android.intent.category.DEFAULT");
        i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        i.putExtra("SmsNum", "123-456-7890");
        startActivity(i);
    }
}
```

### Listing 3: Component A of VicApp2
```java
public class Component_A extends Activity {
    public void OnClickListener(View v) {
        TelephonyManager telmanager = (TelephonyManager)
        getSystemService(Context.TELEPHONY_SERVICE);
        String id = telmanager.getDeviceId();
        Intent i = new Intent("showID");
        i.putExtra("DeviceID", id);
        sendBroadcast(i);
    }
}
```

The above examples demonstrate that the way these attacks are administered is not different from ordinary intent exchanges between apps. For instance, recall the first example from Listing 2: the attack took place when MalApp1 sent an intent by calling startActivity(), but this is a standard Android method for sending an intent. This characteristic of inter-app attacks makes their identification and restriction especially challenging. Moreover, since an ICC can be performed in an essentially invisible way (e.g., sendBroadcast()), it is difficult for end-users to recognize when the attacks are actually committed. Finally, even though the Android framework provides a security mechanism for restricting ICC through permission enforcement [21, a malicious app can bypass the permission check as discussed in the context of Figure 1(c).

Although these types of security violations have been studied in domains such as computer networks and distributed event-based systems [29, 36, 47, 52–54], those techniques cannot be directly applied to Android due to the specifics of its communication mechanism and features. For example, role-based access control [29, 52], a popular technique for securing event-based systems, has been applied on Android as a form of permission grants; however, it can be violated by privilege escalation attacks. Encryption [47, 53], another popular technique, is not a good fit for Android due to encryption-key distribution issues and limited mobile resources.

### 3. SEALANT

This section introduces SEALANT, a technique that automatically identifies vulnerable ICC paths between Android apps, and enables an end-user to control the ICCs on those paths during runtime.

#### 3.1 Overview

SEALANT recognizes each instance of ICC as a relation between (1) a sender, (2) a receiver, and (3) 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 between them. SEALANT builds an ICC graph in which each vertex indicates a component, and two vertices are connected if an ICC relation between them exists. Edges are directed from a sender to a receiver.

SEALANT extracts all possible vulnerable ICC paths in a given ICC graph, and monitors each of these paths during runtime. When an instance of ICC at runtime matches one of the extracted vulnerable paths, SEALANT may block it based on the end-user’s choice.

As presented in Figure 2, two key components are cooperating in SEALANT: (1) Analyzer leverages static analysis to generate a list of vulnerable ICC paths between apps; (2) Interceptor, an extension to the Android framework, performs runtime monitoring and enables advanced ICC control such as blocking of specific ICCs identified by Analyzer. The overall process of SEALANT is as follows:

1. **Analyzer** processes the APK files of the installed apps and identifies the vulnerable ICC paths between them.
2. **Analyzer** contacts an end-user to confirm which ICC paths should be monitored among the identified vulnerable paths. This information can take the form of specific components that are connected by a vulnerable path (for expert users) or simply the apps that are connected by such a path (for non-experts).
3. **Analyzer** feeds the information about vulnerable ICC paths to the **Interceptor** in a pre-defined format, called SEALANT List.
4. At runtime, whenever an intent is sent, **Interceptor** captures the detailed information of the corresponding ICC path (e.g., sender component’s name) from Android’s ActivityManager. If the captured path information matches one of the vulnerable ICC paths in the SEALANT List, **Interceptor** contacts the end-user to determine whether to propagate the intent.
5. 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.

#### 3.2 Analyzer

Figure 3 shows an overview of Analyzer’s operation. Analyzer performs a static analysis that uses APK files of installed apps as its input, and exports the SEALANT List as its output. It operates in three phases: (1) analyze target apps, (2) find vulnerable paths, and (3) generate SEALANT List. Analyzer is novel in that it creates a single compositional program model from multiple apps, analyzes that model for three vulnerability types in a single pass by focusing on ICCs and ignoring other aspects of constituent apps, and allows reusing previously completed analysis when a new app is added.

#### 3.2.1 Phase I: Analyze Target Apps

In this phase, Analyzer first extracts each app’s architectural information from the manifest file and bytecode that comprise the APK file, and then transforms them into ICC paths.

2 APK is an archive file format that distributes and installs Android application software on top of the Android operating system.
3 ActivityManager is the Android component that governs ICC.
Analyzer leverages COVERT [27], a static analysis technique for ICC call method and reuses the prior analysis of the remaining apps. Analyzer components and the edges representing intents of the ICC relations: intra-component paths and architectural objects of an app.

The ICO call method is a standard Android method for sending or receiving an intent (e.g., startActivity()). When apps are installed or updated subsequent to running Analyzer, Analyzer extracts only the architectural information from the new/modified apps and reuses the prior analysis of the remaining apps.

3.2.2 Phase II: Find Vulnerable Paths

In this phase, Analyzer first extracts the information regarding intents that are sent or received by each component from the ICO graph, and stores it in two tables. This pre-processing enables Analyzer to avoid redundant traversal of the ICO graph, and consequently makes our approach more scalable. The information in each table can be retrieved by using two methods:

- **LookUpIntent(i)** returns all components that send an intent of type i.
- **LookUpComponent(c)** returns all the explicit or implicit intents that can be received by component c.

Analyzer then implements Algorithm 1 on the extracted graph in order to identify vulnerable ICC paths. An ICC path is vulnerable if a vulnerable component is present at one or both of its ends. Since a transitive ICC path to a vulnerable component may also exist between apps (recall Figure 1(c), Analyzer recursively identifies a set of connected ICC paths that can access a vulnerable component by calling the PathFinder method (Algorithm 2).

Algorithms 1 and 2 use sets of components, intents, and an ICC graph as inputs, and report the set of vulnerable ICC paths. Algorithm 1 iterates over each component (c; lines 3-13) and considers two cases. The first case (line 4) is when cvulType is Active (i.e., intent spoofing and privilege escalation). In this case, Algorithm 1 calls PathFinder (line 6). As depicted in Algorithm 2, PathFinder iterates over each intent j that can be received by c. For each intent j, PathFinder checks if c and the sender of j belong to different packages and if the sender component has not already been visited in the path (line 2). If all these conditions are satisfied, PathFinder determines the type of attack by calling PermCompare (line 3).

PermCompare(A,B,M) is a method that returns a type of attack by comparing the permissions of two components (A and B). If component B has a permission that component A does not have, and if the permission is required to use method M [25], PermCompare returns the attack type as privilege escalation; otherwise, it returns intent spoofing. If M is null, PermCompare returns null.

Once a type of attack is determined, PathFinder appends the information of the path (i.e., attack type, name of sensitive method, sender, receiver, and intent’s attributes) to a variable, P, that tracks

**Algorithm 1: Identifying vulnerable ICC paths**

| Input: | Comps ← a set of components, Intents ← a set of intents, and an ICC graph |
| Output: | VulPaths |
| 1. | Let T be a variable for an attack type |
| 2. | Let P be a variable for a path between two components |
| 3. | foreach c ∈ Comps do |
| 4. | if cvulType = “Active” then |
| 5. | set P to 0 |
| 6. | PathFinder(c, P) |
| 7. | if cvulType = “Passive” then |
| 8. | foreach i ∈ Intents do |
| 9. | if (cName = cSendComp) then |
| 10. | foreach d ∈ LookUpIntent(i) do |
| 11. | if (dPermReq = cPermReq) then |
| 12. | T := “unauthorized intent receipt” |
| 13. | add (T, cvulMethod, cName, cPkg, dName, dPermReq, d) to VulPaths |

**Algorithm 2: PathFinder**

| Input: | c ∈ Comps, P |
| Output: | (cont.) |
| 1. | foreach j ∈ LookUpComponent(c) do |
| 2. | if (cPermReq ≠ jSendPkg) & & P doesn’t contain jSendComp then |
| 3. | T := PermCompare(jSendComp, cName, cvulMethod) |
| 4. | add (T, cvulMethod, jSendComp, jSendPkg, cName, cPkg, j) to P |
| 5. | add P to VulPaths |
| 6. | PathFinder(jSendComp, P) |

We indicate an attribute y of entity x with xy.
the vulnerable edges between components (line 4). P is then added to the set VulPaths (line 5) which collects all the detected vulnerable ICC paths. To identify transitive ICC paths, PathFinder recursively identifies the other components that are connected to a vulnerable component through edges on the ICC graph. PathFinder adds each path that is introduced by these transitive edges to VulPath. PathFinder stops its processing when it visits all the transitive connected components to c or reaches an already visited component.

The second case (lines 7-13 in Algorithm 1) deals with the situation when the vulnerability of c is passive (i.e., unauthorized intent receipt). Algorithm 1 finds an intent i that is sent from c (line 8-9). If found, Algorithm 1 looks up a component d that can receive i and checks whether d and c belong to different packages (line 11). If so, the type of attack is set to unauthorized intent receipt and the information about the path is added to VulPaths (lines 12-13).

For the examples in Figure 1, Algorithm 1 will return two vulnerable ICC paths (from Component C in MalApp1 to B in VicApp1, and from Component A in VicApp2 to C in MalApp2), as well as one vulnerable transitive ICC path (from Component E in MalApp3 to D in VicApp4 to B in VicApp3).

### 3.2.3 Phase III: Generate SEALANT List

As a last step, Analyzer generates the SEALANT List based on the output of the previous phase, VulPaths. Analyzer first normalizes the output by checking for redundant paths. Thereafter, it transforms the information about identified paths into a pre-defined format that is compatible with SEALANT’s Interceptor component.

The SEALANT List contains two types of information: (1) independent vulnerable ICC paths, each of which comprises two components connected by a single edge in the ICC graph, and their information (e.g., component names); and (2) independent vulnerable transitive ICC paths, each of which comprises multiple ICC paths listed in the order of their connection in the ICC graph.

### 3.3 Interceptor

Interceptor monitors and analyzes each instance of inter-component communication (ICC). Whenever an ICC is requested, Interceptor checks whether it is specified in the SEALANT List. To increase the accuracy in blocking ICCs, Interceptor distinguishes each ICC instance based on its sender, receiver, and intent. To increase the effectiveness of blocking, Interceptor maintains and applies previously made choices until they are modified.

#### 3.3.1 Interceptor’s Architecture

Interceptor extends the Android framework by four components, as depicted in Figure 4. Three components—Blocker, ChoiceDatabase, and ListProvider—are newly added, while one—ActivityManager—is a modification of an existing Android component.

Blocker performs Interceptor’s core functionalities: monitoring, matching, and blocking. 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. Blocker is also responsible for notifying the end-user about the requested ICC.

ActivityManager is an extension of a standard Android component, which controls every instance of ICC processed through the Android framework. The original role of ActivityManager is routing each requested ICC by collaborating with other standard Android components (e.g., PackageManager and PowerManager). We extended ActivityManager to capture the information of each ICC instance, share the information with Blocker, and block a particular instance of ICC upon Blocker’s request. The information about ICC instances comprises identities of the sender and receiver components, and of the intent (e.g., action, category, and data type).

ChoiceDatabase stores an end-user’s choices for each vulnerable ICC path. Whenever a choice is made, it is stored with the corresponding ICC path information. Stored choices are automatically applied when the same ICC is requested, and can be also removed upon the end-user’s request. When a new SEALANT List is imported to the framework, ChoiceDatabase expunges every stored choice.

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.

#### 3.3.2 Interceptor’s Operation

Figure 4 illustrates the interaction among Interceptor’s four components in blocking an ICC. For clarity, the depicted six-step scenario is based on the example from Listings 1 and 2. However, this scenario is reflective of Interceptor’s operation in general.

1. When Component C of MalApp1 tries to send intent i by calling startActivity(), the request is routed to ActivityManager.
2. ActivityManager extracts the sender’s (i.e., Component C’s) information and searches for the component that is allowed to receive intent i. If a receiver is identified (i.e., Component B of VicApp1), ActivityManager passes the information about the ICC to Blocker and pauses the ICC’s routing.
3. After receiving the information about the ICC, Blocker first examines the previous choices stored in ChoiceDatabase. If a choice for the corresponding ICC already exists, Blocker induces ActivityManager to act (block or route the requested ICC) based on the stored choice, 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 the information captured in the SEALANT List, Blocker will ask the end-user whether the ICC should be blocked. The end-user is given four options: (1) allow once, (2) block once, (3) allow always, and (4) block always. If the user selects option (3) or (4), her choice will be stored in ChoiceDatabase for future use.
6. If the end-user chooses to allow the requested ICC, Blocker will force ActivityManager to send intent \(i\) to Component B. Otherwise, Blocker will instruct ActivityManager to forego the sending of intent \(i\) to Component B.

3.3.3 Interceptor’s Strategy for Blocking ICC

Interceptor’s blocking process is performed between the time when an intent is first requested and the time when the intent is actually dispatched to its destination. Interceptor may thus cause a delay in processing a requested intent. This delay may be exacerbated based on the number of vulnerable ICC paths specified in the SEALANT List. However, since Android’s ICC is performed via asynchronous API calls, we hypothesize that the delay will not significantly impact the system’s operation. In Section 5.3, we empirically evaluate this 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. However, since Android implements ICC asynchronously, those apps will simply “skip” the corresponding operation without causing runtime crashes.

Interceptor’s ICC control strategy is distinguished from existing techniques that target inter-app attacks due to its finer-grained characterization of ICC paths based on (1) sender, (2) receiver, and (3) intent. Since Interceptor distinguishes ICC paths based on their attributes (e.g., name of sender component, data scheme of intent, and name of receiver’s package), it enables more elaborate control in blocking ICCs. For example, it can block a malicious ICC without hampering a benign ICC, even when both are initiated by the intent containing the same attributes. Detailed evaluation of Interceptor’s ICC control strategy is discussed in Sections 5.1 and 5.2.

To block a vulnerable transitive ICC, Interceptor makes use of the ICC paths that comprise each vulnerable transitive ICC path in the SEALANT List. When the requested ICC matches the first path of a vulnerable transitive ICC path, Interceptor sets its transitive_flag to true. This flag remains true as long as the subsequently requested ICCs match the subsequent paths in the vulnerable transitive ICC 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 Component \(E \rightarrow D \rightarrow B\) is already in the SEALANT List. If \(E\) launches \(D\) via an intent, Interceptor will set transitive_flag to true. Then, if \(D\) subsequently launches \(B\) via an intent, Interceptor will alert the end-user and reset the flag.

4. IMPLEMENTATION

We have implemented a proof-of-concept of SEALANT in order to demonstrate that our solution can be applied to real-world Android devices and apps, and to evaluate its accuracy and performance. SEALANT’s Analyzer is implemented as a Java application, and Interceptor is implemented as an extension to the Android framework.

SEALANT’s Analyzer was built on top of COVERT [6]. Specifically, Analyzer employs COVERT’s model extractor module to obtain app information and identify vulnerable components. We implemented all other parts of Analyzer in Java. Analyzer runs as a stand-alone Java application while using a set of APKs as an input and exporting the SEALANT List in the predefined XML format.

We implemented SEALANT’s Interceptor on top of the source code from the Android Open Source Project (AOSP) 4.4.4 KitKat [3], since KitKat is currently the most popular version of Android [7]. We directly modified the source code of 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, no standard components or methods were removed. 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 [2] and on the Google Nexus 7 device.

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

To use SEALANT, a device owner must compile Interceptor’s source code along with the provided drivers, and install the image files obtained through compilation using two freely available Android tools: Android debug bridge [1] and Fastboot [19]. SEALANT’s source code, required drivers, and compiled tools are available at http://softarch.usc.edu/sealant/.

5. EVALUATION

We evaluate SEALANT’s effectiveness as compared to competing techniques (Section 5.1), accuracy when applied to real-world apps (Section 5.2), performance (Section 5.3), and usability (Section 5.4).

5.1 Effectiveness

We compare SEALANT’s effectiveness to two existing techniques that share the goal of protecting end-users from inter-app attacks: DroidGuard [28] and XmanDroid [30, 31]. DroidGuard identifies vulnerable surfaces of a set of apps installed on an Android device via static analysis. In order to address the identified vulnerabilities, it directly inserts policy-enforcement code into the vulnerable apps. For example, in the scenario depicted in Figure 1(b), DroidGuard would identify the vulnerabilities of Component A (i.e., sendBroadcast()) and place enforcement code that wraps sendBroadcast(). This ensures that whenever sendBroadcast() sends an intent, the end-user will be notified and will be able to choose whether to block the intent exchange. XmanDroid is a technique that only targets privilege escalation attacks by leveraging an extension to the Android framework. XmanDroid enables an end-user to pre-define a list of ICC restriction policies, and it automatically blocks ICCs that match any one of those policies. For example, XmanDroid can have the following policy: An app that can obtain location information must not communicate with an app that has outgoing network access [30].

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 DroidGuard is missing certain features covered by the underlying technique (e.g., the policy enforcement module). In Section 5.2, we do evaluate SEALANT directly against one of the implemented features of DroidGuard. We tried unsuccessfully to build an installation of XmanDroid on several recent versions of Android. Unfortunately, given the number of Android releases, continuing with this strategy proved impractical. For these reasons, we decided to analytically compare the three techniques, relying on the published algorithms of DroidGuard [28] and XmanDroid [30, 31].

5.1.1 Comparison with DroidGuard

SEALANT raises fewer false inter-app attack alarms to end-users compared to DroidGuard. We validate this by projecting the number of inter-app attack alarms each of the two techniques would raise...
in the scenario depicted in Figure 1(b). We define the following probabilistic random variables in order to illustrate the projection.

- \( N \) is the number of user interactions with a device in any given period of time.
- \( V \) is the random variable for user device containing VicApp2.
- \( P(V) \) is the probability that a user has VicApp2 on her device.
- \( M \) is the random variable for user device containing MalApp2 that can receive intent \( i \) from Component A in VicApp2.
- \( P(M) \) is the probability that a user has MalApp2 on her device.
- \( P(E) \) is the probability that a user interacts with her device and triggers Component A to broadcast intent \( i \).
- \( P(I) \) is the probability that MalApp2 eavesdrops on intent \( i \).
- \( A \) is the event that raises an alarm. \( A_{SLT} \) indicates an alarm from SEALANT, and \( A_{DG} \) indicates an alarm from DroidGuard.

The expected value for an alarm raised by DroidGuard is:

\[
E(A_{DG}) = N \cdot P(V, E)
\]

The expected value for an alarm raised by SEALANT is:

\[
E(A_{SLT}) = N \cdot P(V, M, E, I)
\]

\( E(A_{SLT}) \) can never be higher than \( E(A_{DG}) \) for the following reasons. Assuming \( M = I \) (i.e., MalApp2 can always eavesdrop on intent \( i \)), the expected value for SEALANT raising an alarm will be

\[
E(A_{SLT}) = N \cdot P(V, M, E, I).
\]

Since \( 0 \leq P(M) \leq 1 \), we can infer the following two cases regarding the dependency of \( M \) on \( V \):

1. \( E(A_{SLT}) = E(A_{DG}) \) if \( M \) is completely dependent on \( V \) (i.e., \( M = V \)). However, this is unlikely since there is no guarantee that the two different apps are installed on the same device.
2. \( E(A_{SLT}) = N \cdot P(V, E) \cdot P(M) \) if \( M \) is independent of \( V \). This means that DroidGuard will raise \( N \cdot E(A_{DG})(1 - P(M)) \) more alarms than SEALANT. As \( P(M) \) converges to zero, the rate at which DroidGuard raises a false positive alarm will increase.

Even if the device did not have an app that receives intent \( i \), DroidGuard would raise an alarm for every broadcast of intent \( i \) from Component A. For example, even for a safe, intended intent exchange within VicApp2 such as sending intent \( i \) from Component A to B, DroidGuard will raise the (false positive) alarm. By contrast, SEALANT raises an alarm only when intent \( i \) between Component A and C is initiated. SEALANT filters only the vulnerable ICCs using the path information which includes the ICC’s sender and receiver.

SEALANT also allows end-users to silence alarms that they do not wish to receive. Suppose that an end-user using DroidGuard blocked an ICC when she received an alarm in the scenario depicted in Figure 1(b), in which an ICC was initiated from sendBroadcast() of Component A. Regardless of the choice she made, for each ICC initiated subsequently by sendBroadcast(), DroidGuard would raise another alarm, even if the user preferred not to receive new alarms for that particular ICC. That is because DroidGuard does not have a mechanism to remember end-users’ choices. The mechanism employed by DroidGuard to detect potentially vulnerable ICCs—attaching monitoring code directly to target apps’ source code—is part of the reason why DroidGuard cannot dynamically adjust its behavior. Unlike DroidGuard, SEALANT can store the end-users’ choices and allow end-users to manage their subscriptions to different types of alarms (recall Section 3.3.2).

5.1.2 Comparison with XmanDroid

SEALANT suffers from fewer false negatives than XmanDroid. The detection mechanism of XmanDroid requires an end-user to explicitly specify policies indicating the types of inter-app attacks she wishes to detect and ICC paths to monitor at runtime. This carries the risk of not expecting to define critical inter-app attacks. Recall the privilege escalation attack scenario depicted in Figure 1(c).

When Component E in MalApp3 requests an ICC to access D in VicApp4, XmanDroid raises an alarm only when intent \( i \) and \( D \) match policies specifying the sender and receiver permission combinations that should be blocked. However, this mechanism 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 MalApp4 in the scenario depicted in Figure 1(c), and the device had another app, BenignApp, which holds identical permissions to MalApp3. In that case, even if BenignApp initiated an ICC to a method of VicApp4 that does not require \( i \), XmanDroid would block that ICC. In contrast, SEALANT only monitors specific ICC paths based on the detailed sender, receiver, and intent information provided by Analyzer, triggering no such false alarms.

5.2 Applicability and Accuracy

To evaluate SEALANT’s accuracy, we created three test suites containing real-world Android apps (Section 5.2.1). Using those apps, we evaluated the accuracy of Analyzer in identifying vulnerable ICC paths. We compared Analyzer’s results to those of DroidGuard [28] as well as IccTA [12, 43], a state-of-the-art tool for privacy leak detection in ICCs (Section 5.2.2). We also evaluated the ability of Interceptor to block vulnerable ICC paths at runtime (Section 5.2.3).

5.2.1 Experimental Setup

In order to build our test suites, we first selected 19 real-world apps that were vulnerable to inter-app attacks. Bagheri et al. previously identified a set of vulnerable apps [28] from repositories such as Google Play [11], F-Droid [10], MalGenome [58], and Bazaar [5]. We selected 13 of these apps that were exposed to the three types of inter-app attacks that SEALANT targets (i.e., intent spoofing, unauthorized intent receipt, and privilege escalation). We also added six apps from DroidBench 2.0 [9], an open-source app collection for benchmarking data leaks. We manually inspected the source code of the 19 apps to ensure they are vulnerable to inter-app attacks.

We ensured that each vulnerable app in our test suites has at least one malicious app that targets it. Since some of the vulnerable apps from the public repositories did not have preexisting malicious apps targeting them, we had to construct some of the malicious apps ourselves. Each malicious app we built had the sole purpose of performing an inter-app attack to exploit its target vulnerable app. Our test suites contained a total of 51 malicious apps.

We categorized the 19 vulnerable and 51 malicious apps into three test suites, based on the type of attack to which a vulnerable app is exposed, as shown in Table 2. The test suites contained 25 vulnerable ICC paths and 13 vulnerable transitive ICC paths in total.

5.2.2 Evaluation of Analyzer

Analyzer’s evaluation focused on measuring its accuracy in identifying vulnerable ICC paths compared to DroidGuard and IccTA. Specifically, we measured the three approaches’ (1) precision, i.e., identified ICC paths that were actually vulnerable and (2) recall, i.e., vulnerable ICC paths that were not identified. We installed one vulnerable app at a time along with the malicious apps that target
the vulnerable app on a Google Nexus 7, and manually triggered the malicious operations. We confirmed that the inter-app attacks from the malicious apps 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 [17]. As depicted in Table 2, only Analyzer among the three had perfect precision and recall, identifying all 25 direct and 13 transitive vulnerable ICC paths.

DroidGuard had perfect precision, but its recall was only 13%. This is because DroidGuard was designed to identify the vulnerable components or interfaces in a set of apps (coarser-grained) rather than the ICC paths between them (finer-grained). Moreover, DroidGuard was designed to return an ICC path only when both the sender and receiver components contain sensitive Android API methods [48], which limits its applicability to other types of inter-app attacks such as privilege escalation via a transitive ICC.

IccTA had 50% precision and 8% recall. IccTA only targets a particular type of attack (privacy leaks) and, like DroidGuard, it returns an ICC path only when the sender and receiver components contain sensitive API methods [48]. Moreover, the latest available version of IccTA’s implementation did not fully support analyzing inter-app communication or scenarios involving more than two apps, which additionally hampered its precision and recall.

5.2.3 Evaluation of Interceptor

We evaluated Interceptor’s precision and recall in detecting and blocking malicious ICCs at runtime. In order to monitor all ICCs exchanged on an Android device, we integrated a logging module that outputs information of each ICC instance via logcat [17] into ActivityManager (recall Section 3.1). For each of the three test suites, we installed its apps on a Google Nexus 7 with Interceptor set up, ran Analyzer on the device, and provided the SEALANT List from Analyzer to Interceptor.

To run test scripts that trigger ICCs, we used monkeyrunner [18], an automated 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 also configured the scripts to automatically choose to block an ICC when Interceptor identifies the ICC as vulnerable and prompts for a blocking choice. We repeated executing each script until we accumulated 30 blocked ICCs.

Interceptor was able to block vulnerable ICCs with perfect accuracy. 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 the vulnerable ICC paths specified in the SEALANT List, and (2) recall, i.e., if Interceptor allowed any ICC attempts over the vulnerable ICC paths. In total, Interceptor blocked all 390, 360, and 390 ICCs for the three test suites, respectively (see Table 2).

5.3 Performance Overhead

Interceptor monitors each ICC at runtime, which may impact performance. To evaluate this impact, we measured the differences in execution times between Android with Interceptor and an unmodified version of Android. For simplicity, in the remainder of this section, we will refer to the former Interceptor and to the latter AOSP, which stands for Android Open Source Project [3].

We configured the Interceptor and AOSP environments to be highly similar to each other and to reasonably reflect the real-world. We employed identical devices (Google Nexus 7) in both environments and configured both to use the same version of Android (KitKat 4.4.4; recall Section 4). We installed the 50 most popular third-party apps [8] on the devices. In order to observe the worst-case performance overhead of Interceptor, we manually created a SEALANT List that would induce the longest execution time. The SEALANT List contained 10 ICC paths, amounting to 20% of the installed apps. None of the ICC paths in the SEALANT List matched the actual ICC paths between the 50 installed apps. This maximized the overhead of Interceptor’s detection operation that 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 according to previous studies: an average smartphone user regularly uses about 30 apps per month [20], and around 10% of real-world Android apps are vulnerable to inter-app attacks [28].

To trigger a large number of ICCs on the test devices, we used Monkey [22], an Android testing tool that generates pseudo-random streams of user events (e.g., clicks, touches, or gestures) 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 of the environments and measured the time it took to process each event. We repeated this five times, totaling 25,000 events to process in each environment, to mitigate the impact of uncontrollable conditions such as changes in battery status.

Table 3 describes the results we obtained. The difference in mean execution times was less than 1ms. The difference in maximum execution times was greater, but still under 40ms. Differences of that degree are negligible because the threshold at which an end-user begins noticing slowdown in mobile app response is 100–200ms [16]. 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.

5.4 Usability

In this section, we report the results of a user study and a survey we conducted in order to assess how challenging the adoption of SEALANT is for end-users. When an intent exchange matches a vulnerable ICC path, SEALANT requires the end-user to make the choice of either blocking or allowing the exchange in order to secure her device. This gives the end-user freedom to control any suspicious intents at her discretion. However, it is also possible that the end-user (1) may feel it is difficult to make such choices or (2) may make a choice that can lead to an inter-app attack.

In our evaluation, we are guided by two hypotheses:

### Table 2: Applying SEALANT on Real-World Apps

<table>
<thead>
<tr>
<th>Attack Type of Test Suite</th>
<th>Number of Apps</th>
<th>Vulnerable ICC Paths (Vulnerable)</th>
<th>Vulnerable ICC Paths (Malicious)</th>
<th>ICCs Exchanged</th>
<th>ICCs Blocked</th>
<th>Analyzer</th>
<th>All</th>
<th>Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>13</td>
<td>390</td>
</tr>
<tr>
<td>UIR</td>
<td>6</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>360</td>
</tr>
<tr>
<td>PE</td>
<td>13</td>
<td>26</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>2818</td>
<td>390</td>
</tr>
</tbody>
</table>

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

### Table 3: Differences in Execution Times (in milliseconds)

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Min</th>
<th>Max</th>
<th>Std Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interceptor</td>
<td>25.51</td>
<td>11.31</td>
<td>81.12</td>
<td>10.22</td>
</tr>
<tr>
<td>AOSP</td>
<td>25.20</td>
<td>10.09</td>
<td>45.85</td>
<td>7.18</td>
</tr>
<tr>
<td>Difference</td>
<td>0.31</td>
<td>1.22</td>
<td>35.27</td>
<td>3.04</td>
</tr>
</tbody>
</table>
## 5.4.1 Experimental Setup

Our user study and survey were both designed to simulate situations in which users make the types of choices they would make in daily Android use (e.g., whether to install an app after being shown the list of permissions the app requires). Among those choices, we injected the choices required by SEALANT, and then observed the participants’ actions. We also asked the participants how difficult it was to make each choice and how confident they were in making the choice, in order to compare SEALANT to stock Android.

The user study included 34 participants, all of whom were graduate students at the University of Southern California (USC). We recruited them via the USC graduate student e-mail list. The background survey showed that the participants had used a mobile device for 59.28 months on average. 25 of the participants (73.52%) responded that they were using Android as their primary mobile platform or that they had experience using it. The remaining 9 (26.47%) had not used Android previously. 5 of the participants (14.71%) had no such experience. The survey covered a much wider range of age groups and occupations than the user study. 11 respondents were aged 18-24 (7.09%), 46 were 25-34 (26.67%), 37 were 35-44 (23.87%), 35 were 45-54 (22.18%), and 26 were 55+ (16.77%). The respondents included 46 students (29.67%), 27 doctors (27%), 20 business people (12.9%), 11 housewives (7.09%), 10 software engineers (6.4%), 9 professors (5.8%), 5 retailers (3.2%), 5 lawyers (3.2%), and 22 others (14.19%).

More detailed information regarding the user study and the survey, as well as a link to the complete survey, are available at http://softarch.usc.edu/sealant/.

### 5.4.2 Results

We evaluate hypotheses H1 and H2 by analyzing the data obtained from the user study and the survey. For simplicity, we refer to the user study participants and survey respondents as “participants”.

**H1** – We compared (1) the degree of difficulty perceived by the participants in making their choices, (2) the degree of confidence the participants had in their choices, and (3) the duration of time it took for the participants to make their choices across the native-Android dialogs (Type 1-3) and the SEALANT dialogs (Type 4).

Table 4 presents the values we obtained and subsequently compared. First, we compared the mean degree of difficulty, and did not find enough evidence that the means differed significantly between the two groups of scenarios, either in the user study or the survey. (Student’s t-test; p-value of 0.928 and 0.972, respectively). Second, we compared the mean degree of difficulty, and did not find enough evidence that the means differed significantly between the two groups of scenarios, either in the user study or the survey. (Student’s t-test; p-value of 0.928 and 0.972, respectively). Third, we compared the median response time, and we found that the median of Type 4 was significantly lower than that of Type 1-3 (the Mann-Whitney-Wilcoxon test; p-value of 0.008). These results together support the conclusion that the intent-exchange control choices of SEALANT 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 that would expose the device to an inter-app attack in a Type 4 scenario. When SEALANT detects a malicious intent exchange, it is possible that choosing to allow the intent exchange may align with the end-user’s actual intention. However, in general, because inter-app attacks are irreversible, we regard blocking such intent exchanges as the preferred
We also separately analyzed the user study and the survey results, we regard blocking an ICC that
("correct") choice. Recall that one half of the apps employed in
the Type 4 scenarios were from credible sources and the other half from unreliable sources. In the combined case of Type 1 (credible) and Type 2 (unreliable) scenarios, the participants chose to
cancel installing apps 51.3% of the time. That tendency, halting an
on-going activity as an effort to avoid security threats, was much
higher for Type 4 scenarios. The 34 user study participants chose
blocking 70.32% of the time; similarly, the 155 survey participants
chose blocking 68.44% of the time. Therefore, our data indicates
that participants were able to make intent-exchange control choices
that did not lead to inter-app attacks in well over one half of the
cases and at a much higher rate than their "average" behavior.

5.5 Threats to Validity

Our test suites contained 19 vulnerable apps, which may be con-
sidered too few. However, we ensured that they represent a wide
spectrum by selecting different types of apps (e.g., email, SMS,
utility, etc.) from various sources, as discussed in Section 5.2.1. Our
user study participants were students who were relatively close in
age (18-34). We tried to address any bias this may have introduced
by additionally conducting the survey that included a variety of age
groups and occupations among the respondents. On the other hand,
the survey merely emulated a mobile environment by presenting
the participants with screenshots from Android, which may have influ-
enced their choices. As a mitigation, we carefully described each
scenario in the survey to provide the participants with the context
that they would have had if they had used an actual Android device.
We also separately analyzed the user study and the survey results,
both of which support the conclusion that the adoption of SEALANT
does not present a significant challenge to end-users. Lastly, while
we regard blocking an ICC that SEALANT identifies as vulnerable
to be the appropriate choice in most cases, and consider the users’
choice to block ≈70% of ICCs that would otherwise have remained
uncaught as a positive result, the participants did elect to allow a fair
portion (≈30%) of the vulnerable ICCs. This indicates that improve-
ments may be possible with regards to how SEALANT presents the
vulnerable ICCs to the end-users.

6. RELATED WORK

A lot of effort has been dedicated to identifying Android’s vulnera-
bilities and preventing potential exploitations. Proposed approaches
use program analysis, ICC analysis, policy enforcement, or a com-
bination of them. Here we highlight the most closely related work.

Program Analysis – Several approaches employ program analy-
thesis to detect vulnerabilities in Android apps [33, 37, 39, 50, 59].
ComDroid [33] categorizes the vulnerabilities in Android inter-app
communication and detects the vulnerabilities residing in a target
app via static analysis. SCanDroid [37] statically analyzes the
data flow of a target app and provides security-relevant decisions,
CHEX [44] also leverages data-flow analysis; it inspects a target
app to discover component hijacking vulnerabilities. These tech-
niques mainly focus on identifying vulnerabilities of a single app,
and require developer intervention. On the other hand, SEALANT
targets end-users and protects them against compound inter-app
vulnerabilities by identifying and monitoring exposed surfaces.

ICC Analysis – DidFail [42] uses taint-flow analysis to locate
sensitive data-flows across application boundaries. However, tar-
geting only Android’s Activity components impairs its precision.
IceTA [43] is another taint-flow analysis technique that identifies
inter-component privacy leaks. While instrumenting source code to
resolve the connections between different components improves its
precision, it does not target other types of inter-app attacks such as
an intent spoofing. COVERT [27] introduces a static compositional
analysis of inter-app vulnerabilities, especially against permission
leakage. However, it neither targets other types of inter-app attacks,
such as unauthorized intent receipt, nor does it provide runtime
protection against identified vulnerabilities. do not provide any
protection against the identified ICC vulnerabilities.

Policy Enforcement – Two different strategies are employed for
remedying Android’s security vulnerabilities by policy enforcement:
(1) instrumenting app code [26, 28, 34, 35, 41, 49, 57] and (2) extend-
the Android framework [31, 32, 46, 55].

Aurasium [57] enforces arbitrary policies at runtime by introduc-
ing interposition code to the target app in order to limit the app’s
security and privacy violations. DroidForce [49] enforces custom
data-centric policies on a given app by instrumenting its bytecode.
DroidGuard [28] automatically synthesizes and enforces security
policies, allowing end-users to safeguard the apps installed on their
devices from inter-app vulnerabilities. While rewriting apps can be
effective, incomplete implementations of bytecode rewriting may
result in a number of potential attacks [40]. Furthermore, since
the repackaging process assigns different signatures to target apps,
a rewritten app is not able to share the history information of the
original app and cannot be further updated by the original issuer.

Saint [46] presents an extended Android framework that enables
advanced control of an app’s behavior by leveraging app provider’s
policies. XmanDroid [31] also extends the monitoring mechanism of
Android to detect and prevent app-level privilege escalation attacks
at runtime based on permission-based policies. DeepDroid [55]
provides enterprise policy enforcement by applying dynamic mem-
ory instrumentation to the app runtime environment of Android.
These approaches require a degree of expertise in devising policies,
which end-users generally lack. This implies that end-users have to
rely on general coarse-grain policies written by experts. SEALANT
alleviates this problem and provides finer-grain protection by auto-
matically generating and enforcing what amounts to target-specific
policies for a set of installed apps.

7. CONCLUSION

This paper presented SEALANT, an integrated technique that
monitors and protects the vulnerable ICC paths on which Android
inter-app attacks can take place. Through a novel combination of
static and dynamic analysis, SEALANT improves upon the current
state-of-the-art techniques in automatically identifying the vulnera-
able ICC paths between a given set of apps, monitoring each instance
of ICC to detect potential attacks, and empowering end-users to
immediately stop the attacks as they happen. Our evaluation demon-
strates SEALANT’s effectiveness, efficiency, accuracy, and usability.
Notably, we have shown that SEALANT outperforms the existing
alternatives in blocking inter-app attacks and can be applied in
real-world scenarios, while introducing a negligible performance
overhead and a minor adoption barrier to end-users.

There are several possible avenues of future work. For example,
inter-app attacks can be launched via covert channels in the Android
core system components as well as via kernel-controlled channels,
while completely bypassing the middleware reference monitor (e.g.,
confused deputy attacks over a locally established Internet socket
connection, or collusion attacks over the file system). We believe
that we can effectively counter those types of attacks by combining
our framework-level solution with existing kernel-level solutions
such as SELinux [51] and FlaskDroid [32]. Another direction for
our work is to collect end-users’ choices over time and build a
statistical model in order to provide more specific guidance to end-
users regarding the potential vulnerabilities on their devices.
8. REFERENCES


