# SCLib: A Practical and Lightweight Defense against Component Hijacking in Android Applications

Daoyuan Wu
School of Information Systems,
Singapore Management University
dywu.2015@smu.edu.sg

Yao Cheng
Institute for Infocomm Research,
A*STAR, Singapore
cheng_yao@i2r.a-star.edu.sg

Debin Gao, Yingjiu Li, and
Robert H. Deng
Singapore Management University
{dbgao,yjli,robertdeng}@smu.edu.sg

## ABSTRACT

Cross-app collaboration via inter-component communication is a fundamental mechanism on Android. Although it brings the benefits such as functionality reuse and data sharing, a threat called *component hijacking* is also introduced. By hijacking a vulnerable component in victim apps, an attack app can escalate its privilege for operations originally prohibited. Many prior studies have been performed to understand and mitigate this issue, but *no* defense is being deployed in the wild, largely due to the deployment difficulties and performance concerns. In this paper we present SCLib, a *secure component library* that performs in-app mandatory access control on behalf of app components. It does not require firmware modification or app repackaging as in previous works. The library-based nature also makes SCLib more accessible to app developers, and enables them produce secure components in the first place over fragmented Android devices. As a proof of concept, we design six mandatory policies and overcome unique implementation challenges to mitigate attacks originated from both system weaknesses and common developer mistakes. Our evaluation using ten high-profile open source apps shows that SCLib can protect their 35 risky components with negligible code footprint (less than 0.3% stub code) and nearly no slowdown to normal intra-app communication. The worst-case performance overhead is only about 5%.

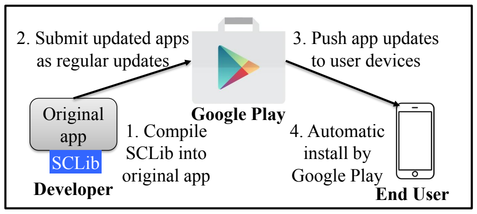
**ACM Reference Format:**
Daoyuan Wu, Yao Cheng, and Debin Gao, Yingjiu Li, and Robert H. Deng. 2018. SCLib: A Practical and Lightweight Defense against Component Hijacking in Android Applications. In *Proceedings of Eighth ACM Conference on Data and Application Security and Privacy (CODASPY'18).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3176258.3176336

## 1 INTRODUCTION

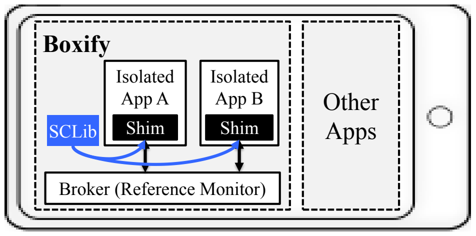
Android has been the dominant player in smartphone markets in the last few years. On Android, different apps collaborate with each other via inter-component communication. Although such flexible cross-app collaboration brings the benefits of functionality reuse and data sharing, *component hijacking* [26] is also introduced in which an attack app hijacks a vulnerable component in victim apps to bypass Android sandbox and escalate its privilege [15], causing confused deputy problems [22] such as permission misuse [21], data manipulation [26], and content leaks [43].

Many approaches have been proposed to mitigate component hijacking. One major line of the research [10, 12, 19, 30] is to modify and extend the Android operating system to supervise inter-component communication. The other direction [41] is to patch app binaries with repackaging [38]. Both are useful if they could be deployed in the wild, but nearly no proposal has been integrated into Android or adopted by Google Play to date, largely due to the compatibility and performance concerns. For example, repackaging violates Android's app verification mechanism and thus is not favorable by app markets and developers who own the source code. Consequently, component hijacking remains a serious open problem in the Android ecosystem. As one of our contributions, we make a comprehensive comparison on existing defenses in §3.2.

**Key idea.** In this paper, we provide a new perspective to practically defending against component hijacking. Our solution is a *secure component library*, shorted as SCLib, which performs in-app mandatory access control on behalf of app components. Due to its library-based nature, SCLib requires neither firmware modification nor app repackaging, significantly reducing the deployment difficulties. Specifically, we propose two deployment models as shown in Figure 1, the developer-driven and the end user-driven deployment.


2. Submit updated apps as regular updates
3. Push app updates to user devices
Google Play
Original app
SCLib
Developer
1. Compile SCLib into original app
4. Automatic install by Google Play
End User

(a) Developer-driven deployment via regular app updates.



(b) End user-driven deployment via Boxify [11].

**Figure 1: Two deployment models of SCLib.**

**Deployment models.** SCLib can be compiled by app developers into their original apps via the regular app updates (e.g., for functionality improvement), which are then pushed to user devices and automatically installed by Google Play. This deployment model introduces minimal burden to developers because they have accumulated experiences to integrate third-party libraries, such as OkHttp [8] and advertisement libraries. Further, SCLib can help developers secure their apps in the first place (rather than applying patches after apps have been released) over fragmented Android devices [2] (a major limitation of firmware modification approaches).

To further enable end users to secure their apps directly, we envision the second deployment model through state-of-the-art app sandboxing technology, e.g., Boxify, which sandboxes any other app into its own process space and delegates their inter-component communication via a reference monitor called Broker. Note that Boxify does not require root privilege, firmware modification, or app repackaging. We refer interested readers to [11] for more details. As shown in Figure 1(b), SCLib can be plugged into Boxify as part of its policy module or its shim code in each isolated app. Since SCLib's design is generally the same for both deployment models, we present it under the first deployment in the rest of this paper.

**SCLib design.** As a major component of SCLib, we devise a set of practical in-app policies to defend against component hijacking. Our policy checking is based on enforcement primitives that previous efforts have not fully leveraged, including various component attributes and input data of incoming requests. SCLib automatically collects these primitives at entry points of the protected components, and enforces "just-enough" policies from the pre-defined policy set. As a proof of concept, we design six mandatory policies that either directly deny illegal requests or alert users via a pop-up dialog for suspicious requests. These policies can mitigate component hijacking originated from both system weaknesses and common developer mistakes, half of which have not been tackled by previous efforts. Moreover, we design SCLib to cover all four types of components for the first time (see §3.2).

In the course of implementing SCLib, we identify and overcome three major challenges that are unique in our context. First, Android currently fails to provide the caller identity information to most callee components, as explained in [10]. This caller identity, however, is essential in implementing our mandatory access control policies. Unlike the previous solution [10] that modifies Android source code, SCLib leverages the Binder side channel to recover caller app identities at the application layer (see §4.3). Second, it is nearly infeasible to pop up alert dialog in the intercepted components due to the lack of appropriate user interface context and limited function return timing thresholds. We solve this problem by a novel dialog-like Activity transition technique, which overcomes both context and timing restrictions while maintaining user experience and policy enforcement logic (see §4.4). Third, there is lack of API support to collect certain component attributes (e.g., whether an exported component is explicitly or implicitly exported). SCLib performs runtime Android manifest analysis by itself (see §4.5).

SCLib is a lightweight solution by design. It enforces policy checking only at the entry points, and thus has no additional overhead of information flow tracking that is required in some existing approaches, e.g., AppSealer [41]. In addition, SCLib only affects the performance of *certain* exported components that require protection. In contrast, hooking-based checking, e.g., Aurasium [38], adds overhead to both exported and non-exported components. Firmware modification approaches introduce overhead to all inter- and intra-app communications in all apps within the system.

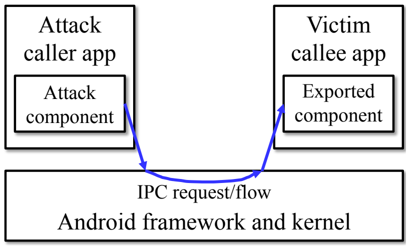


**Figure 2: The threat model of component hijacking.**

**Evaluation.** We evaluate SCLib using ten high-profile open source apps. We show that these well-tested apps contain 35 risky components that SCLib can contribute more protection. Our measurement further finds that SCLib introduces negligible code footprint — less than 0.3% stub code in all cases. Furthermore, by performing eight detailed security case studies, we demonstrate SCLib's unique values as compared to developers' own patches and Android platform updates. Finally, our performance evaluation shows that SCLib incurs modest overhead to those protected components (no overhead at all to other components).

The remainder of this paper is organized as follows. We first introduce the threat model in §2, outline the objectives and analyze existing solutions in §3. The design and implementation of SCLib are presented in §4. In §5, we evaluate SCLib's efficacy and efficiency. Finally, we conclude the paper in §6.

## 2 THREAT MODEL

Figure 2 presents our threat model of component hijacking on Android. The adversary is a *caller app*, and the victim is a *callee app* that contains a component that is *exported*. The attack component in the caller app sends a crafted *IPC* (inter-process communication)[1] request to the exported component to maliciously trigger its code execution for a privileged operation, e.g., permission misuse [21] and data manipulation [26]. In this sense, component hijacking belongs to the classic confused deputy problem [22]. Note that although Figure 2 shows only two parties, our defense can handle hijacking via one or multiple middle app(s).

More specifically, we underline two in-scope threats that are not considered in some related works.

- Unlike some existing work [21, 43], we do *not* assume that exported components protected with above-`normal` permissions [9] are always safe. We do not consider it safe because for an exported component protected with a `dangerous`-level permission, an attack app can still register the corresponding permission for sending IPC requests. Additionally, a recent report [5] showed that even components with a `signature`-level permission could be compromised, because the attack app can pre-claim that permission as `normal` if it is installed earlier than the victim app.
- Similarly, for the attack app, we do *not* assume that it always has zero or few permissions since it can claim the same permission as the misused permission in a victim app. The benefit for doing so is that it may deceive the IPC call chain-based permission checking [16, 19]. We *do* assume that the attack app has no root privilege though.

[1]A.k.a. ICC (inter-component communication) [29] on Android.

Note that unauthorized Intent receipt [14] and malicious app colluding [27] is out of the scope of this paper.

## 3 OBJECTIVES AND RELATED WORKS

### 3.1 Design Objectives

To defend against component hijacking in the wild, we identify the following four objectives:

O1 **No firmware modification.** The approach does not rely on firmware customization. It should also work without the root privilege.

O2 **No app repackaging.** The approach does not repackage target apps in bytecode or binary rewriting [25, 42].

O3 **Handling all four types of components.** The proposed solution shall protect all four types of Android components, including Intent-based components (i.e., Activity, Service, and Receiver) and non-Intent based components (i.e., Provider). Note that in this paper, we simplify BroadcastReceiver and ContentProvider as Receiver and Provider, respectively.

O4 **Minimal impact on normal operations.** The solution has minimal performance impact on normal app functionality and intra-app communication.

### 3.2 Analysis of Existing Solutions

We now analyze how existing solutions defend against component hijacking and the extent to which they achieve the aforementioned four objectives. Table 1 summarizes our analysis on major defenses against component hijacking.

First, most existing defenses require firmware modification (O1: ✗). This includes Saint [30] for adding install- and run-time policies that require developers to specify, IPC Inspection [19], Quire [16], TrustDroid [13], Scippa [10], Bugiel et al. [12] for using system-wide reference monitors to check inter-component call chains to prevent privilege (mainly permissions [21]) escalation, and Kantola et al. [24] for inferring and restricting unintentional component exposure. More recently, IEM [40] extends the Android framework to enable user-layer Intent firewall apps. All these prior works take advantage of the open-sourced nature of Android to make code changes and provide more secure OS design principles, but in the real world, they were not adopted by smartphone vendors. Additionally, end users have no capability to flash the modified firmware in general.

A particularly interesting example is IntentFirewall [1]. Although it was introduced into the Android Open Source Project (AOSP) repository over four years ago, it is still experimental and not an officially supported feature of the Android framework [39], probably due to its limitations [39]. The SEAndroid community is exploring the idea of using IntentFirewall as a potential replacement of their experimental Intent MAC mechanism [7], because in the original form of SEAndroid [32], it does not audit app-layer IPC. SEAndroid tries to reconstruct Android's sandbox from the previous Linux UID-based discretionary access control to the present SELinux-confined mandatory access control. It can restrict certain app flaws such as direct file leak [32] but not component hijacking or indirect file leak [33, 34], because it is challenging to efficiently audit every IPC at the system level without affecting normal app functionality. Even if one day a solid Intent MAC might be activated, it still faces the deployment challenges to protect fragmented and outdated devices.

**Table 1: A comparison of major defenses against component hijacking.**

| | Core Idea | Objectives (see §3.1) | | | |
|---|---|---|---|---|---|
| | | O1 | O2 | O3 | O4 |
| Saint [30] | Adding install- and run-time policies | ✗ | ✔ | ◑ | ✗ |
| IPC Inspection [19] | Checking IPC call chains | ✗ | ✔ | ✗ | ✗ |
| Quire [16] | Checking IPC and RPC call chains | ✗ | ✔ | ✗ | ✗ |
| TrustDroid [13] | Mediating IPC in middleware layer | ✗ | ✔ | ◑ | ✗ |
| Bugiel et al. [12] | Mediating IPC in different layers | ✗ | ✔ | ◑ | ✗ |
| Aurasium [38] | Intercepting sensitive API calls | ✔ | ✗ | ◑ | ◑ |
| Kantola et al. [24] | Restricting component exposure | ✗ | ✔ | ✗ | ✔ |
| IntentFirewall [1] | A system-layer firewall to check Intents | ◑ | ✔ | ✗ | ✗ |
| AppSealer [41] | Flow checking of incoming Intents | ✔ | ✗ | ✗ | ✗ |
| Scippa [10] | Building system-centric IPC call chains | ✗ | ✔ | ✗ | ✗ |
| IEM [40] | Enabling user-layer Intent firewall apps | ✗ | ✔ | ✗ | ✗ |

✔= applies; ◑= partially applies; ✗= does not apply.

Second, other defenses typically need to perform app repackaging (O2: ✗). Aurasium [38] and AppSealer [41] are the two notable examples. Specifically, Aurasium repackages apps to insert API hooking code to intercept sensitive API calls. It mainly aims to prevent malware but can also be used to mitigate component hijacking in which sensitive APIs (in a victim app) are triggered by an attack app. On the other hand, AppSealer is specialized to generate patched apps that introduce flow tracking to avoid critical APIs being triggered by malicious Intents. Both approaches are attractive from the security's perspective, but they also face the deployment difficulties: (i) repackaging is unlikely adopted by app developers because they own the source code and do not want repackaging to affect the original code quality; (ii) app stores are unlikely to deploy repackaging-based approaches because they (including Google Play) have no access to the developers' private signing keys.

Third, none of the prior efforts has fully handled all four types of components (O3: ✗; O3: ◑). Most of them only protect the Intent-based components, whereas the non-Intent based Provider component is largely under-treated. Although all underlying Android IPC communications go through the Binder driver [23, 31], Intent and Provider are two different higher-level abstractions of Binder [20]. Existing approaches either just modify the Android framework to supervise Intent IPC (e.g., [16, 19]), or only recover Intents' semantic from the kernel-layer Binder (e.g., [10, 12]). Approaches such as AppSealer [41] and IEM [40] also explicitly target at Intent-based components. Note that although some approaches (e.g., [13, 30]) mentioned the protection for Provider to some extent, the generic and broader app Provider vulnerabilities [43] were not touched because the problem itself was discovered only afterwards.

Fourth, nearly no defenses satisfy the requirement of minimal checking on normal operations (O4: ✗). Except the work from Kantola et al. [24], firmware modification approaches have to monitor all IPC communications in all apps within the system. Although they achieve the whole-system coverage, the performance was sacrificed by not focusing on apps or components that need protection. Moreover, they often need to retrieve the corresponding permission for each call chain, which also increases the overhead. On the other hand, Intent flow based repackaging approaches such as AppSealer [41] can concentrate on protecting risky app components, but its data flow tracking is expensive. Hooking-based API checking in Aurasium [38] is lightweight; however, it does not differentiate sensitive API calls resulting from user operations or malicious IPC.

## 4 SCLIB: SECURE COMPONENT LIBRARY

This section covers the design and implementation details of SCLib. We begin with an overview of SCLib in §4.1, and then present some important MAC policies that SCLib is capable of enforcing (§4.2). After that, we discuss the detailed implementation of SCLib with the focus on our novel ways of handling the challenges.

### 4.1 Design Overview

Figure 3 presents the overall design of SCLib. At the high-level view, SCLib is a regular user-space library that could be easily integrated into apps on different Android platforms. SCLib aims to be a secure component library that performs in-app mandatory access control (MAC) on behalf of app components to defend against component hijacking. With a set of pre-defined MAC policies in SCLib, developers can overcome the by-default system weaknesses and common mistakes. Moreover, as a library, SCLib inherently requires *no* firmware modification or app repackaging (O1: ✔; O2: ✔).

Using SCLib consists of two phases, i.e., the compile- and run-time phase as shown in Figure 3. Firstly in the compile-time phase, developers include SCLib into their app projects and run our tool suite to help SCLib identify risky components that need protection. Then limited amounts of stub codes are added into entry functions of risky components (usually two LOC per entry) so that SCLib can intercept incoming IPC flows. Note that the whole procedure could be automatic with just a run of our tool suite. Due to the page limit, we skip the details of compile-time designs in this paper, and leave them to the technical report version [35].

In the run-time phase, SCLib automatically collects enforcement primitives and enforces policy checking without developers' involvement. To make practical in-app policies to facilitate the access control, SCLib collects a number of enforcement primitives that previous efforts have not fully leveraged and takes all four types of Android components into consideration (O3: ✔). Considering that SCLib's checking is conducted only at entry points and only for risky components, it makes SCLib lightweight (O4: ✔). With SCLib, the incoming IPC flow no longer directly executes the component codes. Instead, it has to first go through SCLib's checking that could generate three possible outputs: `deny`, `alert`, and `allow`. Only in the `allow` case will the execution flow go back to the component code immediately. In the case of `alert`, SCLib pops up an alert dialog for users to make a decision — flow resumes if the users choose to allow the call. If SCLib determines to deny to call, control will return to the calling environment immediately.

### 4.2 In-app MAC Policy Design

It is important to understand the policies SCLib is designed to enforce before we present other details. Remember that our objective is to have mandatory access control (MAC) policies to stop common component hijacking issues that result from system flaws or developer mistakes. Table 2 lists six representative MAC policies (P1 to P6) we have designed. From a high-level view, policies P1 and P2 patch the system weaknesses, P3 to P5 mitigate common developer mistakes, and P6 filters a common attack. Note that we do not claim that they cover all hijacking issues; instead, our purpose is to show *how to design* in-app SCLib policies for major categories of attacks and for different components. Our policies thus serve as templates or baselines for more enhanced or customized policies.

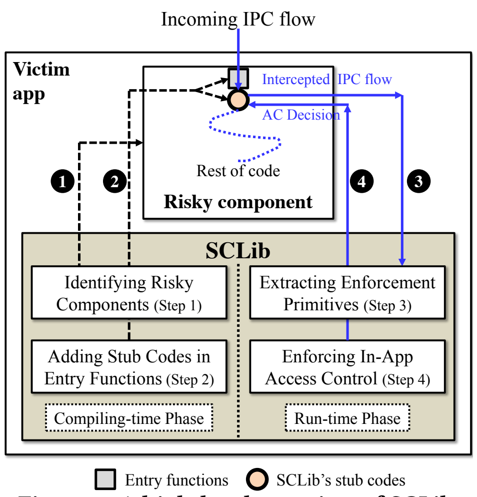

Incoming IPC flow
Victim app
Intercepted IPC flow
AC Decision
Rest of code
Risky component
1
2
4
3
SCLib
Identifying Risky Components (Step 1)
Extracting Enforcement Primitives (Step 3)
Adding Stub Codes in Entry Functions (Step 2)
Enforcing In-App Access Control (Step 4)
Compiling-time Phase
Run-time Phase
Entry functions
SCLib's stub codes

**Figure 3: A high-level overview of SCLib.**

**Trusting intra-app IPC by default.** A common point among all six policies is that we consider the IPC calls initiated from the same app/developer trusted. That is, only an external IPC call from a third-party app will be checked, i.e., $ID_a \neq ID_v$. This by-default rule is important in two aspects. First, it effectively minimizes the usability issues for normal user operations, because only the *external* IPC for certain exported components (i.e., risky components) will trigger the alerts. Second, it strengthens SCLib's access control capabilities because another app from the same developer now can be trusted through the app identity and its developer certificate checking. In contrast, solutions such as IntraComDroid [24] and Android Lint simply stop all incoming IPC calls, including those from the same developer, by un-exporting components.

**Fixing system weaknesses with P1 and P2.** We now show how to design SCLib policies to mitigate system flaws. To this end, we design policies P1 and P2 to fix two major weaknesses in Android. The first is that Android prior to 4.2 by default exports all Provider components even if they do not claim the `exported` attribute. This over-ambitious exposure rule led to thousands of vulnerable Provider components [43]. Although Google later disabled it, there are still many by-default exported Provider components in apps compiled with old SDK according to recent studies [28, 36]. To make the new rule available to all apps and all phones (including those under 4.2), we design policy P1 to mimic the current system rule at the app layer. Specifically, SCLib directly denies an external IPC request to the callee Provider that does not claim the `exported` attribute ($\neg ExportedAttr$).

P2, on the other hand, fixes a more complicated and less-known system flaw [5] where an attack app installed earlier can pre-claim a custom permission in the victim app with the purpose of downgrading its protection level, e.g., from `signature` to `normal`. Consequently, the attack app can hijack a "private" component that was originally protected with `signature`-level permissions. Based on this root cause, policy P2 first determines whether there is a custom permission defined in the callee component, i.e., $\exists(PermAttr_v \notin SysPerms)$. If it exists, we further check whether or not it has been pre-claimed by the caller app, i.e., $PermAttr_v = PermAttr_a$. In practice, we can simplify policy P2 for the `signature`-level custom permissions by leveraging the fact that an external IPC can come only when the `signature` permission has been downgraded.

Table 2: MAC policies in SCLib. Here are the six representative policies (P1 to P6) we have designed.

| ID | Policy Name | † | Policy Representation | ‡ |
|---|---|---|---|---|
| P1 | No By-default Exported Provider | P | **if** $ID_a \neq ID_v \land \neg ExportedAttr$: **deny** | ◑ |
| P2 | No Pre-claimed Custom Permission | All | **if** $ID_a \neq ID_v \land \exists(PermAttr_v \notin SysPerms) \land PermAttr_v = PermAttr_a$: **deny** | ◑ |
| P3 | Alerting Implicitly Exported Components with Custom Action | A, S, R | **if** $ID_a \neq ID_v \land \neg ExportedAttr \land ActionAttr \notin SysActions$ : **alert** | ✔ |
| P4 | Alerting Explicitly Exported Provider | P | **if** $ID_a \neq ID_v \land ExportedAttr = true$: **alert** | ✗ |
| P5 | Checking System-only Broadcasts | R | **if** $ID_a \neq ID_v \land \exists(ActionAttr \in SysActions) \land InputAction \neq ActionAttr$: **deny** | ✔ |
| P6 | Filtering Sql Injection for Provider | P | **if** $ID_a \neq ID_v \land \exists(AttackStr \in InputPara)$: **deny** | ✗ |

† lists which components this policy is applicable to. All: all four components; A: Activity; S: Service; R: Receiver; and P: Provider.
‡ indicates whether a policy has been covered by previous efforts. ✔= covers by [24]; ✗= does not cover; ◑= partially covers by system updates. Note that [24] simply un-exports components in policy P3, which would cause incompatibility issues while ours will not.

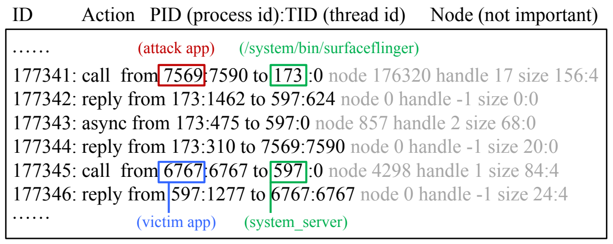
**Preventing developer mistakes with P3 to P5.** In this part, we show that how SCLib prevents three common developer mistakes using policy P3 to P5. We first discuss policy P3 and P4 to take care of developers who mistakenly export their components or simply did not realize the threats from exported components. Specifically, for policy P3, our insight is that if developers register custom Intent actions for their implicitly exported components, very likely they do not intend to export those components. While policy P4 is based on the measurement results in [28, 43] that many explicitly exported Provider components can also leak sensitive data. To prevent these two types of mistakes, policy P4 checks *ExportedAttr* and policy P3 further checks the custom actions ($ActionAttr \notin SysActions$). To reduce false positives, we choose the `alert` for policy P3 and P4. Moreover, since custom actions and Provider components are much less-called by inter-app IPC, we expect that our `alert` policies would not disrupt user experience.

Then we have policy P5 to mitigate a developer mistake that appears at the code level (instead of manifest). That is, a Receiver component that registers system-only broadcasts is still hijack-able if it does not check the incoming Intent action explicitly in the code [37]. Our measurement of ten high-profile open source apps in §5 shows that a couple of them made this mistake. To defend, policy P5 automatically checks the input action (on behalf of callee component) against the system-only action claimed in manifest, i.e., $\exists(ActionAttr \in SysAction) \land InputAction \neq ActionAttr$.

**Stopping a common attack with P6.** Finally, we propose policy P6 as a prominent example to show SCLib's capability of stopping common attacks. Specifically, policy P6 aims to filter SQL injection for Provider. As demonstrated in [43], an attack app can hijack a Provider component to inject malicious SQL statements. For example, the adversary sets the `projection` parameter of the `query` function as a special phase "`* from private_table;`". As these special inputs are different from normal queries, we use keyword-based filtering (such as the expression like "`xxx from yyy;`") to stop them. Similarly, we can stop the directory traversal attack [43] in `openFile` entry of Provider by leveraging some file path patterns. Furthermore, we can devise `alert` policies to protect permission-protected components to stop an adversary that claims corresponding `dangerous` permissions, as we will conduct case studies in §5.2.

## 4.3 Recovering Caller Identity via the Binder Side Channel

Having discussed the policies that SCLib is designed to enforce, we now turn to some implementation details to show how SCLib managed to overcome the design challenges. In this specific subsection, we show how SCLib recovers the caller identify (C2) via the Binder



```
……
177341: call  from 7569:7590 to 173:0 node 176320 handle 17 size 156:4
177342: reply from 173:1462 to 597:624 node 0 handle -1 size 0:0
177343: async from 173:475 to 597:0 node 857 handle 2 size 68:0
177344: reply from 173:310 to 7569:7590 node 0 handle -1 size 20:0
177345: call  from 6767:6767 to 597:0 node 4298 handle 1 size 84:4
177346: reply from 597:1277 to 6767:6767 node 0 handle -1 size 24:4
……
```

**Figure 4: An example of the Binder transaction_log.**

side channel at the path of `/sys/kernel/debug/binder/transaction_log`. More specifically, for each risky IPC call intercepted, we retrieve the recent Binder logs from this side channel and analyze them to recover the corresponding caller app identity. Figure 4 shows a `transaction_log` example when an attack app exploits an Activity component in the victim app. Each Binder log starts with a unique transaction ID followed by the Binder action and the process/thread IDs of the caller and callee processes. The last part, node information, is not important — so we skip here. Note that in the kernel-layer Binder driver, app processes do not directly interact with each other. Instead, the high-level IPC always involves a number of interactions between apps and system processes (see [10] for more details). For example, the attack app (PID: 7569) and the victim app (PID: 6767) here leverage the `surfaceflinger` and `system_server` processes to delegate their communication.

Our extensive tests of Binder logs in different components show that they all follow the same pattern, based on which we propose a simple yet effective algorithm to recover caller identities. We still use Figure 4 to illustrate this algorithm. The first step is to locate the Binder log that "calls from" the callee PID for the first time, i.e., the transaction 177345. Then we trace back to identify the first app process, i.e., PID 7569, which is the caller app we are looking for. Since there is no fixed PID pattern for non-system processes, we further extract the corresponding UID and package name for analysis. More specifically, if the UID is smaller than 10000 or if the package name is a system binary, it must be a system process.

Since there is a timing window to retrieve the recent Binder logs, SCLib performs the Binder analysis before other modules. To further decrease the delay, we focus on extracting and saving the logs first, and postpone the actual analysis. Our tests show that in this way, we can reliably retrieve the required Binder logs. Moreover, we found that accessing the Binder transaction log is allowed even in some smartphones with SEAndroid. We tested more than ten Android device models and found that the majority of them allow the access in the SEAndroid enforcing mode, including Samsung Galaxy S6 Edge+, Nexus 4/5/5X/6P, and several Huawei/Samsung/XiaoMi phones.

## 4.4 Popping Up Alerts via the Dialog-like Activity Transition

To enforce the alert policies, SCLib needs to pop up an alert dialog for users to choose "allow" or "deny". However, this is a challenging task due to the following reasons:

- Background components such as Provider and Service do not have an appropriate UI `Context` to display alert dialogs. Even for Activity, it cannot pop up dialogs when `onCreate()` is still being intercepted, (i.e., not return yet).
- Some components' entry functions (e.g., Activity's `onCreate()` and Receiver's `onReceive()`) need to return in a short time. Therefore, we cannot hold on the execution of these functions and wait for users' decisions.

To address these issues, we opt for a different strategy instead of directly displaying an alert dialog. The basic idea is to launch a dialog-like Activity from the intercepted component via the `startActivity()` API. For entry functions that are sensitive to execution time, SCLib immediately returns the execution flow to them by assuming users choosing "deny". If users select "allow" later, SCLib re-sends the same Intent[2] content on behalf of the original caller app. Since the callee app has no way to distinguish the caller identity, the original execution flow can resume. While for other time-insensitive entry functions, SCLib can pause their component execution and wait for users to make a decision on the alert dialog.

Due to the page limit, we refer interested readers to our technical report [35] for the implementation details of this dialog-like Activity transition approach.

## 4.5 Extracting Component Attributes by Run-time Manifest Analysis

To overcome the challenge that we do not have API support to collect certain component attributes, SCLib performs Android manifest analysis by itself. In particular, we choose the run-time analysis instead of compile-time analysis because it does not bother developers and neither needs the additional file storage. Also, it is immune to app updates and can handle new components well.

The basic procedure is to *dynamically* retrieve and parse `AndroidManifest.xml` of the callee app. Specifically, for the callee component, we extract its raw `exported` status, the registered Intent actions, and the associated permissions. We then correlate the app permission entries to obtain their protection levels and determine whether an associated permission is defined by the system or the callee. We also build a list of system-defined and system-only Intent actions based on the Stowaway result [17] and Android source code so that we can determine whether a given component listens to system Intent actions or not.

## 5 EVALUATION

In this section, we evaluate SCLib in three aspects. Firstly in §5.1, we measure the component statistics of ten high-profile open source apps to find out how many risky components could benefit from SCLib and how much code footprint SCLib introduces. Then in §5.2, we assess the security effectiveness of SCLib against attacks in different components. Finally in §5.3, we measure the performance overhead of SCLib under different scenarios.

[2]Provider's entry functions are not sensitive to execution time, so we focus on Intent-based communications here.

**Table 3: Size of stub code to protect risky components.**

| Application | Lines of Java codes | Lines of stub codes | Extra code percentage |
|---|---|---|---|
| Telegram | 222,074 | 32 | 0.014% |
| Zxing Barcode | 43,221 | 24 | 0.056% |
| Terminal Emulator | 11,507 | 30 | 0.261% |
| K-9 Mail | 51,416 | 62 | 0.121% |
| WordPress | 81,076 | 22 | 0.027% |
| Signal Messenger | 63,137 | 34 | 0.054% |
| Wire | 52,808 | 2 | 0.004% |
| Bitcoin Wallet | 18,695 | 40 | 0.214% |
| ChatSecure | 36,911 | 18 | 0.049% |
| Zirco Browser | 9,638 | 26 | 0.270% |

## 5.1 Applying SCLib

We first get an idea about the extent to which typical Android apps export their components to others, and the corresponding code footprint when we apply SCLib to protect these components. To this end, we collect the latest source code of ten high-profile open source apps from their GitHub sites at the time of our research (November 2016). The detailed statistics of these apps are available in our technical report [35]. We find that every tested app exports some of its components, and 67.3% are *implicitly* exported. Moreover, a total of 35 component are risky and thus require SCLib's protection.

We further measure the additional stub code introduced by SCLib. Specifically, we calculate the number of additional lines of code based on the type of risky component and the number of entry functions of that type. The results in Table 3 show that the code footprint introduced by SCLib is negligible at less than 0.3% in all cases. Note that since SCLib is implemented in Java and will be instrumented in Java environments, we only compare our code footprint based on the number of lines of Java code in each apps, though some tested apps also contain many C/C++ codes. Additionally, the jar file of SCLib itself is also very small — only around 30KB before compression.

## 5.2 Security Evaluation

To perform security evaluation, we identify eight vulnerable or risky components from the aforementioned ten apps. As shown in Table 4, these cases cover all four types of Android components and all six policies we designed. In this subsection, we present our detailed analysis of the eight case studies to demonstrate SCLib's unique values in mitigating developer mistakes and system weaknesses when compared to developers' own patches and Android platform updates.

**Case 1: Fixing vulnerable components without losing compatibility.** The first case, Terminal Emulator (`jackpal.androidterm`), is a good example to illustrate that the developers' own patches sometimes could cause incompatibility issues that SCLib can avoid. Terminal Emulator contained a vulnerable component called `RemoteInterface` in its version 1.0.63. The component is implicitly exported and can be triggered by a crafted Intent to execute arbitrary commands without any user interaction. To fix this vulnerability, the developers removed the programmatic command execution functionality in `RemoteInterface` [3, 4]. However, there were

**Table 4: Security case studies: Using SCLib to protect vulnerable/risky components.**

| ID | Target Component (†) | App | Policy |
|---|---|---|---|
| 1 | `RemoteInterface` (A; I) | Term Emulator | P3 (alert) |
| 2 | `MessageProvider` (P; E) | K-9 Mail | P4 (alert) |
| 3 | `RemoteControlReceiver` (R; I) | K-9 Mail | P3 (alert) |
| 4 | `TermService` (S; I) | Term Emulator | P3 (alert) |
| 5 | `ZircoBookmarksProvider` (P; I) | Zirco Browser | P1 (deny) |
| 6 | `New/Clear KeyReceiver` (R; I) | Signal | P2 (deny) |
| 7 | `AppStartReceiver` (R; I) | Telegram | P5 (deny) |
| 8 | `WeaveContentProvider` (P; I) | Zirco Browser | P6 (deny) |

† means `Type; Export`, i.e., the component type (four types of components) and the export status (implicitly or explicitly exported).

other apps[3] that continue to utilize this programmatic interface and the patch thus caused an incompatibility issue[4] on those apps. Additionally, simply un-exporting the component as proposed by IntraComDroid [24] would cause the same incompatibility issue.

In contrast, SCLib fixes this vulnerability in a more elegant way that results in no incompatibility issue and no additional developer effort. Specifically, since `RemoteInterface` registers an Intent filter to take a custom Intent action, it satisfies our policy P3 (see Table 2). As a result, SCLib pops up an alert dialog when an external app tries to trigger the programmatic command execution in `RemoteInterface`. In this way, SCLib notifies users on potential attacks while keeping the app compatible with other legacy apps (that call `RemoteInterface`). SCLib also saves the developers' effort in making the patches — Terminal's developers performed around 200 lines of code changes to construct the patch [3].

**Case 2 & 3 & 4: Enforcing security beyond Android's existing security mechanisms.** In this part, we first present how SCLib enhances protection of two risky components in K-9 Mail (`com.fsck.k9`) — `MessageProvider` as in case 2 and `RemoteControlReceiver` as in case 3. Both components are exported and have self-defined `dangerous`-level permissions. The rationale behind this design is that K-9 Mail has a number of extension apps [6] which need to access these two components. To share components to other apps with different signatures, the most secure way Android currently provides is to define a `dangerous`-level permission, as what K-9 Mail did. However, this is too coarse-grained and cannot prevent a malicious app from claiming the corresponding permissions to steal users' emails via `MessageProvider`. Indeed, according to a comprehensive survey [18], users generally skip the permission inspection during app installation or simply cannot understand the permission meanings, which makes the attacks here realistic.

With SCLib, K-9 Mail now can achieve a more fine-grained access control by enabling users allow/deny a *particular* external app on the alert dialog. K-9 Mail would not have been capable of achieving such fine-grained security because: (i) Intent-based components such as `RemoteControlReceiver` have no existing method of obtaining caller identity, an important primitive Android currently fails to provide; and (ii) even though `MessageProvider` has an API to extract the caller identity, it cannot pop up alert dialogs.

[3]To name a few, see code snippets in https://goo.gl/HK3HgJ, https://goo.gl/0t78J8, https://goo.gl/xPjlv3, and https://goo.gl/xOs5zN.
[4]A bug report was actually issued after the patch, but the developers of Terminal closed all links after the project was finished.

Further, `TermService` in case 4 demonstrates a clearer example where developers actually demand the capability of differentiating different caller app identities. According to its code at http://tinyurl.com/termservice, we see that the developers want to determine whether an external app or its own Activity makes the incoming IPC. However, `TermService` tries to achieve this by checking whether the incoming Intent contains a custom action that is claimed in the `<intent-filter>`. Developers believe that an external app would use that custom action to launch IPC, but actually an attack app can explicitly call `TermService` without setting that action. Consequently, `TermService`'s action-based checking can be bypassed. With SCLib, we can prevent such attacks and provide developers a solid mechanism to differentiate external IPC calls.

**Case 5 & 6: Fixing system weaknesses with a broader platform and app coverage.** Next, we introduce two cases to illustrate that SCLib can fix system weaknesses with a broader platform and app coverage than Android's system updates. In case 5, Zirco Browser (`org.zirco`)'s `ZircoBookmarksProvider` is by default exported by Android system, causing the leakage of users' bookmarks. Although Android changed this by-default policy since 4.2, the new exposure policy is not applicable to apps with a target SDK version below 4.2. In contrast, SCLib leverages the policy P1 to protect all implicitly exported Provider components even when they run on legacy phones or are compiled with target SDKs of older versions.

As another example, Signal Private Messenger (`org.thoughtcrime.securesms`) contains two dynamically registered Receiver components, `NewKeyReceiver` and `ClearKeyReceiver`, which are protected with a custom `signature`-level permission called `ACCESS_SECRETS`. As explained in §4.2, they are subject to the permission pre-occupy attack. Android fixes this weakness only after 5.0, whereas SCLib can eliminate its impact even on Android versions prior to 5.0.

**Case 7 & 8: Fixing common developer mistakes and stopping common attack patterns.** We now present case 7 and 8 to illustrate how SCLib helps fix a common developer mistake and stop a common attack pattern, respectively. In case 7, Telegram (`org.telegram.messenger`) defines an `AppStartReceiver` component to listen to the `BOOT_COMPLETED` broadcast, but the developers forgot to check this system-only action in its code (see http://tinyurl.com/startreceiver), making it possible that the component execution be triggered by any app. With SCLib, developers no longer need to worry about such checking because SCLib automatically performs the checking based on policy P5. We further mimic a SQL injection attack on `WeaveContentProvider` in case 8, which can be defended by our policy P6, as shown in http://tinyurl.com/sqlweave.

## 5.3 Performance Evaluation

We now evaluate the performance overhead of SCLib under different scenarios. Here we focus only on the results. Interested readers can refer to our technical report [35] for more details about evaluation methodology and experimental setup. Table 5 shows the performance results for the tested Activity and Provider, respectively. We can see that the cumulative overhead (i.e., the worst-scenario overhead) is below 5% for both components, with 4.42% for Activity

Table 5: Breakdown of SCLib's overheads.

| Scenario | Category | Time cost | Overhead % |
|---|---|---|---|
| **Activity** | | | |
| Original scenario | Normal IPC latency: $t_0$ | 464.40ms | - |
| Overheads introduced by SCLib | Binder analysis: $t_1$ | 8.73ms | 1.88% |
| | Manifest analysis: $t_2$ | 0.24ms$^\dagger$ | 0.05%$^\dagger$ |
| | Policy assessment: $t_3$ | 0.05ms | 0.01% |
| | Popping up alerts: $t_4$ | 11.53ms | 2.48% |
| | Sum (worst-scenario) | 20.55ms | 4.42% |
| **Provider** | | | |
| Original scenario | Normal IPC latency: $t_0{'}$ | 10.82ms | - |
| Overheads introduced by SCLib | Getting caller identity: $t_1{'}$ | 0.24ms | 2.22% |
| | SQL filtering: $t_{1.5}{'}$ | 0.001ms | 0.01% |
| | Manifest analysis: $t_2{'}$ | 0.146ms$^\dagger$ | 1.35%$^\dagger$ |
| | Policy assessment: $t_3{'}$ | 0.014ms | 0.13% |
| | Sum (worst-scenario) | 0.401ms | 3.71% |

$^\dagger$ SCLib analyzes manifest only once for the entire lifecycle of the app. In the Activity context, manifest analysis takes 2.41ms in the first run and zero for the rest of runs. Similarly, the analysis of the first run on Provider takes 1.46ms. Therefore, we calculate an estimated value by assuming that there are ten IPC transactions in a lifecycle of the app.

and 3.71% for Provider. Also, the absolute cumulative timing overhead is only 20.55ms and 0.4ms, which is unnoticeable to human users. Moreover, we would like to underline that SCLib brings overheads only at the entry points of risky components, while existing defenses cause slowdown to the entire app or system.

## 6 CONCLUSION AND FUTURE WORKS

In this paper, we presented a practical and lightweight approach called SCLib to defend against component hijacking in Android apps. SCLib is essentially a secure component library that performs in-app mandatory access control on behalf of app components. We designed six mandatory policies for SCLib to stop attacks originated from both system weaknesses and common developer mistakes. We have implemented a proof-of-concept SCLib prototype and demonstrated its efficacy and efficiency. In the future, we will try to integrate SCLib into Boxify [11] after its code is released.

## ACKNOWLEDGEMENTS

We thank all the reviewers of this paper for their valuable comments. This work is partially supported by the Singapore National Research Foundation under NCR Award Number NRF2014NCR-NCR001-012.

## REFERENCES

[1] 2013. The IntentFirewall code. http://tinyurl.com/IFcode. (2013).

[2] 2015. Android Fragmentation Report. http://tinyurl.com/frag1508. (2015).

[3] 2015. A Fix to the Issue #374 in Android-Terminal-Emulator. http://tinyurl.com/fixissue374. (2015).

[4] 2015. Issue #374 in Android-Terminal-Emulator. https://tinyurl.com/pull375. (2015).

[5] 2015. The Custom Permission Problem. http://tinyurl.com/CusPerm. (2015).

[6] 2015. Works with K-9 Mail. http://tinyurl.com/WorksWithK9. (2015).

[7] 2017. IntentFirewall in SEAndroid. http://tinyurl.com/IFwall. (2017).

[8] 2017. OkHttp. http://square.github.io/okhttp/. (2017).

[9] 2017. The Protection Levels of Permissions. http://tinyurl.com/permlevel. (2017).

[10] Michael Backes, Sven Bugiel, and Sebastian Gerling. 2014. Scippa: System-Centric IPC Provenance on Android. In *Proc. ACM ACSAC*.

[11] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp Von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. USENIX Security Symposium*.

[12] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad Sadeghi, and Bhargava Shastry. 2012. Towards Taming Privilege-Escalation Attacks on Android. In *Proc. ISOC NDSS*.

[13] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2011. Practical and Lightweight Domain Isolation on Android. In *Proc. ACM SPSM*.

[14] Erika Chin, Adrienne Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-Application Communication in Android. In *Proc. ACM MobiSys*.

[15] Lucas Davi, Alexandra Dmitrienko, Ahmad Sadeghi, and Marcel Winandy. 2010. Privilege Escalation Attacks on Android. In *Proc. Springer ISC*.

[16] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan Wallach. 2011. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proc. USENIX Security Symposium*.

[17] Adrienne Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proc. ACM CCS*.

[18] Adrienne Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. ACM SOUPS*.

[19] Adrienne Felt, Helen Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses. In *Proc. USENIX Security*.

[20] Aleksandar Gargenta. 2013. Deep Dive into Android IPC/Binder Framework. http://tinyurl.com/diveIPC. (2013).

[21] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proc. NDSS*.

[22] Norm Hardy. 1988. The Confused Deputy: (or why capabilities might have been invented). In *ACM SIGPOS Operating Systems Review*.

[23] Ahn Joonseok. 2012. Binder: Communication Mechanism of Android Processes. http://tinyurl.com/bindercomm. (2012).

[24] David Kantola, Erika Chin, Warren He, and David Wagner. 2012. Reducing Attack Surfaces for Intra-Application Communication in Android. In *Proc. SPSM*.

[25] Yu Liang, Xinjie Ma, Daoyuan Wu, Xiaoxiao Tang, Debin Gao, Guojun Peng, Chunfu Jia, and Huanguo Zhang. 2015. Stack Layout Randomization with Minimal Rewriting of Android Binaries. In *Proc. Springer International Conference on Information Security and Cryptology (ICISC)*.

[26] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. ACM CCS*.

[27] Claudio Marforio, Hubert Ritzdorf, AurÃľlien Francillon, and Srdjan Capkun. 2012. Analysis of the Communication between Colluding Applications on Modern Smartphones. In *Proc. ACM ACSAC*.

[28] Patrick Mutchler, Yeganeh Safaei, Adam Doupe, and John Mitchell. 2016. Target Fragmentation in Android Apps. In *Proc. IEEE MoST*.

[29] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proc. USENIX Security Symposium*.

[30] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. 2009. Semantically Rich Application-Centric Security in Android. In *Proc. ACSAC*.

[31] Thorsten Schreiber. 2012. Android Binder: Android Interprocess Communication. http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf. (2012).

[32] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. ISOC NDSS*.

[33] Daoyuan Wu and Rocky K. C. Chang. 2014. Analyzing Android Browser Apps for file:// Vulnerabilities. In *Proc. Springer Information Security Conference (ISC)*.

[34] Daoyuan Wu and Rocky K. C. Chang. 2015. Indirect File Leaks in Mobile Applications. In *Proc. IEEE Mobile Security Technologies (MoST)*.

[35] Daoyuan Wu, Yao Cheng, Debin Gao, Yingjiu Li, and Robert H. Deng. 2018. SCLib: A Practical and Lightweight Defense against Component Hijacking in Android Applications. *CoRR* abs/1801.04372 (2018). https://arxiv.org/abs/1801.04372

[36] Daoyuan Wu, Ximing Liu, Jiayun Xu, David Lo, and Debin Gao. 2017. Measuring the Declared SDK Versions and Their Consistency with API Calls in Android Apps. In *Proc. Conference on Wireless Algorithms, Systems, and Applications*.

[37] Daoyuan Wu, Xiapu Luo, and Rocky K. C. Chang. 2014. A Sink-driven Approach to Detecting Exposed Component Vulnerabilities in Android Apps. *CoRR* abs/1405.6282 (2014). http://arxiv.org/abs/1405.6282

[38] Rubin Xu, Hassen Saidi, and Ross Anderson. 2012. Aurasium: Practical Policy Enforcement for Android Applications. In *Proc. USENIX Security*.

[39] Carter Yagemann. 2016. IntentFirewall Unofficial Document. http://www.cis.syr.edu/~wedu/android/IntentFirewall/. (2016).

[40] Carter Yagemann and Wenliang Du. 2016. Intentio Ex Machina: Android Intent Access Control via an Extensible Application Hook. In *Proc. ESORICS*.

[41] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proc. ISOC NDSS*.

[42] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *ACM CODASPY*.

[43] Yajin Zhou and Xuxian Jiang. 2013. Detecting Passive Content Leaks and Pollution in Android Applications. In *Proc. ISOC NDSS*.