Detecting control flow in Smarphones: Combining static and dynamic analyses

Mariem Graa, Nora Cuppens-Bouhlahia, Frédéric Cuppens, Ana Cavalli

Detecting control flow in Smarphones: Combining static and dynamic analyses

Mariem Graa[1,2], Nora Cuppens-Boulahia[1], Frédéric Cuppens[1], Ana Cavalli[2]

[1]Telecom-Bretagne, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
{mariem.benabdallah,nora.cuppens,frederic.cuppens}@telecom-bretagne.eu
[2]Telecom-SudParis, 9 Rue Charles Fourier, 91000 Evry - France
{mariem.graa,ana.cavalli}@it-sudparis.eu

Abstract. Security in embedded systems such as smartphones requires protection of confidential data and applications. Many of security mechanisms use dynamic taint analysis techniques for tracking information flow in software. But these techniques cannot detect control flows that use conditionals to implicitly transfer information from objects to other objects. In particular, malicious applications can bypass Android system and get privacy sensitive information through control flows. We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies by using the static analysis in embedded system such as Google Android operating system. By using this new approach, it becomes possible to protect sensitive information and detect most types of software exploits without reporting too many false positives.

1 Introduction

Today security is a requirement for an increasing number of embedded systems, such as smartphones. These systems are usually used to connect to the internet and download third-party applications. Apple recently announced that more than three billion apps have been downloaded from its groundbreaking App Store by iPhone and iPod touch users worldwide [2]. These downloaded applications can access and manipulate privacy data. An attacker can exploit a malicious application and launch overwrite attacks (such as worms, injection attacks and flow control attacks) to compromise the confidentiality and integrity of the Android system. Many mechanisms are used to protect the Android system against attacks, such as the dynamic taint analysis that is implemented in TaintDroid [11]. The principle of dynamic taint analysis is to "taint" some of the data in a system and then propagate the taint to data for tracking the information flow in the program. Two types of flows are defined: explicit flows such as $x = y$, where we observe an explicit transfer of a value from x to y, and implicit flows (control flows) shown in Figure 1 were there is no direct transfer of value from a to b, but when the code is executed, b would obtain the value of a. we will interest only on direct and implicit flows; we do not consider covert channels such as timing, power channels, probabilistic channels, etc.

```
1.boolean b = false;
2.boolean c = false;
3.if (!a)
4. c = true;
5.if (!c)
6.  b = true;
```

Fig. 1. Implicit flow example.

The dynamic taint analysis mechanism is used primarily for vulnerability detection and protection of sensitive data. To detect the exploitation of vulnerabilities, the sensitive transactions must be monitored to ensure that they are not tainted by outside data. But this technique does not detect control flows which can cause an under-tainting problem *i.e.* that some values should be marked as tainted, but are not. Consider the code in Figure 1 that presents an under tainting problem. When $a = false$ and a is tainted, the first branch is executed but the second is not, thus b is not tainted while b depends on a (b depends on c that depends on a). This can cause a failure to detect a leak of sensitive information. Thus, malicious applications can bypass the Android system and get privacy sensitive information through control flows. We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies to track implicit flows in embedded systems such as the Google Android operating system. In this paper, guiding dynamic taint analysis by static analysis, we show how to solve the under-tainting problem and detect most types of software exploits without reporting too many false positives. This paper is organized as follows: section 2 presents a motivating example. Related work about static and dynamic taint analysis is discussed in section 3 and we analyze existing solutions to solve the under tainting problem. Section 4 describes our formal specification of under tainting. Section 5 presents our solution based on a hybrid approach that improves the functionality of TaintDroid by integrating the concepts introduced by Trishul (an information flow control system that correctly handle implicit flows). Finally, section 6 concludes with an outline of future work.

2 Motivating example

An attacker can exploit an indirect control dependency to exploit a vulnerability. For example, consider the attack shown in Figure 2.

The variables a and b are both initialized to false. On Line 4, the attacker tests the user's input for a specific value. Let us assume that the attacker was lucky and the test was positive. In this case, Line 5 is executed, setting a to true and a is tainted. Variable b keeps its false value, since the assignment on Line 7 is not executed and b is not tainted because dynamic tainting occurs only along the branch that is actually executed. Since b has not been modified, the condition on b (Line 10) evaluates to true. As b is not tainted, no tainted scope is generated for this branch, and the attacker is free to enter malicious code at

```
1.a= false;
2.b=false;
3.char c[256];
4.if( gets(c) == "aaa" )
5.     a=true;
6.else
7.     b=true;
8.if (a==false)
9.  //Line 7 was executed, and a is not tainted
10.if (b==false)
11. //Line 5 was executed, and b is not tainted
```

Fig. 2. Attack using indirect control dependency

this point in the program. In this case, it is not possible to detect all information flows by dynamic analysis [27] because dynamic tainting occurs only along the branch that is actually executed. Variable b should be tainted, but is not. Thus the code presents an under-tainting problem that can be the cause of a failure to detect a leak of sensitive information.

3 Related work

Many works exist in the literature to track information flows. They are based on data tainting and static and dynamic analyses for detection of vulnerabilities. One of the most familiar work on data tainting is Perl's taint mode [30], an interpreted language which explicitly marks as tainted any data originating from outside a program, and prevents it from being used as arguments for certain sensitive functions that affect the local system - such as running local commands, creating and writing files and sending data over the network. Like Perl, the Ruby programming language [19] has a taint checking mechanism but with finer-grained taint levels than Perl. It has five safety levels ranging from 0 to 4, different security checks being performed at each level. One of the limits of Perl and Ruby approaches is that they can protect only against vulnerabilities in language-specific code.

Static taint analysis has been used to detect bugs in C programs. For example, Evans' Splint static analyzer [12] and Cqual [32] take C source codes as input annotated with "tainted" and "untainted" annotations to find security vulnerabilities such as string format vulnerabilities and buffer overflows. The static analysis based approach of Splint and Cqual presents many limitations due to undecidability problems including reachability and determining possible aliases. Shankar et al [28] use a similar approach but they add a new qualifier, tainted, to tag data that originated from an untrustworthy source. Denning [7, 8] defines a certification mechanism that determines the consistency of the data flow with the flow relation on given security classes specified by the programmer. To construct this mechanism, a lattice model is used at the analysis phase of

compilation to certify the security of a program. JFlow compiler [22] statically checks programs for correctness using information flow annotations and formal rules to prevent information leaks through storage channels. The major disadvantage of all the static analysis approaches is that they require a source code, they have some limitations due to undecidability problems [21] and they might report a number of false positives [6].

TaintCheck [25] is a mechanism that can perform dynamic taint analysis at binary level by instrumenting the code using Valgrind [24]. TaintTrace [5] is more efficient than TaintCheck. It is based on DynamoRIO [10] and consists of a number of optimizations to keep the overhead low. Also, LIFT [26] presents a low overhead information flow tracking system at software level. [17] presents a taint analysis for Java that instruments different classes to implement untrustworthy sources and sensitive sinks. [31] proposes an approach of dynamic instrumentation to keep track of how tainted data propagates throughout the whole system. The previous instrumentation-based approaches, implemented with a dynamic taint analysis, insert additional code into original application to trace and maintain information about the propagation. Thus they suffer from significant performance overhead that does not encourage their use in real-time applications.

TaintDroid [11] implements Dynamic taint analysis in real-time applications. Its design was inspired by these prior works, but addresses different challenges specific to mobile phones like the resource limitations. Hauser and al [18] present an approach for confidentiality violation detection based on dynamic data tainting. They extended Blare [16], an information flow monitor at the operating system level. Blare is able to dynamically observe information propagation. However, a significant limitation of standard approaches based on dynamic taint analysis is that they do not propagate taint along control dependencies. This can cause an under-tainting problem.

Some works have been undertaken to solve this under-tainting problem. BitBlaze [29] presents a novel fusion of static and dynamic taint analysis techniques to track implicit and explicit flow. DTA++ [20], based on the Bitblaze approach, presents an enhancement of dynamic taint analysis to limit the under-tainting problem. However DTA++ is evaluated only on benign applications but malicious programs in which an adversary uses implicit flows to circumvent analysis are out of scope. Furthermore, DTA++ is not implemented in embedded application. Trishul [23] correctly identifies implicit flow of information to detect a leak of sensitive information. The Data Mark Machine is an abstract model created by Fenton to handle implicit flows. It associates a security class $\underline{p}$ to a program counter p. This class is defined as follows: Whenever a conditional structure $c : S_1, \ldots, S_m$ is entered, $\underline{p}$ is set to $\underline{p} \oplus \underline{c}$. If a statement S is conditioned on the value of k condition variables $c_1, \ldots, c_k$ then $\underline{p}$ is set to $\underline{p} = \underline{c_1} \oplus \ldots \oplus \underline{c_k}$. If S represents an explicit flow from objects $a_1, \ldots, a_n$ to an object b, the instruction execution mechanism verifies that $\underline{a_1} \oplus \ldots \oplus \underline{a_n} \oplus \underline{p} \rightarrow \underline{b}$. Fenton [13] proves that this mechanism is sufficient to ensure the security of all implicit flows. But it is insufficient to guarantee security. Considering the implicit flow example shown in Figure 1 proposed by Fenton [14] where at the end of the execution, b attains the

value of a whereas $\underline{b} <> \underline{a}$. The problem is that the updating mechanism does not take into account the implicit flow when a branch is not executed. Thus the first branch is not followed ($a = true$) but it contains information which is then leaked using the next if. To solve this problem, Fenton [13] and Gat and Saal [15] proposed a solution which restores the value and class of objects changed during the execution of conditional structure to the value and security class it had before entering the branch. But, this approach cannot be applied in practice because existing application code does not modify control structures to consider information flow leaks. Furthermore, the Data Mark Machine is an abstract concept that lacks formal proofs of its soundness and was never implemented. By contrast, Gat and Saal's approach is based on specialized hardware architecture to control information flow. Aries [4] considers that writing to a particular location within a branch is disallowed when the security class associated with that location is equal or less restrictive than the security class of $\underline{p}$. So, in the example shown in Figure 1, if a is false then the compile time system prohibits the program to write to c since the security class of c (Low) is less or equal then the security class of p ($Low <= \underline{p}$). But, p is not known at compile time and this approach is based only on high and low security classes. Beres and Dalton [3] suggest that taking the Aries approach would preclude many applications from executing correctly, and might potentially still leak some information through other covert channels. Therefore, they are based on the principle that in practice, when running real applications it is acceptable to ignore certain information leaks. In Denning's approach [9], whether the branch is taken or not, the compiler inserts in the compiled program updating instructions that determine the required class updated to reflect the information flow. In the previous example, it inserts an instruction at the end of the if ($!a$) $c = true$ code block to update $\underline{c}$ to $\underline{p}$ ($= \underline{a}$). In the approach we propose, we draw our inspiration from the Denning approach, but we perform the required class update when Java methods are invoked, as we track Java applications, instead of performing the update at compile time. These approaches are not implemented in embedded systems like smartphones. Thus, to secure a running process of a smartphone, we propose to prevent the execution of the malicious code by monitoring transfers of control in a program. Then we show that this approach is effective to detect control flow attacks and solve the under-tainting problem.

4 Formal specification of the under tainting problem

Denning [7] defined an information flow model as:

$$FM =< N, P, SC, \oplus, \rightarrow > .$$

N is a set of logical storage objects (files, program variables, ...). P is a set of processes that are executed by the active agents responsible for all information flow. SC is a set of security classes that are assigned to the objects in N. SC is finite and has a lower bound L attached to objects in N by default. The class combining operator "$\oplus$" specifies the class result of any binary function on

values from the operand classes. A flow relation "$\rightarrow$" between pairs of security classes A and B means that "information in class A is permitted to flow into class B". A flow model FM is secure if and only if execution of a sequence of operations cannot produce a flow that violates the relation "$\rightarrow$".

We draw our inspiration from the Denning information flow model to formally specify under tainting. However, we assign taint to the objects instead of assigning security classes. Thus, the class combining operator "$\oplus$" is used in our formal specification to combine taints of objects.

We use the following syntax to formally specify under tainting: A and B are two logical formulas and x and y are two variables.

- $A \Rightarrow B$: If A Then B
- $x \rightarrow y$: Information flow from object x to object y
- $x \leftarrow y$: the value of y is assigned to x
- $Taint(x) \oplus Taint(y)$: specifies the taint result of combined taints.

Definition: We have a situation of under tainting when x depends on a *condition*, the value of x is modified in the conditional branch and *condition* is tainted but x is not tainted.

Formally, we can define the under tainting when there is a variable x and a *condition* such that:

$$\begin{aligned} Ismodified(x) \wedge Dependency(x, condition) \wedge Tainted(condition) \\ \wedge \neg Tainted(x) \end{aligned} \tag{1}$$

where:

- $Ismodified(x)$ associates with x the result of $explicitflowstatement$.

$$Ismodified(x) \stackrel{def}{\equiv} (x \leftarrow explicitflowstatement)$$

- $Dependency(x, condition)$ defines an information flow from *condition* to x when x depends on the *condition*.

$$Dependency(x, condition) \stackrel{def}{\equiv} (condition \rightarrow x)$$

Axioms: Let us consider the following axioms:

$$(x \rightarrow y) \Rightarrow (Taint(y) \leftarrow Taint(x)) \tag{2}$$

$$(x \leftarrow y) \Rightarrow (y \rightarrow x) \tag{3}$$

$$\begin{aligned} (Taint(x) \leftarrow Taint(y)) \wedge (Taint(x) \leftarrow Taint(z)) \\ \Rightarrow (Taint(x) \leftarrow Taint(y) \oplus Taint(z)) \end{aligned} \tag{4}$$

Proof of non Under-tainting: We will prove that our system cannot be in an under tainting situation. We perform a proof *reductio ad absurdum*. We assume that the conditions necessary to be in an under tainting situation are satisfied. Thus, (1) is valid. Therefore, $Dependency(x, condition)$ is true,

$Ismodified(x)$ is true, $Tainted(condition)$ is true and $Taint(x)$ is false. But, $Dependency(x, condition)$ is true implies that $condition \rightarrow x$. Then, by applying axiom (2), we have $Tainted(condition)$ is true. Then x is tainted and $Taint(x) \leftarrow Taint(condition)$ which contradicts with $Taint(x)$ is false. ■

Proof of propagation taint rules: We consider that $ContextTaint$ is the taint of the *condition*. To know the exact taint of x we prove that the two rules that specify the propagation taint policy are valid:

- Rule 1: if the value of x is modified and x depends on the *condition* and the branch is taken, we will apply the first rule to taint x.

$$\frac{Ismodified(x) \wedge Dependency(x, condition) \wedge BranchTaken}{Taint(x) \leftarrow ContextTaint \oplus Taint(explicitflowstatement)}$$

- Rule 2: if the value of x is modified and x depends on the *condition* and the branch is not taken, we will apply the second rule to taint x.

$$\frac{Ismodified(x) \wedge Dependency(x, condition) \wedge \neg BranchTaken}{Taint(x) \leftarrow Taint(x) \oplus ContextTaint}$$

Let us start with the first rule and suppose that $Dependency(x, condition)$ is true, $Ismodified(x)$ is true and $BranchTaken$ is true, we will demonstrate that $(Taint(x) \leftarrow ContextTaint \oplus Taint(explicitflowstatement))$ is valid.
Given that $Dependency(x, condition)$ is true, thus $condition \rightarrow x$, using axiom (2) we obtain $Taint(x) \leftarrow Taint(condition)$. As $ContextTaint = Taint(condition)$, then $Taint(x) \leftarrow ContextTaint$. Now, $Ismodified(x)$ is true, then $x \leftarrow explicitflowstatement$. Using axiom (3), we obtain:
$Taint(x) \leftarrow Taint(explicitflowstatement)$. Finally, using axiom (4), we get:
$Taint(x) \leftarrow ContextTaint \oplus Taint(explicitflowstatement)$. ■

We will now prove the second rule. Let us first assume that $Dependency$ $(x, condition)$ is true, $Ismodified(x)$ is true and $BranchTaken$ is false, we will demonstrate that $(Taint(x) \leftarrow Taint(x) \oplus ContextTaint)$ is valid.

The relation " $\rightarrow$ " is reflexive then $x \rightarrow x$, we use (2): $Taint(x) \leftarrow Taint(x)$. $Dependency(x, condition)$ is true then $condition \rightarrow x$, we use (2) to obtain $Taint(x) \leftarrow Taint(condition)$. As, $ContextTaint = Taint(condition)$ then $Taint(x) \leftarrow ContextTaint$.
We use (4) : $Taint(x) \leftarrow Taint(x) \oplus ContextTaint$. The predicate $BranchTaken$ specifies that branch is executed. So, an explicit flow which contains x is executed. Otherwise, branch is not taken so x depends only on implicit flow and does not depend on explicit flow. ■

5 Detecting control flow in embedded systems

TaintDroid cannot detect control flows because it only uses dynamic taint analysis. We aim to enhance the TaintDroid approach by tracking control flow in

the Android system to solve the under-tainting problem. To do so, we adapt and integrate the implicit flow management approach defined in Trishul. We use also a hybrid approach that combines and benefits from the advantages of static and dynamic analyses. To solve the under-tainting problem, we use the previous rules of taint propagation that we proved in section 4. We present in the following TaintDroid and Trishul from which we took our inspiration to implement our approach in real-time applications such as smartphone applications.

5.1 Background

TaintDroid : architecture and principles Third-party smartphone applications can access to sensitive data and compromise confidentiality and integrity of Android systems. To solve this problem, TaintDroid, an extension to the Android mobile-phone platform, implements dynamic taint analysis to track the information flow in real-time and control the handling of private data.

Architecture of TaintDroid Figure 3 presents the TaintDroid architecture. After tainting data in the trusted application (1), a native method called by the taint interface stores the taint in the virtual taint map (2). The taint tags are propagated by the Dalvik VM referencing (3) data flow rules. When the tainted information is used in an IPC transaction, the modified binder library (4) verifies that the taint tag parcel is equivalent to combined taint marking of all data in the parcel. The parcel is sent through the kernel (5) and received by the remote untrusted application (only the interpreted code is untrusted). The modified binder library assigns the taint tag from the parcel to all values read from it (6). The taint tags are propagated by the remote Dalvik VM instance (7) identically to the untrusted application. When tainted data is used in a taint sink (network sink) (8), the library specifies the taint sink, gets the taint tag (9) and reports the event.

Handling Flows with TaintDroid TaintDroid uses the dynamic taint analysis to track explicit flows on smartphones. First, it defines a sensitive source. Each input data is tainted with its source taint. Then, TaintDroid tracks propagation of tainted data at the instruction level. The taint propagation is patched by running the native code without instrumentation. To minimize IPC overhead, it implements message-level tracking between applications and file-level tracking. Finally, vulnerability can be detected when tainted data are used in taint sink (network interface). One limit of TaintDroid is that it cannot detect control flows because it uses dynamic taint analysis. We aim to enhance the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. To do so, we adapt and integrate the Trishul approach. We describe this approach with more details in the following.

Trishul Trishul is an information flow control system. It is implemented in a Java virtual machine to secure execution of Java applications by tracking

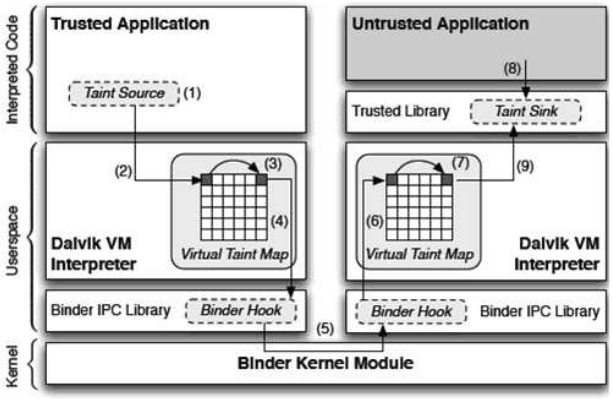

Fig. 3. TaintDroid architecture [11]

data flow within the environment. It does not require a change to the operating system kernel because it analyzes the bytecode of an application being executed. Trishul is based on the hybrid approach to correctly handle implicit flows using the compiled program rather than the source code at load-time.

Architecture of Trishul When an application calls a function, the Trishul's run-time policy enforcement architecture provides a mechanism to trap these function calls. By using a policy decision engine that prevents tainted data to be propagated to insecure locations (network channels), it checks the policy and decides whether or not to allow the calls. To do that, Trishul is placed between the Java application and the operating system. The Trishul architecture is based on two parts illustrated in Figure 4: the core Trishul JVM system and the pluggable policy engine. The core JVM allows information flow tracking and provides the policy engine with the hooks needed to trap the calls performed by the untrusted application. Using these hooks, the policy engine loads appropriate policies into the Trishul system based on the access and propagation of tainted data in the application to allow or not the application function call. When the application code is loaded, a policy engine is also loaded in the JVM. If the application reads a piece of data from the hard disk (1), these data are loaded into Trishul (2). The policy engine hooks onto the call and taints the data. The information flow control functionality of Trishul ensures that the taint remains associated with the data when it is propagated (3). When the tainted data is used by the application (4) (sent over a socket connection) (5), Trishul interposes (6) and transfers the control to the policy decision engine (7). The engine checks with the respective data's usage policy (8) and decides whether or not to allow the application to proceed.

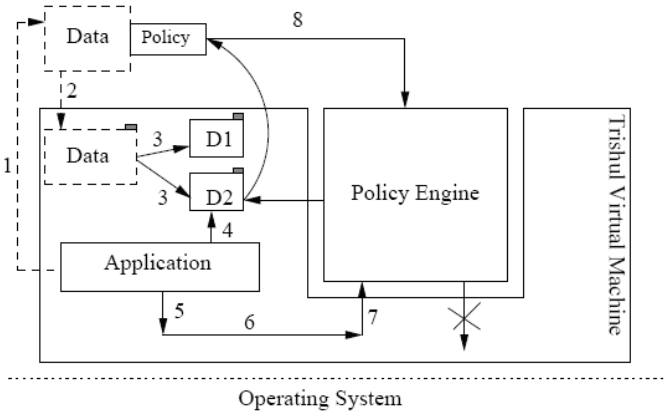

Fig. 4. Architecture of Trishul [23]

Handling Flows with Trishul Trishul assigns a taint to each value that appears as an operand on the JVM working stack (local variable, parameter and return value). It handles explicit flows by instrumenting the Java bytecode instructions to combine the taint values when the corresponding values are used as operands of a logic or arithmetic operation. To detect the implicit flow, Trishul uses:

- Static analysis of the bytecode at load time: To define the conditional control flow instruction, Trishul creates the control flow graph (CFG) which is analyzed to determine branches in the method control flow. A basic block is assigned to each control flow branch. When the basic block is executed, the condition taint is included in its context taint, because the information flow in that block depends on the condition. This taint is removed from the context taint when all paths have converged and the condition does not influence the control flow. A dataflow analysis (postdominator analysis) is determined to detect branching and merging of the flow of control in the graph. A context bitmap summarizes the result of this dataflow and is used to update the context-taint appropriately at run-time.

- The dynamic system uses information provided by the static analysis and run-time enforcement: The run-time enforcement allows policies to be attached when the program is executed. Trishul attaches an array of context taints to each method that is stored in the method's stack frame. When the conditional flow instruction is executed, the condition taint is stored in an appropriate entry of the array.

Trishul solves the under-tainting problem by updating the context taint and maintaining a list of all the variables that are modified in a basic block of control flow graph to handle not executed branches. But, it is not implemented in

embedded system such as smartphone. We adapt and integrate the implicit flow management approach of Trishul and we follow not executed branches to solve the under-tainting problem in the Android system. We present, in the following, extensions of the TaintDroid and Trishul works that we implement in real-time applications such as smartphone applications.

5.2 Handling implicit flow in Android system

To solve the under-tainting problem in the Android system [1] we use a hybrid approach that improves the functionality of TaintDroid by integrating the concepts introduced by Trishul. TaintDroid is composed of four modules: (1) Explicit flow module that tracks variable at the virtual machine level, (2) IPC Binder module that tracks messages between applications, (3) File module that tracks files at the storage level and (4) Taint propagation module that is implemented in the native methods level.

To track implicit flow, we propose to add an implicit flow module in the Dalvik VM bytecode verifier which checks instructions of methods at load time. We define two additional rules that we prove in section 4 to propagate taint in the control flow. At class load time, we build an array of variables that are modified to handle the branch that is not executed. Figure 5 presents the modified architecture to handle implicit flow. Our process is summarized in Figure 6.

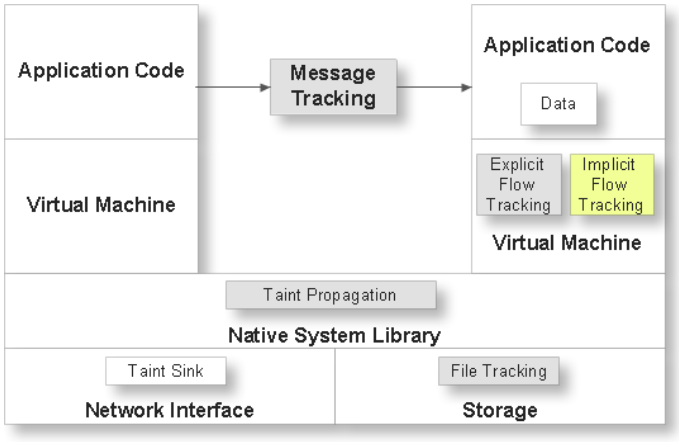

Fig. 5. Modified architecture to handle implicit flow.

– Static analysis at load time:
  - We create the control flow graphs which will be analyzed to determine branches in the method control flow. In a control flow graph, each node in the graph represents a basic block. Directed edges are used to represent jumps in the control flow.

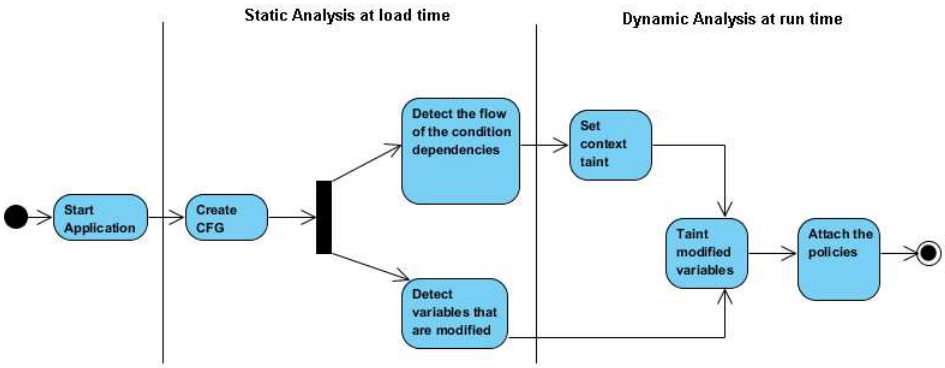

Fig. 6. Handling implicit flow in Android system.

- We detect the flow of the condition-dependencies from blocks in the graph.
- We detect variables that are modified in a basic block of the control flow graph to handle not executed branches.

– Dynamic analysis, at run time, uses information provided by the static analysis:
  - We create an array of context taints that includes all condition taints.
  - By referencing to the condition-dependencies from block in the graph, we set the context taint of each basic block.
  - We taint modified variables that exist in the conditional instruction according to the rules of taint propagation (see section 4): If the branch is taken: $Taint(x) = ContextTaint \oplus Taint(explicitflowstatement)$. If the branch is not taken: $Taint(x) = ContextTaint \oplus Taint(x)$
  - We attach the policies that prevent the use of tainted data in defined taint sink.

Application of our approach:

```
boolean x;
boolean y;
if ( x == true )
y = t rue;
else
y = false;
return(y);
```

Fig. 7. Source code for implicit flow

```
0: iload_0
1: ifeq 9
4: iconst_1
5: istore_1
6: goto 11
9: iconst_0
10: istore_1
11: iload_1
12: ireturn
```

Fig. 8. Bytecode for implicit flow

The source code in Figure 7 presents an indirect transfer of value from x to y. If $x = true$ the first branch is executed but the second is not, thus $y = true$. The

same result when $x = false$. In this case, we have an implicit flow of information from x to y. To detect such flows, a control flow graph, represented in Figure 9,

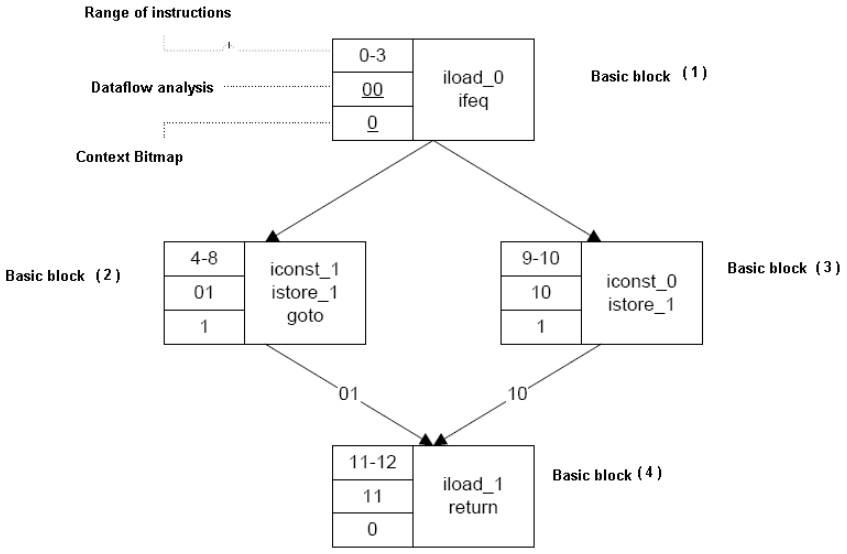

Fig. 9. Control flow graph corresponding to Figure 7 example.

is created at load time using the byte code shown in Figure 8. The bytecode is composed of basic blocks that present the node of CFG. Each basic block groups instructions in the control flow block. The range of instructions covered by this basic block is $< pc\_start, pc\_end >$. An analysis flow allows detecting the flow of the condition-dependencies from the blocks in the graph. It is composed of bits. When both bits are set (basic block 4 of the graph), the flow of control is merged. Thus this block is not controlled by control condition. It is controlled when one bit is set (basic blocks 2 and 3 of the graph). If no bit is set, this block represents the conditional instruction (basic block 1 of the graph). The context bitmap is composed of one bit per conditional instruction. It represents the analysis flow value on one bit. At run time, we use the context bitmap to know dependency of blocks to the conditional instruction. This dependency is detected when the context bitmap is set. Thus, we include the condition taint in the context taint. In our example, we include the taint of x in the context taint. We use the two propagation rules of section 4 and assume that $x = true$. The first branch is taken, so we use the first rule: $Taint(y) = ContextTaint \oplus Taint(explicitflowstatement)$ or $ContextTaint = Taint(x)$ then $Taint(y)$ depends on $Taint(x)$. The second branch is not taken, so we use the second rule: $Taint(y) = ContextTaint \oplus Taint(y)$ or $ContextTaint = Taint(x)$ then $Taint(y)$ depends on $Taint(x)$. So, implicit flow of information from x to y is correctly identified with this approach. We have a similar result when $x = false$.

6 Conclusion

In order to protect embedded systems from software vulnerabilities, it is necessary to have automatic attack detection mechanisms. In this paper, we show how to enhance dynamic taint analysis with static analysis to track implicit flows in the Google Android operating system. We prove that our system cannot create under tainting states. Thus, malicious applications cannot bypass the Android system and get privacy sensitive information through control flows. The implementation of our approach "static analysis at the load time" to handle implicit flows is underway. We perform a static verification on a single method by checking its instructions at load time. When it is a control instruction (if, Go to, etc.), we allocate and insert a BasicBlock at the end of the basic blocks list. We specify its target and allocate a BitmapBits for tracking condition dependency. Future work will be to create the CFG from the method and implement the dynamic analysis at run time based on information provided by the static analysis. Once the implementation is finished, we will be able to evaluate our approach in terms of overhead and false alarms. We will also demonstrate the completeness of the propagation rules.

References

1. Android, `http://www.android.com/`
2. APPLE, I.: Apple store downloads top three billion (January2010), `http://www.apple.com/pr/library/2010/01/05Apples-App-Store-Downloads-Top-Three-Billion.html`
3. Beres, Y., Dalton, C.: Dynamic label binding at run-time. In: Proceedings of the 2003 Workshop on New security paradigms. pp. 39–46. ACM (2003)
4. Brown, J., Knight Jr, T.: A minimal trusted computing base for dynamically ensuring secure information flow. Project Aries TM-015 (November 2001) (2001)
5. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: Tainttrace: Efficient flow tracing with dynamic binary rewriting. In: ISCC'06. Proceedings. 11th IEEE Symposium on. pp. 749–754. IEEE (2006)
6. Chess, B., McGraw, G.: Static analysis for security. Security & Privacy, IEEE 2(6), 76–79 (2004)
7. Denning, D.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
8. Denning, D., Denning, P.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
9. Denning, D.: Secure information flow in computer systems. Ph.D. thesis, Purdue University (1975)
10. Derek Bruening, Q.Z.: Dynamorio : Dynamic instrumentation tool platform. `http://dynamorio.org/`
11. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–6. USENIX Association (2010)
12. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. Software, IEEE 19(1), 42–51 (2002)

13. Fenton, J.: Information protection systems. Ph.D. thesis, University of Cambridge (1973)
14. Fenton, J.: Memoryless subsystem. Computer Journal 17(2), 143–147 (1974)
15. Gat, I., Saal, H.: Memoryless execution: a programmers viewpoint. Ibm tech. rep. 025, IBM Israeli Scientific Center (1975)
16. George, L., Viet Triem Tong, V., Mé, L.: Blare tools: A policy-based intrusion detection system automatically set by the security policy. In: Recent Advances in Intrusion Detection. pp. 355–356. Springer (2009)
17. Haldar, V., Chandra, D., Franz, M.: Dynamic taint propagation for java. In: Proceedings of the 21st Annual Computer Security Applications Conference. pp. 303–311. Citeseer (2005)
18. Hauser, C., Tronel, F., Reid, J., Fidge, C.: A taint marking approach to confidentiality violation detection. In: Proceedings of the 10th Australasian Information Security Conference (AISC 2012). vol. 125. Australian Computer Society (2012)
19. Hunt, A., Thomas, D.: Programming ruby: The pragmatic programmer's guide. New York: Addison-Wesley Professional. 2 (2000)
20. Kang, M., McCamant, S., Poosankam, P., Song, D.: Dta++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego, CA (2011)
21. Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) 1(4), 323–337 (1992)
22. Myers, A.: Jflow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 228–241. ACM (1999)
23. Nair, S., Simpson, P., Crispo, B., Tanenbaum, A.: A virtual machine based information flow control system for policy enforcement. Electronic Notes in Theoretical Computer Science 197(1), 3–16 (2008)
24. Nethercote, N., Seward, J.: Valgrind:: A program supervision framework. Electronic notes in theoretical computer science 89(2), 44–66 (2003)
25. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer (2005)
26. Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 135–148. IEEE Computer Society (2006)
27. Sabelfeld, A., Myers, A.: Language-based information-flow security. Selected Areas in Communications, IEEE Journal on 21(1), 5–19 (2003)
28. Shankar, U., Talwar, K., Foster, J., Wagner, D.: Detecting format string vulnerabilities with type qaualifiers. In: Proceedings of the 10th conference on USENIX Security Symposium-Volume 10. pp. 16–16. USENIX Association (2001)
29. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. Information Systems Security pp. 1–25 (2008)
30. Wall, L., Christiansen, T., Orwant, J.: Programming perl. O'Reilly Media (2000)
31. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 116–127. ACM (2007)
32. Zhang, X., Edwards, A., Jaeger, T.: Using cqual for static analysis of authorization hook placement. In: Proceedings of the 11th USENIX Security Symposium. pp. 33–48 (2002)