# IVDroid: Static Detection for Input Validation Vulnerability in Android Inter-component Communication*

Zhejun Fang, Qixu Liu, Yuqing Zhang, Kai Wang, and Zhiqiang Wang

National Computer Network Intrusion Protection Center,
University of Chinese Academy of Sciences, Beijing, China
{fangzj,wangk,wangzq}@nipc.org.cn,
{liuqixu,zhangyq}@ucas.ac.cn

**Abstract.** Input validation vulnerability in Android inter-component communication is a kind of severe vulnerabilities in Android apps. Malicious attacks can exploit the vulnerability to bypass Android security mechanism and compromise the integrity, confidentiality and availability of Android devices. However, so far there is not a sound approach at source code level designed for app developers to detect such vulnerabilities. In this paper we propose a novel approach aiming at detecting input validation flaws in Android apps and implement a prototype named IVDroid, which provides practical static analysis of Java source code. IVDroid leverages backward program slicing to abstract application logic from Java source code. On slice level, IVDroid detects flaws of known pattern by security rule matching and detects flaws of unknown pattern by duplicate validation behavior mining. Then IVDroid semi-automatically confirms the suspicious rule violations and report the confirmed ones as vulnerabilities. We evaluate IVDroid on 3 versions of Android spanning from version 2.2 to 4.4.2 and it detects 37 vulnerabilities including confused deputy and denial of service attack. Our results prove that IVDroid can provide a practical defence solution for app developers.

**Keywords:** Input Validation Vulnerability, Static Analysis, Program Slicing, Vulnerability Detection, Android Security

## 1 Introduction

Android security mechanism based on permission and sandbox has improved app security effectively. However, Android inter-component communication(ICC) mechanism brings in some new threat. In some cases if app developers didn't validate the input in Android inter-component communication, malicious Intent(the carrier of ICC data) would be injected and perform security-sensitive behaviors. It is so-called input validation vulnerability [1]. In this paper we talk about

* This research is supported in part by National Information Security Special Projects of National Development and Reform Commission of China under grant (2012)1424, National Natural Science Foundation of China under grants 61272481 and 61303239.


J. Lopez and Y. Wu (Eds.): ISPEC 2015, LNCS 9065, pp. 378–392, 2015.
DOI: 10.1007/978-3-319-17533-1_26

the detection of such vulnerability in the context of Android inter-component communication mechanism, which can lead to various attacks such as confused deputy, deny of service, etc. The former includes capability leaks [2], permission re-delegation [3], content leaks and pollution [4], component hijacking [5][6], etc. The latter includes null point dereference [7], array index exception, illegal state exception, etc.

Prior work primarily focuses on automatic detection of confused deputy attack. Most approaches [2][4][5][6] predefine a certain kind of vulnerability pattern based on expert knowledge and identify confused deputy attack through pattern matching on the reachable execution path. Their approaches are all designed in the perspective of online market and detect vulnerabilities in thousands of executable apps (in form of .APK files). However, there should be a tool designed for app developers to detect confused deputy attack at source code level for two reasons: (1)Analyzing the source code prior to compilation provides a scalable method of security code review[8]. Besides, evolving techniques of anti-tamper and anti-decompiler greatly increase the difficulty of bytecode analysis. (2) App developers are the first battle line defending against vulnerabilities but have little security training. Unfortunately, there is no free and sound tool designed for app developers to secure their development lifecycle.

Detection of DoS attack and other input validation vulnerabilities is not given enough attention to. That's because app crashes are supposed as trivial flaws, and these vulnerabilities are application-specific and hard to be extracted into general vulnerability pattern. To better detect these vulnerabilities relevant to application logic, some researchers extract security policy from source code[11][12] and binary code[13], and check whether the implementation is inconsistent with the stated policy. However, there is so far no sound work focusing on these vulnerabilities in Android inter-component communication.

In this paper we propose a novel approach aiming at detecting input validation vulnerability in Android ICC mechanism and implement a plugin named IVDroid, which provides practical static analysis of Java source code in Eclipse. We employ backward program slicing on the control flow graph (CFG) to precisely capture application logic at slice level. Then we leverage predefined security rules to detect input validation vulnerability in known patterns. Besides, we extract the repeated validation behaviors as implicit and undocumented security rules to detect flaws in unknown pattern. Finally, we infer the inputs of suspicious flaws and confirm them semi-automatically on a running virtual machine.

The contributions of this paper are as follows:

- We propose a detecting technique to defeat input validation vulnerability in Android ICC mechanism, which could be used by app developers to avoid serious threats before app submission.
- We develop a practical plugin of Eclipse called IVDroid in 37 thousand lines of Java code, which is precise for known vulnerabilities and flexible to be expanded for new ones.
- We evaluate IVDroid on original apps in Android 2.2 ,4.0.3 and 4.4.2 and have detected 37 input validation vulnerabilities (16 confused deputy

vulnerabilities and 21 denial of service errors), among which 23 vulnerabilities are undisclosed. The vulnerability report to Android Open Source Project(AOSP) is in progress.

The rest of this paper is structured as below: Section 2 presents an illustrative example and the goal of this paper. In section 3 we give an overview and detail of our approach; Section 4 evaluates the performance of IVDroid together with case studies of discovered vulnerabilities; Section 5 discusses the limitation of IVDroid; Related work is presented in Section 6. Section 7 is a brief conclusion.

## 2 Motivating Example and Problem Statement

### 2.1 Running Example of Input Validation Vulnerability

Here is an example of capability leak vulnerabilities( a typical and severe kind of input validation vulnerabilities) existing in *Settings* app [10], an original Android app providing a GUI to configure system settings and user preferences. The Class *com.android.settings.ChooseLockGeneric* lacks necessary security check and in consequence some running malicious app could exploit the flaw to remove existing device lock and unlock the device. The affected platforms range from Android 4.0 to 4.3.

In Android security model, the device has to ask the user for confirmation of the previous lock for modification or removal in password settings. However, the activity ChooseLockGeneric, which has the capability of remove device lock without any confirmation, is exported unexpectedly without any protection. Fig 1 is a code snippet of ChooseLockGeneric.java in Android 4.0. Line 93, 178, 359, 360 are an execution path leading to the function in charge of clearing screen lock. The malicious Intent has to be crafted well to satisfy the path branch conditions(line 85, 167, 309, 318, 338, 355, 358). Ironically, app developer wrote the annotation(line 308) about password confirmation and do the check(line 309) which could be easily bypassed. The exploit is available in [10]. The vulnerability is fixed in 4.4 by setting ChooseLockGeneric activity *unexported* in manifest.xml of Settings app.

In summary, two factors must be met for the formation of input validation vulnerability as above. Firstly, there exists a reachable path from the entry of component to the call site of sensitive API. Secondly, the input validations are not designed properly. The two factors can lead to many security-sensitive behaviors. As [1] says, "this is a recipe for disaster".

### 2.2 Problem Statement

If we want to detect an input validation vulnerability, we should find the reachable paths and check the validation behaviors carefully. In this paper, we extract the reachable path on the control flow graph(CFG) and take all the branch conditions on the path as validation behaviors. The vetting of validation behaviors is a challenge we need to solve.

```
69  public void onCreate(Bundle savedInstanceState) {
77      final boolean confirmCredentials = getActivity().getIntent()
                .getBooleanExtra(CONFIRM _CREDENTIALS, true);
71      mPasswordConfirmed = ! confirmCredentials;
85      if (mPasswordConfirmed) {
93          updatePreferencesOrFinish();
...
164 private void updatePreferencesOrFinish() {
165     Intent intent = getActivity().getIntent();
166     int quality = intent.getIntExtra(
                LockPatternUtils.PASSWORD_TYPE_KEY, -1);
167     if (quality == -1) {...}
177     else
178         updateUnlockM ethodAndFinish(quality, false);

307 void updateUnlockMethodAndFinish(int quality, boolean disabled) {
308     // Sanity check. We should never get here without
            confirming user's existing password.
309     if (!mPasswordConfirmed)
310         throw new IllegalStateException();
316     quality = upgradeQuality(quality, null);
318     if (quality >= PASSWORD_QUALITY_NUMERIC) {...}
338     else if (quality == PASSWORD_QUALITY_SOMETHING) {...}
355     else if (quality == PASSWORD_QUALITY_BIOMETRIC_WEAK) {...}
358     else if (quality == PASSWORD_QUALITY_UNSPECIFIED) {
359         mChooseLockSettingsHelper.utils().clearLock(false);
360         mChooseLockSettingsHelper.utils().setLockScreenDisabled(disabled);
```

**Fig. 1.** Code snippet of class "ChooseLockGeneric" of Settings app

We anticipate our proposed technique to be leveraged as a vetting plugin of Android IDE and achieve the following goals:

- **Security Development.** During the development of a new app, developers can run our plugin to detect input validation vulnerabilities, even when the app may not be runnable at that check point.
- **Minimal demand on users.** Even if developers do not have enough security knowledge, we hope our plugin would be helpful in most cases and be able to guide them about how to confirm and patch.
- **Extensible for future flaws.** The tool should be extensible for unknow vulnerabilities.

## 3 Approach Overview

Fig 2 depicts the workflow of our proposed technique. It works in the following steps:

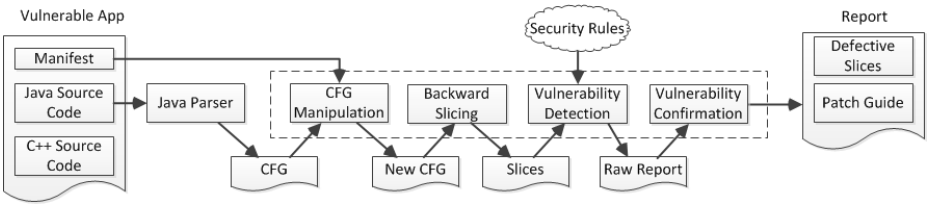


**Fig. 2.** The architecture of IVDroid

**(1) Java Parser.** The source code of an Android app consists of Java, C++ (native code) and xml (manifest.xml etc.) code, and We focus on Java source code and manifest.xml file. We leverage JavaParser, an open-source parser written by

*jgsser*, to parse Java code and generate the Abstract Syntax Tree(AST), Control Flow Graph(CFG) and System Dependence Graph(SDG) automatically.

**(2) CFG Manipulation.** The main task of this step is to modify the CFG according to Android features and remove unnecessary control flow for the next slicing step.

**(3) Backward Slicing.** At this step, we perform *flow-sensitive context-sensitive inter-procedural backward slicing* [14] to extract *transaction slices* and *constraint slices*. *Transaction slice* contains a statement of sensitive system API invocation as slicing criterion and its minimal set of data-dependent statements. *Constraint slice* is similar to transaction slice but its slicing criterion is an "if" statement.

**(4) Vulnerability Detection.** At slice level, the problem of detecting input validation vulnerability is transformed into how to detect pattern violation. The core of this step is to detect known flaws by security rule matching and detect unknown flaws by frequent validation behavior mining.

**(5) Vulnerability Confirmation.** We implement a semi-automatic vulnerability confirming module to validate the raw report. We can infer the inputs of simple suspicious flaws and confirm them on a running Android virtual machine dynamically. The complex suspicious flaws will be left for manual validation. Then the final report is generated with defective slices and patch guides.

### 3.1 CFG Manipulation

We gain basic CFGs of an app by leveraging JavaParser, and modify them according to Android features and remove unnecessary control flow. The new modified CFG only has basic blocks and conditional jumps, and does not have any loops, or throw statements.

**Special Considerations for Android.** Some access control policy are saved in the manifest file in the form of XML attribute "*exported*" or "*permission*". To collect them in the program slicing, we transform these access control policy to the form of "if" statement and add them to the entry point of corresponding components.

**Remove Unnecessary or Uncertain Control Flow.** Java Assert statements would test assumption and throws an exception if the test fails. Android *checkPermission(perm)* API family is similar to them and checks if the permission *perm* has been granted to the calling application and throws a security exception when the check fails. To avoid such unexpected control flow transfer, IVDroid extracts the condition expressions of them and enforce them as implicit constraints for the following statements. For the similar reason, to "loop" statements we take the inverse expression of its condition as constraints for the following statements.

### 3.2 Backward Slicing

With the modified CFG, we now leverage backward program slicing to extract transaction and constraint slices. The basic algorithm is fairly standard and

similar to other work such as [14]. In comparison, our slicing works in the context of Android platform and thus needs to be somewhere different.

**Basic Algorithm.** The algorithm begins from the last statement of a function (often is "return" statement) and searches all invocations of security-sensitive system API as slicing criterions backward on the control flow. From each slicing criterion, we compute all data-dependent statements via backward slicing until we get to the start point of the input. Then we get a transaction slice, which consists of a slicing criterion and all its dependent statements. In the same way, we extract constraint slices for each transaction slice by starting slicing from each "if" statement of the transaction.

**Special Considerations for Android Apps.** To leverage the above algorithm in Android platform, we have several special considerations for Android environment.

*System API.* We choose the security-sensitive system APIs as slice criterions especially when the API accesses Internet or mobile communication, or manipulates database or file system. The work of Pscout[16] about API calls mappings helps us construct the API list. In addition, the list also includes inter-component communication API such as *startActivity(), sendBroadcast()*.

*Handle.* A Handler(android.os.Handler) allows developers to send and process Message and Runnable objects associated with a thread's MessageQueue. It is an asynchronous message handling mechanism. To deal with such implicit method invocation, in the CFG we add a link between *Handle.sendMessage()* and *Handle.handleMessage()*.

**Slice Example.** Fig. 3 is a transaction and constraint slice example of Fig. 1. The last statement of transaction slice is the slicing criterion *clearLock(false)*, which is a security-sensitive function and maps the permission WRITE_SETTINGS. Also, we present the constraint slices of this transaction. It consists of seven security checks, one from a Throw statement (line 10) and six from "if" statements (line 8,9,11-14). These constraints check some property of variable *quality* and *mPasswordConfirmed*, whose values are both assigned from the input. Due to space limitations, we do not detail the constraint slices.

### 3.3 Vulnerability Detection

With transaction and constraint slices in hand, we can perform vulnerability detection at the slice level. We firstly leverage the predefined security rules to detect known-pattern vulnerabilities. Then we extract the duplicated constraints as implicit security rules from the slices and verify them to detect vulnerabilities with unknown pattern. Finally we generate the suspicious violations as raw report, which will be confirmed in the next step.

**(1) Detecting Vulnerabilities of Known Pattern**
In this section we validate each transaction slice according to predefined security rules. Prior work has undisclosed a lot of vulnerability patterns for input validation flaws, especially for confused deputy. We leverage Pscout [16] to write rules

```
1   Transaction Slice:
2   Activity newvar1 = this.getActivity();
3   mChooseLockSettingsHelper = new ChooseLockSettingsHelper();
4   LockPatternUtils newvar2 = mChooseLockSettingsHelper.utils();
5   newvar2.clearLock(false);
6
7   Constraint Slices:
8   if (mPasswordConfirmed)
9   if (!(quality == -1))
10  if (!(!mPasswordConfirmed))
11  if (!(quality >= PASSWORD_QUALITY_NUMERIC))
12  if (!(quality == PASSWORD_QUALITY_SOMETHING))
13  if(!(quality == PASSWORD_QUALITY_BIOMETRIC_WEAK))
14  if (quality == PASSWORD_QUALITY_UNSPECIFIED)
```

**Fig. 3.** Transaction and constraint slices from class "ChooseLockGeneric" of Settings app

```
    ics_allmappings.txt
21273  Permission:android.permission.WRITE_SETTINGS
21274  1098 Callers:
21561  <com.android.internal.widget.LockPatternUtils:
                                    void clearLock(boolean)> (2)
    Security Rule
1   public void rule(Context context)
2     ChooseLockSettingsHelper mChooseLockSettingsHelper =
                    new ChooseLockSettingsHelper(this.getActivity());
3     if(Outer.checkPermission("android.permission.WRITE_SETTINGS")
                    || Outer.isExported(false))
4       mChooseLockSettingsHelper.utils().clearLock(false);
```

**Fig. 4.** An example of security rules

for detection and the final rule list contains 33,624 items. To make reading and writing rules as readily as possible, the rules are designed to be written in Java language. They are just similar to a transaction slice of Java code invocating critical system API but with necessary permission validation. Fig. 4 is an example to illustrate how we write a security rule to detect the flaw in Section 2.1. The first three lines are from the result of Pscout(ics_allmappings.txt). First, We analyse it and map the permission *WRITE_SETTINGS* with the API *clearLock()*. Second, we extract all the data-dependent statements from the source code of *com.android. settings.ChooseLockGeneric* manually to fill minimal execution context of the rule. After these steps a rule is constructed completely, which checks whether the permission WRITE_SETTINGS is validated and whether the component is exported.

The rule matching procedure is as follows: IVDroid first leverage JavaParser to parse the security rules and compare them with all extracted transaction slices. If a transaction slice and a security rule have the same slicing criterion, we demand that slice should perform the same permission checks. If not, the violation would be reported as suspicious flaw. Given slicing criterions of two transactions as $o_1.fun_1(p_1, p_2, ..., p_n)$ and $o_2.fun_2(q_1, q_2, ..., q_n)$, they are equal only if the classes of $o_1$ and $o_2$ are the same or inherit from the same parent class, function names of $fun_1$ and $fun_2$ are equal and parameter types are equal. Suppose $O_1$ is $o_1$'s class, $O_2$ is $o_2$'s class, the symbol "$<$" stands for the relationship of inheritance, $P_x$ is $p_x$'s class and $Q_x$ is $q_x$'s class, then $o_1.fun_1(p_1, p_2, ..., p_n) = o_2.fun_2(q_1, q_2, ..., q_n)$ if and only if $(O_1 = O_2 \vee \exists Class\ O, (O_1 < O) \wedge (O_2 < O)) \wedge fun_1.name = fun_2.name \wedge (\forall x \in [1, n], (P_x = Q_x \vee \exists Class\ R, (P_x < R) \wedge (Q_x < R)))$.

**(2) Detecting Vulnerabilities of Unknown Pattern**

Input validation vulnerability, particularly DoS flaw, is very relevant to application logic and it is hard to extract application-specific validation behaviors in general vulnerability patterns. To enhance the capability of IVDroid, we propose

an algorithm to extract duplicated validation behaviors as implicit security rules by frequent pattern mining and detect violations at slice level.

Firstly, IVDroid divides the transactions into different categories by equality of slicing criterion. Then IVDroid traverses all constraints in the category, and take the constraints appearing frequently as implicit security rules. Second, we verify all extracted rules and record suspicious violations. The details are described as below.

**Extracting Implicit Security Rules.** In this step, we classify transactions according to the equality of their slicing criterions. That's because the slicing criterion is the core statement of a transaction and stands for its functionality. Specially, inter-component communication APIs such as *startActivity(), startService() and sendBroadcast()* are treated as equal slicing criterions because their functionalities are just the same.

Secondly, we infer the security specifications for each transaction category. We traverse all constraints from each transaction in a category and only treat the constraints appearing more than once in different transactions as implicit security rules we need. The equality of two constraints is similar to the equality of two transactions.

**Verification of Implicit Rules.** After obtaining these implicit security rules, we apply them as mandatory property to the original transaction category and record the suspicious violation. One challenge is how to judge the relevance of specific transaction and implicit security rules. The judging criterion is: for any extracted security rule $SR_x$ from a transaction category, only when the dependent variable set of a transaction contains the dependent variable set of $SR_x$, the transaction should contain a constraint which is equal to $SR_x$. When we infer the relationship of dependent variable sets, we actually use variables' type instead of variables themselves and ignore primitive types. For example, if the dependent variable set of a transaction is *Intent, Message, SmsMessage, Context* and the dependent variable set of security rule is *Context, Intent, String, int*, we can tell that the security rule is necessary for and relevant to the transaction.

### 3.4 Vulnerability Confirmation

In this part, we confirm the above suspicious violations semi-automatically. Through data flow dependence analysis, we can collect all the constraints the input should satisfy. Then the work divides into two parts: (1)if the constraints only contain boolean expressions of string or integer, IVDroid generates exploit code and validates them automatically. (2)if the constraints are too complex to resolve, IVDroid leaves them for manual validation.

As Fig.5 shows, we implement a very simple resolver to infer the value of input automatically, such as *Extra* property. Then IVDroid generates the exploit code and send it to Android virtual machine (VM). The VM has bee deployed a modified system, in which *ActivityManagerService* is reprogrammed to monitor API calls[9]. IVDroid also leverages *logcat* to monitor app crash. If either of the above two situations is monitored, the vulnerability is recorded as confirmed

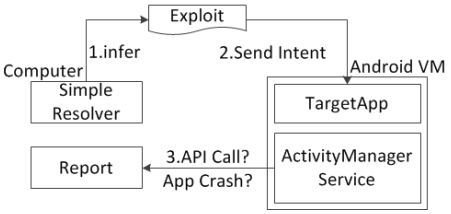


**Fig. 5.** Automatically Validate Exploitable Vulnerabilities

```
1  CONFIRMED
2  From:com.android.settings. ChooseLockGeneric   updateUnlockMethodAndFinish
3  Activity newvar1 = this.getActivity();
4  mChooseLockSettingsHelper = new ChooseLockSettingsHelper();
5  LockPatternUtils newvar2 = mChooseLockSettingsHelper.utils();
6  newvar2.clearLock(false);
7  Lost Validation :
8  if(Outer.checkPermission("android.permission.WRITE_SETTINGS")
9  if(Outer.isExported(false))
10 Patch Guide:
11 Add permission restriction "android.permission.WRITE_SETTINGS" in the manifest.xml
12 Set "exported = false" in the manifest.xml
```

**Fig. 6.** An example of report of IVDroid

one, or it would be recorded as false alarm. The vulnerability would be recorded as unconfirmed one if the input can not be resolved, or there is no executable app(.apk).

After all these steps, a final report is generated, in which suspicious vulnerabilities are listed with the defective slices and possible patch guide.

## 4 Evaluation

We evaluate the performance of IVDroid on different Android distributions including 2.2, 4.0 and 4.4. We choose original apps in the folder *packages/apps* as test cases because they are available in almost every version of Android ROMs. In total, there are 27 apps in 2.2_r1.1, 34 in 4.0.3_r1 and 45 in 4.4.2_r1.

### 4.1 Results Overview

IVDroid running on each distribution produces a lot of suspicious input validation vulnerability reports. We then manually verify the reports by checking the corresponding source code.[1]. The results are shown in Table 1. Column "Apps" indicates the name of app which contains at least one input validation vulnerability. The dot mark means we do not find any vulnerability of that category in target app. The cross mark means the app doesn't exist in that version( NFC is available since Android 2.3). Column "Vulnerable Component" indicates the name of component which contains the vulnerabilities.

In total, IVDroid finds 37 input validation vulnerabilities, 23 of which are undisclosed. Among them, there are 16 confused deputy attacks(2 are undisclosed) and 21 null pointer dereferences(all are undisclosed). The experiment results provide encouraging evidence proving the effectiveness of IVDroid.

The results also tell that the code quality of Android is improved while the number of confused deputy is decreasing. However, null dereference vulnerabilities are still not taken seriously and are not patched in all systems. In next section we'll give out an example to demonstrate that null dereference can lead to severe DoS (Denial of Service) attack. There is another interesting fact that some vulnerabilities only exist in the version 4.x not 2.2.

[1] The vulnerability details are available on `http://ivdroid.sinaapp.com`

**Table 1.** Detected Input Validation Vulnerabilities (C: confused deputy; N: null dereference)

| ID | Apps | 2.2_r1.1 | | 4.0.3_r1 | | 4.4.2_r1 | | Vulnerable Component |
|---|---|---|---|---|---|---|---|---|
| | | C | N | C | N | C | N | |
| 1 | com.android.mms | 1 | 1 | 1 | 1 | · | 1 | .transaction.SmsReceiverService |
| 2 | com.android.bluetooth | · | 1 | · | 1 | · | 1 | .pbap.BluetoothPbapService |
| 3 | com.android.deskclock | 1 | 1 | · | 1 | · | 1 | AlarmInitReceiver |
| 4 | com.android.music | 1 | · | · | · | · | · | MediaPlaybackService |
| 5 | com.android.phone | 2 | 1 | 2 | 1 | 2 | 1 | PhoneAppBroadcastReceiver |
| 6 | com.android.settings | 5 | 1 | 1 | · | · | · | .widget.SettingsAppWidgetProvider&ChooseLockGeneric |
| 7 | com.android.stk | · | 2 | · | 2 | · | 2 | StkCmdReceiver&BootCompletedReceiver |
| 8 | com.android.nfc | × | × | · | 1 | · | 1 | .handover.HandoverManager |
| | vulnerabilities in total | 10 | 7 | 4 | 7 | 2 | 7 | |

### 4.2 Detail analysis

**Vulnerabilities Detected by Security Rule Matching.** The security rule database is effective to detect confused deputy attacks. In detail, confused deputy vulnerabilities in App 3, 4, 5, 6(.widget.SettingsAppWidgetProvider) are simple. Intent only containing "action" field and simple "extra" field would trigger the vulnerability. So IVDroid could confirmed these flaws automatically. In comparison, confused deputy vulnerabilities in App 1 and 6(ChooseLockGeneric) are much more complex. The content of Intent should be constructed manually to reach the sensitive API and manipulate privacy information.

In Fig.3, we depict the DeskLock capability leak vulnerability in App 7(ChooseLockGeneric). Fig.4 is the security rule to detect that vulnerability. Fig. 6 is a snippet of the final report, containing defective slice and patch guide. As the report says, capability leak vulnerability can be mitigated by adding access restriction in manifest.xml file. In fact, many capability leaks are simply patched in higher version by setting "*exported*" property *false*.

**Vulnerabilities Detected by Duplicate Validation Behavior Mining.** Vulnerabilities of unknown patterns can be detected by duplicate validation behavior mining. This method is effective to detect deny of service attacks, which are particularly dangerous when an adversary leverage them to stop critical service [7], i.e. anti-virus and security enhancement software.

In detail, DoS attacks in App 1, 2, 3, 5, 6, 7 are null dereference vulnerabilities missing necessary vetting for the input. In App 8 there is an array bound error, which we could only validate it manually. The rest of vulnerabilities are simple enough to be confirmed automatically. In our observation, null pointer dereference appears frequently but there are not many null point dereference flaws reported because of two reasons: one is that null dereferences in activity components have minimal impact [7]; the other is that some potential null dereferences can't be triggered actually because the vulnerable components may be not exported or the pointer is checked somewhere else.

In Fig. 7, we depict a null pointer dereference in app 7 *com.android.stk*, which are undisclosed before. App com.android.stk is a STK (short for SIM Application

```
1   Transaction 1 From: stk.BootCompletedReceiver  onReceive
2   Bundle args = new Bundle();
3   args.putInt(StkAppService.OPCODE, StkAppService.OP_BOOT_COMPLETED);
4   context.startService(new Intent(context, StkAppService.class).putExtras(args));
5   Constraint Slices:
6   if(action.equals(Intent.ACTION_BOOT_COMPLETED))

7   Transaction 2 From: phone.InCallScreen onNewIntent
8   startActivity(intent.setClassName(this, EmergencyCallHandler.class.getName()));
9   Constraint Slices:
10  if(intent == null || intent.getAction() == null)
11  if(!(action.equals(ACTION_SHOW_ACTIVATION))
12  if(!action.equals(Intent.ACTION_ANSWER))
13  if(action.equals(Intent.ACTION_CALL)||action.equals(Intent.ACTION_CALL_EMERGENCY))
14  if(okToCallStatus != InCallInitStatus.SUCCESS)
15  if(isEmergencyNumber && (okToCallStatus == InCallInitStatus.POWER_OFF))
```

**Fig. 7.** Null pointer dereference vulnerability of STK app

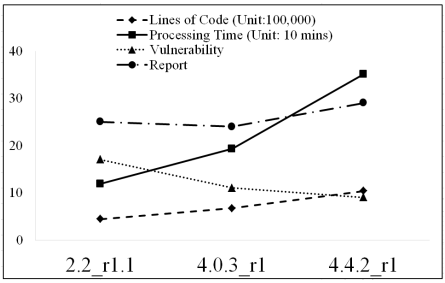


**Fig. 8.** Performance of IVDroid

Toolkit) app in charge of value-added services based on GSM communication. If STK app crashes, the mobile communication of the phone will be cut off.

The flaw exists in the *onReceive()* function of *BootCompletedReceiver* component, as shown in Fig. 7. Transaction 1 is extracted from *onReceive()* and neither of its constraint slices check whether the Intent's "*Action*" property is null. If the incoming Intent's "*Action*" property is set null on purpose, the application will crash when it is dereferenced, which, in consequence, leads to the crash of Phone app. A persistent attack would prevent mobile communication totally, neither in nor out. IVDroid gets the needed security rule from transaction 2, which is extracted in another application(com.android.phone) and supplies the validation behavior we need. Transaction 1 & 2 are divided into the same category because *startService()* and *startActivity()* are treat equally. This kind of situation is not rare especially when the programmer has weak security concepts.

Null pointer dereference can be mitigated by adding content checking statements in source code. For example, the content of Intent needs null check, and the index of Array needs bound check.

### 4.3 Performance Measurement

In this section we evaluate the performance of IVDroid. The results are shown in Table 2. Column "Apps" indicates the number of apps and providers in certain Android version. Column "LoC" indicates the sum-up of lines of code of apps and providers in certain Android version. Column "Time" indicates the running time it takes to process a version of Android distribution. Column "Report" indicates the number of reported suspicious violations. Column "Vul" indicates the number of vulnerabilities IVDroid has detected. Column "Confirmed Vul" indicates the number of vulnerabilities IVDroid has confirmed automatically. Column "False Alarm" indicates the number of false alarm IVDroid has confirmed automatically. Column "Unconfirmed Alarm" indicates the number of unconfirmed alarms, in which the first number indicates the number of vulnerabilities we have confirmed manually.

**Table 2.** Performance of IVDroid

| Version | Apps | LoC | Time | Report | Vul | Confirmed Vul | False Alarm | Unconfirmed Alarm |
|---|---|---|---|---|---|---|---|---|
| 2.2_r1.1 | 27 | 445,836 | 119min | 25 | 17 | 16 | 7 | 1/2 |
| 4.0.3_r1 | 34 | 674,156 | 193min | 24 | 11 | 9 | 11 | 2/4 |
| 4.4.2_r1 | 45 | 1,040,688 | 351min | 29 | 9 | 9 | 16 | 0/4 |

We measure the processing time by running IVDroid on an Intel Core i7 2.93GHz machine with 8GB of memory and Windows 7 SP1 OS. We believe the average processing time (about 6 minutes per app) is reasonable for offline detection. From Fig 8 we can tell that the processing time is increasing faster than the speed of lines of code because the implicit validation mining is applied in all transaction categories. The larger the category is, the more time it takes for IVDroid to collect constraint set and calculate the relevance of transaction and constraint.

We can also tell that the vulnerability confirmation module of IVDroid is effective to decrease false positive rate. Most of the suspicious vulnerabilities IVDroid reported are simple enough to be confirmed automatically. Specially, DoS vulnerabilities' input is easy to generate and app crashes information can always be monitored by logcat. Unconfirmed vulnerabilities are complex ones detailed in 4.2. The analysis of complex vulnerabilities can also benefit from the vulnerability confirmation module.

We do not measure false negatives because we don't have enough ground truth in the target apps. Here are two reasons which may lead to false negatives. First, dynamic vulnerability confirmation would raise false negatives if some system behavior is not monitored or the resolved input is wrong. Second, since the implicit security specifications are inferred from the extracted transactions, we can't get enough specifications for flaw detection if the programmer didn't make any security-sensitive check.

## 5 Discussion

IVDroid has so far uncovered a lot of input validation vulnerabilities in three Android distributions. It is important to give a further discussion about its own advantages and disadvantages.

**Advantages of IVDroid.** IVDroid has meet the goals we proposed in Section 2.2. IVDroid is a practical framework of input validation vulnerability detection for app developers. With predefined security rules and semi-automatic vulnerability confirmation module, we have lowered knowledge demand of IVDroid impressively. In addition, we leverage duplicate validation behavior mining to detect vulnerabilities of unknow pattern.

**Limitation of IVDroid.** IVDroid is only our first step for input validation vulnerability detection. Although it has identified several serious vulnerabilities in current Android version, it's still neither sound nor complete. IVDroid can not handle the complex situation when several system API call co-work together to

accomplish a transaction. The vulnerability confirmation module is very simple so far and it can be enhanced by leveraging current constraint resolve technique in the future. Meanwhile, IVDroid can not handle the situation when apps use Java reflection techniques.

**Detection for Other Vulnerabilities.** Logic vulnerability is another common kind of vulnerabilities in Android apps, which misleads the legitimate processing flow of an application into unexpected negative consequence. IVDroid can detect some certain kinds of logic vulnerabilities. Application logic can be extracted by backward slicing and the validation behavior mining can help us understand undocumented application logic.

## 6 Related Work

**Automatic Detection of Vulnerabilities in Android Inter-component Communication.** When researchers turn to Android platform, they focus on a subset of input validation flaws, such as permission re-delegation [3], capability leak [2], and denial of service [7]. Felt etc. [3] firstly discover the permission re-delegation problem in the Android IPC mechanism and propose defence mechanism. Stowaway [17] detects overprivilege in compiled Android applications by comparing the required and requested permissions. To our best knowledge, there is no approach addressing static analysis of input validation vulnerabilities on source code level comprehensively in Android apps before.

Woodpecker [2] employs inter-procedural data flow analysis to systematically expose possible capability leaks. Our approach tries to detect capability leaks in a different way and focus on the execution path with necessary checks. Another difference is that IVDroid aims at the source code and Woodpecker operates on binary code. This leads to different scenarios. IVDroid tries to provide defence for app developers but Woodpecker are designed for automatic vetting in online app market. In comparison, IVDroid covers all the vulnerabilities Woodpecker detects and IVDroid can detect more types of input validation vulnerabilities than Woodpecker. Additionally, IVDroid can extract application-specific rules and Woodpecker does not have that capability.

Appsealer [6] focuses on the component hijacking attacks in Android applications and proposes a patch automatic generation technique. Appsealer injects minimal required code in vulnerable apps and provides a runtime defence for component hijacking attacks. Both IVDroid and Appsealer leverage static backward program slicing to extract application logic. The main difference between IVDroid and Appsealer is, however, that IVDroid can extract new vulnerability patterns from the code automatically. Similarly to Woodpecker, another difference is that Appsealer takes binary code as input.

**Automatic Infering and Understanding of Security Specifications of Android Applications**. SCanDroid [11] and Kirin [13] validate manifest files containing the access control policy of an application. Mustafa et al. [12]

extracts the implemented access control policy existing in the form of checkPermission APIs from Android system services with the help of program slicing. They admit that their approach would miss some security checks. Compared with their approach, IVDroid focus on Android ICC mechanism and gets more but smaller slices. We argue that IVDroid extracts more kinds of constraints and more fine-grained application logic, and cover all the policy Mustafa gains. Some vulnerabilities detected by IVDroid cannot be identified by Mustafa, as Mustafa doesn't analyse the specifications in manifest and "if" statements. All of the approaches before can not extract the whole implemented security specifications of Android apps and IVDroid is the first approach that extracts all security policies including manifest, checkPermission() APIs and "if" statements.

To describe access control policies formally, Kirin [13], Mustafa [12] and Sohr [18] use auxiliary language such as Java Modeling Language (JML), Kirin Security Language (KSL) and Object Constraint Language (OCL). It needs extra effort to understand the grammar of these languages and create new policy. IVDroid overcomes that disadvantage by directly using Java language to describe security rules.

**Comprehensive Study on Android Vulnerabilities.** Enck etc. [7] propose a study of Android application security based on static analysis of 21 million lines of recovered codes. Their approach uncovers different kinds of pervasive vulnerabilities and bugs, such as misuse of personal information and null pointer dereference. They think that many application-specific errors are often ignored, which inspired us a lot.

**Dynamic Testing on Android.** Yang etc.[9] detect capability leaks of Android apps by dynamic fuzzing the Intent. We are inspired by their work and implement our own dynamic vulnerability confirmation module.

## 7 Conclusions

This paper proposes a static approach to detect input validation vulnerabilities in Android inter-component communication. We employ program slicing to extract application logic. Then we detect vulnerabilities of known patterns through predefined security rules and detect vulnerabilities of unknown patterns through implicit validation mining. The suspicious flaws are validated by dynamic testing to lower the false positives. We implement a prototype plugin named IVDroid and evaluate it on Android 2.2, 4.0.3 and 4.4.2. The results prove that IVDroid has good precision. In the future work we will leverage more accuracy analysis such as symbolic execution and improve the dynamic vulnerability confirmation module.

## References

1. Category:input validation on owasp, `https://www.owasp.org/index.php/Category:Input_Validation`

2. Grace, M., Zhou, Y., Wang, Z., et al.: Systematic detection of capability leaks in stock Android smartphones. In: NDSS (2012)
3. Felt, A.P., Wang, H.J., Moshchuk, A., et al.: Permission Re-Delegation: Attacks and Defenses. USENIX Security Symposium (2011)
4. Zhou, Y., Jiang, X.: Detecting Passive Content Leaks and Pollution in Android Applications. In: NDSS (2013)
5. Lu, L., Li, Z., Wu, Z., et al.: Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 229–240 (2012)
6. Zhang, M., Yin, H.: AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In: Proceedings of the 21th Annual Network and Distributed System Security Symposium, NDSS 2014 (2014)
7. Enck, W., Octeau, D., McDaniel, P., et al.: A Study of Android Application Security. In: USENIX security symposium (2011)
8. SDL Process: Implementation, `http://www.microsoft.com/security/sdl/process/implementation.aspx`
9. Yang, K., Zhuge, J., Wang, Y., et al.: IntentFuzzer: detecting capability leaks of android applications. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 531–536. ACM (2014)
10. CVE-2013-6271: Security Advisory Curesec Research Team, `http://dl.packetstormsecurity.net/1311-advisories/CURE-2013-1011.txt`
11. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: SCanDroid: Automated security certification of Android applications Manuscript, Univ. of Maryland. Citeseer (2009), `http://www.cs.umd.edu/avik/projects/scandroidascaa`
12. Mustafa, T., Sohr, K.: Understanding the Implemented Access Control Policy of Android System Services with Slicing and Extended Static Checking. Technical report, University of Bremen (2012)
13. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 235–245 (2009)
14. Fang, Z., Zhang, Y., Kong, Y., et al.: Static detection of logic vulnerabilities in Java web applications Security and Communication Networks. Security and Communication Networks 7(3), 519–531 (2014)
15. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Security & Privacy 7, 50–57 (2009)
16. Au, K.W.Y., Zhou, Y.F., Huang, Z., et al.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 217–228 (2012)
17. Felt, A.P., Chin, E., Hanna, S., et al.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638 (2011)
18. Berger, B.J., Sohr, K., Koschke, R.: Extracting and Analyzing the Implemented Security Architecture of Business Applications. In: 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 285–294 (2013)