Android Implicit Information Flow Demystified

Wei You, Bin Liang*, Jingzhe Li, Wenchang Shi
Renmin University of China, Beijing, P. R. China
{youwei, liangb, lijingzhe, wenchang}@ruc.edu.cn

Xiangyu Zhang
Purdue University, Indiana, USA
xyzhang@cs.purdue.edu

ABSTRACT

In this paper, a comprehensive analysis of implicit information flow (IIF) on the Android bytecode is presented to identify all potential IIF forms, determine their exploitability, and mitigate the potential threat. By applying control-transfer-oriented semantic analysis of the bytecode language, we identify five IIF forms, some of which are not studied by existing IIF literature. We develop proof-of-concepts (PoCs) for each IIF form to demonstrate their exploitability. The experimental results show that all these PoCs can effectively and efficiently transmit sensitive data, as well as successfully evade the detection of a state-of-the-art privacy monitor TaintDroid. To mitigate the threat of IIF, we propose a solution to defending against IIF leveraging a special control dependence tracking technique and implement a prototype system. The evaluation shows that the prototype can effectively detect information leak by all the identified IIF forms and also real-world malware with an acceptable overhead. In summary, our study gives in-depth insight into Android IIF from both offensive and defensive perspectives, and provides a foundation for further research on Android IIF.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection

Keywords

implicit information flow; exploitation; mitigation; Android

1. INTRODUCTION

Information flow analysis (IFA) is an important technique, which aims to track the propagation of information between program variables. Information flow occurs from a source object x to a target object y, if the information stored in object x can be revealed by the value of object y. Information flow can be explicit or implicit [4]. Explicit information flow (EIF) results from *data dependence*; and implicit information flow (IIF) results from *control dependence*.

Intuitively, a program with the pattern $low = high$; exhibits privacy leakage via EIF. Confidential data stored in variable *high* is directly passed to variable *low* whose value can be observed from outside. On the other hand, the following code snippet contains IIF from *high* to *low*. Although there is no direct data flow between these two variables, one can still learn something about *high* by observing *low*.

```
1  if (high == 0) low = 0;
2  else if (high > 0) low = 1;
3  else low = -1;
```

In general, IIF is more difficult to be tracked than EIF. Most existing IFA techniques [10, 12] concentrate mainly on EIF, ignoring the impact of IIF. As a result, it is possible for malware to leverage IIF to evade detection. A recent anti-IFA study [2] has demonstrated the feasibility of evasion for C programs. To mitigate the threat of IIF, some approaches [1, 9] are proposed to selectively track control dependence. For example, the study in [1] tracks only a special type of control dependence whose nature highly resembles data dependence. However, the existing studies only consider IIF resulted from regular conditional structures. Little attention is paid to IIF resulted from other language structures.

Information leak is the most commonly seen malicious behavior on Android [14]. Most existing malware samples steal sensitive information through EIF such that the existing defense schemes on Android focus on tracking only EIF [6, 13]. However, as we found in the wild, sophisticated malware starts to leverage IIF to evade the existing defense systems. We envision such attacks will become more popular in near future due to the difficulty of defending against IIF. Hence, there is a pressing need to study IIF on Android.

To better understand the possible attack and the corresponding defense on Android, there are three open research questions to be answered: 1) How many IIF forms there may exist; 2) Given an IIF form, is it exploitable, allowing to effectively transmit sensitive information; 3) How to defend against information leak through such IIF forms with an acceptable overhead. In this paper, we answer these questions by theoretical analysis as well as empirical experiments.

Our study begins with a control-transfer-oriented analysis of Android's Dalvik bytecode in a formal structured operational semantic model. With this model, 54 instructions are identified that may cause control dependence. IIF can be constructed by using any of these instructions. Identifying all these IIF inducing instructions is meaningful for conducting a sophisticated IIF attack or performing a com-

*Corresponding Author

ASIA CCS'15, April 14–17, 2015, Singapore..

http://dx.doi.org/10.1145/2714576.2714604.

prehensive IIF detection. We find that besides conditional structures (i.e., *if* and *switch*), there are other instructions (i.e., *throw*, *exception-prone* instructions and *polymorphism* related instructions) that can also be used to construct IIF attacks. Details are shown in Section 2.

We develop proof-of-concepts (PoCs) for the identified IIF forms and examine their exploitability. Without loss of generality, we respectively integrate these PoCs to a real-world application *FMajor*, which leaks phone number via EIF. We modify the original apk, inserting an additional propagation step leveraging IIF forms to transmit data. We evaluate the PoCs on an emulator equipped with TaintDroid [6], a state-of-the-art IFA-based privacy monitor for Android. We find that all PoCs can effectively steal phone number and successfully evade the detection of TaintDroid. In other words, these PoCs can be easily turned into real attacks. Indeed, by manually analyzing the samples provided by the Android Malware Genome Project [14], we found two samples leveraging IIF in a way similar to some of our PoCs. We believe more real malware samples similar to our (other) PoCs will be found in the future. Details are shown in Section 3.

Compared with exploiting IIF, it is more difficult to defend against it. In general, it is intractable to perform sound and complete tracking of IIF. In this paper, we reduce the problem to tracking a special kind of control dependence called *strict control dependence* (SCD). Observe that practical IIF attacks most likely have to leverage SCD, we hence enhance an existing runtime SCD tracking algorithm [1] to detect IIF. The algorithm selectively taints a predicate if it has *strong* correlation with sensitive information. All assignments guarded by such a tainted predicate are tainted. For better efficiency, we improve the SCD detection algorithm by adopting a lazy tainting policy, which postpones the tainting of control dependence to the post-dominator of a control structure. More importantly, our technique formulates the various Dalvik instructions vulnerable to IIF exploits to SCD, such that our technique can detect all possible IIF forms on Android. We implement a prototype system. The evaluation results show that the prototype can successfully detect all the PoCs and the real-world malware samples with acceptable overhead. Details are shown in Section 4.

2. SEMANTICS OF DALVIK BYTECODE

IIF is caused by control dependence, which is determined by control transfer relationship between instructions. Hence in this section, we study the semantics of Android's Dalvik bytecode (DVML) to identify all the control transfer instructions that can be exploited to construct IIF. The goal is to assist in conducting a sophisticated IIF attack or performing a comprehensive IIF detection. To get a concise and precise semantic specification, we abstract an imperative language DVML$_I$ from DVML and present it with a structured operational semantic (SOS) model. This model focuses on control transfer behavior of DVML$_I$ instructions.

Note that in order to exploit IIF, the attacker should be able to infer a value from the control transfer determined by the value. Hence, if an instruction always transfers control to a definite target address, or in other words, the target is not determined by any runtime value, there is no IIF. We call the control flow instructions whose targets are determined by runtime values as *indirect control transfer* (ICT) instructions. ICT instructions are the building blocks of various forms of IIF, and hence the focus of our discussion.

2.1 The SOS Model

The SOS model focuses on the control-transfer related semantics of DVML$_I$ and leaves out other unnecessary details. A DVML$_I$ program consists of classes. Each class contains a set of fields and a set of methods. Without loss of generality, we do not distinguish static fields (or methods) from instance fields (or methods). Besides, we assume that the first instruction of a method locates at offset 0 and the offset difference between two consecutive instructions is 1.

2.1.1 Memory Model and Execution State

A DVML$_I$ program has a local memory space *RegisterArray* that stores local variables of the current method, and a global memory space *Heap* that stores dynamically allocated objects. A configuration $\langle cm, off, H, V\rangle$ describes execution state with cm the current method, off the offset of the current instruction inside the method, H the heap and V the register array. Note that the current program counter can be denoted as $pc = \langle cm, off\rangle$.

2.1.2 Instruction Set and Operational Semantics

The DVML$_I$ instruction set is abstracted from the DMVL instruction set by grouping the DMVL instructions with similar semantics into a single DVML$_I$ instruction. The operational semantic rule of DVML$_I$ is of the form:

$$\frac{instruction \wedge computation}{\langle \text{current configuration}\rangle \Rightarrow \langle \text{next configuration}\rangle}$$

Rules are read from top to bottom, left to right. Given an instruction, we search for an applicable rule that matches the instruction, and apply the computation given in the top of the rule. The execution state of the program will transform from the current configuration to the next configuration.

For example, instruction *invoke* m v_{this} $(v_i)_1^n$ has a rule:

$$\frac{\begin{array}{l} I{=}\text{invoke } m\ v_{this}\ (v_i)_1^n \wedge loc{=}V[v_{this}] \wedge o{=}H(loc) \wedge \\ m'{=}\text{methodDispatch}(m,o) \wedge \text{pushState}(\langle cm,off,V\rangle) \wedge \\ V'{=}loc{::}(V[v_i])_1^n \end{array}}{\langle cm,off,H,V\rangle \Longrightarrow \langle m',0,H,V'\rangle} \quad \textsc{Invoke}$$

The instruction is used to conduct a polymorphic invocation. When executing the instruction, dynamic method dispatching is performed to determine which polymorphic version m' of the method should be invoked. The method dispatching is based on the dynamic type of the receiver object, which can be accessed via the object reference stored in register v_{this}. The current program state is pushed onto an internal execution stack for restoring at return site. The instruction execution results in a new program state where the register array stores the receiver object reference concatenated with the parameters (i.e., $loc :: (V\,[v_i])_1^n$) and the program counter is set to the start of the invoked method (i.e., $pc{=}\langle m', 0\rangle$).

An instruction may trigger a runtime exception, if runtime values of its operands violate related constraints. Such instruction is called *exception-prone* instruction. It has two execution rules: one for the normal execution, the other for the exceptional execution. When executing an *exception-prone* instruction, checks are conducted on its operands to determine which rule is applied. For example, instruction *binop op* v_A v_B v_C is an *exception-prone* instruction. At runtime, a check is conducted to examine whether the operator *op* is *division* and the divisor v_C is 0. If so, an exception is thrown and the control is transferred to the corresponding exception handler. Otherwise, the binary operation is performed and the control is transferred to the next instruction.

2.2 ICT Instructions

For identifying indirect control transfer instructions, we examine all DVML$_I$ instructions. The *exception-prone* instructions have two execution rules. Different rules transfer the control to different target addresses, and the rule applied is determined by runtime checks. Therefore, all *exception-prone* instructions can cause indirect control transfer. For an instruction with a single execution rule, if the program counter of the next configuration is dynamically dependent on some operands, it can cause indirect control transfer.

By examining all the DVML$_I$ instructions, we identify 12 ICT instructions. These ICT instructions correspond to 54 DMVL instructions, and can be divided into five categories according to their related control structures, as shown in Table 1. We can see that besides the common conditional structures (i.e., *if* and *switch*), there are three other control structures (i.e., *throw*, *exception-prone* instructions and *polymorphism* related instructions) can also cause IIF.

2.3 IIF Forms

In practice, given an ICT instruction, there are various ways to leverage it to construct IIF. In this paper, we focus on IIF in which the attacker can uniquely determine a high-confidentiality value from the observable low-confidentiality value. In other words, we focus on IIF that induces a one to one (1-2-1) mapping between the two values. Specifically, we consider IIF in which sensitive data is used as the operand that distinctively determines the target address of the instruction, i.e., different values of operand v_H will lead to different control transfer targets. Besides, there is an observable variable v_L that will be assigned with a distinctive value in each target. Consequently, adversaries can directly deduce the value of v_H by examining the value of v_L.

IIF instances can be categorized into five forms, each based on one of the identified control structures. Leveraging the *switch* structure is straightforward. The attacker can use a branch for each possible value of v_H and assign v_L with a unique value in each branch. For the *if* structure, it has only two branches thus can distinguish only two possible values. The attacker can use multiple *if* structures in nesting, or use a single *if* structure multiple times in looping.

Other control structures are analogous to the *if* structure or the *switch* structure. For example, the *exception-prone* structure can be treated as a special *if* structure, which tests whether a runtime exception is triggered. Another example is the *polymorphism* structure. It can be modeled as a special *switch* structure, which tests the runtime type of a receiver object to determine the most appropriate version of a polymorphic method to be invoked.

3. EXPLOITABILITY

We develop proof-of-concepts (PoCs) for the identified IIF forms and examine their exploitability. Our PoCs exploit IIF to precisely transmit confidential data from a secret variable *high* to a public variable *low*, so that we can unambiguously infer *high* from *low*. In other words, our PoCs exploit IIF to generate a 1-2-1 mapping from *high* to *low*.

3.1 PoCs

Without loss of generality, each of our PoCs takes a sensitive 8-bit digital character as input, and leverages different IIF forms to transmit it to the publicly accessible output.

Table 1: Identified ICT Instructions.

DVML$_I$	DVML Instruction(s)	Structure
if	if-eq, if-ne, if-lt, if-le, if-gt, if-ge, if-eqz, if-nez, if-ltz, if-lez, if-gtz, if-gez	if
switch	packed-switch, sparse-switch	switch
invoke	invoke-virtual, invoke-virtual/range invoke-interface, invoke-interface/range	polymorphism
throw	throw	throw
binop	div-int, div-long, rem-int, rem-long	exception-prone
check_cast	check_cast	exception-prone
fget	iget, iget-wide, iget-object, iget-byte, iget-short, iget-char, iget-boolean	exception-prone
fput	iput, iput-wide, iput-object, iput-byte, iput-short, iput-char, iput-boolean	exception-prone
new_array	new_array	exception-prone
array_length	array_length	exception-prone
aget	aget, aget-wide, aget-object, aget-byte, aget-short, aget-char, aget-boolean	exception-prone
aput	aput, aput-wide, aput-object, aput-byte, aput-short, aput-char, aput-boolean	exception-prone

3.1.1 If-Based IIF

The *if* statement is a common control structure for constructing IIF. A single *if* statement is not sufficient to transmit 8-bit information. In PoC 1, multiple nested *if* statements are used to enumerate the possible values of *high*. In PoC 2, a loop is used to traverse the range of the value of *high*. After execution, *low* will hold the same value as *high*.

```
1  if (high == '0') low = '0';
2  else if (high == '1') low = '1';          PoC 1
3  ......
```

```
1  for (low = '0'; low != high; low++);      PoC 2
```

3.1.2 Switch-Based IIF

The *switch* statement is another common structure for constructing IIF. PoC 3 uses sufficient amount of *case* branches to explicitly enumerate the possible values of *high*.

```
1  switch (high) {
2    case '0': low = '0'; break;
3    case '1': low = '1'; break;             PoC 3
4    ......
5  }
```

3.1.3 Exception-Prone-Based IIF

An *exception-prone* statement is analogous to an *if* statement that tests whether a runtime exception is triggered. PoC 4 uses a loop to iterate on variable *low* over the value range of *high*. In each iteration, a *division* operation is performed with the difference between *high* and *low* as divisor. An arithmetic exception will be thrown when *high* == *low*.

```
1  for (low = '0'; low <= '9'; low++){
2    try {int tmp = 1 / (high - low);}
3    catch (Exception e) {break;}           PoC 4
4  }
```

3.1.4 Throw-Based IIF

A *throw* statement can throw exception of different types, which can be caught by different handlers. It is essentially equivalent to a *switch* statement that tests the type of the exception variable. In PoC 5, the array *excepts* has different elements of different types. The value of *high* decides which array element is assigned to the exception variable *except*. Different types of *except* lead to different handlers to be taken, which in turn lead to different values assigned to *low*. As such, we get a 1-2-1 mapping from *high* to *low*.

```
1  Exception except = excepts[high - '0'];
2  try { throw except; }{
3  catch (Exception_0 e) {low = '0';}          PoC 5
4  catch (Exception_1 e) {low = '1';}
5  ......
```

3.1.5 Polymorphism-Based IIF

Polymorphism is an essential feature for Object Oriented languages. In Android programs, polymorphism includes four manners: virtual invocation, reflective invocation, message dispatching, and event dispatching.

PoC 6 demonstrates the IIF via virtual invocation. Method $f()$ has different versions that return different values to *low*. The value of *high* decides the runtime type of the receiver object *poly*, which in turn decides which version of $f()$ to be executed. As such, we get a 1-2-1 mapping from *high* to *low*.

```
1  class Poly_0 extends Poly {
2    char f() {return '0';}{
3  }
4  ......                                      PoC 6
5  Poly poly = polys[high - '0'];
6  low = poly.f();
```

PoC 7 demonstrates the IIF via reflective invocation. The name of the method to be invoked is determined by the value of *high*. Different methods return different values to *low*. As such, we get a 1-2-1 mapping from *high* to *low*.

```
1  class Reflect {
2    static char method_0() {return '0';}
3    ......
4  }                                           PoC 7
5  String name = "method_" + high;
6  Method m = Reflect.class.getMethod(name);
7  low = m.invoke(null);
```

PoC 8 demonstrates the IIF via message dispatching. Different values of *high* cause different message handlers to be executed, which assign different values to *low*. As such, we get a 1-2-1 mapping from *high* to *low*.

```
1  class Handler_0 extends Handler {
2    void handleMessage(Message msg) {low = '0';}
3  }
4  ......                                      PoC 8
5  Handler handler = handlers[high - '0'];
6  Message message = handler.obtainMessage();
7  handler.sendMessage(message);
```

PoC 9 demonstrates the IIF via event dispatching. Multiple *Button* components are attached with different listeners, which assign different values to *low*. The value of *high* decides which *Button* to obtain the focus. The event of focus changing will be dispatched to the listener of the focused *Button*. As such, we get a 1-2-1 mapping from *high* to *low*.

```
1  class Listener_0 extends Listener {
2    void onFocusChange() {low = '0';}
3  }                                           PoC 9
4  ......
5  Button button = buttons[high - '0'];
6  button.requestFocus();
```

3.2 Evaluation of PoCs

To evaluate these PoCs, we respectively integrate them to a real-world application *FMajor*, which leaks phone number via EIF. We modify the original apk, inserting an additional propagation step leveraging IIF to transmit data. The evaluation is performed on an emulator equipped with TaintDroid [6] and run on a computer with Intel(R) Core(TM) 2.50 GHz CPU and 2G memory. All PoCs can effectively leak phone number and successfully evade the detection of TaintDroid.

Table 2: Performance of PoCs.

PoC	Δ Size	Δ Mem	t(PN)	t(1M)
if-based (nesting)	0.4 KB (0.06%↑)	11.3 KB (0.18%↑)	42 ms	294 ms
if-based (looping)	0.3 KB (0.05%↑)	8.2 KB (0.13%↑)	45 ms	303 ms
switch-based	0.4 KB (0.06%↑)	10.2 KB (0.16%↑)	37 ms	283 ms
exception-prone-based	0.4 KB (0.06%↑)	28.7 KB (0.46%↑)	51 ms	391 ms
throw-based	0.9 KB (0.14%↑)	55.3 KB (0.89%↑)	49 ms	314 ms
polymorphism-based (virtual invocation)	1.1 KB (0.17%↑)	69.6 KB (1.12%↑)	47 ms	310 ms
polymorphism-based (reflective invocation)	0.7 KB (0.11%↑)	78.8 KB (1.27%↑)	49 ms	596 ms
polymorphism-based (message dispatching)	1.4 KB (0.21%↑)	84.0 KB (1.35%↑)	57 ms	844 ms
polymorphism-based (event dispatching)	1.9 KB (0.29%↑)	109.6 KB (1.76%↑)	96 ms	1126 ms

We evaluate the performance of each PoC from four aspects: size increment, memory increment, time consumption in transmitting a small amount of data (i.e., phone number) and a large amount of data (i.e., 1M randomly-generated data). The evaluation result is shown in Table 2. The PoCs incur less than 0.3% size increment, and less than 2% memory increment. All PoCs can efficiently transmit data. The time consumption in transmitting phone number is less than 0.1 second. For transmitting 1M data, even the least efficient PoC can finish in about 1 second.

3.3 Real-World Threat

We want to note that IIF is not only a theoretical threat. It has been used in the real world either inadvertently or by design. Indeed, we found two samples (i.e., *DroidKungFu3* and *AnserverBot*) from the Android Malware Genome Project [14] leveraging IIF in a way similar to some of our PoCs. Although we have not found other IIF forms in malware samples, we believe they are likely to be leveraged by malware in the future due to their effectiveness.

3.3.1 IIF of DroidKungFu3

DroidKungFu3 will send certain bits of the victim's integrated circuit card identifier (i.e., ICCID) to a remote server. Instead of directly sending ICCID, it adopts IIF to encode each bit. The following shows the highly simplified IIF-related code of *DroidKungFu3*. We can see that it leverages the *if*-based IIF in a way similar to PoC 1.

```
1  if (high.equals("0")) low = 0;
2  else if (high.equals("1")) low = 1;
3  ......
```

3.3.2 IIF of AnserverBot

AnserverBot will send the victim's international mobile equipment identity (IMEI) string to a remoter server. Instead of directly sending IMEI, *AnserverBot* adopts IIF to encode each character as the index of its occurrence in a character map *value*. The encoding is implemented by invoking method *String.indexOf()*, whose highly simplified code is shown in the following. We can see that it leverages *if*-based IIF in a similar way to PoC 2.

```
1  for (low = 0; low < length; low++)
2    if (value[low] == high) break;
```

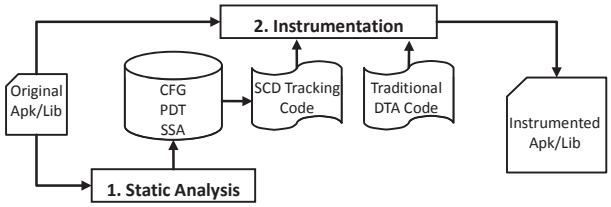

Figure 1: Overview of our IIF mitigation solution.

4. MITIGATION

From the above discussion, we can see that IIF poses serious threat to data confidentiality. In theory, it is intractable to perform sound and complete IIF tracking. In this paper, we propose a solution concentrating on tracking 1-2-1 IIF, since such IIF can precisely transmit sensitive data and hence is more likely to be exploited by real-world malware.

A key observation is that 1-2-1 IIF can be formulated as a special kind of control dependence called *strict control dependence* (SCD) [1]. A statement s is strictly control dependent on a predicate statement with v_p as predicate variable, if the execution of s can precisely infer the value of v_p. The branch leading to the execution of s is called the SCD branch. Consider the code snippet in Section 1. There is an SCD between the assignment statement at line 1 and the *if* statement at the same line. If the *true* branch (i.e., SCD branch) of the *if* statement is taken, the attacker can precisely infer that the value of the predicate variable *high* is 0. As a counter example, the control dependence between the assignment statement at line 2 and the *if* statement at the same line is not an SCD. Because from the execution of this assignment statement, we can only infer that the predicate variable *high* is larger than 0. Little information is revealed.

4.1 Overview

The overview of our solution is shown in Figure 1. Given an application or a library, we first perform static analysis on its bytecode to compute control flow graph (CFG), post dominator tree (PDT), and static single assignment (SSA). The computed information is used in generating SCD tracking code. Then we instrument the application or library with both traditional DTA tracking code and SCD tracking code. The static analysis and instrumentation are performed on a desktop computer. The instrumented application or library will be installed and executed on the smartphone device.

4.2 SCD-Based Tracking

The basic idea of SCD-based tracking is to propagate the taint of the predicate variable of a control structure to the variables that are assigned in an SCD branch. To this end, we first identify the SCD branches of each control structure. Then we adopt a lazy tainting policy that is equally effective as the original SCD tracking algorithm [1] but more efficient.

4.2.1 Identifying SCD Branches

We perform static analysis to identify SCD branches. Currently, the analysis only focuses on SCDs caused by equivalence testing. For an *if* structure, if it is an equivalence predicate, the *true* branch is an SCD branch; if it is a non-equivalence predicate, the *false* branch is an SCD branch. For a *switch* structure, if a branch can be reached from only one *case* value, it is an SCD branch.

Rule	Event	Instrument
T1	Encoutering a predicate statement: x is the predicate variable.	$t = \tau(x)$
T2	Encoutering a post-dominator: variable x is assigned in an SCD branch i; $\nexists j \neq i$: x is assigned in branch j and $\|x_i\| = \|x_j\|$.	$\tau(x) = \tau(x) \mid t$
T3	Encoutering a statement s: s is contained in an SCD branch; s may immediately propagate the value outward to a global variable x.	$\tau(x) = \tau(x) \mid t$

Figure 2: Rules for lazy SCD-based tainting. Here, $\tau(x)$ is the taint of variable x, and $\|x_i\|$ denotes the value assigned to variable x at branch i.

Our solution also features the capability of handling other IIF inducing structures. It explicitly converts them to *if* structures or *switch* structures, depending on the number of their branches. For example, an *exception-prone* instruction is converted to an explicit *if* structure that tests whether the specific condition triggering a runtime exception is satisfied. Another example is the *polymorphism* structure, which is converted to an explicit *switch* statement that tests the dynamic type of the receiver object. The explicit *if* or *switch* statements are then instrumented to track SCD at runtime.

4.2.2 Lazy Tainting Policy

For better performance, we propose lazy tainting policy, instead of the original on-the-fly SCD-based tainting. The rules for lazy SCD-based tainting are shown in Figure 2. When encountering a predicate statement, the taint of the predicate variable is stored in a temporary variable (Rule T1). At each immediate post-dominator, the algorithm examines each variable that is assigned in an SCD branch to check whether its value is distinctive from other branches. If so, it will propagate the taint of the current control structure to the identified variable (Rule T2).There is a special consideration for the lazy tainting policy. An SCD branch may contain some assignment statements whose values may immediately escape the branch (e.g. assignments to global variables). When encountering these statements in an SCD branch, we propagate taints immediately (Rule T3).

4.3 Evaluation of Mitigation Solution

We implement a prototype system and deploy it in HTC T528w smartphone equipped with Android-4.0. The prototype tracks both data dependence and strict control dependence. It can successfully detect all the aforementioned 1-2-1 IIF exploit PoCs, as well as real-world malware samples.

We evaluate the performance of our prototype with the *FMajor* application, the *DroidKungFu3* and *AnserverBot* malware samples, and some popular applications collected from serveral markets. We compare the code size before and after instrumentation, and find that the instrumented versions are averagely 40% larger than the original ones. We also compare the execution times of the native runs (without tracking), the runs with DD-only tracking, and the runs with DD+SCD tracking. The DD+SCD runtime overhead is approximately 65% on average compared with the native runs. Comparing with the DD-only tracking, the DD+SCD tracking increases the overhead by 30% at most. As a comparison, Dytan [3], which blindly tracks IIF, brings approximate 50x performance overhead. We argue that the performance overhead of our prototype system is acceptable.

5. RELATED WORKS

Information flow analysis (IFA) has been studied for a long time. It can be static or dynamic. Static IFA approaches [7, 8] infers the dependencies between variables via data flow analysis and control flow analysis. However, the existing techniques do not have sufficient supports for tracking information flow caused by object oriented features, such as polymorphism. On the other hand, dynamic IFA approaches [10, 12], also known as dynamic taint analysis (DTA), provides the ability to track information flow at runtime. However, most DTA frameworks can only handle explicit information flow (EIF), resulting in under-tainting [11].

Recent studies trend to combine dynamic taint analysis with static analysis to track implicit information flow (IIF). Dytan [3] statically identifies control dependence structures, and propagates the taint of predicate variables to variables assigned in each branch without discrimination. As an improvement, DTA++ [9] only tracks IIF within information-preserving transformations, such as data conversion from one format to another. Similarly, the study in [1] only tracks strict control dependence (SCD), a special kind of control dependence whose nature highly resembles data dependence. Although neither sound nor complete, these studies provide a cost-effective approach to tracking IIF. Our mitigation solution is based on the concept of SCD. We improve the original SCD detection algorithm by adopting a lazy tainting policy to improve efficiency. In addition, our solution can handle all IIF inducing control structures on Android, with many not covered by the existing IIF tracking techniques.

Recent years, with the widespread usage of smartphones, some studies are concentrated on information flow tracking of mobile apps. PiOS [5] conducts static information flow analysis on iOS apps to detect privacy leakage. TaintDroid [6] and DroidScope [13] conduct dynamic taint analysis on Android apps with the same goal. Unfortunately, all of them only consider EIF, ignoring the impact of IIF. However, our experiments show that it is feasible to leverage IIF to effectively and efficiently transmit sensitive data and evade the existing IFA-based detection. Moreover, we find that real-world smartphone malware starts to leverage IIF. It indicates that IIF is not only a theoretical threat but a reality. To the best of our knowledge, we are the first to propose and implement practical mitigation solution to defending against IIF on smartphone.

6. CONCLUSION

This paper presents a comprehensive study of Android implicit information flow (IIF). We propose a systematical method to identify potential IIF forms. With the method, we find that in addition to the commonly studied conditional structures, there are other control structures on Android that can also be used to construct IIF. We develop proof-of-concepts (PoCs) for each identified IIF form and examine their exploitability. The experimental result shows that these PoCs can effectively and efficiently transmit sensitive data and evade the detection of a state-of-the-art information flow tracking system TaintDroid. Moreover, we find two real-world malware samples leveraging IIF in a way similar to some of our PoCs. We believe all of our identified IIF forms are likely to be leveraged by malware in the future due to their effectiveness. In theory, it is intractable to perform sound and complete IIF tracking. We propose a practical mitigation solution concentrating on a subclass of IIF that can precisely transmit sensitive data hence poses the most serious threat to data confidentiality. Based on the solution, we develop a prototype system that tracks both data dependence and strict control dependence. The evaluation of our prototype shows that it can successfully detect all PoCs and real-world malware samples with an acceptable overhead.

7. ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. The work is supported by National Natural Science Foundation of China under grants 61170240, 91418206, and 61472429, and Beijing Natural Science Foundation under grant 4122041.

8. REFERENCES

[1] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of ISSTA*, 2010.

[2] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of DIMVA*, 2008.

[3] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of ISSTA*, 2007.

[4] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in ios applications. In *Proceedings of NDSS*, 2011.

[6] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of OSDI*, 2010.

[7] S. Genaim, R. Giacobazzi, and I. Mastroeni. Modeling secure information flow with boolean functions. In *Proceedings of ITS*, 2004.

[8] S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *Proceedings of VMCAI*, 2005.

[9] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of NDSS*, 2011.

[10] L. Lam and T. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of ACSAC*, 2006.

[11] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of S&P*, 2010.

[12] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*, 2006.

[13] L. Yan and H. Yin. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the USENIX Security Symposium*, 2012.

[14] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of S&P*, 2012.