SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

May 2017, Vol. 60 052111:1–052111:16
doi: 10.1007/s11432-015-5422-7

A static technique for detecting input validation vulnerabilities in Android apps

Zhejun FANG[1,2], Qixu LIU[1,4], Yuqing ZHANG[1*], Kai WANG[1], Zhiqiang WANG[3] & Qianru WU[1]

[1]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing* 101408*, China;*
[2]*National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing* 100029*, China;*
[3]*Beijing Electronic Science and Technology Institute, Beijing* 100070*, China;*
[4]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093*, China*

Received April 7, 2016; accepted June 3, 2016; published online September 12, 2016

Abstract Input validation vulnerabilities are common in Android apps, especially in inter-component communications. Malicious attacks can exploit this kind of 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 the source code level for app developers aiming to detect input validation vulnerabilities in Android apps. In this paper, we propose a novel approach for detecting input validation flaws in Android apps and we implement a prototype named EasyIVD, which provides practical static analysis of Java source code. EasyIVD leverages backward program slicing to extract transaction and constraint slices from Java source code. Then EasyIVD validates these slices with predefined security rules to detect vulnerabilities in a known pattern. To detect vulnerabilities in an unknown pattern, EasyIVD extracts implicit security specifications as frequent patterns from the duplicated slices and verifies them. Then EasyIVD semi-automatically confirms the suspicious rule violations and reports the confirmed ones as vulnerabilities. We evaluate EasyIVD on four versions of original Android apps spanning from version 2.2 to 5.0. It detects 58 vulnerabilities including confused deputy attacks and denial of service attacks. Our results prove that EasyIVD can provide a practical defensive solution for app developers.

Keywords input validation, static analysis, program slicing, vulnerability detection, Android security

Citation Fang Z J, Liu Q X, Zhang Y Q, et al. A static technique for detecting input validation vulnerabilities in Android apps. Sci China Inf Sci, 2017, 60(5): 052111, doi: 10.1007/s11432-015-5422-7

1 Introduction

Android's security mechanism is based on permissions and sandbox, which has improved app security effectively. Apps are forced to request individual application permissions before accessing system resources. Apps are isolated in the sandbox and the common outward interfaces of an app are inter-component communication (ICC), Internet and Bluetooth sockets, and external files. However, if app

* Corresponding author (email: zhangyq@ucas.ac.cn)

developers do not validate the input from an external and untrusted source properly, malicious code can be injected and perform security-sensitive behaviours. This is the so-called input validation vulnerability [1], which is the most serious threat to app security and can lead to various attacks such as a confused deputy, denial of service (DoS), etc. A confused deputy attack includes capability leaks [2], permission re-delegation [3], content leaks and pollution [4], component hijacking [5, 6], etc. A DoS attack includes null pointer dereference [7], array index exception, illegal state exception, etc. Because the Android ICC mechanism is very flexible in inter-process communication, an input validation vulnerability often exists in the implementation of an app's ICC module.

A confused deputy attack is very dangerous for app security so prior work primarily focuses on automatic detection of it. Most approaches [1, 4–6] predefine a certain kind of vulnerability pattern based on expert knowledge and detect a confused deputy attack through pattern matching on the reachable execution path. These approaches are all designed from the perspective of an online market and have detected vulnerabilities in thousands of executable apps (in the form of APK files). However, there should be a tool designed for app developers to detect a confused deputy attack at the source code level for three reasons: (1) Static analysis of source code provides a more precise way of detecting vulnerabilities than on bytecode. Besides, the evolving anti-tamper and anti-decompiler techniques greatly increase the difficulty of bytecode analysis. (2) According to Microsoft SDL (Security Development Lifecycle)[1)], analysing the source code prior to compilation provides a scalable method of conducting a security code review. Security policies can be enforced during development even if the app is in inexecutable state. (3) App developers are the front line in defending against attacks but have little security training. It is necessary to prevent vulnerabilities at source. Unfortunately, there is no free and sound tool designed for app developers to secure their apps.

In contrast with the confused deputy attack, DoS attacks and other input validation vulnerabilities are not given enough attention. App developers frequently do not perform enough checks on Intent [7]. There are two reasons for this: one is that many validation behaviours are application-specific and hard to extract to a general vulnerability pattern; the other is that app crashes are not severe flaws and probably neglected by developers. To better detect these vulnerabilities relevant to application logic, some approaches extract a security policy from source code [8, 9] or binary code [10], and check whether the implementation is inconsistent with the stated policy. However, there is no sound way of statically detecting these kinds of input validation vulnerabilities in an Android app.

To solve the above problems, in this paper we propose a novel approach for detecting input validation vulnerabilities in Android apps and implement a prototype named EasyIVD, which provides practical static analysis of Java source code. We employ backward program slicing on the manipulated control flow graph (CFG) to precisely capture application logic at slice level. Then we leverage predefined security rules to detect input validation vulnerabilities in known patterns. In addition, we extract the implicit validation behaviours as undocumented security rules using frequent-pattern mining. 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 detection technique to detect input validation vulnerabilities in Android apps, which could be used by app developers to prevent serious threats before app submission. It detects known-pattern flaws using pattern matching and it detects unknown-pattern flaws using implicit validation mining.

• We develop a practical prototype in 37000 lines of Java code. Our tool firstly performs backward slicing to extract transaction and constraint slices. Then, it leverages predefined security rules to detect the vulnerability patterns we already know. It also extracts and analyses the implicit and undocumented constraints using frequent-pattern mining to detect vulnerabilities of unknown patterns.

• We evaluate EasyIVD on original apps for Android 2.2 ,4.0.3, 4.4.2 and 5.0 and have detected 58 input validation vulnerabilities (22 confused deputy vulnerabilities and 36 DoS errors), among which 44 vulnerabilities have not been found before. We analysed these vulnerabilities in detail and wrote proof

1) SDL Process: Implementation. http://www.microsoft.com/security/sdl/process/implementation.aspx.

```
public int onStartCommand(Intent intent, int flags, int startId) {
125   mResultCode = intent != null ? intent.getIntExtra("result", 0) : 0;
126
127   Message msg = mServiceHandler.obtainMessage();
128   msg.arg1 = startId;
129   msg.obj = intent;
130   mServiceHandler.sendMessage(msg);
131   return Service.START_NOT_STICKY;
132 }

public void handleMessage(Message msg) {
159   int serviceId = msg.arg1;
160   Intent intent = (Intent)msg.obj;
161   if (intent != null) {
162     String action = intent.getAction();
163
164     if (MESSAGE_SENT_ACTION.equals(intent.getAction())) {
165       handleSmsSent(intent);
166     } else if (SMS_RECEIVED_ACTION.equals(action)) {
167       handleSmsReceived(intent);

private void handleSmsReceived(Intent intent) {
279   SmsMessage[] msgs = Intents.getMessagesFromIntent(intent);
280   Uri messageUri = insertMessage(this, msgs);
…
private Uri insertMessage(Context context, SmsMessage[] msgs) {
331   // Build the helper classes to parse the messages.
332   SmsMessage sms = msgs[0];
333
334   if (sms.getMessageClass() == SmsMessage.MessageClass.CLASS_0) {
335     displayClassZeroMessage(context, sms);
336     return null;
337   } else if (sms.isReplace()) {
338     return replaceMessage(context, msgs);
339   } else {
340     return storeMessage(context, msgs);
341   }
342 }
```

Figure 1 Code snippet of class *SmsReceiverService* of the MMS app.

of concept for them. The vulnerability report to Android Open Source Project is in progress.

The rest of this paper is structured as follows: Section 2 presents an illustrative example. Section 3 gives the problem definition. In Section 4, we present an overview of our approach. Section 5 details the implementation of EasyIVD. Section 6 evaluates the performance of EasyIVD together with case studies of discovered vulnerabilities. Section 7 discusses the limitation of EasyIVD. Related work is presented in Section 8. Section 9 is a brief conclusion.

2 Running example

In this section, we propose an example of capability leak vulnerabilities (a typical and severe kind of input validation vulnerability) in MMS application, an original Android app in charge of storing, sending and receiving short messages and multimedia messages [11]. A service component of MMS lacks the necessary security check due to the carelessness of programmers. In consequence, a running malicious app can fake arbitrary incoming SMS text messages. This vulnerability can be exploited through phishing or fraud attacks to compromise user privacy and mobile payments. The affected platforms range from Android 2.2 to 4.1.2.

In the Android security model, only applications requiring *write_SMS* permission explicitly in *manifest.xml* can invoke system API *storeMessage*() to manipulate SMS database. In principle, the application should check the caller's permission when its SMS management interface is exported to other applications. Figure 1 shows the functions of *SmsReceiverService*, which is a service component handling the

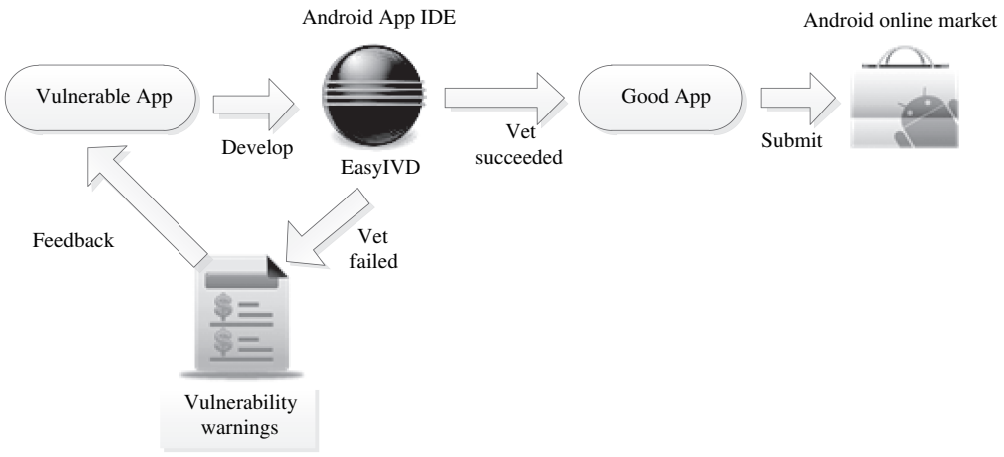

Figure 2 Deployment of EasyIVD.

incoming Intent. Lines 130, 167, 280, and 340 in Figure 1 show that a fake message can be injected into *SmsReceiverService* by arbitrary applications without any restriction of permissions. The MMS app violates the guideline for developers[2)], and it leads to unexpected exposure of SMS manipulation. The exploit is available in [12].

In addition to the typical capability leak vulnerability, a null pointer dereference flaw also exists in the application. Line 332 shows that variable *msgs* is dereferenced before checking whether it is null, which can be leveraged by malicious apps to launch a persistent attack to block SMS communication.

In summary, two factors must be met for an input validation vulnerability as above. Firstly, malicious input can be injected into an app's components through the ICC interface. Secondly, the validations are not designed properly. The two factors lead to many security-sensitive behaviours. As [1] says, "this is a recipe for disaster".

3 Problem definition

In this section we present a formal definition of the input validation vulnerability.

Definition 1 (Transaction)**.** A transaction is a statement collection that consists of a functionality-invocation statement and other data-dependent statements. Suppose transaction $T = \{s_1, s_2, \ldots, s_n\}$. $s_1, s_2, \ldots, s_n$ are continuous program statements and s_n is the functionality-invocation statement. For $\forall s_i$ $(i \in [2, n])$, there is s_j $(j \in [1, i))$ satisfying variable u in s_i defined by variable v at s_j.

Definition 2 (Constraint)**.** Constraints are checking conditions impacting the execution path of a transaction. These constraints are for validation and all of them should be satisfied before the transaction could be executed completely. A checking statement S is the constraint of a transaction T if and only if S is the dominator of T's statements in the CFG. Like a transaction, suppose constraint $C = \{s_1, s_2, \ldots, s_n\}$. $s_1, s_2, \ldots, s_n$ are continuous program statements and s_n is the checking statement. For $\forall s_i$ $(i \in [2, n])$, there is s_j $(j \in [1, i))$ satisfying variable u in s_i defined by variable v at s_j.

Definition 3 (Input validation vulnerability)**.** A input validation vulnerability is a transaction that lacks necessary constraints. Suppose enforced security rule $SR = \{s | s \text{ is necessary constraint}\}$, T is vulnerable if $\exists s \in SR, \forall c \in T.Constraints, s \neq c$. From the definition, the major challenge of detecting an input validation vulnerability is to gain or infer the correct security rules for a typical transaction.

We anticipate our proposed technique will be leveraged as a vetting plugin in the Android IDE, as illustrated in Figure 2. During the development of a new app, developers can run our plugin to detect input validation vulnerabilities. If such a vulnerability is discovered, a report would be generated to warn the app developers and guide them about how to patch it. Thus, vulnerable apps with input

2) Developing secure mobile applications for Android. https://www.isecpartners.com/media/11991/isec_securing_android_apps.pdf.

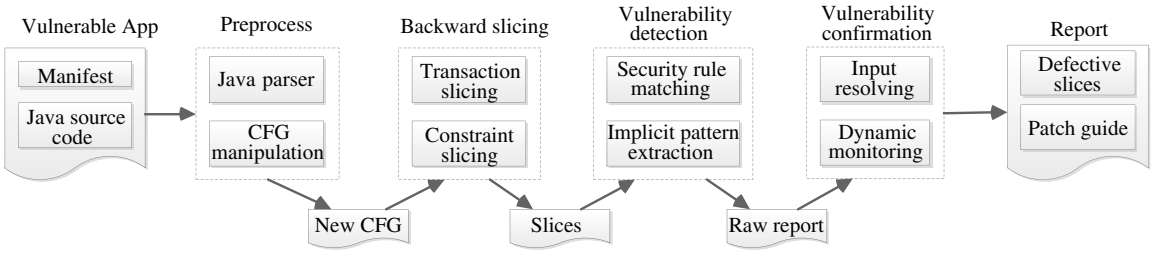

Figure 3 The architecture of EasyIVD.

validation vulnerabilities will not reach the market, and users will not suffer from attacks caused by the vulnerabilities. We would like our tool to be practical and it should not require that the developers have specialist knowledge. The tool should be extensible with new detecting rules provided by security experts.

4 System design

Figure 3 depicts the workflow of our vulnerability-detection technique. It works in the following steps:

(1) Preprocess. The source code of an Android app consists of Java, C++ (native code) and XML (manifest.xml etc.) code. We focus on the Java source code and manifest.xml file. Firstly, we parse the Java code and generate CFG and system dependence graph (SDG). Then we convert the basic CFG into a manipulated CFG, which only has basic blocks and conditional jumps, and does not have any loops or switches.

(2) Backward slicing. In this step, we perform *flow-sensitive and context-sensitive inter-procedural backward slicing* [13] to extract *transaction slices* and *constraint slices*. The backward slicing starts from a statement of a sensitive system API invocation, referred to as a "slicing criterion", and extracts the minimal set of statements that the slicing criterion is dependent on. To distinguish it from other kinds of program slices, we call this slice the *transaction slice*. We also extract the *constraint slice* by starting slicing from each "if" statement in the control flow of every transaction slice. The slicing criterion of the constraint slice contains the enforced conditions of the "if" statement, which determine the branches in the control flow of the transaction.

(3) Vulnerability detection. At slice level, the problem of detecting an input validation vulnerability is transformed into how to detect a pattern violation. The work is divided into two parts: first, a known-pattern vulnerability is detected using security rules. Prior work provides many vulnerability patterns, which help us to write our own security rules.

Second, unknown-pattern vulnerabilities are mostly due to specific application logic. In that case, EasyIVD extracts these undocumented constraints using frequent-pattern mining. Firstly, EasyIVD divides the transaction slices into different categories by comparing slicing criterions. Then in each category, the implicit security specifications are mined as frequent patterns. We verify all the extracted constraints and predefined security rules and report the suspicious violations.

(4) 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. In contrast, complex suspicious flaws will be left for manual validation. Then the final report is generated with defective slices and patch guides.

5 System implementation

5.1 Preprocess

In this part, we parse the Java source code and convert the basic CFG of an app into a manipulated CFG, which only has basic blocks and conditional jumps, and does not have any loops or switches.

(1) Parsing Java code. Firstly, we leverage *JavaParser*, an open-source parser written by *jgsser*, to parse the Java code and generate the abstract syntax tree. All statements in the abstract syntax tree are transformed automatically by *JavaParser* into Static Single Assignment form, which is an intermediate representation well suited to data flow analysis because each variable is assigned exactly once, and every variable is defined before it is used. Finally, *JavaParser* constructs the CFG and SDG.

(2) Adding checks from manifest. In this part, EasyIVD adds checking statements for every component according to its manifest file. If a component has an access control policy for the ICC mechanism, EasyIVD adds a check in the form of an "if" statement to the entry point of the component, which differs among components[3]. An access control policy is mainly in the form of XML attribute "*exported*" or "*permission*". In particular, the access control policy of content provider contains additional security facilities called "*readPermission*", "*writePermission*" and "*grantUriPermissions*".

These security policies are checked when the component is called. So we add these checks at the entry point of the component. For example, a service component declares its access permission *android.permission.CALL_PHONE*, so a check statement is added at the beginning of its entry functions *onStart*(), *onStartCommand*() and *onBind*(). For an activity, the entry points are *onCreate*() and *onNewIntent*(). For a broadcast receiver, it is *onReceive*(). For a content provider, they are *query*(), *insert*(), *delete*() and *update*().

(3) Transform assert statements. An assertion is a statement that enables developers to test their assumptions in the program. An assert statement throws an exception if the check fails. The control flow often jumps to the exit point when the condition of an assert statement is not satisfied. Therefore, EasyIVD extracts the expression of an assert statement and enforces it as an implicit constraint for the following statements. We focus on two kinds of assert statements: Java Assert class and service hook API. The Java Assert class is an assert statement containing a Boolean expression and throws *AssertionError* if the expression evaluates to false.

Service hook [14] is an alias of the *checkPermission*() API family, which allows a developer to employ more fine-grained and flexible access control policies. The *checkPermission*(*perm*) API call checks if the permission *perm* has been granted to the calling application and throws a security exception when the check fails. Like the Java Assert class, service hooks allow developers to perform a custom runtime check. Service hooks help secure the single method of a service with permissions rather than the whole service component. Developers are able to use the *checkPermission*() API family to arbitrarily enforce a more restrictive policy. Service hooks are necessary constraints and we treat them in the same way as the Java Assert class.

(4) Manipulate the CFG. The "loop" and "switch" statements need to be transformed into "if" statements so that the manipulated CFG only has basic blocks and conditional jumps, which is convenient for the slicing step. For "loop" statements, we treat the loop body as a normal block with the loop condition as the constraint and take its negative condition as constraints for the following statements. For "switch" statements, we transform them into "if" and "else" form.

5.2 Backward slicing

With the manipulated CFG, we now leverage backward program slicing to extract the transaction and constraint slices. The basic algorithm is fairly standard and similar to other work such as [13]. In comparison, our slicing works in the context of the Android platform and thus needs to be somewhere different.

Basic algorithm. The algorithm begins from the last statement of a function (often a "return" statement) and searches all invocations of the security-sensitive system API as slicing criterions backward in the control flow. In particular, when an operation accesses the Internet or mobile communication, manipulates a database or file system, or communicates with other components by Intent, we consider it to be a slicing criterion. Starting from each slicing criterion, we compute all data-dependent statements via backward slicing until we get to the start point of the input. A transaction slice consists of a slicing criterion and all its dependent statements.

3) Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html.

Besides, during the backward traversal of the CFG, for each transaction slice we collect constraint slices by starting slicing from each "if" statement of the transaction. The slicing criterion of the "if" statement contains the enforced conditions, which determine the branches in the control flow of the transaction. Like a transaction slice, a constraint slice consists of a slicing criterion and all its dependent statements. As shown in Algorithm 1, *DEF*(s) is redefined variables of statement s and *REF*(s) is referenced variables of statement s.

Algorithm 1 Backward slicing algorithm

Require: $slice\_criterion$; $control\_flow\_graph\langle D, E\rangle$;
Ensure: Transaction T;
  $dependent\_variables \Leftarrow REF(slice\_criterion)$
  $undefined\_variables \Leftarrow REF(slice\_criterion)$
  $statement\_set \Leftarrow slice\_criterion$
  $constraint\_set \Leftarrow \emptyset$
  $s \Leftarrow control\_flow\_graph.get\_last\_statement(slice\_criterion)$
  while s is not an input point **do**
    if s is an "if" statement **then**
      $statement\_set_x, constraint\_set_x, dependent\_variables_x \Leftarrow backward\_slicing(s, control\_flow\_graph)$
      $constraint \Leftarrow new\ constraint(s, statement\_set_x, constraint\_set_x, dependent\_variables_x)$
      $constraint\_set \Leftarrow constraint\_set \cup constraint$
    else
      if DEF(s) in undefined_variables **then**
        $dependent\_variables \Leftarrow dependent\_variables \cup REF(s)$
        $undefined\_variables \Leftarrow undefined\_variables - DEF(s)$
        $undefined\_variables \Leftarrow undefined\_variables \cup REF(s)$
        $statement\_set \Leftarrow statement\_set \cup s$
        $s \Leftarrow control\_flow\_graph.get\_last\_statement(s)$
      end if
    end if
  end while
  return $new\ Transaction(slice\_criterion, statement\_set, constraint\_set, dependent\_variables)$

Special considerations for Android apps. We have several special considerations for Android environment.

System API list. We choose the security-sensitive system API as slice criterions so listing the security-sensitive system API is the first important problem to solve. We construct the list using Pscout [15], which analyses the permission system of Android and extracts the permission specification from the source code. Pscout's API call mapping helps us to construct the list. In addition, the list also includes ICC API such as *startActivity*() and *sendBroadcast*(). How to construct the list is explained in Subsection 5.3.1.

Set and Get functions. When dealing with the data field of a class object or an instance object, we have to analyse the *Set* and *Get* functions to eliminate the uncertainty during the slicing phase. *Set* functions are just used to update the state of objects and *Get* functions are used to read the state of objects. A control-flow-insensitive data flow analysis identifies the *Set* and *Get* functions and associates them with the relevant data field.

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

Android framework code. During the slicing, we do not look at the Android framework code, which greatly reduce the complexity of EasyIVD. That is because our algorithm focuses on data dependencies between variables, not the concrete values of the variables. If a statement is in the form of "x =

```
1   Transaction Slice:
2   Message msg = mServiceHandler.obtainMessage();
3   msg.arg1 = startId;
4   msg.obj = intent;
5   Intent intent = (Intent)msg.obj;
6   SmsMessage[] msgs = Intents.getMessagesFromIntent(intent);
7   Context context = this;
8   storeMessage(context, msgs);

9   Constraint Slices:
10  if(Exported.check("True"))
11  if(intent != null)
12  if(!MESSAGE_SENT_ACTION.equals(intent.getAction()))
13  if(SMS_RECEIVED_ACTION.equals(action))
14  if(!sms.getMessageClass() == SmsMessage.MessageClass.CLASS_0)
15  if(!sms.isReplace())
```

Figure 4 Transaction and constraint slices from class *SmsReceiverService* of the MMS app.

```
        froyo_allmappings.txt
17274   Permission:android.permission.BLUETOOTH
17275   922 Callers:
17379   <com.android.settings.widget.SettingsAppWidgetProvider$
        BluetoothStateTracker:void toggleState(android.content.Context)> (1)

        Security Rule
1       public void rule(Context context){
2       StateTracker sBluetoothState = new BluetoothStateTracker();
3       if(Outer.permissionCheck("android.permission.BLUETOOTH")||
        Outer.isExported("false"))
4               sBluetoothState.toggleState(context);
        }
```

Figure 5 An example of a security rule.

system_api(y)", we assume x is dependent on y. This assumption works well in most cases.

Slice example. Figure 4 is a transaction and constraint slice example for the code shown in Figure 1. The last statement of transaction slice is the slicing criterion *storeMessage*(), which is a security-sensitive system API and maps the permission *READ_SMS* and *WRITE_SMS*. Also, we present the constraint slices of this transaction. They consist of six security checks, one from manifest.xml (line 10) and five from "if" statements (lines 11–15). These constraints check some property of variables *intent* and *sms*. Due to space limitations, we do not detail the constraint slices.

5.3 Vulnerability detection

With transaction and constraint slices in hand, we can detect vulnerabilities at the slice level. We firstly use security rules to detect known pattern vulnerabilities. Then we extract the implemented undocumented implicit patterns from the slices and verify them. The suspicious slices are listed in the raw reports.

5.3.1 *Security rule matching*

In this subsection we validate each transaction slice by matching to security rules. Prior work has identified many vulnerability patterns for input validation flaws, especially for the confused deputy vulnerability. So we leverage prior work to write detection rules.

The construction of a security rule needs expert knowledge while our purpose is to minimize the knowledge required to do so. We hope that when a new input validation vulnerability is disclosed to the public in the future, a user of EasyIVD will be able to analyse the detail of the vulnerability, extract the core transactions and constraints, and add them to the database.

To achieve this goal, the rules are designed to be written in Java and can be easily read. They are similar to a transaction slice of Java code invoking a critical system API but with the necessary permission

validation. If an extracted transaction slice matches a security rule, we require that the slice performs the same permission checks. If not, the rule violation will be reported.

We construct these rules using Pscout [15]. Its database contains 17892 rules. This kind of rule requires the right access checks before a critical function is invoked. Figure 5 is an example to illustrate how we write a security rule. The first three lines are from Pscout(froyo_allmappings.txt). First, we analyse the code and map the permission BLUETOOTH with the API call *BluetoothStateTracker.toggleState*(). Second, we extract all the dependent statements from *com.android.settings.widget.SettingsAppWidgetProvider* manually to fill the minimal execution context of API invocation. After these steps, a rule is completed constructed. It checks whether the permission BLUETOOTH is validated and whether the component is exported. In our perspective, the rule is a minimal transaction slice with the necessary constraint.

5.3.2 *Implicit pattern extraction*

Input validation is very relevant to application logic and sometimes it is hard to extract application-specific validation behaviours in general vulnerability patterns. To enhance the capability of EasyIVD, we propose an algorithm to extract application-specific validation behaviours as implicit security specifications using frequent-pattern mining and detecting violations at slice level.

Firstly, EasyIVD divides transactions into different categories if their slicing criteria are not the same. Then in each category we collect all the constraints into a repeatless set, in which we extract frequent patterns as implicit security specifications. After that, we verify all the extracted specifications and report the suspicious violations. A challenge is how to judge the relevance of a specific transaction and implicit specification. The details are described below.

(1) Extracting the implemented security specifications.

First, we classify transactions into the same category if they have the same slicing criterion. That is because the slicing criterion is the core statement of a transaction and represents its functionality. Given the slicing criterions of two transactions, $o_1.fun_1(p_1, p_2, \ldots, p_n)$ and $o_2.fun_2(q_1, q_2, \ldots, q_n)$, they are equal only if the classes o_1 and o_2 are the same or inherit from the same parent class, the function names fun_1 and fun_2 are the same and the parameter types are identical. 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

$(O_1 = O_2 \vee \exists Class\ O, (O_1 < O) \wedge (O_2 < O)) \wedge fun_1.name = fun_2.name \wedge (\forall x \in [1, n], (P_x = Q_x \vee \exists Class\ R, (P_x < R) \wedge (Q_x < R)))$

$\Rightarrow o_1.fun_1(p_1, p_2, \ldots, p_n) = o_2.fun_2(q_1, q_2, \ldots, q_n)$.

Specially, ICC API calls such as *startActivity*(), *startService*() and *sendBroadcast*() are treated as identical slicing criterions because their functionalities are identical.

Secondly, we infer the security specifications for each transaction category. We join all constraints from each transaction in a category into a set and then delete all trivial constraints that appear only once in the constraint set. We only use the constraints appearing more than once in different transactions as implicit security specifications. Then the constraint set has all the security checks the transaction category should satisfy. The equality of two constraints is also judged by the equality of their slicing criterions.

Consider slicing criterions of two constraints, $o_1.fun_1(\cdots)\ OP\ o_2.fun_2(\cdots)$ and $o_1'.fun_1'(\cdots)\ OP'\ o_2'.fun_2'(\cdots)$, in which OP is a relational operator $(<, <=, =, <>, >=, >)$. $o.fun(\cdots)$ can be a constant such as null so that all "if" condition statements can be modeled in the above form. The slicing criterions of two constraints are equal only if their expressions and the relational operators are equal. The formal definition is

$o_1.fun_1(\cdots) = o_1'.fun_1'(\cdots) \wedge o_2.fun_2(\cdots) = o_2'.fun_2'(\cdots) \wedge OP = OP'$

$\Rightarrow o_1.fun_1(\cdots)\ OP\ o_2.fun_2(\cdots) = o_1'.fun_1'(\cdots)\ OP'\ o_2'.fun_2'(\cdots)$.

After obtaining these implicit security specifications, we apply them as a mandatory property to the transaction category.

(2) Verification of the security specifications.

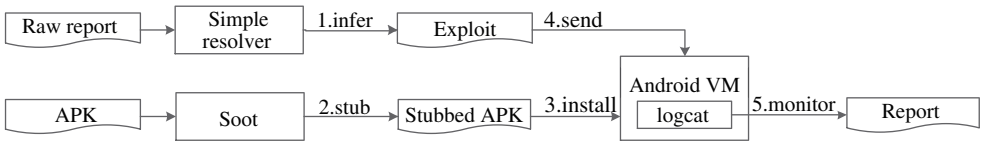

Figure 6 Automatically confirm exploitable vulnerabilities.

Table 1 Four Android distributions

Android version	Number of apps	Number of providers
2.2_r1.1	27	8
4.0.3_r1	34	9
4.4.2_r1	45	8
5.0_r5	46	9

With the help of the extracted security specifications, we start the verification for each transaction category and report any violation as a suspicious vulnerability.

The criterion for judging whether a constraint is necessary and relevant for a transaction is: for any extracted security specification SS_x from a transaction category, if the dependent variable set of a transaction contains the dependent variable set of SS_x, the transaction should contain a constraint which is equal to SS_x. When we infer the relationship for dependent variable sets, we actually use the variable type instead of the 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 the security specification is {*Context, Intent, String, int*}, we can tell that the security specification is necessary for and relevant to the transaction.

5.4 Vulnerability confirmation

In this part, we confirm the raw reports about 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, EasyIVD resolves the constraints, generates exploit code and validates them automatically and (2) if the constraints are too complex to resolve, EasyIVD leaves them for manual validation.

As Figure 6 shows, we implement a very simple resolver to infer the value of the input automatically, such as the *Extra* and *Action* properties. Then we use *Soot* to modify the binary code of the target APK and insert a stub statement, which writes context information to the system log before the security-sensitive API is called. After these steps, EasyIVD installs the stubbed APK onto an Android virtual machine and sends it the generated exploit code. EasyIVD uses *logcat* to see if there is an API invocation or an app crash. If either of these is observed, the vulnerability is recorded as confirmed, else it is recorded as a false alarm. If the input cannot be resolved, or there is no executable app, the vulnerability is recorded as unconfirmed.

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

6 Evaluation

In this section we evaluate the performance of EasyIVD on different Android distributions including 2.2, 4.0, 4.4 and 5.0, as shown in Table 1. We choose original apps in the folder *packages/apps* and *packages/providers* of the Android system source code as test cases because they are available in almost every version of Android ROMs. We load all apps in Eclipse together to evaluate EasyIVD.

Table 2 Detected input validation vulnerabilities (C: confused deputy; N: null pointer dereference)

ID	Apps	2.2_r1.1		4.0.3_r1		4.4.2_r1		5.0_r5		Vulnerable component
		C	N	C	N	C	N	C	N	
1	com.android.mms	1	1	1	1	·	1	·	1	.transaction.SmsReceiverService
2	com.android.bluetooth	·	1	·	1	·	1	·	1	.pbap.BluetoothPbapService
3	com.android.deskclock	1	1	·	1	·	1	·	1	AlarmInitReceiver
4	com.android.launcher	1	·	1	·	1	·	1	·	CloseSystemDialogsIntentReceiver
5	com.android.music	1	·	·	·	·	·	·	·	MediaPlaybackService
6	com.android.phone	2	1	2	1	2	1	2	1	PhoneAppBroadcastReceiver
7	com.android.settings	5	1	1	·	·	·	·	·	SettingsAppWidgetProvider&ChooseLockGeneric
8	com.android.stk	·	2	·	2	·	2	·	2	StkCmdReceiver&BootCompletedReceiver
9	com.android.nfc	×	×	·	1	·	1	·	1	.handover.HandoverManager
10	com.android.providers.media	·	2	·	2	·	2	·	2	MediaScannerService
	In total	11	9	5	9	3	9	3	9	

6.1 Results overview

EasyIVD running on each distribution produces many suspicious input validation vulnerability reports. We then manually verify the reports by checking the corresponding source code. For further verification, we write exploits for each vulnerability and run them on Android emulators[4)]. The results for apps with at least one input validation vulnerability are shown in Table 2. Column "Apps" contains the name of the apps. A dot means we did not find any vulnerability for that category for the target app. The cross means the app was not available for that version (NFC has been available since Android 2.3). Column "Vulnerable component" indicates the name of the component that contains the vulnerabilities.

In total, EasyIVD found 58 input validation vulnerabilities, 44 of which are new and have not been found before. Among them, 22 are confused deputy attacks (8 are new) and 36 are null pointer dereferences (all are new). The experimental results provide encouraging evidence for the effectiveness of EasyIVD.

The results also show that Android's code quality has improved while the number of confused deputy vulnerabilities decreased for later versions. However, null pointer dereference vulnerabilities are still not being taken seriously and the vulnerabilities have not been patched at all in the system updates. In the next subsection, we will give an example to demonstrate that a null pointer dereference can lead to a severe DoS attack. Interestingly, some vulnerabilities only exist in versions 4.x and 5.0 but not 2.2 due to a careless system update.

6.2 Detail analysis

Here we analyse these vulnerabilities in detail to illustrate how EasyIVD works.

Vulnerabilities detected by matching to security rules. The security rule database is effective in detecting confused deputy attacks, which are the most severe kind of input validation vulnerabilities in Android ICC. The backward slicing literally extracts a reachable execution path from the entry point of the Intent to the call site of a security-sensitive system API. The security rule demands that there should be a static permission check or *exported* property check at the entry point (originally in manifest.xml file). Any slices not matching the security rules are reported as suspicious vulnerabilities. App developers can follow the security rule and patch the vulnerabilities.

In detail, the confused deputy vulnerabilities in Apps 3, 4, 5, 6 and 7 (SettingsAppWidgetProvider) are simple confused deputy vulnerabilities. Intent containing only the *action* field and simple *extra* field triggered the vulnerability so EasyIVD was able to confirm these flaws automatically. In comparison, confused deputy vulnerabilities in Apps 1 and 7 (ChooseLockGeneric) are much more complex. The

4) The vulnerability details are available on http://easyivd.sinaapp.com.

```
1  Transaction 1 From:Security.Rules.Bluetooth
2  StateTracker sBluetoothState = new BluetoothStateTracker();
3  sBluetoothState.toggleState(context);
4  Constraint Slices:
5  if(Outer.permissionCheck("android.permission.BLUETOOTH"))
6  if(Outer.isExported("false"))
7
8  Transaction 2 From:SettingsAppWidgetProvider.onReceive
9  sBluetoothState.toggleState(context);
10 Constraint Slices:
11 if(!WifiManager.WIFI_STATE_CHANGED_ACTION.equals(intent.getAction()))
12 if(!BluetoothAdapter.ACTION_STATE_CHANGED.equals(intent.getAction()))
13 if(intent.hasCategory(Intent.CATEGORY_ALTERNATIVE))
14 if(!buttonId == BUTTON_WIFI)
15 if(!buttonId == BUTTON_BRIGHTNESS)
16 if(!buttonId == BUTTON_SYNC)
17 if(!buttonId == BUTTON_GPS)
18 if(buttonId == BUTTON_BLUETOOTH)
```

Figure 7 Capability leak vulnerability of Settings app.

```
1 From:com.android.settings.widget.SettingsAppWidgetProvider  onReceive
2 sBluetoothState.toggleState(context);
3 Lost Constraints :
4
5 if(Outer.permissionCheck("android.permission.BLUETOOTH"))
6 if(Outer.isExported("false"))
7 Patch Guide:
  Add permission restriction "android.permission.BLUETOOTH" in the manifest.xml
8 Set "exported = false" in the manifest.xml
```

Figure 8 An example of an EasyIVD report.

content of the Intent has to be constructed carefully to reach the sensitive API and manipulate privacy information.

In Figure 7, we depict the Bluetooth capability leak vulnerability in App 7 (SettingsAppWidgetProvider). Transaction 1 is a rule stored in the security-rule database. The *exported* property of SettingsAppWidgetProvider should be false or *sBluetoothState.toggleState*() should only be accessed by the caller with the BLUETOOTH permission. Transaction 2 is a transaction extracted by EasyIVD. It has eight constraints checking the *Action*, *Category* and *Data* fields of incoming Intent. Compared with Transaction 1, Transaction 2 misses the constraint "*Outer.check*("*android.permission.BLUETOOTH* || *false*")" and it is reported as input validation vulnerability. Figure 8 is a snippet of the report, showing the defective slice and the patch guide.

As the report says, capability leak vulnerability can be mitigated by adding an access restriction in file manifest.xml. The vulnerable component should set the access permission. In fact, many capability leaks are simply patched in higher versions by setting *exported* property to *false*.

Vulnerabilities detected by implicit pattern extraction. Vulnerabilities of unknown patterns can be detected by implicit pattern extraction. We can find many application-specific constraints and judge whether they are necessary for other transactions. In theory, the more security specifications we extract, the more efficient EasyIVD will become. Even if app developers neglect the vetting process, EasyIVD can discover the necessary validation elsewhere and consider it to be mandatory vetting for all similar transactions.

In detail, the input validation vulnerabilities in Apps 1, 2, 3, 6, 7, 8 and 10 are null pointer dereference vulnerabilities that are missing necessary vetting of the input. App 9 has an array bound error because the malformed content of Intent is used as the index of an array without any checks. These vulnerabilities lead to DoS attacks, which are particularly useful if an adversary wants to stop a critical service [16], e.g., antivirus and security enhancement software. In our observation, a null pointer dereference appears frequently in the input of Android ICC. But not many null pointer dereference flaws have been reported because of two reasons: one is that null pointer dereferences in activity components have minimal impact [16]; the other is that some potential null pointer dereferences cannot actually be triggered because the vulnerable components may be not exported or the pointer is checked somewhere else.

```
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))
```

Figure 9 Null pointer dereference vulnerability of a STK app.

Table 3 Performance of EasyIVD

Version	Apps	LoC	Time (min)	Report	Vul	Confirmed Vul	False alarm	Unconfirmed alarm
2.2_r1.1	35	501485	174	25	20	19	3	1/2
4.1.2_r1	43	764343	223	18	13	11	1	2/4
4.4.2_r1	53	1135733	396	17	11	11	2	0/4
5.0_r5	55	1467650	447	18	11	11	2	0/5

In Figure 9, we depict a null pointer dereference in app *com.android.stk*, which have not been found before. App com.android.stk is a SIM Application Toolkit app, which manages value-added services based on GSM communication. If a STK app crashes, the phone's mobile communications will be cut off.

The flaw exists in the *onReceive*() function of *BootCompletedReceiver* components, as shown in Figure 9. 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 Phone app crashing. A persistent attack would prevent mobile communication totally, both in and out. EasyIVD gets the required security specification from transaction 2, which is extracted from another application (com.android.phone) and supplies the validation we need. Transactions 1 and 2 are divided into the same category because *startService*() and *startActivity*() are treated the same. This kind of situation is not rare, especially when the programmer has weak security concepts.

A null pointer dereference can be mitigated by adding content-checking statements to the source code. For example, the content of Intent needs a null check, and the index of an Array needs a bound check.

6.3 Performance measurement

In this subsection we evaluate the performance of EasyIVD. The results are shown in Table 3. Column "Apps" shows the number of apps and providers for the Android version. Column "LoC" lists the total number of lines of code in the apps and providers for the Android version. Column "Time" shows the running time it takes to process that version of the Android distribution. Column "Report" shows the number of reported suspicious violations. Column "Vul" lists the number of vulnerabilities EasyIVD has detected. Column "Confirmed Vul" shows the number of vulnerabilities EasyIVD has confirmed automatically. Column "False alarm" shows the number of false alarm EasyIVD has confirmed automatically. Column "Unconfirmed alarm" gives the number of unconfirmed alarms, in which the first number is the number of vulnerabilities we confirmed manually.

We measure the processing time by running EasyIVD on an Intel Core i7 2.93 GHz machine with 8 GB of memory and Windows 7 SP1 OS. We believe the average processing time (6 min) per app is reasonable for offline detection. From Figure 10 we can see that the processing time has increased at a faster rate than the number of LoC because the implicit validation mining is applied in all transaction categories.

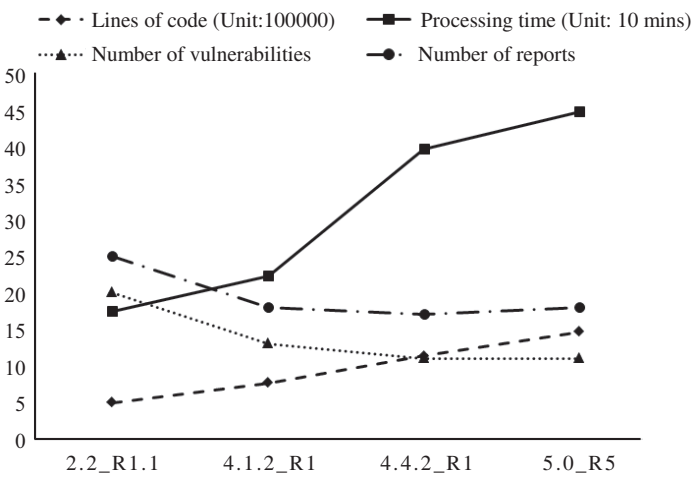

Figure 10 EasyIVD performance.

The larger the category is, the more time it takes for EasyIVD to collect the constraint set and calculate the relevance of transaction and constraint.

We can also see that the vulnerability confirmation module of EasyIVD is effective in decreasing the false positive ratio. Most of the suspicious vulnerabilities EasyIVD reported are simple enough to be confirmed automatically. In particular, it is easy to generate input to check a DoS vulnerability and app crashes can always be monitored by *logcat*. Unconfirmed vulnerabilities are complex and they are described in Subsection 6.2. The analysis of complex vulnerabilities can also benefit from using the vulnerability confirmation module.

We do not measure the number of false negatives because we do not have enough ground truth for the target apps. Here are two circumstances that may lead to false negatives. First, a dynamic vulnerability confirmation can raise false negatives if some system behaviour is not monitored or the resolved input is wrong. Second, since the implicit security specifications are inferred from the extracted transactions, we cannot get enough specifications for flaw detection if the programmer has not included any security-sensitive checks.

7 Discussion

EasyIVD has so far uncovered many input validation vulnerabilities in three Android distributions. It is important to discuss further its advantages and disadvantages.

Advantages of EasyIVD. EasyIVD meets the goals we proposed in Section 3. EasyIVD is a practical framework for detecting input validation vulnerabilities for app developers. With predefined security rules and the semi-automatic vulnerability confirmation module, we have lowered the knowledge requirement impressively. In addition, we have used implicit validation behaviour mining to detect vulnerabilities with an unknown pattern.

Limitations of EasyIVD. EasyIVD is only our first step in detecting input validation vulnerabilities. Although it has identified several serious vulnerabilities in the current Android version, it is still neither sound nor complete. EasyIVD cannot handle the complex situation when several system API calls co-work together to accomplish a transaction. The vulnerability confirmation module is very simple so far and it can be enhanced by leveraging a current constraint resolution technique in the future. Meanwhile, EasyIVD cannot handle the situation when apps use Java reflection techniques.

Detecting other vulnerabilities. A logic vulnerability is another common kind of vulnerability in Android apps, which misleads the legitimate processing flow of an application into an unexpected negative consequence. EasyIVD can detect some logic vulnerabilities. Application logic can be extracted by backward slicing and implicit validation mining can help us to understand undocumented application logic.

8 Related work

Automatic detection of vulnerabilities in Android ICC. When researchers look at Android platform, they focus on a subset of input validation flaws, such as permission re-delegation [3], capability leak [2], and DoS [16]. Felt et al. [3] were the first to discover the permission re-delegation problem in Android ICC and they proposed a defence mechanism. *Stowaway* [17] detects overprivilege in compiled Android applications by comparing the required and requested permissions. To the best of our knowledge, no other approach has addressed static analysis of input validation vulnerabilities at the source-code level comprehensively for Android apps.

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 by focusing on the execution path with necessary checks. Another difference is that EasyIVD looks at the source code and *Woodpecker* at the binary code. This leads to different scenarios. EasyIVD tries to provide solutions for app developers but *Woodpecker* is designed for automatic vetting in an online app market. In comparison, EasyIVD covers all the vulnerabilities that *Woodpecker* detects and EasyIVD can detect more types of input validation vulnerabilities than *Woodpecker*. Additionally, EasyIVD can extract application-specific rules but *Woodpecker* does not have that capability.

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

Automatic inference and understanding security specifications of Android applications. *SCanDroid* [8] and *Kirin* [18] validate manifest files containing the access control policy of an application. Mustafa and Sohr [9] extract the implemented access control policy existing in the form of service hooks from Android system services with the help of program slicing. They admit that their approach would miss some security checks. Compared with their approach, EasyIVD focuses on Android ICC and gets more but smaller slices. We argue that EasyIVD extracts more kinds of constraints and more fine-grained application logic, and covers all the policies that Mustafa and Sohr [9] gains. Some vulnerabilities detected by EasyIVD cannot be identified by Mustafa and Sohr [9], as they do not analyse the specifications in the manifest and "if" statements. None of the other approaches can extract all of the implemented security specifications of Android apps and EasyIVD is the first approach that extracts all security policies including the manifest, service hooks and "if" statements.

To describe access control policies formally, *Kirin* [18], Mustafa and Sohr [9] and Berger et al. [19] use an auxiliary language such as Java Modeling Language (JML), Kirin Security Language (KSL) or Object Constraint Language (OCL). Additional effort is required by a developer to understand the grammar of these languages when creating a new policy. EasyIVD overcomes that disadvantage by directly using Java to describe security rules.

Comprehensive study on Android vulnerabilities. Enck et al. [16] studied Android application security based on the static analysis of 21 million lines of recovered code. Their approach uncovers different kinds of pervasive vulnerabilities and bugs, such as the misuse of personal information and null pointer dereferences. They think that many application-specific errors are often ignored, which significantly inspired us. They found 3925 potential null pointer dereferences in the ICC input of 591 apps (53.7%) with the help of *Fortify SCA*, commercial software for static code analysis. In comparison, EasyIVD found only 9 null pointer dereferences in 35 apps in Android 2.2. In our defence, some potential null pointer dereferences detected by [16] cannot be triggered because the vulnerable components may not be exported or the pointer is checked somewhere else. EasyIVD is able to recognize and discard these false positives. Meanwhile, our research with EasyIVD used original Android apps while [16] uses third-party apps. The quality of the code of the former apps is better than the latter [20].

9 Conclusion

This paper proposes a static approach for detecting input validation vulnerabilities in Android apps. We employ program slicing to extract application logic. Then we detect vulnerabilities with known patterns by matching to security rules and detect vulnerabilities with unknown patterns using implicit pattern extraction. The suspicious flaws are validated by dynamic testing to eliminate some false positives. We implement a prototype plugin named EasyIVD and evaluate it on Android 2.2, 4.0.3, 4.4.2 and 5.0. The results prove that EasyIVD has good precision. In future work, we will leverage more accuracy analysis, such as symbolic execution, and improve the dynamic vulnerability confirmation module.

Acknowledgements This research was supported in part by National Information Security Special Projects of National Development and Reform Commission of China (Grant No. (2012)1424), National Natural Science Foundation of China (Grant Nos. 61572460, 61272481, 61303239), and Open Project Program of the State Key Laboratory of Information Security (Grant No. 2015-MS-04).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Category: input validation on owasp. https://www.owasp.org/index.php/Category:Input_Validation
2 Grace M, Zhou Y J, Wang Z, et al. Systematic detection of capability leaks in stock Android smartphones. In: Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS'12), San Diego, 2012
3 Felt A P, Wang H J, Moshchuk A, et al. Permission re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX Conference on Security (Sec'11), San Francisco, 2011. 22–38
4 Zhou Y J, Jiang X X. Detecting passive content leaks and pollution in Android applications. In: Proceedings of the 20th Network and Distributed System Security Symposium (NDSS'13), San Diego, 2013
5 Lu L, Li Z C, Wu Z Y, et al. Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12), Raleigh, 2012. 229–240
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'14), San Diego, 2014
7 Yang K, Zhuge J W, Wang Y K, et al. IntentFuzzer: detecting capability leaks of Android applications. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS 2014), Kyoto, 2014. 531–536
8 Fuchs A P, Chaudhuri A, Foster J S. SCanDroid: automated security certification of Android applications. Technical Report CS-TR-4991. 2009
9 Mustafa T, Sohr K. Understanding the implemented access control policy of Android system services with slicing and extended static checking. Int J Inf Secur, 2012, 14: 347–366
10 Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09), Chicago, 2009. 235–245
11 Jiang X X. Smishing vulnerability in multiple Android platforms (including Gingerbread, Ice Cream Sandwich, and Jelly Bean). http://www.csc.ncsu.edu/faculty/jiang/smishing.html, 2012
12 Thomascannon. Android sms spoofer. https://github.com/thomascannon/android-sms-spoof, 2012
13 Fang Z J, Zhang Y Q, Kong Y, et al. Static detection of logic vulnerabilities in Java web applications. Secur Commun Netw, 2014, 7: 519–531
14 Enck W, Ongtang M, McDaniel P. Understanding Android security. IEEE Secur Priv, 2009, 7: 50–57
15 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 (CCS'12), Raleigh, 2012. 217–228
16 Enck W, Octeau D, McDaniel P, et al. A study of Android application security. In: Proceedings of the 20th USENIX Conference on Security (SEC'11), San Francisco, 2011. 21–37
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 (CCS'11), Chicago, 2011. 627–638
18 Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09), Chicago, 2009. 235–245
19 Berger B J, Sohr K, Koschke R. Extracting and analyzing the implemented security architecture of business applications. In: Proceedings of 17th European Conference on Software Maintenance and Reengineering (CSMR'13), Genova, 2013. 285–294
20 Zhang Y Q, Liu Q X, Luo Q H, et al. XAS: Cross-API scripting attacks in social ecosystems. Sci China Inf Sci, 2014, 58: 012101