Noname manuscript No.
(will be inserted by the editor)

System-Level Attacks against Android by Exploiting Asynchronous Programming

Ting Chen · Xiaoqi Li · Yajuan Tang · Xiapu Luo · Xiaosong Zhang

Received: date / Accepted: date

Abstract To avoid unresponsiveness, Android developers utilize asynchronous programming to schedule long-running tasks on the background. In this work, we conduct a systematic study on IntentService, one of the async constructs provided by Android using static program analysis, and find that in Android 6, 974 intents can be sent by third-party applications without protection. Based on this observation, we develop a tool, ATUIN, to demonstrate the feasibility of attacking CPU automatically by exploiting the intents that can be handled by Android system. Furthermore, by investigating the unprotected intents, we discover tens of critical vulnerabilities that have not been reported before, including Wi-Fi DoS, telephone signal block, SIM card removal, homescreen hiding and NFC state cheating. Our study sheds light on the researches of protecting the asynchronous programming fashion from being exploited by hackers.

Ting Chen
Cybersecurity Research Center, University of Electronic Science and Technology of China
E-mail: brokendragon@uestc.edu.cn

Xiaoqi Li
Department of Computing, The Hong Kong Polytechnic University, Hong Kong
E-mail: csxqli@comp.polyu.edu.hk

Yajuan Tang
College of Engineering, Shantou University, China
E-mail: yjtang@stu.edu.cn

Xiapu Luo
Department of Computing, The Hong Kong Polytechnic University, Hong Kong
E-mail: csxluo@comp.polyu.edu.hk

Xiaosong Zhang
Cybersecurity Research Center, University of Electronic Science and Technology of China
E-mail: johnsonzxs@uestc.edu.cn

Keywords Asynchronous programming · Android · IntentService · System-level attacks · Wi-Fi DoS · Telephone signal block · SIM card removal · Homescreen hiding · NFC state cheating

1 Introduction

Responsiveness is critical for smartphones. However, smartphones are likely to be unresponsive because of their limited computing resources and frequent network operations. Previous researches show that many Android applications suffer from poor responsiveness and one of the primary reasons is that applications run too much workload in the UI event thread [1, 2]. The primary way to avoid unresponsiveness is to resort to concurrency that puts long-running tasks into background threads and runs the main thread and the background threads asynchronously.

To make asynchronous programming easier, Android provides three major async constructs: `AsyncTask`, `IntentService`, and `AsyncTaskLoader`. `AsyncTask` is designed for short-running tasks while the other two are good choices for long-running tasks. Although `AsyncTask` is the most widely used construct, it may result in memory leaks, lost results, and wasted energy if improperly used [3, 4]. The other two constructs do not suffer from the same problems encountered by `AsyncTask` because they do not hold a reference to GUI [4]. `AsyncTaskLoader` is introduced after Android 3.0, and it only supports two GUI components: activity and fragment. Hence, Lin et al. developed ASYNCDROID [4] to refactor `AsyncTask`-related code into using `IntentService`, a more general and safer async construct.

We conduct a systematic study on `IntentService` to check whether the async construct is used properly and whether hackers can take advantage of unprotected intents to launch attacks. We find that in Android 6, 974 intents are not well protected and hence can be sent by third-party applications. Based on this observation, we develop a tool, ATUIN (short for ATtacks by exploiting Unprotected INtents), to demonstrate the feasibility of attacking CPU automatically by periodically sending unprotected intents that can be processed by Android system. Furthermore, by inspecting unprotected intents, we discover tens of critical vulnerabilities that have not been reported before, such as Wi-Fi DoS, telephone signal block, SIM card removal, homescreen hiding and NFC state cheating.

Overall, our study has three major contributions:

- We conduct the *first* systematic study on IntentService, and discover nearly one thousand unprotected intents in Android 6, which could be exploited to launch Denial-of-Service attacks on the system.
- We develop ATUIN to demonstrate the feasibility of attacking CPU automatically by periodically sending unprotected intents that can be handled by Android system.

– We further discover tens of critical unreported system vulnerabilities that can disable some key functionalities of smartphones (e.g., Wi-Fi, telephone, launch activity).

The rest of this paper is organized as follows. Section 2 gives a motivating example. Section 3 briefly introduce background knowledge. Section 4 presents the approach and results of the systematic study. Section 5 details the implementation of ATUIN and its experimental results. Section 6 elaborates on the discovered vulnerabilities. We discuss the threats to validity in Section 7. Related studies are briefly discussed in Section 8. This paper concludes in Section 9.

2 Motivating Example

This section shows a real vulnerability in Android 6 which involves an unprotected intent `ACTION_STEP_IDLE_STATE`. By exploiting the intent, any third-party applications without requiring any permissions can force a smartphone to leave `IDLE` state immediately after it enters `IDLE` state. Therefore, hackers can deplete the battery power of Android phones quickly.

To save power, Android 6 introduces a new feature, so-called Doze mode, which is able to reduce power consumption aggressively by forbidding or deferring critical tasks when the smartphone is in `IDLE` state. In implementation, Android 6 defines seven states, in which only `IDLE` state can save power. Fig. 1 shows a part of the code implementing state transitions.

```
240    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
241        @Override public void onReceive(Context context, Intent intent) {
……
253            } else if (ACTION_STEP_IDLE_STATE.equals(intent.getAction())) {
254                synchronized (DeviceIdleController.this) {
255                    stepIdleStateLocked();
256                }
……
1252    void stepIdleStateLocked() {
……
1265        switch (mState) {
1266            case STATE_INACTIVE:
1267                // We have now been inactive long enough, it is time to start looking
1268                // for significant motion and sleep some more while doing so.
1269                startMonitoringSignificantMotion();
1270                scheduleAlarmLocked(mConstants.IDLE_AFTER_INACTIVE_TIMEOUT, false);
1271                // Reset the upcoming idle delays.
1272                mNextIdlePendingDelay = mConstants.IDLE_PENDING_TIMEOUT;
1273                mNextIdleDelay = mConstants.IDLE_TIMEOUT;
1274                mState = STATE_IDLE_PENDING;
1275                if (DEBUG) Slog.d(TAG, "Moved from STATE_INACTIVE to       STATE_IDLE_PENDING.");
1276                EventLogTags.writeDeviceIdle(mState, "step");
1277                break;
1278            case STATE_IDLE_PENDING:
1279                mState = STATE_SENSING;
1280                if (DEBUG) Slog.d(TAG, "Moved from STATE_IDLE_PENDING to STATE_SENSING.");
1281                EventLogTags.writeDeviceIdle(mState, "step");
1282                scheduleSensingAlarmLocked(mConstants.SENSING_TIMEOUT);
```

Fig. 1: Vulnerable code in implementing Doze mode

More specifically, the core source file of Doze mode is `/services/core/java/com/android/server/DeviceIdleController.java`, which defines a pending intent `ACTION_STEP_IDLE_STATE`. When a pre-established time slice expires, the `AlarmManager` sends the intent to trigger state transition. We can see that `DeviceIdleController.java` registers a broadcast receiver (Line 240), and if it receives the `ACTION_STEP_IDLE_STATE` intent (Line 253), the function `stepIdleStateLocked` (Line 255) will be invoked. The code (Line 1266 to 1277) demonstrates that Android system transfers from `INACTIVE` state (Line 1266) to `IDLE_PENDING` state (Line 1274) once the `stepIdleStateLocked` is invoked.

However, the implementation of Doze mode is vulnerable since the critical intent `ACTION_STEP_IDLE_STATE` is unprotected, indicating that any third-party applications can send the intent without requiring any permissions. Therefore, an attacker can deplete the battery quickly by tricking innocents to install the malware which detects current state and then sends the specific intent if Android is in `IDLE`. We have to mind that tricking users to install the malware would not be difficult since the malware does not need any permissions. A three-hour testing shows that an LG Nexus 5X under attack consumes 6X more power than the normal situation. A detailed description of the attack can refer to our previous work [5].

3 Background

This section introduces the background knowledge closely related to this study. First, we present the overall architecture of the Android system. Then, we focus on the four components of the application. Finally, we will describe the main techniques examined in this paper, namely ICC (Inter-Component Communication) and asynchronous programming.

3.1 Android System Infrastructure

Android is an operating system for mobile devices such as smartphones and tablet computers, which is developed by the Open Handset Alliance led by Google. Android has evolved quickly since its first commercial version Android 1.0 that was released on September 2008. The newest version, Android 7 whose code name is Nougat was released on August 22, 2016.

Although Android evolves quickly, its infrastructure keeps stable, as shown in Fig.2. Android consists of five parts: Linux Kernel, Android Runtime, Libraries, Application Framework, and Applications. Linux Kernel manages hardware drivers, network, battery, system security and memory etc. Android Runtime consists of Core Libraries which provide most functionalities in Java core libraries, and a DVM (Dalvik Virtual Machine). Android can run multiple DVMs simultaneously, with one application in each DVM.

Application Framework provides a set of services (e.g., `ResourceManager`, `NotificationManager`, `ActivityManager`) for applications, so programmers

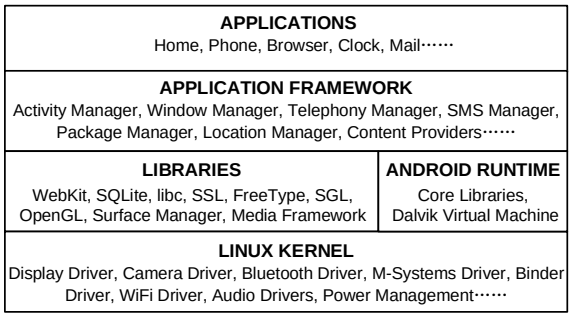

Fig. 2: Infrastructure of Android system

can develop varied applications by invoking framework's APIs. Applications is the top layer of Android which hosts built-in applications and third-party applications. The layered infrastructure ensures that the lower layers provide services for higher layers, and also benefits programmers for different layers concentrating on their own layers.

3.2 Android Application Structure

An Android application has at least one of the following components: `Activity`, `Service`, `BroadcastReceiver` and `ContentProvider`. An activity is the entry point for interacting with the user. It represents a single screen with a user interface. A service is a general-purpose entry point for keeping an application running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.

A broadcast receiver provides a new approach to the Android system for sending events to the app, and empowers the app to receive the announcements broadcasted by the system. Because broadcast receivers are another well-defined entry into the application, the system can deliver broadcasts even to applications that aren't currently running. A content provider manages a shared set of application data that you can store in the file system, in an `SQLite` database, on the web, or on any other persistent storage location that your application can access. Through the content provider, other applications can query or modify the data if the content provider allows it.

3.3 ICC

Different components in an application can communicate using ICC objects, mainly Intents. By the same way, components can also communicate across applications, allowing developers to reuse functionality. For example, `Google`

`Map` provides navigation function, so any restaurant applications just need to give the location coordinates and invoke `GoogleMap` for navigation by sending appropriate intent. Android intents are two types in nature.

- Explicit intents, explicitly define the exact component which should be called by the Android system, by using the Java class are identifier. Explicit intents are often used in ICC within the application because the name of invoked should be given correctly.
- Implicit intents specify the action which should be performed by other components or applications. Implicit intents are usually used for IAC (Inter-Application Communication) since the action should be performed rather than the exact name of the called component should be specified.

3.4 Android Asynchronous Programming

To ease asynchronous programming which is a widely-used approach to reduce application latency, Android provides three major async constructs: `AsyncTask`, `IntentService`, and `AsyncTaskLoader`.

- `AsyncTask` provides a `doInBackground` method for encapsulating asynchronous work. Besides, it provides four event handlers (i.e., `onPreExecute`, `onProgressUpdate`, `onPostExecute` and `OnCancel`) which are run in the UI thread. The `doInBackground` and these event handlers share variables through which the background task can communicate with UI [4]. Fig.3 depicts the workflow of `AsyncTask`. `AsyncTasks` should ideally be used for short operations (a few seconds at the most).

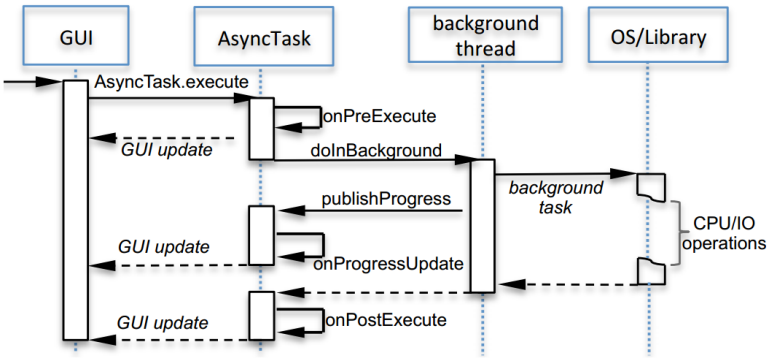

Fig. 3: The workflow of AsyncTask [4]

- `IntentService` is a base class for `Services` that handle asynchronous requests (expressed as `Intents`) on demand [17]. Clients send requests through `startService` (intent) calls; the service is started as needed, handles each intent in turn using a worker thread, and stops itself when

it runs out of work. Please note that all requests are handled on a single worker thread - they may take as long as necessary and will not block the application's main loop [17]. To get the task result, the GUI that starts the service should register a broadcast receiver. After the task is finished, `IntentService` sends its task result via the `sendBroadcast` method. Once the registered receiver on GUI receives this broadcast, its `onReceive` method will be executed on UI thread, so it can get the task result and update GUI [4]. The workflow of `IntentService` is given in Fig. 4. Therefore, `IntentService` is a good choice for long-running tasks.

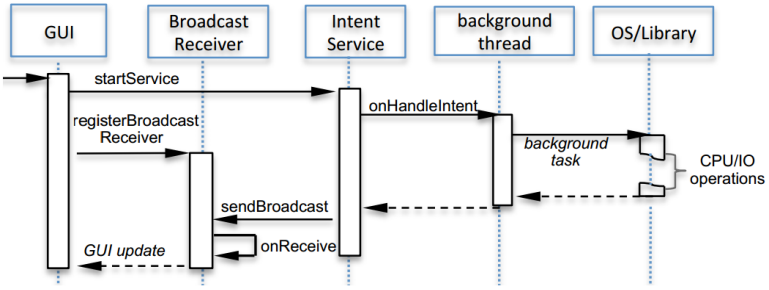

Fig. 4: The workflow of IntentService [17]

– `AsyncTaskLoader`, as it name suggests, is built on top of `AsyncTask`, and it provides similar handlers as `AsyncTask`. Unlike `AsyncTask`, `AsyncTaskLoader` is lifecycle aware: Android system binds/unbinds the background task with GUI according to GUIs lifecycle [4].

4 Unprotected Intents

In this paper, we examine Android 6 because it accounts for 26.4% market share in November 2016 [6] instead of the newest Android N/7 because it is installed on very few smartphones, about just 0.4% [6]. All experiments are conducted on a real smartphone, Huawei Nexus 6P.

We develop a tool to find all unprotected intents defined in Android 6 automatically, which consists of the following steps. First, the tool parses the source code of Android system to search for this pattern "new intent", because all intents should be defined according to the pattern. Then, the tool searches for the types of intents by exploring the fact that Android always defines the types in the beginning of source files as constant strings. After eliminating the duplicate intent types, we get 1,235 intents defined in Android 6.

Then, we determine all protected intents that should not be sent by third-party applications by analyzing the manifest file `/frameworks/base/core/res/AndroidManifest.xml` because Android lists all protected intents in it. We find that all protected intents are defined in a fixed pattern like `<protected-broadcast`

`android:name=XXX/>`, where `XXX` is a string indicating the intent type. Android forbids third-party applications to send protected intents as follows: (1) before forwarding an intent to the target component, Android system checks whether the intent is in the protected list; (2) if so, Android checks whether the application that sends the intent has system privilege; (3) if not, Android system terminates the application with a crash. Our tool parses all manifest files and extracts 261 protected intents from them, and therefore the number of unprotected intents should be 974.

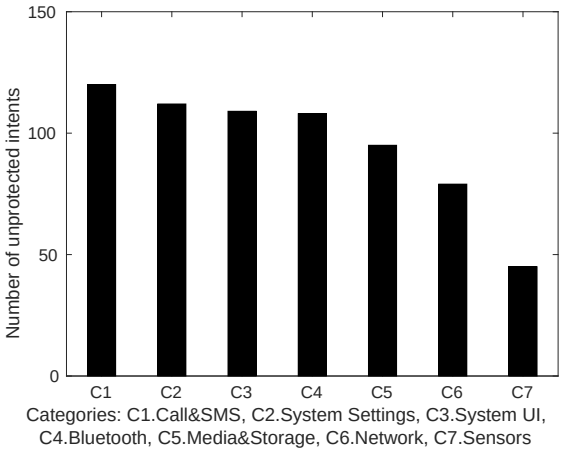

Fig. 5: Numbers of different categories of unprotected intents

We count high-risk unprotected intents that involve hardware and system operations, and classify them into 7 categories, as shown in Fig.5. For example, the intent `android.provider.Telephony.SMS_REJECTED` belongs to the Call&SMS category. One observation from Fig.5 is that system components use unprotected intents for async operations and communications frequently, indicating that unprotected intents would be good choices to attack Android system. For instance, there are 120, 112, 109 unprotected intents belong to Call&SMS, System Settings, System UI, respectively. Section 5 and Section 6 will present the attacking sceneries by exploiting the unprotected intents.

5 ATUIN: Attacks by Exploiting Unprotected Intents Automatically

This section details the design and implementation of our tool, ATUIN, to demonstrate the feasibility of attacking CPU automatically. That is, it will result in high CPU utilization ratio, thus the responsiveness of smartphones can be weakened. The basic idea of this attack is to force Android system

to repeatedly execute heavy-weight functions for handling intents by sending those intents periodically.

The proposed CPU attack is stealthy, although it is not complicated for the following reasons. First, the malware itself does not contain much code to be executed; instead, it forces Android to execute a lot of system code. Second, the malware does not need any permissions so that it can evade permission-based detection approaches. Moreover, a hacker can adjust attacking strength flexibly by setting the speed of sending intents.

To launch an effective and efficient CPU attack, ATUIN aims to use the intents that force Android system to spend computational resources to handle them. Note that Android system will not process all intents. For example, the intents that are used for informing third-party applications about the change of system state will not be processed by Android system. Actually, Android system just sends those intents, rather than receiving them. If any third-party applications send the intents without processing code, Android system simply discards them.

To the end, ATUIN follows the workflow as shown in Fig.6, which consists of three steps. The first step is finding all statically registered broadcast receivers. According to Android programming guides, all statically registered broadcast receivers should be listed in `manifest.xml`. Therefore, ATUIN parses all manifest files and extracts necessary information from them, such as which component can receive broadcasts and which types of broadcasts can be received. Fortunately, Android defines a fixed pattern to register broadcast receivers in manifest files, facilitating the parsing process of ATUIN.

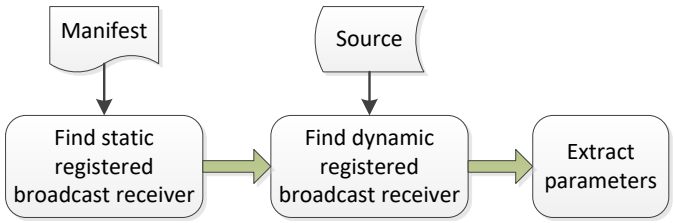

Fig. 6: Workflow of ATUIN

Fig.7 gives a code snippet in `/packages/apps/Bluetooth/AndroidManifest.xml`. The code highlighted by blue lines indicates the keywords searched by ATUIN. For example, ATUIN searches for `<receiver` and `</receiver>` to locate the registration of a broadcast receiver, and searches for `android:name=` to find the component that receives intents. Moreover, ATUIN looks for `<intent-filter>` and `</intent-filter>` to locate intent filters. Then, ATUIN searches for the pattern `<actionandroid:name=` to find the type of intent that can be processed.

The code underlined by red lines contains the information ATUIN required. For example, Fig.7 indicates that the component `.opp.BluetoothOp`

```
104     <receiver
105         android:process="@string/process"
106         android:exported="true"
107         android:name=".opp.BluetoothOppReceiver"
108         android:enabled="@bool/profile_supported_opp">
109         <intent-filter>
110             <action android:name="android.bluetooth.adapter.action.STATE_CHANGED" />
111             <!--action android:name="android.intent.action.BOOT_COMPLETED" /-->
112             <action android:name="android.btopp.intent.action.OPEN_RECEIVED_FILES" />
113         </intent-filter>
114     </receiver>
```

Fig. 7: A statically registered broadcast receiver

`pReceiver` can receive two types of intents, `android.bluetooth.adapter.action.STATE_CHANGED` and `android.btopp.intent.action.OPEN_RECEIVED_FILES`.

The second step is discovering dynamically registered broadcast receivers that are widely-used when developers want to control the life circle of the broadcast receivers. ATUIN finds this kind of broadcast receivers by code analysis. Note that Android enables applications to register broadcast receivers dynamically (i.e., the framework API, `registerReceiver` should be invoked). Fig.8 illustrates how the component `GsmServiceStateTracker` registers a broadcast receiver at runtime to receive the intent `ACTION_RADIO_OFF`.

```
252         filter = new IntentFilter();
253         Context context = phone.getContext();
254         filter.addAction(ACTION_RADIO_OFF);
255         context.registerReceiver(mIntentReceiver, filter);
```

Fig. 8: A dynamically registered broadcast receiver

ATUIN firstly searches for the API invocation, `registerReceiver`, and then gets the second parameter, `filter` in this example. Afterwards, ATUIN looks for the API invocation, `addAction` before the invocation of `registerReceiver`, and then gets the parameter, `ACTION_RADIO_OFF`. Finally, ATUIN searches for the definition of `ACTION_RADIO_OFF` which is a constant string in Android source code.

The third step is extracting the data attached to the intent since ATUIN aims to trigger the processing code of the corresponding intent. If the data is not provided correctly, the processing logic will abort quickly, result in non-obvious attacking effect. ATUIN conducts inter-procedural data flow analysis to discover valid data parameters of intents. For a better understanding of the inter-procedural analysis, we take the code in Fig.27 as an example. The code `getIntExtra(NfcAdapter.EXTRA_ADAPTER_STATE,NfcAdapter.STATE_OFF)` indicates that the parameter is named `EXTRA_ADAPTER_STATE` and it is an integer. Then, inter-procedural data flow analysis shows that the data attached in the intent is passed as a parameter `newState` of the function `handleNfcStateChanged`. After that, ATUIN searches for the statement that `newState` is compared with a constant integer since the comparison is used for executing the corresponding program logic for different data. Hence, ATUIN finds that

the attached data, `NfcAdapter.EXTRA_ADAPTER_DATA` can be set as `NfcAdapter.STATE_OFF`, `NfcAdapter.STATE_ON`, `NfcAdapter.STATE_TURNING_OFF`, or `NfcAdapter.STATE_TURNING_OFF`. According to this process, ATUIN can extract and set parameters attached to the targeted intent.

The experiments consist of three sceneries: no attacks, attack by sending 20 intents with and without processing code respectively, as shown in Fig.9 to Fig.11. The observation is that our tool attacks CPU effectively, i.e., CPU utilization ratio rises from 11.17% to 71.13%. Moreover, to adapt to heavy workload, Android adjusts CPU frequency from 652.8MHz to 1.22GHz. On the contrary, attacking by sending the intents without processing code can only slightly increase CPU utilization ratio by 4.74%.

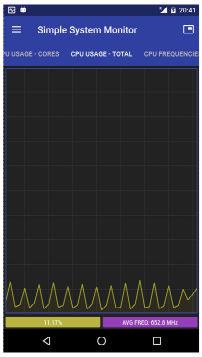

Fig. 9: CPU utilization ratio without attacks

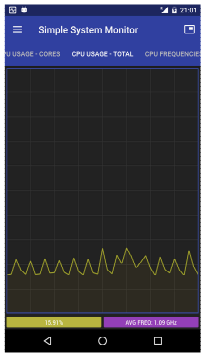

Fig. 10: CPU utilization ratio when sending 20 intents without processing code

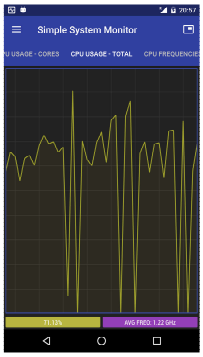

Fig. 11: CPU utilization ratio when sending 20 intents with processing code

6 Case Studies: Critical Vulnerabilities

In this section, we analyze the unprotected intents and examine the negative effect on Android system if they are exploited by an attacker. In particular, we investigate the processing code of selected unprotected intents in depth and find tens of critical vulnerabilities that have not been reported before. This section describes five important vulnerabilities. The attacks exploiting each vulnerability are recorded and the videos can be found at `https://goo.gl/QZn7Rk`.

Since it is time-consuming to analyze each unprotected intent manually, we prefer to the unprotected intents that either (1) change system states; (2) operate hardware component (e.g., Wi-Fi, SIM card, UI); (3) get access to private information (e.g., contact, photos). Then we analyze the processing code and generate attacks manually. We try four heuristic strategies to launch attacks. The first is sending an unprotected intent once with valid data. The second is sending an unprotected intent once with invalid data. The third is sending an unprotected intent with valid data repeatedly at a higher rate. The

last is sending an unprotected intent with invalid data repeatedly at a higher rate.

The last step is checking whether the performance or functionalities of Android system or applications are impaired. To do so, we try each critical functionalities manually, such as Wi-Fi, Bluetooth, Telephone to examine whether they can work as usual. Moreover, we resort to behavior monitoring tool (e.g., DroidBox [34]) to find abnormal behaviors as well as privacy leakage. Furthermore, we use performance profiling tool (e.g., Android Studio Performance Profiling Tools [35]) to discover abnormal performance degradation, such as high CPU utilization ratio, fast power depletion, excessive memory consumption.

6.1 Wi-Fi DoS

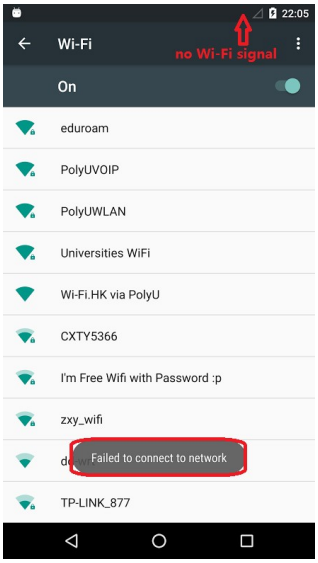

Fig. 12: Symptom after Wi-Fi DoS attack

This attack takes advantage of the unprotected intent: `ACTION_DEVICE_IDLE` that is defined in `/frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiController.java`. It is easy to reproduce the attack

which broadcasts the intent without data attached to the intent. After the attack is successfully launched, Wi-Fi signal will be blocked and Android system cannot connect to any access points, as shown in Fig.12.

The vulnerability-related code is shown in Fig.13. The component `WifiController` registers a broadcast receiver (Line 181), and if one `ACTION_DEVICE_IDLE` intent is received (Line 185), a message termed by `CMD_DEVICE_IDLE` will be sent. `WifiController` defines a routine, `processMessage` (Line 710) to handle all Wi-Fi related commands. In particular, if the message is `CMD_DEVICE_IDLE`, the function `checkLocksAndTransitionWhenDeviceIdle` will be invoked. In this function, we can find state transitions.

```
180         mContext.registerReceiver(
181                 new BroadcastReceiver() {
182                     @Override
183                     public void onReceive(Context context, Intent intent) {
184                         String action = intent.getAction();
185                         if (action.equals(ACTION_DEVICE_IDLE)) {
186                             sendMessage(CMD_DEVICE_IDLE);
……
710         public boolean processMessage(Message msg) {
711             if (msg.what == CMD_DEVICE_IDLE) {
712                 checkLocksAndTransitionWhenDeviceIdle();
……
784     private void checkLocksAndTransitionWhenDeviceIdle() {
785         if (mLocks.hasLocks()) {
786             switch (mLocks.getStrongestLockMode()) {
787                 case WIFI_MODE_FULL:
788                     transitionTo(mFullLockHeldState);
789                     break;
790                 case WIFI_MODE_FULL_HIGH_PERF:
791                     transitionTo(mFullHighPerfLockHeldState);
792                     break;
793                 case WIFI_MODE_SCAN_ONLY:
794                     transitionTo(mScanOnlyLockHeldState);
795                     break;
796                 default:
797                     loge("Illegal lock " + mLocks.getStrongestLockMode());
798             }
799         } else {
800             if (mSettingsStore.isScanAlwaysAvailable()) {
801                 transitionTo(mScanOnlyLockHeldState);
802             } else {
803                 transitionTo(mNoLockHeldState);
804             }
805         }
806     }
```

Fig. 13: Vulnerable code exploited by Wi-Fi DoS attack

Android defines twelve states (as shown in Fig.14) and maintains state transitions in `/frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiController.java`. The twelve states are not in the same hierarchy; instead, some states are the sub-states of another state. For instance, `addState(mApStaDisabledState,mDefaultState)` indicates that `mApStaDisabledState` is a sub-state of `mDefaultState`. The complete hierarchical relation of states is depicted in Fig.15. We can see that `mDefaultState` is the parent state of all other states and all states involving the function `checkLocksAndTransitionWhenDeviceIdle` are the sub-states of `mDeviceInactiveState`.

```
122    private DefaultState mDefaultState = new DefaultState();
123    private StaEnabledState mStaEnabledState = new StaEnabledState();
124    private ApStaDisabledState mApStaDisabledState = new ApStaDisabledState();
125    private StaDisabledWithScanState mStaDisabledWithScanState = new StaDisabledWithScanState();
126    private ApEnabledState mApEnabledState = new ApEnabledState();
127    private DeviceActiveState mDeviceActiveState = new DeviceActiveState();
128    private DeviceInactiveState mDeviceInactiveState = new DeviceInactiveState();
129    private ScanOnlyLockHeldState mScanOnlyLockHeldState = new ScanOnlyLockHeldState();
130    private FullLockHeldState mFullLockHeldState = new FullLockHeldState();
131    private FullHighPerfLockHeldState mFullHighPerfLockHeldState = new FullHighPerfLockHeldState();
132    private NoLockHeldState mNoLockHeldState = new NoLockHeldState();
133    private EcmState mEcmState = new EcmState();
……
146        addState(mDefaultState);
147            addState(mApStaDisabledState, mDefaultState);
148            addState(mStaEnabledState, mDefaultState);
149                addState(mDeviceActiveState, mStaEnabledState);
150                addState(mDeviceInactiveState, mStaEnabledState);
151                    addState(mScanOnlyLockHeldState, mDeviceInactiveState);
152                    addState(mFullLockHeldState, mDeviceInactiveState);
153                    addState(mFullHighPerfLockHeldState, mDeviceInactiveState);
154                    addState(mNoLockHeldState, mDeviceInactiveState);
155            addState(mStaDisabledWithScanState, mDefaultState);
156            addState(mApEnabledState, mDefaultState);
157            addState(mEcmState, mDefaultState);
```

Fig. 14: Definitions of twelve states of Wi-Fi

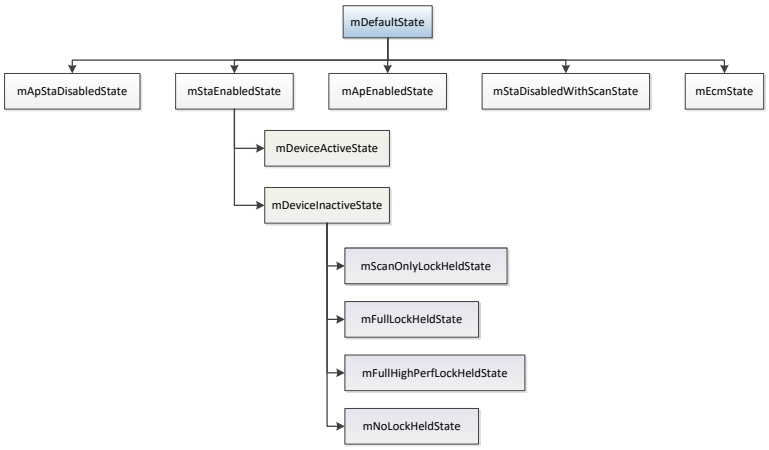

Fig. 15: Hierarchical relation of states

Our attack implements a piece of malware without requiring permissions which sends the unprotected intent, `CMD_DEVICE_IDLE`. After processing by `checkLocksAndTransitionWhenDeviceIdle`, the parent state, `mDefaultStat e` will handle this intent, as shown in Fig.16. At Line 359, a global variable `mDeviceIdle` is set to `true`.

```
358    case CMD_DEVICE_IDLE:
359        mDeviceIdle = true;
360        updateBatteryWorkSource();
361        break;
```

Fig. 16: Processing code in mDefaultState

When users tag the Wi-Fi slider (as shown in Fig.12), the component `wifi Service` will generate an intent, `CMD_WIFI_TOGGLED`. However, none of the five sub-states of `mDeviceInactiveState` can handle this intent. Interestingly, `mD eviceInactiveState` cannot process this intent, and hence this intent will be forwarded to its parent state, `mStaEnabledState`. Afterwards, `mStaEnabledS tate` handles `CMD_WIFI_TOGGLED` as shown in Fig.17, indicating that Android system will transfer to one of the two states, `mStaDisabledWithScanState` and `mApStaDisabledState`.

```
489    class StaEnabledState extends State {
490        @Override
491        public void enter() {
492            mWifiStateMachine.setSupplicantRunning(true);
493        }
494        @Override
495        public boolean processMessage(Message msg) {
496            switch (msg.what) {
497                case CMD_WIFI_TOGGLED:
498                    if (! mSettingsStore.isWifiToggleEnabled()) {
499                        if (mSettingsStore.isScanAlwaysAvailable()) {
500                            transitionTo(mStaDisabledWithScanState);
501                        } else {
502                            transitionTo(mApStaDisabledState);
503                        }
```

Fig. 17: Processing code in mStaEnabledState

`mStaDisabledWithScanState` and `mApStaDisabledState` process `CMD_WI FI_TOGGLED` in a similar way, as shown in Fig.18. If the variable `mDeviceIdle` is `false`, Android system will transfer to `mDeviceActiveState`, the state a smartphone can connect to access points. However, our attack makes `mDevic eIdle` be `true` by sending the unprotected intent, `ACTION_DEVICE_IDLE`. As a consequence, we successfully DoS the Wi-Fi component.

```
554        public boolean processMessage(Message msg) {
555            switch (msg.what) {
556                case CMD_WIFI_TOGGLED:
557                    if (mSettingsStore.isWifiToggleEnabled()) {
558                        if (doDeferEnable(msg)) {
559                            if (mHaveDeferredEnable) {
560                                // have 2 toggles now, inc serial number and ignore both
561                                mDeferredEnableSerialNumber++;
562                            }
563                            mHaveDeferredEnable = !mHaveDeferredEnable;
564                            break;
565                        }
566                        if (mDeviceIdle == false) {
567                            transitionTo(mDeviceActiveState);
568                        } else {
569                            checkLocksAndTransitionWhenDeviceIdle();
570                        }
```

Fig. 18: Processing code in mStaDisabledWithScanState

6.2 Telephone Signal Block

Our attack exploits the unprotected intent, `ACTION_RADIO_OFF` that is defined in `/frameworks/opt/telephony/src/java/com/android/internal/telephony/ServiceStateTracker.java`. Since the definition of this intent cannot be found in Android API, it should be reserved for internal use only. However, Android 6 does not protect the intent, and thus any third-party applications can send the intent without restrictions.

After sending one `ACTION_RADIO_OFF` intent, the signal of the smartphone will be cut immediately and the signal will reappear in a short while. Therefore, by sending the intent periodically, we can block telephone signal at all, as shown in Fig.19. The most obvious symptom is that a smartphone under attack cannot make or receive telephone calls.

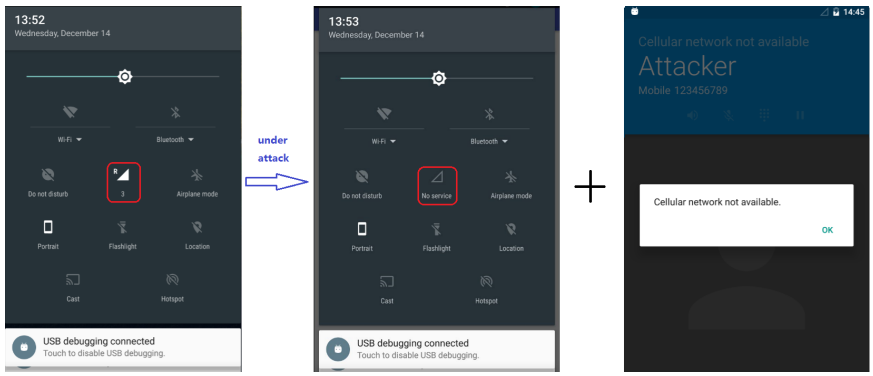

Fig. 19: Symptom after telephone signal block attack

The vulnerable code is located in `/frameworks/opt/telephony/src/java/com/android/internal/telephony/gsm/GsmServiceStateTracker.java` (as shown in Fig.20). It registers a broadcast receiver (Line 171) to receive the intent (Line 183). When an `ACTION_RADIO_OFF` is received, the function `powerOffRadioSafely` is invoked (Line 186), where the function `hangupAndPowerOff` is called (Line 2153). `hangupAndPowerOff` firstly hangs up all active phone calls if any (Line 562 to 566), and then powers off the radio by invoking `setRadioPower` (Line 568). The immediate observation of our attack is that telephone signal vanishes because the radio component is turned off.

6.3 SIM Card Removal

Our attack can disable all SIM-card-related functionalities, such as making/receiving calls, sending/receiving SMS messages by periodically sending an unprotected intent, `ACTION_CARRIER_CONFIG_CHANGED` that is defined in `/frameworks/base/telephony/java/android/telephony/CarrierConfigManager.`

```
560    protected void hangupAndPowerOff() {
561        // hang up all active voice calls
562        if (mPhone.isInCall()) {
563            mPhone.mCT.mRingingCall.hangupIfAlive();
564            mPhone.mCT.mBackgroundCall.hangupIfAlive();
565            mPhone.mCT.mForegroundCall.hangupIfAlive();
566        }
567
568        mCi.setRadioPower(false, null);
569    }
……
171    private BroadcastReceiver mIntentReceiver = new BroadcastReceiver() {
172        @Override
173        public void onReceive(Context context, Intent intent) {
……
183            } else if (intent.getAction().equals(ACTION_RADIO_OFF)) {
184                mAlarmSwitch = false;
185                DcTrackerBase dcTracker = mPhone.mDcTracker;
186                powerOffRadioSafely(dcTracker);
……
2140    public void powerOffRadioSafely(DcTrackerBase dcTracker) {
2141        synchronized (this) {
2142            if (!mPendingRadioPowerOffAfterDataOff) {
……
2153                    hangupAndPowerOff();
```

Fig. 20: Vulnerable code exploited by telephone signal block

`java` as shown in Fig.21. Interestingly, the explanation for the intent definition shows that this intent should be sent by the system. However, Android 6 does not protect the intent from being sent by third-party applications.

```
48    /**
49     * This intent is broadcast by the system when carrier config changes.
50     */
51    public static final String
52            ACTION_CARRIER_CONFIG_CHANGED = "android.telephony.action.CARRIER_CONFIG_CHANGED";
```

Fig. 21: Definition of `ACTION_CARRIER_CONFIG_CHANGED`

The component `SimChangeReceiver` (as shown in Fig.22) registers a broadcast receiver to receive the intent (Line 42) and processes the intent in corresponding callback function (Line 46). When an `ACTION_CARRIER_CONFIG_CHANGED` intent is received, the settings of SIM-card-related components, such as voice mail and phone account, are refreshed. By sending the unprotected intent repeatedly, the attack is capable of denying the services of SIM-card-related components. Fig.23 demonstrates that the tested smartphone can neither find the SIM card nor make calls under the attack.

6.4 Homescreen Hiding

Android's homescreen provides shortcuts to applications, which is an user-friendly design. Our attack is able to hide all shortcuts on the homescreen by just sending unprotected intents periodically. We find that two intents can be

```
42 public class SimChangeReceiver extends BroadcastReceiver {
43     private final String TAG = "SimChangeReceiver";
44
45     @Override
46     public void onReceive(Context context, Intent intent) {
……
58         switch (action) {
……
66             case CarrierConfigManager.ACTION_CARRIER_CONFIG_CHANGED:
……
84                     if (!isUserSet) {
85                         // Preserve the previous setting for "isVisualVoicemailEnabled" if it is
86                         // set by the user, otherwise, set this value for the first time.
87                         VisualVoicemailSettingsUtil.setVisualVoicemailEnabled(context, phoneAccount,
88                                 isEnabled, /** isUserSet */ false);
89                     }
90
91                     if (isEnabled) {
92                         LocalLogHelper.log(TAG, "Sim state or carrier config changed: requesting"
93                                 + " activation for " + phoneAccount.getId());
94
95                         // Add a phone state listener so that changes to the communication channels
96                         // can be recorded.
97                         OmtpVvmSourceManager.getInstance(context).addPhoneStateListener(
98                                 phoneAccount);
99                         carrierConfigHelper.startActivation();
100                     } else {
101                         // It may be that the source was not registered to begin with but we want
102                         // to run through the steps to remove the source just in case.
103                         OmtpVvmSourceManager.getInstance(context).removeSource(phoneAccount);
104                         Log.v(TAG, "Sim change for disabled account.");
105                     }
```

Fig. 22: Vulnerable code exploited by SIM card removal

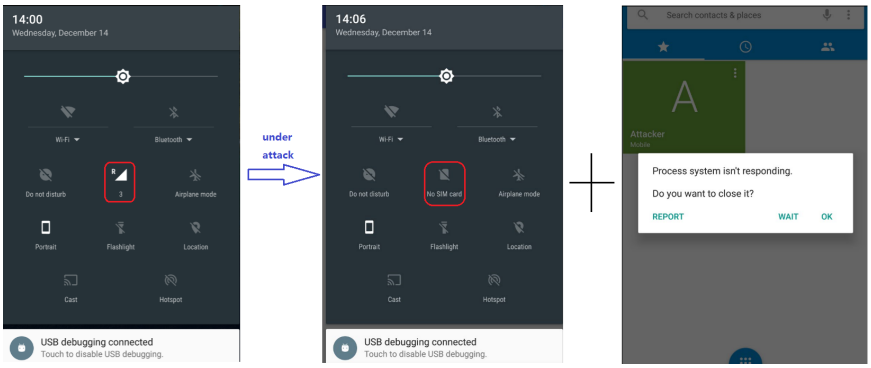

Fig. 23: Symptom after SIM card removal attack

exploited to achieve the same attacking effect, which are `ACTION_MANAGED_PROFILE_ADDED` and `ACTION_MANAGED_PROFILE_REMOVED`.

Android uses the same piece of code to process the two intents, which are in `/packages/apps/Launcher3/src/com/android/launcher3/LauncherModel.java`, as shown in Fig.24. The component `LauncherModel` registers a broadcast receiver to receive the two intents (Line 1281 and 1282). If any one of them is received, the function `forceReload` (Line 1284) will be invoked, in which the launch activity will be reloaded. Note that the launch activity corresponds to the homescreen. Hence, the shortcuts will be hidden if the launch activity reloads in a fast rate, as shown in Fig.25.

```
1269    public void onReceive(Context context, Intent intent) {
……
1281        } else if (LauncherAppsCompat.ACTION_MANAGED_PROFILE_ADDED.equals(action)
1282                || LauncherAppsCompat.ACTION_MANAGED_PROFILE_REMOVED.equals(action)) {
1283            UserManagerCompat.getInstance(context).enableAndResetCache();
1284            forceReload();
1285        }
1286    }
1287
1288    void forceReload() {
1289        resetLoadedState(true, true);
1290
1291        // Do this here because if the launcher activity is running it will be restarted.
1292        // If it's not running startLoaderFromBackground will merely tell it that it needs
1293        // to reload.
1294        startLoaderFromBackground();
1295    }
```

Fig. 24: Vulnerable code exploited by homescreen hiding

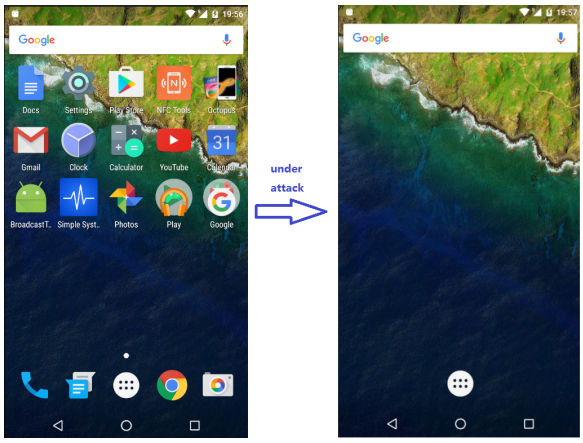

Fig. 25: Symptom after homescreen hiding attack

6.5 NFC State Cheating

Near Field Communication (NFC) is a set of short-range wireless technologies, allowing users to share small payloads of data between an NFC tag and an Android-powered device, or between two Android-powered devices. Our attack can change the UI which presents the state of NFC, and thus users will be cheated. This attack takes advantage of the intent `ACTION_ADAPTER_STATE_CHANGED` that is defined in `/frameworks/base/core/java/android/nfc/NfcAdapter.java`, as shown in Fig.26.

The component `NfcEnabler` processes the intent and updates UI, as shown in Fig.27. In particular, the callback function of the registered broadcast receiver invokes `handleNfcStateChanged` (Line 49) to process the intent. In this function, Android updates UI according to the change of states. Please note that Android defines four states to maintain NFC component (i.e., `STATE_OFF`, `STATE_TURNING_ON`, `STATE_ON`, and `STATE_TURNING_OFF`), as shown in Fig.26.

```
184    public static final String ACTION_ADAPTER_STATE_CHANGED =
185            "android.nfc.action.ADAPTER_STATE_CHANGED";
……
195    public static final String EXTRA_ADAPTER_STATE = "android.nfc.extra.ADAPTER_STATE";
196
197    public static final int STATE_OFF = 1;
198    public static final int STATE_TURNING_ON = 2;
199    public static final int STATE_ON = 3;
200    public static final int STATE_TURNING_OFF = 4;
```

Fig. 26: Definition of `ACTION_ADAPTER_STATE_CHANGED` and several states

```
44    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
45        @Override
46        public void onReceive(Context context, Intent intent) {
47            String action = intent.getAction();
48            if (NfcAdapter.ACTION_ADAPTER_STATE_CHANGED.equals(action)) {
49                handleNfcStateChanged(intent.getIntExtra(NfcAdapter.EXTRA_ADAPTER_STATE,
50                        NfcAdapter.STATE_OFF));
……
109    private void handleNfcStateChanged(int newState) {
110        switch (newState) {
111        case NfcAdapter.STATE_OFF:
112            mSwitch.setChecked(false);
113            mSwitch.setEnabled(true);
114            mAndroidBeam.setEnabled(false);
115            mAndroidBeam.setSummary(R.string.android_beam_disabled_summary);
116            break;
117        case NfcAdapter.STATE_ON:
118            mSwitch.setChecked(true);
119            mSwitch.setEnabled(true);
120            mAndroidBeam.setEnabled(!mBeamDisallowed);
121            if (mNfcAdapter.isNdefPushEnabled() && !mBeamDisallowed) {
122                mAndroidBeam.setSummary(R.string.android_beam_on_summary);
123            } else {
124                mAndroidBeam.setSummary(R.string.android_beam_off_summary);
125            }
126            break;
127        case NfcAdapter.STATE_TURNING_ON:
128            mSwitch.setChecked(true);
129            mSwitch.setEnabled(false);
130            mAndroidBeam.setEnabled(false);
131            break;
132        case NfcAdapter.STATE_TURNING_OFF:
133            mSwitch.setChecked(false);
134            mSwitch.setEnabled(false);
135            mAndroidBeam.setEnabled(false);
136            break;
137        }
```

Fig. 27: Vulnerable code exploited by NFC state cheating

UI changes according to state transitions. To be specific, if `NfcEnabler` thinks the state is `STATE_OFF`, the state of `NFC` slider (as shown in Fig.28) will be unchecked and can be changed by tapping. If `NfcEnabler` considers the state to be `STATE_ON`, the state of `NFC` slider will be checked and can be changed. If `NfcEnabler` considers the state to be `STATE_TURNING_ON`, the state of `NFC` slider will be checked, but it can not be changed by finger tapping. If `NfcEnabler` thinks the state is `STATE_TURNING_OFF`, the state of `NFC` slider will be unchecked and it can not be changed by finger tapping. Though source code inspection, we find the that both `STATE_TURNING_ON` and `STATE_TURNING_OFF` are intermediate states between `STATE_OFF` and `STATE_ON`. Therefore, if UI is in either intermediate states, the `NFC` slider can not respond to user interactions.

Our attack sends the unprotected intent that sets the attached data `EXTRA_ADAPTER_STATE` as `STATE_ON`. As a consequence, the UI indicates that NFC is enabled, however, the real state of NFC component is still turned off, as

shown in Fig.28. Our attack can also send the unprotected intent with `EXTRA_ADAPTER_STATE` setting as other values, making UI present other misleading states.

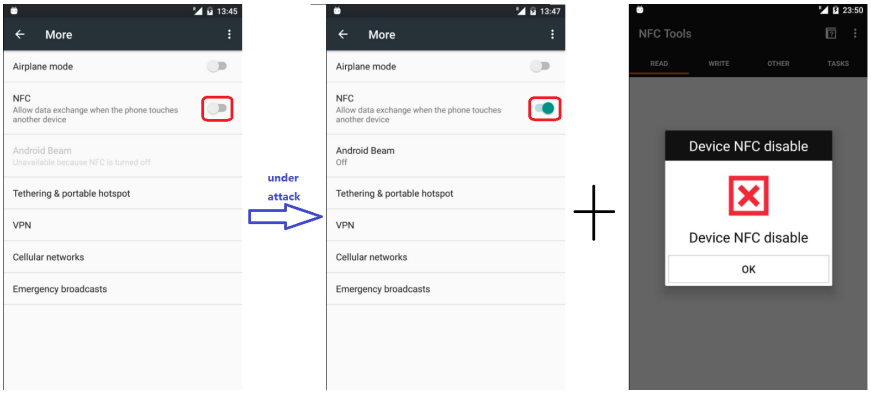

Fig. 28: Symptom after NFC state cheating attack

7 Threats to Validity

7.1 Internal Threats

There are some internal threats to the confidence in saying the study's results are correct. First, this work uses some programming patterns to find the definitions of intents, the declaration of protected intents. However, the patterns are concluded by manual code inspection. Hence, the number of intents, protected intents, and unprotected intents may not accurate since developers can declare intents while not following the patterns. Moreover, we classify unprotected intents according to their names. However, one intent may be processed by different components, increasing the difficulty of accurate classification.

Additionally, the explanations of the vulnerabilities presented in Section 6 depend on manual code analysis, that may be inaccurate. The reason lies that after the observation of attacks, we check the source manually, which is not guaranteed to be accurate. In future, we will validate the causes of attacks through dynamic debugging. Besides, the experiments of ATUIN randomly select 20 intents. However, the number and selection of intents can influence the experimental results. For example, the process of an intent involving heavy I/O operations may cause higher CPU utilization ratio than the intent whose processing code is much simpler. We leave the investigation of the selection strategy as future work.

7.2 External Threats

There are several external threats to the confidence in stating whether the study's results are applicable to other groups. First, we conduct all experiments on one device, Huawei Nexus 6P. We plan to carry out similar studies on other Android devices in future. Second, this work only studies Android 6, leaving the investigation of other versions as future work. Third, this study uses fixed programming patterns to find the definitions of intents and the declarations of protected intents. Other versions of Android may not use the same patterns.

8 Related Work

This work relates to the following research topics on Android, which are asynchronous-programming-related bugs, DoS attacks, resources-depletion attacks, intents-related bugs, and other performance bugs. This section briefly discusses the five categories of related studies separately.

8.1 Asynchronous-Programming-Related Bugs

The heavy workload in main thread is a well-known cause of many performance problems [1]. Android provides several async constructs (e.g., `AsyncTask`, `IntentService` and `AsyncTaskLoader`) that enable developers to put long-running tasks into background threads. However, existing studies [3–5, 9] show that developers have to use `AsyncTask` carefully to avoid security vulnerabilities.

Asynchronizer [3] is an automated refactoring tool that enables developers to extract long-running operations into `AsyncTask` and uses a points-to static analysis to determine the safety of the transformation. ASYNCDROID [4] is a refactoring tool which enables Android developers to transform existing improperly-used async constructs (i.e., `AsyncTask`) into correct constructs (i.e., `IntentService`).

DiagDroid [9] is a UI performance diagnosis tool, which is able to profile the asynchronous executions in a task granularity, equipping it with low-overhead and high compatibility merits. [5] is our previous work which investigates a new feature, Doze mode in Android 6. [5] finds one unprotected intent in the code for implementing Doze mode and proposes several approaches to deplete battery power by exploiting the intent.

8.2 DoS Attacks

This work discovers tens of vulnerabilities. By exploiting them, hackers can deny some critical services of Android 6, such as Wi-Fi, telephone signal, SIM card, and launch activities. As far as we know, none of the proposed attacks were covered by related studies. Huang et al. [18] proposed a new type of vulnerabilities, ASV (Android Stroke Vulnerabilities) which can lead to system

Services freezing and system server shutdown. ASV corresponds to a flaw in the design of the coarse-grained concurrency control in the core of Android, `SystemServer`, leading to a chance of DoS attacks. Based on the vulnerability, hackers can launch attacks in a straightforward way: writing a simple loop to call normal Android APIs to easily craft several exploits.

Different with their previous work [18], Liu's group discovers another vulnerability in `SystemServer`, that is the flaw in designing of synchronous callback mechanism [19]. By exploiting the vulnerability, they enable a malicious application to freeze critical system functionalities or soft-reboot the system immediately. After elaborative construction, they successfully to launch other meaningful attacks, such as anti anti-virus, anti process-killer, hindering app updates or system patching.

Armando et al. propose a DoS attack that makes devices become totally unresponsive [20]. Their work bases on the observation that Android sets security policies to protect `Zygote`, a process enables fast start-up for new processes from being exploited by attacks. However, the protection is weak that can be bypassed easily, resulting in a large number of dummy processes until all memory resources are exhausted. Eian and Mjolsnes [21] use formal method to identify deadlock vulnerability that causes DoS attacks in IEEE 802.11w protocol.

8.3 Resources-Depletion Attacks

This study implements ATUIN to attack CPU, leading to a high CPU utilization ratio. The related study aforementioned [20] can exhaust all memory resources. This section mainly focuses battery-draining attacks, which should be a severe threat to mobile devices since they are power-limited and not always plugged. Our previous work [5] drain battery silently by exploiting an unprotected intent.

Fiore et al. proposed to drain battery stealthily by sending the victims browser with unhearable audio files [22], for example, sounds below 20Hz. As a result, the power is wasted by playing unhearable music. Researchers found that Android applications can deplete battery (deliberately or unintentionally) by misusing power management APIs. To reduce battery consumption aggressively, Android exports wakelock-related APIs to application programmers. Hence, applications can keep the smartphone awake by acquiring a wakelock, and then allow it to sleep after releasing the wakelock. However, programmers sometimes forget to release wakelocks in each path [23, 24], or place wakelocks in wrong places [25, 27], incurring power waste or faulty program logic. NANSA [26] holds a partial wakelock, preventing CPU going to sleep and then stimulates benign applications to do power-intensive work when the screen is off.

8.4 Intents-Related Bugs

IntentFuzzer [14] generates intents to discover capability leaks of Android applications that finds more than 100 applications in Google play has at least one permission leak. Android system can be attacked by the web browsers which support intent scheme URLs. When parsing an intent scheme URL, the web browser will generate intents to launch activities. By exploiting intent scheme URLs, hackers can launch attacks including cookie file theft and universal XSS [15]. Schartner and Bürger [16] propose to insert malicious processing functions into Android system to handle intents as hackers' will. Based on the idea, they successfully hack the applications secured by mTANs, such as web-banking.

8.5 Other Performance Bugs

Guo et al. [28] developed a static analysis tool, `Relda` which detects energy and memory leaks as well as the resources never being released. Liu et al. [29] analyze 70 real performance bugs from eight Android applications and conclude three categories of performance bugs (i.e., GUI lagging, memory bloat, energy leak). Additionally, the authors propose `PerfChecker`, a static program analysis tool to identify two types of performance bugs: lengthy operations in the UI thread and violations of the view holder pattern. Linares-Vásquez et al. propose a taxonomy of practices and tools for detecting an fixing performance bottlenecks based on the survey with 485 developers [30]. [31–33] leverages cost-benefit analysis to detect whether an Android application uses sensory data in a cost-ineffective way.

StrictMode [10] is a developer tool provides by Android that aims at finding blocking operations in main thread. To reduce application latency, Tango [11] offloads some workload from the smartphone to a remote server. Tango replicates the application and executes it on both the client and the server, and allows either replica to lead the execution. Similar with Tango, Outatime [12] performs game execution and rendering on remote servers on behalf of thin clients that simply send input and display output frames. SmartIO [13] reduces the application delay by prioritizing reads over writes, and grouping them based on assigned priorities.

In summary, our work differs from related studies in the following aspects.

- We focus on the async construct, `IntentService`.
- We reveal that a lot of intents are unprotected from being manipulated by third-party applications.
- We discover tens of critical vulnerabilities which have not been reported before. To the best of our knowledge, our work is the *first* systematic study about hacking Android system by exploiting `IntentService`.

9 Conclusion

To reduce application latency, Android provides asynchronous programming which enables developers to put long-running tasks into background threads. This paper focuses on one async construct, `IntentService`. Through static program analysis, our work finds nearly one thousand unprotected intents which can be sent by third-party applications. Moreover, we implement a tool which is able to attack CPU by exploiting the unprotected intents automatically. Furthermore, we discover tens of critical vulnerabilities that have not been reported before.

We plan to extend our work from the following aspects. First, we are of great interest in designing an automated approach to discover critical vulnerabilities like those in Section 6. Second, we plan to conduct a similar systematic study on other Android versions, such as Android 7 (the newest version), Android 5 which still leads the market share at 34% [6]. Moreover, we plan to test ATUIN in different settings (e.g., different numbers of intents, selecting different intents).

Acknowledgements This work is supported in part by the Hong Kong GRF (PolyU 152279/16E), the HKPolyU Research Grants (G-YBJX), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), the National Natural Science Foundation of China (No.61402080, No.61572115, No.61502086, No.61572109), and China Postdoctoral Science Foundation founded project (No.2014M562307).

References

1. Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In Proc. ICSE, 1013–1024, 2014.
2. S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in Android applications. In Proc. MOBS, 1–6, 2013.
3. Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for android applications through refactoring. In Proc. FSE, 2014, 341-352, 2014.
4. Y. Lin, C. Radoi, and D. Dig. (2015, November). Study and Refactoring of Android Asynchronous Programming. In Proc. ASE, 224–235, 2015.
5. T. Chen, H. Tang, K. Zhou, X. Zhang, and X. Lin. Silent Battery Draining Attack Against Android Systems by Subverting Doze Mode. In Proc. GlobeCom, 2016.
6. Bandla. Android Version Share: Lollipop still leads with 34%, Nougat at 0.4%. `http://www.gadgetdetail.com/android-version-market-share-distribution/`.
7. C. Amrutkar, M. Hiltunen, T. Jim, K. Joshi, O. Spatscheck, P. Traynor, and S. Venkataraman. Why is my smartphone slow? on the fly diagnosis of underperformance on the mobile internet. In Proc. DSN, 1–8, 2013.
8. L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In Proc. SOSP, 85–100, 2013.
9. Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu. DiagDroid: Android performance diagnosis via anatomizing asynchronous executions. In Proc. FSE, 410–421, 2016.
10. StrictMode. `http://developer.android.com/reference/android/os/StrictMode.html`.
11. M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In Proc. of MobiSys, 137–150, 2015.
12. K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In Proc. of MobiSys, 151–165, 2015.

13. D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang. Reducing smartphone application delay through read/write isolation. In Proc. of Mobisys, 287–300, 2015.
14. K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. IntentFuzzer: detecting capability leaks of android applications. In Proc. of ASIACCS, 531–536, 2014.
15. T. Terada. Attacking Android browsers via intent scheme URLs. `http://www.mbsd.jp/Whitepaper/IntentScheme.pdf`.
16. P. Schartner, and S. Bürger. Attacking Android's Intent Processing and First Steps towards Protecting it. Technical Report TR-syssec-12-01, Universität Klagenfurt.
17. IntentService. `https://developer.android.com/reference/android/app/IntentService.html`.
18. H. Huang, S. Zhu, K. Chen, and P. Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In Proc. of CCS, 1236–1247, 2015.
19. K. Wang, Y. Zhang, and P. Liu. Call Me Back!: Attacks on System Server and System Apps in Android through Synchronous Callback. In Proc. of CCS, 92–103, 2016.
20. A. Armando, A. Merlo, M. Migliardi, and L. Verderame. Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In Proc. of IFIP SEC, 13-24, 2012.
21. M. EIAN, AND S. MJOLSNES. A formal analysis of IEEE 802.11w deadlock vulnerabilities. In Proc. of INFOCOM, 2012.
22. U. Fiore, F. Palmieri, A. Castiglione, V. Loia, and A. De Santis. Multimedia-based battery drain attacks for android devices. In Proc. of CCNC, 145–150, 2014.
23. A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. Hypnos: understanding and treating sleep conflicts in smartphones. In Proceedings of EuroSys, 253–266, 2013.
24. A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff. What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In Proc. of MobiSys, 267–280, 2012.
25. A. Jindal, A. Pathak, Y. C. Hu, and S. Midkiff. On death, taxes, and sleep disorder bugs in smartphones. In Proc. of HotPower, 1–5, 2013.
26. M. Bauer, M. Coatsworth, and J. Moeller. NANSA: a no-attribution, nosleep battery exhaustion attack for portable computing devices, 2015.
27. F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan. Energy optimization in Android applications through wakelock placement. In Proc. of DATE, 1–4, 2014.
28. C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in android applications. In Proc. of ASE, 389-398, 2013.
29. Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In Proc. of ICSE, 1013-1024, 2014.
30. M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk. How developers detect and fix performance bottlenecks in android apps. In Proc. of ICSME, 352–361, 2015.
31. G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In Proc. of PLDI, 174–186, 2012.
32. L. Zhang, M. S. Gordon, R. P. Dick, Z. Mao, P. A. Dinda, and L. Yang. ADEL: an automated detector of energy leaks for smartphone applications. In Proc. of CODES+ISSS, 363–372, 2012.
33. Y. Liu, C. Xu, and S. C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In Proc. of ICSE, 1013–1024, 2014.
34. DroidBox. `https://github.com/pjlantz/droidbox`.
35. Performance Profiling Tools. `https://developer.android.com/studio/profile/index.html`.