AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction

Jianjun Huang
Department of Computer Science
Purdue University, USA
huang427@cs.purdue.edu

Xiangyu Zhang
Department of Computer Science
Purdue University, USA
xyzhang@cs.purdue.edu

Lin Tan
Electrical and Computer Engineering
University of Waterloo, Canada
lintan@uwaterloo.ca

Peng Wang
School of Information
Renmin University of China, China
pengwang@ruc.edu.cn

Bin Liang
School of Information
Renmin University of China, China
liangb@ruc.edu.cn

ABSTRACT

Android smartphones are becoming increasingly popular. The open nature of Android allows users to install miscellaneous applications, including the malicious ones, from third-party marketplaces without rigorous sanity checks. A large portion of existing malwares perform stealthy operations such as sending short messages, making phone calls and HTTP connections, and installing additional malicious components. In this paper, we propose a novel technique to detect such stealthy behavior. We model stealthy behavior as the program behavior that mismatches with user interface, which denotes the user's expectation of program behavior. We use static program analysis to attribute a top level function that is usually a user interaction function with the behavior it performs. Then we analyze the text extracted from the user interface component associated with the top level function. Semantic mismatch of the two indicates stealthy behavior. To evaluate AsDroid, we download a pool of 182 apps that are potentially problematic by looking at their permissions. Among the 182 apps, AsDroid reports stealthy behaviors in 113 apps, with 28 false positives and 11 false negatives.

Categories and Subject Descriptors

D2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D2.5 [**Software Engineering**]: Testing and Debugging—*Code inspection and walk-throughs*

ICSE'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2756-5/14/05
http://dx.doi.org/10.1145/2568225.2568301

General Terms

Security

Keywords

Android, Stealthy Behaviors, User Interface, Program Behavior Contradiction

1. INTRODUCTION

Android smartphones are becoming increasingly popular. Gartner's analysis shows that 72.4% of smartphones are based on Android [14]. A prominent characteristic of Android phones is that users can easily install miscellaneous apps downloaded from third-party marketplaces without jailbreaking. However, the downside is that Google and other vendors can hardly control the quality of apps on third-party marketplaces. Adversaries can submit their malicious apps and tempt users to install with various lures. Juniper Networks Mobile Threat Center reported a dramatic growth in Android malware population from roughly 400 samples in June 2011 [24] to 175,000 in the third quarter of 2012 [32]. Most are present on third-party marketplaces.

A very popular category of Android malware features stealthy malicious operations such as making phone calls, sending SMS messages to premium-rate numbers, making undesirable HTTP connections and installing other malicious components. It was reported by three recent studies [12, 34, 26] that 52-64% of existing malwares send stealthy premium-rate SMS messages or make phone calls. Note that these actions cause unexpected charges to phone bills [7, 19]. It was observed that stealthy HTTP requests are also very common undesirable behavior in malwares [12]. Besides leaking user information, they could also cause unexpected data plan consumption. In China, it was reported in March 2012 that more than 210,000 Chinese mobile devices were affected by a kind of malwares that could make stealthy HTTP connections inducing charges. They caused around 8 million dollars loss [3].

Despite the pressing need, detecting such malware is challenging as the malicious behavior appears to be indistin-

guishable from that of benign apps. For example, an online shopping app usually provides operation interfaces to help users conveniently call a service number or send a query SMS message. Apps providing travel-aid and adult content often allow users to make phone calls or send messages. Many benign apps allow establishing background HTTP connections (e.g. weather, stock trading and gaming apps). Many also allow users to install additional components.

Existing techniques are insufficient in detecting/preventing stealthy malicious behaviors. A very important protection mechanism on Android is to allow users to perform access control by setting application privileges. However, the access control is very coarse-grained. For example, the SMS messaging capability can either be enabled or completely disabled. It is hard to decide if we should disable for a given app as many benign apps do send SMS messages. Taint analysis [10, 15, 13] allows detecting information leak in apps. But the stealthy behavior in malwares may not leak any private information. Recently, Google provides the capability of blacklisting certain premium-rate phone numbers [17], which provides a potential way of preventing stealthy SMS messages or phone calls. However, keeping such a blacklist up-to-date is a non-trivial challenge. In some countries such as China, there is no difference between a premium-rate number and a regular phone number.

In this paper, we propose a novel technique to detect stealthy malicious behaviors in Android apps. We model stealthy behavior as *the program behavior mismatches with user interface*. The intuition is that user interface (UI) represents the user's expectation of program behavior. Hence, it can naturally serve as an oracle to detect behind-the-scene behavior. For example, an SMS message send triggered by a user interaction that is supposed to set the background color should be considered malicious. The technique consists of two components. One is the static program analysis component that attributes the behavior of interest (e.g. SMS send and HTTP connection) to a top level function with associated UI (e.g. the `onClick()` function of a button). The other is the UI analysis component that makes use of text analysis to analyze the intent described by the corresponding interface artifacts (e.g. the text associated with the button). Any mismatch will be reported as potentially malicious. In the program analysis component, we classify Android APIs into different groups. Each group is assigned an intent type such as SMS send and phone calls. Reachability analysis is performed on control flow graph (CFG) and call graph (CG) to propagate such intents from the API call sites to top level functions. Note that in event driven programming, an invocation of a top level function usually denotes an action or a task that can be considered as a natural unit to reason about stealthiness. The interface analysis component identifies the text of the UI artifact associated with a top level function. Then compatibility check is performed between the intents from program analysis and those extracted from the interface text.

Our contributions are summarized as follows.

- We propose a method to detect Android malware that performs stealthy operations including SMS message send, phone calls, HTTP connections and component installations. It is based on the novel idea of detecting mismatches between program behavior and user interface.
- We found that in many cases even though there is no direct match between an API intent (e.g. SMS send) and the UI text, the API may be correlated with other APIs that explicitly expose the behavior (e.g. an API call that logs the SMS send to the mail box). In such cases, the behavior should not be considered stealthy. We propose an in-depth analysis that considers program dependences between APIs to identify their correlations and hence improve precision.
- We formally present our design using *datalog* rules. The design handles a number of Android-specific challenges.
- We implement a prototype called AsDroid (*Anti-Stealth Droid*). We collect a pool of 182 apps that have the permissions to perform the malicious operations of interest. AsDroid reports that 113 of them have stealthy behaviors, with 28 false positives and 11 false negatives.

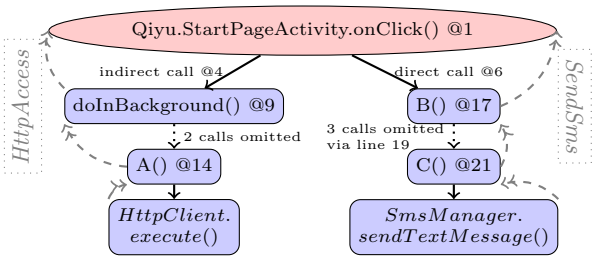
2. MOTIVATING EXAMPLE

We use a real application *Qiyu* to motivate our technique. It is a location-based social networking service application on Android. Some relevant code snippets are shown in Fig. 1(a) and part of the corresponding call graph is in Fig. 1(b). The entry function `onClick()` (at line 1) is the handler of a button with text "`One-Click Register & Login`". The scenario is as follows. When the user clicks the button, the app checks the current environmental settings. In most cases, the true branch is taken, in which an asynchronous task is appended to the task queue and executed (line 4). This causes an indirect invocation to a predefined handler `doInBackground()` at line 9, which is always implicitly called by the Android runtime to perform some background processing when a task starts to execute. The function transitively calls method `A()` (in class `Woa.BA`) at line 14. The method connects to a website through `HttpClient.execute()` at line 15 to perform registration or login. The chain of function calls is also shown on the left of Fig. 1(b). When the test at line 2 fails, the else branch (line 5) is taken. A different chain of function invocations are made, eventually leading to an SMS message being sent inside method `C()` (in class `Woa.AK`) at line 23 without the user's awareness. The chain is shown on the right of Fig. 1(b). Note that we omit three function calls between the asynchronous task execution at line 19 and method `C()` for brevity.

To detect stealthy behaviors, our program analysis component first attributes top level functions with intents by analyzing the operations of interest directly or transitively performed by such functions. We classify Android APIs to a few pre-defined intent types. In this example, `HttpClient.execute()` at line 15 denotes the **HttpAccess** intent and `SmsManager.sendTextMessage()` at line 23 denotes the **SendSms** intent. The intents get propagated upward along the call edges (see Fig. 1(b)) and eventually aggregated on the top level node `onClick()`, which is a user interaction function, suggesting the operations performed by this function should reflect what the UI states. The UI analysis component identifies the UI artifacts corresponding to the `onClick()` function, i.e. the button and its residence dialog. It further extracts the text on these interface artifacts and performs text analysis to identify a set of keywords. In this example, they are "`Register`" and "`Login`". AsDroid

```
// In class Qiyu.StartPageActivity
01:  public void onClick(View v){
02:    if(/*test environment*/){
03:      Woa.F f = new Woa.F(v, this);
04:      f.execute(new String[0]);//trigger line 9
05:    } else ...{
06:      Woa.AG.B();//invoke line 17
07:    }
08:  }
// In class Woa.F
09:  public Object doInBackground(Object[] objs){
10:    //transitively calls Woa.BA.A() at line 14
11:  }
// In class Woa.BA
12:  private org.apache.http.client.HttpClient h;
13:  private org.apache.http.client.methods.HttpGet d;
14:  public void A(){
15:    this.h.execute(this.d); //HttpClient.execute(...)
16:  }
// In class Woa.AG
17:  public static void B(){
18:    Woa.U u = new Woa.U();
19:    u.execute(...);//transitively calls C() at line 21
20:  }
// In class Woa.AK
21:  public static boolean C(Context c, String s1, String s2){
22:    SmsManager sm = SmsManager.getDefault();
23:    sm.sendTextMessage(s1, null, s2, null, null);
24:  }
```

(a) Simplified Code Snippet



(b) Call Graph and Intent Propagation

Figure 1: Motivating Example in app *Qiyu*.

looks-up the compatibility of the keywords and the intents identified by the program analysis component from a dictionary generated before-hand in a training phase. In this case, the **HttpAccess** intent is compatible but **SendSms** is not. Our tool hence reports the contradiction.

There are cases that multiple intents of a top level function are correlated. For example, a dialog may be popped up after a SMS message send to indicate the success of the send, even though the button that initiates the send does not have any textual hint about sending messages. In this case, the SMS send is not stealthy. The display of a dialog has the **UiOperation** intent. Both the **UiOperation** and **SendSms** intents reach the top level function. We hence analyze if the intents are correlated by analyzing their program dependences. Since **UiOperation** is not stealthy, the correlation between the **UiOperation** and **SendSms** intents suggests the sanity of the SMS send behavior.

3. DESIGN

In this section, we first define six types of intents that are of our interest. The corresponding APIs are commonly used in Android apps.

SendSms. This intent corresponds to SMS send APIs, including `sendTextMessage()`, `sendDataMessage()` and `sendMultipartTextMessage()` declared in class `SmsManager`. These API functions are usually executed in the background. An SMS send through a separated messaging app is not taken into consideration in this paper because it requires the user to explicitly interact with the messaging app to finish the process and hence is not stealthy.

PhoneCall. It corresponds to a direct phone call, namely, invoking `startActivity()` with action `android.intent.action.CALL`. Malware can leverage the automated calling mechanism to dial a number without the user's awareness. Phone calls can also be made through `startActivity()` with an action `android.intent.action.DIAL`. However, we do not model this API because explicit user approval is needed when the API is used.

HttpAccess. This intent describes HTTP access APIs. It includes `URL.openConnection()`, `URL.openStream()`, `AbstractHttpClient.execute()`, and so on. HTTP access is commonly used in Android apps for a wide range of purposes.

Install. It describes API functions that are for installing other components or applications. Many Android malwares have their payload as installing another piece of malicious code. Benign apps may also need to perform installation, which is however usually authorized or explicitly guided by the user. Modeled functions include `Runtime.exec()` with `"pm install"` as the argument, and `ProcessBuilder.start()` using `"pm"` and `"install"` to build a new process.

SmsNotify. In some cases, the user does not need to (or cannot) authorize a message send operation. But after the operation, the app may automatically notify the user that there was an SMS send. In this case, we should not consider the message send as a stealthy action even though the user interface that leads to the SMS send operation does not have any textual implication of the operation. One typical example is that a copy of the message is saved to the user's mail-box to record what just happened. Hence, we model the following API to the **SmsNotify** intent: `ContentResolver.insert()` and the destination table is given by a URL "`content://sms`". It means inserting data into the preloaded database for short messages.

UiOperation. A top level user interaction function may display more user interface elements to allow further interactions with the user. In some cases, UI display operations may be correlated to some of the aforementioned intents. For example, a dialog may be popped up after an SMS send to notify the user about the send. In such cases, the SMS send is not stealthy. To reason about these cases, we associate the UI display API functions such as `AlertDialog$Builder.setMessage()`, `ImageView.setImageBitmap()`, and `View.setBackgroundDrawable()`, with the **UiOperation** intent.

3.1 Intent Propagation

In this section, we describe how intents are propagated to top level functions such that we can check compatibility with the corresponding UI text. We also describe how to detect correlation between intents. Intent propagation is based on call graph. The calling convention of Android apps has its unique features, which need to be properly handled. Intent correlation analysis is mainly based on program dependences. However, correlated intents do not simply mean there are (transitive) dependences between them.

The analysis is formally described in the *datalog* language

Atoms

$apiIntent(L,T)$	:	API call at program point L has intent type T.
$def(L,X)$	:	variable X is defined at program point L.
$use(L,X)$	:	variable X is used at program point L.
$actual(L,M,X)$	:	variable X is the M^{th} actual argument at call site L.
$formal(F,M,X)$	:	variable X is the M^{th} formal argument of function $F()$.
$inFunction(F,L)$	:	program point L is in function $F()$.
$funEntry(F,L)$	:	program point L is the entry of function $F()$.
$hasDefFreePath(L_1,L_2,X)$	:	there is a path from L_1 to L_2 along which X may not be defined.
$componentEntry(X,F)$	:	$F()$ is the entry of Android component X. e.g. *onCreate*() of an *Activity* or a *Service* component.
$immediateCD(L_1,L_2)$	:	program point L_2 is immediately control dependent on L_1 in the same function.
$directInvoke(F_1,F_2,L)$	:	F_1 invokes F_2 at program point L
$indirectInvoke(F_1,F_2)$	:	F_2 is the actual destination of $F_1()$ in event-driven circumstances, e.g. (1) *Thread.start*() → *Runnable.run*(); (2) *Handler.sendMessage*() → *Handler.handleMessage*().
$iccInvoke(F_1,F_2,L)$	:	F_1 invokes a function F_2 for inter-component communication purpose at L. F_2 should be APIs like *startActivity*(), *startService*().

Rules

/*$invoke(F_1,F_2,L)$: F_1 invokes F_2 at program point L.*/
$invoke(F_1,F_2,L)$:- $directInvoke(F_1,F_2,L)$
$invoke(F_1,F_2,L)$:- $iccInvoke(F_1,F_3,L)$ & $actual(L,1,X)$ & "L_1: $X.setClass(...)$" & $actual(L_1,2,Y)$ & $componentEntry(Y,F_2)$
$invoke(F_1,F_2,L)$:- $invoke(F_1,F_3,L)$ & $indirectInvoke(F_3,F_2)$
$invoke(F_1,F_2,L)$:- $invoke(F_1,F_3,L)$ & $invoke(F_3,F_2,L)$

/*$hasIntent(F,T,L)$: $F()$ has *intent* type T and the corresponding API call is at L.*/
$hasIntent(F,T,L)$:- $invoke(F,A,L)$ & $apiIntent(L,T)$
$hasIntent(F,T,L_1)$:- $hasIntent(F_1,T,L_1)$ & $invoke(F,F_1,L_2)$

/*$controlDep(L_1,L_2)$: program point L_2 is control dependent on L_1.*/
$controlDep(L_1,L_2)$:- $immediateCD(L_1,L_2)$
$controlDep(L_1,L_2)$:- $inFunction(F_1,L_1)$ & $inFunction(F_2,L_2)$ & $invoke(F_1,F_2,L_3)$ & $controlDep(L_1,L_3)$

/*$defUse(L_1,L_2)$, $useUse(L_1,L_2)$: data at L_1 and L_2 are data correlated.*/
$defUse(L_1,L_2)$:- $def(L_1,X)$ & $use(L_2,X)$ & $hasDefFreePath(L_1,L_2,X)$
$defUse(L_1,L_2)$:- $invoke(F_1,F_2,L_1)$ & $actual(L_1,M,X)$ & $formal(F_2,M,Y)$ & $funEntry(F_2,L_3)$ & $hasDefFreePath(L_3,L_2,Y)$ & $use(L_2,Y)$
$useUse(L_1,L_2)$:- $defUse(L_3,L_1)$ & $defUse(L_3,L_2)$
$useUse(L_2,L_1)$:- $defUse(L_3,L_1)$ & $defUse(L_3,L_2)$

/*$correlated(L_1,L_2)$: L_1 and L_2 are data/control correlated.*/
$correlated(L_1,L_2)$:- $controlDep(L_1,L_2)$
$correlated(L_1,L_2)$:- $defUse(L_1,L_2)$
$correlated(L_1,L_2)$:- $useUse(L_1,L_2)$
$correlated(L_1,L_2)$:- $correlated(L_1,L_3)$ & $correlated(L_3,L_2)$

/*$correlatedIntent(F,T_1,L_1,T_2,L_2)$: In function F, intent T_1 at L_1 is correlated to T_2 at L_2*/
$correlatedIntent(F,T_1,L_1,T_2,L_2)$:- $hasIntent(F, T_1, L_1)$ & $hasIntent(F, T_2, L_2)$ & $correlated(L_1,L_2)$

Figure 2: Datalog Rules for Intent Propagation and Correlations

[5], which is a Prolog-like notation for relation computation. It provides a representation for data flow analysis in the form of formulated relations. The inference rules on these relations are shown in Fig. 2. Relations are in the form $p(X_1, X_2, ..., X_n)$ with p being a predicate. $X_1, X_2, ..., X_n$ are terms of variables or constants. In our context, variables are essentially program artifacts such as statements, program variables and function calls. A predicate is a declarative statement on the variables. For example, $inFunction(F,L)$ denotes if a statement with label L is in function F.

Rules express logic inferences with the following form.

$$H \text{ :- } B_1 \text{ \& } B_2 \text{ \& ... \& } B_n$$

H and $B_1, B_2,...B_n$ are either relations or negated relations. We should read the :- symbol as "if". The meaning of a rule is if $B_1, B_2,...B_n$ are true then H is true.

Relations can be either inferred or atoms. We often start with a set of atoms that are basic facts derived from the compiler and then infer the other more interesting relations through our analysis. We use WALA [22] as the underlying analysis infrastructure. We leverage its *single static assignment* (SSA) representation, control flow graph, part of call graph, and the MAY-points-to analysis to provide the atoms.

Atom $apiIntent(L,T)$ denotes an intent T is associated with an API call at L, reflecting our API classification. Atom $hasDefFreePath(L_1,L_2,X)$ indicates there is a program path from program point L_1 to L_2 and along the path (not including L_1 or L_2), variable X may not be defined. This is to compute the $defUse(L_1, L_2)$ relation that denotes if a variable is defined at L_1 and used at L_2. To generate the atom relation, we leverage the SSA form and the points-to analysis. The analysis is conservative. If we are not sure X must be re-defined along the path, we assume the path is definition free. The paths we are considering include both intra- and inter-procedural paths.

Android apps are component based. Generally, there are four types of basic components: *Activity*, *Service*, *Broadcast Receiver* and *Content Provider*. Activity component is for a single UI screen. Service component is for long-running operations in the background (without any UI). Broadcast receiver responds to system-wide broadcast announcements. Content provider is used for application data management [18]. Inter-Component Communication (ICC) is used to deliver data between components, which is similar to traditional function invocations. We have to model

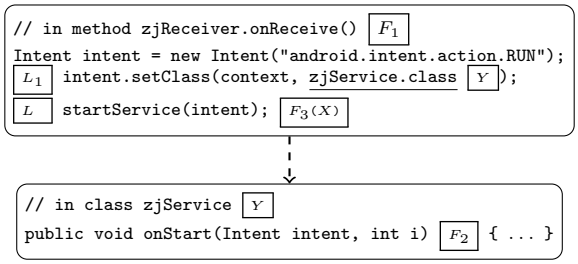


Figure 3: ICC call chain example in *GoldDream*.

such communication as a function may transitively invoke API functions with intent of interest through ICC. However, the calling convention of ICC is so unique that the underlying WALA infrastructure cannot recognize ICC invocations. Fig. 3 shows an example from a real world app *GoldDream*. Inside the `zjReceiver.onReceive()` function, there is an ICC call to the `onStart()` function of the `zjService` component. Observe that the invocation is performed by creating an Android `Intent` object[1], which can be considered as a request that gets sent to other components to perform certain actions. The target component is set by explicitly calling `setClass()` of the Android Intent object. The request is sent by calling `startService()` with the Android Intent object. The Android runtime properly forwards the request to the `onStart()` function of the `zjService` component.

To capture such call relation, we introduce the *componentEntry*(X,F) atom with X a subclass of `Service`, `Activity` or `BroadcastReceiver`. The entry point F denotes `onCreate()`, `onStart()`, and `onReceive()`, which are also called *lifecycle* methods by Android developers. We introduce atom *iccInvoke*(F_1,F_2,L) with F_2 denoting special ICC functions, such as `startActivity()`, `startService()` and `sendBroadcast()`. The second inference rule of the *invoke*(F_1, F_2,L) relation describes how we model ICC as a kind of function invocation. Let's use the example in Fig. 3 to illustrate the rule. It allows us capture the call chain `zjReceiver.onReceive()` → `startService()` → `zjService.onStart()`. Labels L, L_1, F_1, F_3, and Y in Fig. 3 correspond to those in the second *invoke*() rule.

Atom *directInvoke*(F_1,F_2,L) denotes regular function calls including virtual calls, leveraging WALA. Atom *indirectInvoke*(F_1,F_2) denotes another special kind of function invocations in Android apps, namely, implicit calls in thread execution and event handling. A typical indirect call is a thread-related invocation, e.g., actual call destination of `Thread.start()` is the `run()` method of the corresponding class. The function call `f.execute()` → `doInBackground()` in Fig. 1 (i.e., line 4 → line 9) is an example for event handling indirect invocation. We detect these implicit calls through pre-defined patterns.

Relation *hasIntent*(F,T,L) denotes function F is tagged with an intent T initiated by the API call at program point L. For example, in Fig. 1, we can infer the following:

hasIntent (F = `StartPageActivity.onClick()`,
T = **SendSms**,
23 /*`sm.sendTextMessage(...)`*/) = TRUE.

Observe that the first *hasIntent*() rule tags the enclosing function of an API call. The second rule propagates a tag from a callee to the caller. Note that a function may have multiple intents. These intents may be of the same type (but initiated at different API call locations).

The remaining relations and rules are for intent correlations. Relation *correlated*(L_1,L_2) determines if two program points L_1 and L_2 are correlated. Correlation can be induced by definition-use, use-use, and control dependence relations, described by relations *defUse*(), *useUse*(), and *controlDep*(), respectively. The fourth *correlated*() rule suggests that the relation is transitive.

The first rule of *defUse*(L_1,L_2) is standard. In our implementation, we leverage SSA form to derive definition-use relation for local and global variables. We leverage points-to relation to reason about definition-use relation for object fields. The second rule is to capture definition-use relation by parameter passing, including those through Android specific calling conventions. The basic idea is that we consider a formal argument Y used inside the callee at L_2 is defined at the call site L_1 (in the caller) if it is not re-defined along the path from the callee entry to the use site.

The relation *useUse*(L_1,L_2) denotes that there are uses at L_1 and L_2 coming from the same definition point. For example, L_1 and L_2 could be the two uses of the same variable in the two branches of a predicate. Considering use-use relation in the *correlated*() relation is the key difference from standard program dependence analysis that considers only definition-use and control dependence relations.

Computation of *controlDep*(L_1,L_2) is standard except that it also models inter-procedural control dependence. Particularly, all statements in a callee have control dependence with a predicate in the caller that guards the call site.

Finally, the relation *correlatedIntent*(F,T_1,L_1,T_2,L_2) denotes if two intents T_1 and T_2 at function F are correlated.

Example. Fig. 4 shows a correlation analysis example in app *Shanghai 1930*. `ContentResolver.insert()` at line 15 stores the sent text message into the mail box and it hence has intent type **SmsNotify**. It is determined to be correlated to the SMS sending operation with **SendSms** intent at line 7. According to the definition-use graph in Fig. 4(b), line 15 is correlated with line 10 (both use *cv* defined at line 9) by the *useUse*() rules. Line 10 is further correlated with line 7 because of variables *v8*, again by the *useUse*() rules. Hence, we have *correlatedIntent*(`PaySmsActivity.a()`, **SendSms**, 7, **SmsNotify**, 15)=TRUE. Intuitively, the two intents are correlated because the same content is being sent over a short message and written to the mail box. Thus, the message send is not stealthy.

3.2 UI Compatibility Check

After intents are propagated to top level functions, the next step is to check their compatibility with the text of the corresponding user interface artifacts.

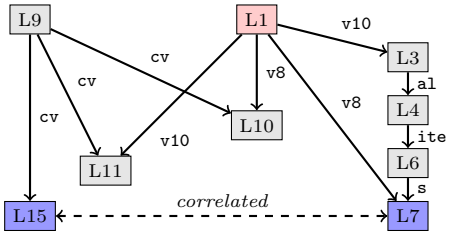
Acquiring User Interface Text. Given a top level function, we need to first extract the corresponding text. User interface components in an Android app are organized in a view tree. A view is an object that renders the screen that the user can interact with. Views can be organized as a tree to reflect the layout of interface. There are two ways to construct the layout: (1) statically through an XML resource file; (2) dynamically by constructing the view tree at runtime.

With the static layout construction, upon the creation

[1] `Intent` is a standard class in Android. We call it Android Intent in order to distinguish with the intents we associate with API functions.

```
// in class PaySmsActivity
01:  void a (String v8, String v9, String v10){
02:     SmsManager sm = SmsManager.getDefault();
03:     ArrayList al = SmsManager.divideMessage(v10);
04:     Iterator<String> ite = al.iterator();
05:     while (ite.hasNext()){
06:       String s = ite.next();
07:       sm.sendTextMessage(v8,v9,s,null,null);
08:     }
09:     ContentValues cv = new ContentValues();
10:     cv.put("address",v8);
11:     cv.put("body",v10);
12:     cv.put("type",2);
13:     ContentResolver cr = getContentResolver();
14:     Uri uri = Uri.parse("content://sms");
15:     cr.insert(uri,cv);
16:  }
```

(a) Code Snippet



(b) Part of Definition-Use Relations. Solid arrows labeled with variable names indicate def-use relation.

Figure 4: Intent Correlation Example in app *Shanghai 1930*.

of an activity, the corresponding user interface is instantiated by associating the activity with the corresponding XML file by calling `setContentView([XML layout id])`. The Android core renders the interface accordingly. A UI object has a unique ID. The ID is often specified in the XML file. Inside the app code, the handle to a UI object is acquired by calling `findViewById([object id])`. For example, the following text defines a button in the XML file. Note that the button text is also specified.

```
<Button android:id="@+id/my_button"...
     android:text="@string/my_button_text"/>
```

Its handle can be acquired as follows. Note that the lookup id matches with that in the XML file.

```
Button btn = (Button)findViewById(R.id.my_button);
```

The event handler for an UI object is registered as a listener. For example, one can set the listener class for the previous button by making the following call.

```
btn.setOnClickListener(new MyListener(...));
```

In this case, the `onClick()` method of the `MyListener` class becomes the top level user interaction function associated with the button. Next we describe how we extract text for different kinds of functions.

For a top level *interactive* function F (e.g. `onClick()`), AsDroid identifies the corresponding UI text as follows. It first identifies the registration point of the listener class of F. From the point, AsDroid acquires the UI object handle, whose ID can be acquired by finding the corresponding `findViewById()` function. The ID is then used to scan the layout XML file to extract the corresponding text. AsDroid also extracts the text in the parent layout. For example, the parent layout of a button may be a dialog. Important information may be displayed in the dialog and the button may have only some simple text such as "`OK`". We currently cannot handle cases in which the text is dynamically generated. We found such cases are relatively rare.

Algorithm 1 Generating Keyword Cover Set.

```
train(S, F)
  KWD=φ /*the keyword cover set*/
  while F ≠ φ do
    sort S by keyword (or keyword pair) frequency
    k=the top ranked keyword (or pair) in S
    X= the functions in which k occurs
    KWD=KWD ∪ k
    F= F-X
    S=S-{all the keywords (pairs) in X}
  end while
```

Some *non-interactive* top level functions also have associated UIs, for instance, the lifecycle methods `onCreate()` and `onStart()` of activity components. These methods are invoked when the screen of an activity is first displayed. While no user interactions are allowed when executing these methods, the displayed screen may have enough information to indicate the expected behavior of these methods, such as loading data from a remote server. Hence, for an activity lifecycle method, AsDroid extracts the text in the XML layout file associated with the activity.

Text Analysis. Once we have the text, we build a dictionary that associates a type of intent to a set of keywords through training. We use half of the apps from the benign sources[2] as the training subjects, which account for about 28% of all the apps we study. During evaluation, we use the dictionary generated from the 28% apps to scan over the entire set of apps. Here, we assume the training apps are mostly benign. If an intent appears together with some text in a benign case, then the intent and the text are compatible. We use keywords to represent text, and build compatible keyword cover set for each intent. In particular, For each intent type T of interest, we identify all the top level functions F that have T annotated and collect their corresponding texts. We then use Stanford Parser [25] to parse the text to keywords. We populate a universal set S to include all individual keywords and keyword pairs that appear in these functions. We then use Algorithm 1 to identify the smallest set of keywords (or pairs) that have the highest frequency and cover all the top level functions tagged with T.

The algorithm is similar to the greedy set cover algorithm [8]. It picks the most frequently occurring keyword k at a time and adds it to the keyword set. Then it removes all the keywords that appear in the top level functions in which k occurs, as they can be covered by k. It repeats until the set of functions are covered.

We consider keyword pairs are semantically more predictive. Hence, we first apply the algorithm to keyword pairs and keep the pairs that can uniquely cover at least 10% of functions. Then we apply the algorithm to singleton keywords on the remaining functions.

Fig. 5 shows the generated keyword cover set for the **SendSms** intent. Observe some keywords are semantically re-

[2]We collect apps from both benign and malicious sources as shown in Section 4.

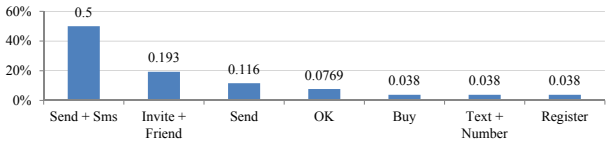


Figure 5: The keyword cover set for the SendSms intent. The y axis denotes the percentage of top level functions that can be *uniquely* covered by a keyword (pair).

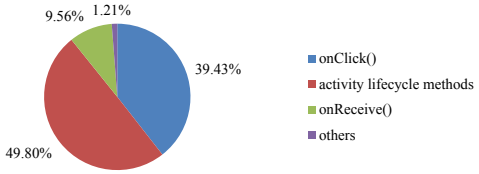


Figure 6: Breakdown of the top level functions with intents. Activity lifecycle methods include `onCreate()` and `onStart()` of an activity. `onReceive()` and the other categories do not have associated UI.

lated to the intent but some are not, e.g. "OK" and "Register", which occur rarely but do uniquely cover some functions. Further inspection shows that it is due to the malwares in the training pool. Hence, we also use human semantic analysis to prune the keyword set, e.g. filtering out "OK" and "Register". The keyword set of **HttpAccess** is similarly constructed, containing keywords "Download", "Login", "Load", "Register", and so on. The cover set of **PhoneCall** is much simpler, containing only one keyword "Call".

Once we get the keyword cover set, we further populate it with its synonyms, using Chinese WordNet [28] to have the final dictionary.

Compatibility Check. The compatibility check is performed as follows.

- Given a top level function F with UI text S and an intent T, if S is incompatible with T and all the intents correlated with T, it is considered a mismatch. Note that we consider empty text is incompatible with any intent.

- If T is a **SendSms** intent and has a correlated **SmsNotify** intent. It is not a mismatch regardless of the UI text.

- If T is **HttpAccess**, the technique checks if the corresponding UI text is compatible. If not, it further checks if T is correlated to any **UiOperation** intent. If not, the intent is consider stealthy. Intuitively, it suggests that even an HTTP access is not explicit from the GUI text, if the data acquired through the HTTP connection are used in some UI component (e.g. fetching and then displaying advertisements from a remote server), the HTTP access is not considered stealthy.

4. EVALUATION

We implement a prototype called AsDroid (*Anti-Stealth Droid*). We transform the DEX file of an app to a JAR file with dex2jar [31] and then use WALA [22] as the analysis engine. Our implementation is mainly on top of WALA.

We have collected apps from three different sources. We aim to detect those with the following stealthy behavior: SMS sends, phone calls, HTTP connections and component installations. Hence, we only focus on those having the permissions for such behaviors. Particularly, since almost all apps have the HTTP permission, we select those that have at least one of the other three permissions. Note that despite we introduce six intents in Section 3, **SmsNotify** and **UiOperation** do not describe stealthy behavior but rather suppress false alarms. The 3 sources are the following.

◇ **Contagio Mini Dump** [1]. It collects a large pool of (potential) malware reported by users and existing security tools. These malicious apps may perform stealthy operations, leak user private information, or compromise the operating system like a rootkit. We acquired 96 apps holding the needed permissions.

◇ **Google Play** [2]. This is the official apps market holding a lot of Android games. We checked the top 180 free game apps and only 12 of them satisfy our selection criteria.

◇ **Wandoujia** [4]. This is a popular general Android app market in China. We have checked the 1000 most popular game apps on the market and downloaded 74 of them with the needed permissions.

All experiments are performed on an Intel Core i7 3.4GHz machine with 12GB memory. The OS is Ubuntu 12.04.

The detection results are shown in Table 1. In the table, #App in the second column denotes the number of tested apps from a specific source. #Intent is the number of API invocations with one of the four kinds of potential stealthy intents. #Rep is the number of intent points reported by AsDroid as stealthy. #FP is the number of false positives and #FN is the number of false negatives. The corresponding #App in parentheses denotes the number of apps in which these intents appear. Note that one app may have multiple intents. The last three columns show the total numbers. #App in the last three columns is not the simple sum of the #App in the corresponding preceding columns. For example, the number of total reported apps is 77 for the `Contagio` source. It is not the sum of the reported apps in the four categories as one app may be reported in multiple categories. We make the following observations.

- AsDroid is able to detect a lot of stealthy behaviors in these apps. Totally, AsDroid detects that 113 apps perform stealthy operations, with 85 true positives, i.e. having at least one true stealthy API call. Note that there are some apps that do not have the intents (i.e. API calls) of interest even though they hold the permissions. Since there are no existing oracles to determine stealthy behavior, we identify true positives by manually inspecting the results in two ways. For those API calls that can be reached by testing, we determine their stealthiness by executing the apps. Many of the API calls are difficult to reach without a complex sequence of user actions. Since we lack automatic test generation support, we perform code inspection instead. AsDroid detects a lot of stealthy behavior in the apps from `Contagio`, which is supposed to be a source hosting (highly likely) malwares. Most of the detected stealthy SMS sends and phone calls may cause unexpected charges. Most of the stealthy HTTP accesses are to notify the remote servers the status of device or the app (e.g. a mobile device becomes online). Some of them also leak critical user information.

Table 1: Experiment Result

	#App	HTTP			SMS			CALL			INSTALL					
		#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)	#Intent (#App)	#Rep (#App)	#FP/#FN (#App)
Contagio	96	189(69)	136(64)	28/7(14/2)	90(57)	86(55)	0	4(4)	2(2)	0	4(2)	4(2)	0/7(0/6)	287(82)	228(77)	28/14(14/8)
Google Play	12	19(9)	12(7)	3/0(2/0)	6(5)	6(5)	2/0(1/0)	2(1)	0	0	0	0	0	27(10)	18(8)	5/0(3/0)
Wandoujia	74	166(39)	70(23)	23/5(10/1)	46(24)	13(10)	3/2(2/2)	8(5)	0	0	0	0	0	220(47)	83(28)	26/7(11/3)
Total	182	374(117)	218(94)	54/12(26/3)	142(86)	105(70)	5/2(3/2)	14(10)	2(2)	0	4(2)	4(2)	0/7(0/6)	534(139)	329(113)	59/21(28/11)

- AsDroid produces some false positives (28 out of the 113 reported apps). They are induced by the following reasons: (1) AsDroid cannot analyze dynamically generated text associated with a UI component; (2) The dictionary we use is incomplete; (3) Some reported intents are along infeasible paths but AsDroid does not reason about path feasibility. The detection outcome for individual apps is denoted by the symbols on top of the bars and their colors in Fig. 7. Also observe that most false positives belong to the category of HTTP accesses. Some of them are due to the incompleteness of our keyword dictionary. However most of them are essentially HTTP accesses in advertisement libraries. These accesses often download advertisement materials and store them to *external files* that are later read and displayed. Ideally, they are not stealthy as the materials are displayed. However AsDroid currently cannot reason about correlations through external resources, leading to false positives. Note that most existing static data flow analysis engines on Android have the same limitation. It should be easy to have an additional post-processing phase to suppress warnings from advertisement libraries.
- The number of false negatives is small (11 apps total). We manually inspect the apps that are not reported by AsDroid to determine false negatives. In particular, we use WALA to report all the API calls of interest and then we inspect them one by one manually. There are 182−113=69 such apps. We found that AsDroid missed 11 malicious apps. Most of them are in the category of stealthy install. As such, the detection rate of AsDroid is 85/(85+11)=88%. The main reason for false negatives is that the current implementation cannot model some of the implicit call edges. There are also cases that native libraries are used to perform stealthy behavior, which is not handled by AsDroid. The false negative HTTP accesses mainly result from the in-accuracy of the text analysis. While AsDroid extracted keywords such as "`download`" and "`login`" that make the (stealthy) HTTP accesses compatible and thus not being reported, these accesses doesn't match the textual semantics.
- Stealthy HTTP connections are very common, although many of them may not be as harmful as the other stealthy behaviors (please refer to our case study). SMS sends are another dominant category of stealthy behaviors, which echoes the recent studies [12, 34].

Comparison with FlowDroid. FlowDroid [13] is a state-of-the-art open-source static taint analysis for Android apps. We ran it on the 96 apps from `Contagio`. We use the default taint sources (e.g. methods retrieving private information). For the taint sinks, we only keep the SMS send and HTTP access methods. FlowDroid ran out of memory for 55 of the apps hence we compare the results for the remaining 41. FlowDroid reports 4 SMS sends in 3 apps and 1 HTTP access in 1 app that have information leak. In contrast, in the 41 apps, AsDroid reports 26 stealthy HTTP connections in 18 apps, including the one reported by FlowDroid, with 1 false positive in 1 app and 7 false negatives in 2 apps. It also reports 35 SMS sends in 21 apps, including 2 SMS sends reported by FlowDroid. For the other 2 SMS sends (by FlowDroid), the UIs explicitly indicate the behavior. Hence they are not stealthy although they do leak information. From the comparison, we clearly see that FlowDroid and AsDroid focus on problems with different natures.

Fig. 6 shows the breakdown of the top level functions that are attributed with intents. There are totally 743 such functions. Observe that 39% of such functions are the interactive `onClick()` function and almost 50% of them are activity lifecycle methods that are not interactive but nonetheless have associated UI. About 10% of them are `onReceive()` of external events and 1.2% of other functions such as the timer handler function `TimerTask.run()`. These functions are often not associated with any UI.

We present the analysis time for the 182 apps in Fig. 7. Most apps (about 93%) can be detected in 3 mins and a few in 13 mins. Three apps require more than 30 mins. Human inspection disclosed that that they are very complex apps such that AsDroid consumes exceptionally large amount of memory, which slows down the analysis significantly. We plan to further look into this issue.

4.1 Case Studies

Next, we present two more cases.

iCalendar is a calendar app infected by malicious code that sends a SMS message subscribing to a premium-rate service. The malicious operation is triggered by user interaction in a stealthy way. The user clicks the app to change a background image and the app increases a counter. When the counter gets to 5, a message is sent. Fig. 8 shows a simplified code snippet of the process.

Variable `main` represents the main interface layout. As soon as the app is launched, it registers a click listener in `onCreate()`. When the user clicks the interface, `showImg()` is invoked in `onClick()` to reset the background image. In the mean time, the app checks the counter to see if `sendSms()` should be called to send a premium-rate SMS.

In our analysis, two intents: **UiOperation** and **SendSms**, are associated with $\boxed{L_1}$ and $\boxed{L_2}$ in Fig. 8, respectively. The intents are propagated to the top level function `onClick()` through the call graph. The UI component associated with the function is the background image without any text, which does not imply the **SendSms** indent. The correlation analysis also determines that these two intents are not correlated. It is hence reported as a mismatch. Note that taint analysis tools [13, 10] cannot report the problem because the data involved in the SMS send are hardcoded.

HitPP is a game app downloaded from Google Play. Fig. 9 shows the code snippet in which a stealthy HTTP access is

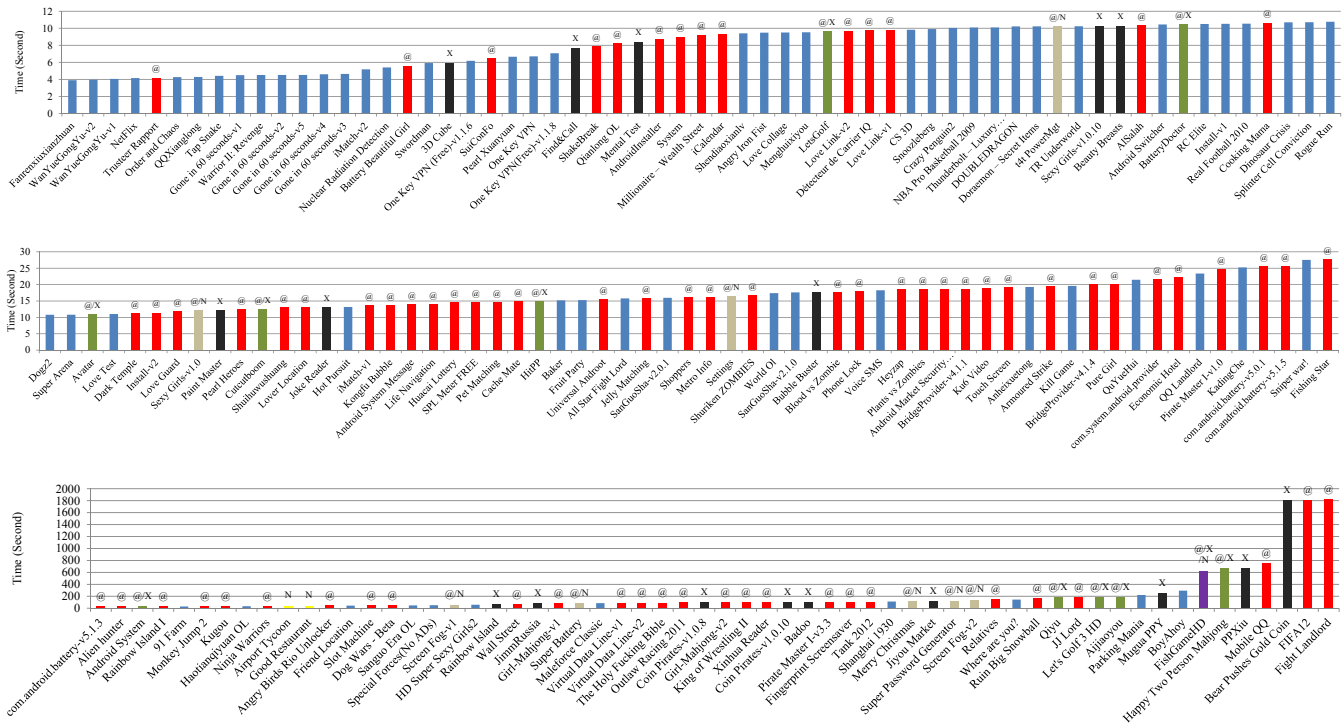


Figure 7: Analysis time. The detection results are also annotated on top of each bar with '@' denoting true positive(*red*), 'X' false positive(*black*) and 'N' false negative(*yellow*). Since an app may have multiple intents, it may be annotated with multiple labels. The last 3 apps exceeded the max timeout 30 mins.

```
// in class iCalendar
public void onCreate(Bundle bundle)
 { main.setOnClickListener(this); }
public void onClick(View view)
 { showImg(); }
private void showImg()
 { if(index == 5) sendSms();
   [L1] main.setBackgroundDrawable(drawable1); }
public void sendSms()
 { [L2] smsmanager.sendTextMessage(
            "106xxxx", null, "921X1", p, p); }
```

Figure 8: *iCalendar* example.

```
// in class HitPP extends Activity
01:  void onCreate(Bundle bundle) {
02:     // initialization ...
03:     WiGame.init(this,"f11947a...","Df6mBy...",true,true);
04:  }
// in class WiGame
05:  static void init(Context ctx, String s1, String s2, ...)  {
06:     b.a(ctx,s1);
07:  }
// in class b
08:  static void a(Context ctx, String str) {
09:     (new b.1(str,ctx)).start();//→b$1.run() at line 11
10:  }
// in class b$1 extends Thread
11:  void run() {
12:     String str="http://d.wiXXX.com/was/r?u=" +
                              WiGame.getDeviceId();
13:     HttpGet httpGet=new HttpGet(str);//HttpAccess
14:     httpClient.execute(httpGet);//without a LHS variable
15:     httpClient.getConnectionManager().shutdown();
16:  }
```

Figure 9: *HitPP* example.

made when the app is initialized. The initialization at line 3 transitively starts a thread at line 9. The thread entry is at line 11. The thread starts an HTTP connection at line 14 and then shuts it off right after at line 15. The app does not receive or display any data from the remote server. We suspect the HTTP access is to inform the remote server about the start of the app. Since there is no UI text associated with the top level `onCreate()` method and there are no correlated intents, the HTTP access is reported by AsDroid. This is a very typical kind of stealthy HTTP access reported by AsDroid.

5. LIMITATIONS

AsDroid has the following limitations. (1) The current UI analysis is simply based on textual keywords, which may be insufficient. It is possible that apps use images or obfuscated texts (e.g. text containing keyword "send" but having no relation with sending a message). AsDroid will have difficulty in catching the intention of the UI. We will study applying more advanced text analysis or image analysis. (2) Currently, to avoid false positives, AsDroid relies on certain rules in detecting intent correlation and avoids reporting some intents incompatible with UI if their correlated intents are compatible. This seems to be working fine given that Android malwares are still in their early stage. In the future, if an adversary has the prior knowledge of AsDroid, he could obfuscate a malicious app to induce bogus correlations to avoid being reported. We envision a more sophisticated program analysis component will be needed, which may leverage testing or symbolic analysis (e.g. use symbolic analysis to determine if two intents are truely correlated). (3) AsDroid currently cannot reason about correlations through external resources, leading to false positives. Note that most existing static data flow analysis engines on Android have

the same limitation. It could be mitigated by modeling external accesses. (4) Currently, AsDroid does not support native code or reflection. (5) AsDroid misses some Inter-Component Communication correlations. We could leverage Epicc [29] to get better coverage in our future work.

6. RELATED WORK

TaintDroid applies dynamic taint analysis to Android apps [10] to prevent information leak. Gilbert et al. extended the technique to track implicit flows [16]. Hornyack et al. developed AppFench to impose privacy control on Android applications [21]. Arzt et al. investigated the limitations of using runtime monitoring for securing Android apps [6]. They used unintended SMS sending as an example. The essence of the technique is information flow tracking. FlowDroid [13] is a very recent static taint analysis tool. These techniques cannot detect stealthy behavior as such operations may not leak information, as evidenced by the comparison with FlowDroid in Section 4.

Enck et al. developed a simple static analysis [11] that can detect SMS sends with hardcoded SMS numbers and phone calls, such as prefix "`tel:`" and substring "`900`". However, these patterns are very limited and not all such operations are malicious.

Elish et al. proposed to detect malicious Android apps [9] by determining the absence of data dependence path between user input/action and a sensitive function. However, dependence is not the key characteristic of stealthy behavior. In our experience, SMS sends triggered by user inputs can be malicious. Furthermore, many benign HTTP accesses are not triggered by any user action, e.g. an email app might connect to the server frequently to check new emails in background.

DroidRanger developed by Zhou et al. employs both static and dynamic techniques to detect malware [35], based on signatures derived from known malware such as premium-rate numbers and content of SMS messages. Hence, DroidRanger has to maintain a signature database that may change significantly overtime. And it also has runtime overhead.

Some existing work tries to capture Android GUI errors [33] or improve privacy control via GUI testing [23]. Gross et al. developed EXSYST [20] that uses search based testing to improve GUI testing coverage. Mirzaei et al. applied symbolic execution to generate test cases for Android apps [27]. AsDroid could potentially leverage these techniques to generate test cases for bug report validation.

Recently, Pandita et al. proposed WHYPER to analyze an app's text description and then determine if the app should be granted certain permissions [30]. Both WHYPER and AsDroid leverage text analysis. However, they have different goals and AsDroid works by analyzing both apps and UIs.

7. CONCLUSION

We propose AsDroid, a technique to detect stealthy malicious behavior in Android apps. The key idea is to identify contradiction between program behavior and user interface text. We associate intents to a set of API's of interest. We then propagate these intents through call graphs and eventually attribute them to top level functions that usually have associated UIs. By checking the compatibility between the intents and the text of the UI artifacts, we can detect stealthy operations. We test AsDroid on 182 apps that are potentially problematic by looking at their permissions. AsDroid reports 113 apps that have stealthy behaviors, with 28 false positives and 11 false negatives.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. This research is supported, in part, by National Science Foundation (NSF) under grants 0845870, 0917007, 1218993 and by the National Natural Science Foundation of China (NSFC) under grants 61170240 and 61070192, and the National Science and Technology Major Project of China under grant 2012ZX01039-004. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF or NSFC.

9. REFERENCES

[1] Contagio mobile malware mini dump. http://contagiominidump.blogspot.com/.

[2] Google play market. https://play.google.com/store/apps/.

[3] Money-stealing apps are hosting in the mobile devices. http://finance.sina.com.cn/money/lczx/20120410/0703 11783396.shtml.

[4] Wandoujia. http://www.wandoujia.com/apps/.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson Education, Inc., 2006.

[6] S. Arzt, K. Falzon, A. Follner, S. Rasthofer, E. Bodden, and V. Stolz. How useful are existing monitoring languages for securing Android apps? In *ATPS'13*.

[7] M. Becher, F. C. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *S&P'11*.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[9] K. Elish, D. D. Yao, and B. G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *MoST'12*.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *USENIX Security'11*.

[12] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *SPSM'11*.

[13] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android applications. Technical report, TU Darmstadt, 2013.

[14] Gartner. Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent. http://www.gartner.com/it/page.jsp?id=2237315.

[15] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST'12*.

[16] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *MCS'11*.

[17] Google. Android 4.2 compatibility definition. http://source.android.com/compatibility/4.2/android-4.2-cdd.pdf.

[18] Google. Android developer guide. http://developer.android.com/guide/.

[19] P. Gosling. Trojan: Trojans & spyware: an electronic achilles. *Netw. Secur.*, 2005(3):17–18, Mar. 2005.

[20] F. Gross, G. Fraser, and A. Zeller. EXSYST: search-based GUI testing. In *ICSE'12*.

[21] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications. In *CCS'11*.

[22] IBM T.J. Watson Research Center. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[23] A. Jääskeläinen. *Design, Implementation and Use of a Test Model Library for GUI Testing of Smartphone Applications*. Doctoral dissertation, Tampere University of Technology, Tampere, Finland, Jan. 2011.

[24] Juniper Networks. Juniper mobile security report 2011 - unprecedented mobile threat growth. http://forums.juniper.net/t5/Security-Mobility-Now/Juniper-Mobile-Security-Report-2011-Unprecedented-Mobile-Threat/ba-p/129529.

[25] R. Levy and C. D. Manning. Is it harder to parse Chinese, or the Chinese Treebank? In *ACL'03*.

[26] D. Maslennikov. IT threat evolution: Q1 2013. http://www.securelist.com/en/analysis/204792292/.

[27] N. Mirzaei, S. Malek, and R. M. Corina S. Păsăreanu, Naeem Esfahani. Testing Android apps through symbolic execution. In *JPF'12*.

[28] National Taiwan University. Chinese wordnet. http://lope.linguistics.ntu.edu.tw/cwm/.

[29] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysi. In *USENIX Security'13*.

[30] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security'13*.

[31] pxb1988. dex2jar: Tools to work with android .dex and java .class files. http://code.google.com/p/dex2jar/.

[32] TrendLabs. 3Q 2012 security roundup - Android under siege: Popularity comes at a price. http://www.trendmicro.com/us/security-intelligence/.

[33] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ISSTA'12*.

[34] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *S&P'12*.

[35] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *NDSS'12*.