Formal Characterization of Illegal Control Flow in Android System
Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Ana Cavalli

To cite this version:
Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Ana Cavalli. Formal Characterization of Illegal Control Flow in Android System. SITIS 2013 : 9th International Conference on Signal Image Technology & Internet Systems, Dec 2013, Kyoto, Japan. 2013. <hal-00924480>

HAL Id: hal-00924480
https://hal.archives-ouvertes.fr/hal-00924480
Submitted on 6 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract—The dynamic taint analysis mechanism is used to protect sensitive data in the Android system. But this technique does not detect control flows which can cause an under-tainting problem. This means that some values should be marked as tainted, but are not. The under-tainting problem can be the cause of a failure to detect a leak of sensitive information. To solve this problem, we use a set of formally defined rules that describes the taint propagation. We prove the completeness of these rules. Also, we provide a correct and complete algorithm based on these rules to solve the under-tainting problem.

Keywords—dynamic taint analysis; android system; control flows; under-tainting; formal rules; complete algorithm

I. INTRODUCTION

Recent years have witnessed an increase in the use of embedded systems such as smartphones. According to a recent Gartner report [1], 417 million of worldwide mobile phones were sold in the third quarter of 2010, which corresponds to 35 percent increase from the third quarter of 2009. To make mobile phones more fun and useful, users usually download third-party applications. For example, we can show an increase in third-party apps of Android Market from about 15,000 third-party apps in November 2009 to about 150,000 in November 2010. These applications are used to capture, store, manipulate, and access to data of a sensitive nature in mobile phone. An attacker can launch flow control attacks to compromise confidentiality and integrity of the Android system and can leak private information without user authorization. In the study presented in Black Hat conference, Daswani [2] analyzed the live behavior of 10,000 Android applications and show that more than 800 were found to be leaking personal data to an unauthorized server. Therefore, there is a need to provide adequate security mechanisms to control the manipulation of private data by third-party apps. The dynamic taint analysis mechanism is used to protect sensitive data in the Android system against attacks [3]. 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. Let us consider the attack shown in Figure 1 that presents an under-tainting problem which can cause a failure to detect a leak of sensitive information. The variables $x$ and $y$ 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 $x$ to true and $x$ is tainted. Variable $y$ keeps its false value, since the assignment on Line 7 is not executed and $y$ is not tainted because dynamic tainting occurs only along the branch that is actually executed. As $y$ is not tainted, it is leaked to the network (Line 8) without being detected. Since $y$ has not been modified, it informs the attacker about the value of the user private contact. Thus, an attacker can circumvent an android system through the control flows.

In a previous work [4], we have proposed an approach that combines static and dynamic taint analysis to propagate taint along control dependencies and to track implicit flows in embedded systems such as the Google Android operating system. In this paper, we formally specify the under-tainting problem and we provide an algorithm to solve it based on a set of formally defined rules describing the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm.

The rest of this paper is organized as follows. Section 2 gives a technical overview of our approach. Section 3 presents some definitions and theorems that are used in other sections. Section 4 describes our formal specification of the under-tainting problem. In section 5, we specify an algorithm based on a set of formally defined rules describing the taint propagation policy that we use to solve the under-tainting problem. Related work about existing solutions to solve the under-tainting problem is analyzed in section 6. Finally, section 7 concludes with an outline of future work.

II. APPROACH OVERVIEW

TaintDroid [3], an extension of the Android mobile-phone platform, implements dynamic taint analysis to track the information flow in real-time and control the handling of private data. It addresses different challenges specific to mobile phones like the resource limitations. 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. It only tracks explicit flows and does not track control flows. In a previous work [4], we have proposed a technical approach that enhances the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. To track implicit flow, we have added an implicit flow module in the Dalvik VM bytecode verifier which checks instructions of methods at load time. We have defined two additional rules to propagate taint in the control flow. At class load time, we have built an array of variables to which a value is

```
1. x = false;
2. y = false;
3. char c[256];
4. if ( gets(c) != user_contact )
5.   x = true;
6. else
7.   y = true;
8. NetworkTransfer(y);
```

Figure 1. Attack using indirect control dependency
assigned to handle the branch that is not executed. Figure 2 presents the modified architecture to handle implicit flow in TaintDroid system.

In this paper, we prove the completeness of the two additional rules and we provide a correct and complete taint algorithm based on these rules. Our algorithm is based on a hybrid approach that combines and benefits from the advantages of static and dynamic analyses. We use static analysis to detect control dependencies. This analysis is based on the control flow graphs [5], [6] which will be analyzed to determine branches in the conditional structure. A basic block is assigned to each control flow branch. Then, we detect the flow of the condition-dependencies from blocks in the graph. Also, we detect variable assignment in a basic block of the control flow graph to handle not executed branches. The dynamic analysis uses information provided by the static analysis and allows tainting variables to which a value is assigned in the conditional instruction. To taint these variables, we create an array of context taints that includes all condition taints. We use the context taints array and the condition-dependencies from block in the graph to set the context taint of each basic block. Finally, we apply the propagation rules to taint variables to which a value is assigned whether the branch is taken or not.

III. NOTATIONS, DEFINITIONS AND THEOREMS

Definition 1. Direct graph
A directed graph \( G = (V, E) \) consists of a finite set \( V \) of vertices and a set \( E \) of ordered pairs \((v, w)\) of distinct vertices, called edges. If \((v, w)\) is an edge, \( w \) is a successor of \( v \) and \( v \) is a predecessor of \( w \).

Definition 2. Complete directed graph
A complete directed graph is a simple directed graph \( G = (V, E) \) such that every pair of distinct vertices in \( G \) are connected by exactly one edge. So, for each pair of distinct vertices, either \((x, y)\) or \((y, x)\) (but not both) is in \( E \).

Definition 3. Control flow graph
A control flow graph \( G = (V, E, r) \) is a directed graph \((V, E)\) with a distinguished \( Exit \) vertex and start vertex \( r \), such that for any vertex \( v \in V \) there is a path from \( r \) to \( u \). The nodes of the control flow graph represent basic blocks and the edges represent control flow paths.

The concept of post-dominator and dominator tree are used to determine dependencies of blocks in the control flow graph.

Definition 4. Dominator
A vertex \( v \) dominates another vertex \( w \neq v \) in \( G \) if every path from \( r \) to \( w \) contains \( v \).

Definition 5. Post-Dominator
A node \( v \) is post-dominated by a node \( w \) in \( G \) if every path from \( v \) to \( Exit \) (not including \( v \)) contains \( w \).

Theorem 1. Every vertex of a flow graph \( G = (V, E, r) \) except \( r \) has a unique immediate dominator. The edges \( \{(idom(w), w)\} | w \in V - \{r\} \) form a directed tree rooted at \( r \), called the dominator tree of \( G \), such that \( v \) dominates \( w \) if and only if \( v \) is a proper ancestor of \( w \) in the dominator tree [7], [8].

Computing post-dominators in the control flow graph is equivalent to computing dominators [5] in the reverse control flow graph. Dominators in the reverse graph can be computed quickly by using the Fast Algorithm [9] or a linear-time dominators algorithm [10] to construct the dominator tree. Using these algorithms, we can determine the post-dominator tree of a graph.

Definition 6. Control Dependency
Let \( G \) be a control flow graph. Let \( X \) and \( Y \) be nodes in \( G \). \( Y \) is control dependent on \( X \) noted \( \text{Dependency}(X, Y) \) if:

1) There exists a directed path \( P \) from \( X \) to \( Y \) with any \( Z \) in \( P \) (excluding \( X \) and \( Y \)) post-dominated by \( Y \) and
2) \( X \) is not post-dominated by \( Y \).

Given the post-dominator tree, Ferrante et al. [11] determine control dependencies by examining certain control flow graph edges and annotating nodes on the corresponding tree paths.

Definition 7. Context Taint
Let \( G \) be a control flow graph. Let \( X \) and \( Y \) be basic blocks in \( G \). If \( Y \) is control dependent on \( X \) that contains \( \text{Condition} \) then we associate to \( Y \) a Context Taint with \( \text{Context Taint}(Y) = \text{Taint}(\text{Condition}) \).

We use the completeness theorem to prove the completeness of the taint propagation rules in section V-A. We use the soundness theorem to prove this completeness from left to right and the compactness theorem and theorem 2 to prove from right to left. These theorems [12] are given below.

Completeness Theorem. For any sentence \( G \) and set of sentences \( \mathcal{F} \), \( \mathcal{F} \models G \) if and only if \( \mathcal{F} \vdash G \).

Soundness Theorem. For any formula \( G \) and set of formulas \( \mathcal{F} \), if \( \mathcal{F} \vdash G \), then \( \mathcal{F} \models G \).

Compactness Theorem. Let \( \mathcal{F} \) be a set of formulas. \( \mathcal{F} \) is unsatisfiable if and only if some finite subset of \( \mathcal{F} \) is unsatisfiable.

Definition 8. CNF formula
A formula \( F \) is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. That is,
where each $L_{i,j}$ is either atomic or a negated atomic formula.

**Theorem 2.** Let $F$ and $G$ be formulas of the first-order logic. Let $H$ be the CNF formula obtained by applying the CNF algorithm [12] to the formula $F \land \neg G$. Let $\text{Res}^*(H)$ be the set of all clauses that can be derived from $H$ using resolvents. The following are equivalent:

1. $F \models G$
2. $F \models \neg G$
3. $\emptyset \in \text{Res}^*(H)$

**IV. THE UNDER-TAINTING PROBLEM**

In this section we formally specify the under-tainting problem based on Denning’s information flow model. Denning [13] 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.

**Syntactic definition of connectors $\{\rightarrow, \oplus, \leftarrow\}$:**

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.

**Semantic definition of connectors $\{\rightarrow, \leftarrow, \oplus\}$:**

- The $\rightarrow$ connector is reflexive: If $x$ is a variable then $x \rightarrow x$.
- The $\rightarrow$ connector is transitive: $x$, $y$ and $z$ are three variables, if $(x \rightarrow y) \land (y \rightarrow z)$ then $x \rightarrow z$.
- The $\leftarrow$ connector is reflexive: If $x$ is a variable then $x \leftarrow x$.
- The $\leftarrow$ connector is transitive: $x$, $y$ and $z$ are three variables, if $(x \leftarrow y) \land (y \leftarrow z)$ then $x \leftarrow z$.
- The $\rightarrow$ and $\leftarrow$ connectors are not symmetric.
- The $\oplus$ relation is commutative: $Taint(x) \oplus Taint(y) = Taint(y) \oplus Taint(x)$
- The $\oplus$ relation is associative: $Taint(x) \oplus (Taint(y) \oplus Taint(z)) = (Taint(x) \oplus Taint(y)) \oplus Taint(z)$

**Definition 9. Under-Tainting**

We have a situation of under-tainting when $x$ depends on a condition, the value of $x$ is assigned in the conditional branch and condition is tainted but $x$ is not tainted.

Formally, an under-tainting occurs when there is a variable $x$ and a formula condition such that:

$$\text{IsAssigned}(x, y) \land \text{Dependency}(x, \text{condition}) \land \text{Tainted}(\text{condition}) \land \neg \text{Tainted}(x)$$  \hspace{1cm} (1)

where:

- $\text{IsAssigned}(x, y)$ associates with $x$ the value of $y$.
- $\text{IsAssigned}(x, y) \overset{\text{def}}{=} (x \leftarrow y)$
- $\text{Dependency}(x, \text{condition})$ defines an information flow from condition to $x$ when $x$ depends on the condition.
- $\text{Dependency}(x, \text{condition}) \overset{\text{def}}{=} (\text{condition} \rightarrow x)$

**V. THE UNDER-TAINTING SOLUTION**

In this section, we specify a set of formally defined rules that describe the taint propagation. We prove the completeness of these rules. Then, we provide an algorithm to solve the under-tainting problem based on these rules. Afterwards, we analyse some important properties of our algorithm such as Correctness and Completeness.

**A. The taint propagation rules**

Let us consider the following axioms:

$$x \rightarrow y \Rightarrow (Taint(y) \leftarrow Taint(x))$$  \hspace{1cm} (2)

$$x \leftarrow y \Rightarrow (y \rightarrow x)$$  \hspace{1cm} (3)

$$(Taint(x) \leftarrow Taint(y)) \land (Taint(x) \leftarrow Taint(z)) \Rightarrow (Taint(x) \leftarrow Taint(y) \oplus Taint(z))$$  \hspace{1cm} (4)

**Theorem 3.** We consider that $\text{Context_Taint}$ is the taint of the condition. To solve the under-tainting problem, we use the two rules that specify the propagation taint policy:

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

$$\text{IsModified}(x, \text{explicit flowstatement}) \land \text{Dependency}(x, \text{condition})$$

$$\land \text{BranchTaken}(br, \text{conditional statement})$$

$$\Rightarrow (Taint(x) \leftarrow \text{Context_Taint} \oplus \text{Taint(explicit flowstatement)})$$

where:

- The predicate $\text{BranchTaken}(br, \text{conditional statement})$ specifies that branch $br$ in the conditional statement is executed.
- An explicit flow which contains $x$ is executed. $\text{IsModified}(x, \text{explicit flowstatement})$ associates with $x$ the result of an explicit flow statement.

$$\text{IsModified}(x, \text{explicit flowstatement}) \overset{\text{def}}{=} (x \leftarrow \text{explicit flowstatement})$$

- Rule 2: if the value of $y$ is assigned to $x$ and $x$ depends on the condition and the branch $br'$ in the conditional statement is not taken ($x$ depends only on implicit flow and does not depend on explicit flow), we will apply the following rule to taint $x$. 

$$\text{IsAssigned}(x, y) \land \text{Dependency}(x, \text{condition}) \land \neg \text{Tainted}(x)$$  \hspace{1cm} (1)
Proof of taint propagation rules

To prove completeness of propagation taint rules, we use the basic rules cited in Table I for derivations.

<table>
<thead>
<tr>
<th>Premise</th>
<th>Conclusion</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F \vdash G \rightarrow (F &amp; G)$</td>
<td>$F \vdash G$</td>
<td>Assumption</td>
</tr>
<tr>
<td>$F \vdash F \rightarrow G$</td>
<td>$F \vdash F &amp; G$</td>
<td>Monotonicity</td>
</tr>
<tr>
<td>$F \vdash (F &amp; G)$</td>
<td>$F \vdash (F &amp; G)$</td>
<td>∧-Introduction</td>
</tr>
<tr>
<td>$F \vdash (F &amp; G)$</td>
<td>$F \vdash (F &amp; G)$</td>
<td>∧-Symmetry</td>
</tr>
</tbody>
</table>

Table I

Basic rules for derivations

We start by proving completeness of the first rule. We suppose that $F = \{ \text{IsModified}(x, \text{explicitflowstatement}), \text{Dependency}(x, \text{condition}), \text{BranchTaken}(br, \text{conditionalstatement}) \}$ and $G = \text{Taint}(x) \leftarrow \text{Context}_\text{Taint} \oplus \text{Taint}($explicitflowstatement$)$.

We prove soundness, left to right, by induction. If $F \vdash G$, then there is a formal proof concluding with $F \vdash G$ (see Table II). Let $M$ be an arbitrary model of $F$, we will demonstrate that $M \models G$. $G$ is deduced by Modus ponens of $G_1, G_2 \rightarrow G$ then by induction, $M \models G_1$ and $M \models G_2 \rightarrow G$ and it follows $M \models G$.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $(\text{condition} \rightarrow x) \vdash (\text{Taint}(x) \rightarrow \text{Taint}(\text{condition}))$</td>
<td>Axiom (2)</td>
</tr>
<tr>
<td>2. $(\text{condition} \rightarrow x) \vdash (\text{Taint}(x) \rightarrow \text{Context}_\text{Taint})$</td>
<td>Monotonicity applied to 2</td>
</tr>
<tr>
<td>3. $F \vdash (\text{Taint}(x) \rightarrow \text{Context}_\text{Taint})$</td>
<td>Monotonicity applied to 2</td>
</tr>
<tr>
<td>4. $(x \rightarrow \text{explicitflowstatement}) \vdash (\text{This}(x) \rightarrow \text{Taint}(\text{condition}))$</td>
<td>Axiom (3)</td>
</tr>
<tr>
<td>5. $(x \rightarrow \text{explicitflowstatement}) \vdash (\text{Taint}(x) \rightarrow \text{Taint}(\text{explicitflowstatement}))$</td>
<td>Axiom (2)</td>
</tr>
<tr>
<td>6. $F \vdash (\text{Taint}(x) \rightarrow \text{Taint}(\text{explicitflowstatement}))$</td>
<td>Monotonicity applied to 2</td>
</tr>
<tr>
<td>7. $F \vdash ((\text{Taint}(x) \rightarrow \text{Context}_\text{Taint}) \rightarrow (\text{Taint}(x) \rightarrow \text{Taint}(\text{explicitflowstatement})))$</td>
<td>∧-Introduction applied to 3 and 6</td>
</tr>
<tr>
<td>8. $F \vdash G$</td>
<td>Modus ponens</td>
</tr>
</tbody>
</table>

Table II

Formal proof of the first rule

Conversely, suppose that $F \models G$. Then $F \cup \neg G$ is unsatisfiable. By compactness, some finite subset of $F \cup \neg G$ is unsatisfiable. So there exists finite $F_0 \subset F$ such that $F_0 \cup \neg G$ is unsatisfiable and, equivalently, $F_0 \models G$. Since $F_0$ is finite, we can apply Theorem 2 to get $F_0 \vdash G$. Finally, $F \models G$ by Monotonicity.

B. The algorithm

The tainting algorithm that we propose, Taint_Algorithm, allows solving the under-tainting problem. It takes as input a control flow graph of a binary program. In this graph, nodes represent a set of instructions that represent basic blocks. Firstly, it determines the control dependency of the different blocks in the graph using Dependency_Algorithm [11]. Afterwards, we parse the Dependency_List generated by Dependency_Algorithm and we set the context taint of blocks to include the taint of the condition that depends on whether the branch is taken or not. Finally, using the context taint and the two propagation rules, we taint all variables to which a value is assigned in the conditional branch.

Algorithm 1 Taint_Algorithm (Control flow graph G)

Input: $G = (V, E, r)$ is a control flow graph of a binary program
Output: $\text{Tainted}_\text{Variables}_\text{List}$ is the list of variables that are tainted.

```
x \in V
y \in V

dependency_list \leftarrow \text{Dependency}_\text{Algorithm}(G)

while (x, y) \in dependence_list do
    set context_taint(x, y, list_context_taint)
    taint_variables_list \leftarrow taint_assigned_variable(y)
end while
```
C. Running example

We analyze the control flow graph $G = (V, E, r)$ (see Figure 5) of the bytecode given in Figure 4 to illustrate the operation of the $Taint$ Algorithm. The source code is given in Figure 3. The $Taint$ Algorithm takes as input the control flow graph $G = (V, E, r)$ where:

- $V = \{BB(1), BB(2), BB(3), BB(4)\}$
- $E = \{(BB(1), BB(2)), (BB(1), BB(3)), (BB(2), BB(4)), (BB(3), BB(4))\}$

The $Dependency$ Algorithm checks the dependency of the blocks in the control flow graph. It generates a $Dependency$ List $= \{(BB(1), BB(2)); (BB(1), BB(3))\}$. As, $BB(2)$ depends on $BB(1)$ and $BB(3)$ depends on $BB(1)$, the $Taint$ Algorithm sets context_taint of $BB(2)$ and context_taint of $BB(3)$ to condition taint in $BB(1)$. If $x = true$, the first branch is executed but the second is not. The first rule is used to taint modified variable $y$ in $BB(2)$: $Taint(y) = ContextTaint \oplus Taint(explicit)flowstatement$. The second rule is used to taint the variable $y$ in $BB(3)$ : $Taint(y) = ContextTaint \oplus Taint(y)$. So, all variables that depend on the condition will be tainted and stored in $Tainted$ Variables List whether the branch is taken or not and we do not have an under-tainting problem.

D. Properties of the algorithm

First, we prove the correctness of the $Taint$ Algorithm and then we prove its completeness.
Correctness proof for Set_Context_Taint
We include the taint of the condition in the context taint of the dependent blocks. As the condition taint is valid thus the inclusion operation is valid. We can conclude that Set_Context_Taint is correct.

Correctness proof for Taint_Assigned_Variable
We use the two propagation rules to taint variables to which a value is assigned. We proved the completeness of the two propagation rules in section V-A, thus we can conclude that Taint_Assigned_Variable is complete. Therefore, we can conclude the completeness of the Taint_Algorithm.

2) Completeness: Let us assume that the control flow graph is complete (see Definition 2). To prove the completeness of the Taint_Algorithm, we will prove the completeness of Dependency_Algorithm and Taint_Assigned_Variable.

The Dependency_Algorithm takes as input the post-dominator tree of the control flow graph. The post-dominator tree can be constructed using the complete algorithm defined in [10]. The Dependency_Algorithm is based on the set of the least common ancestor (L) of A and B in the post-dominator tree for each edge (A, B) in S. According to the value of L, Ferrante et al. define two cases to determine the control dependency. To prove the completeness of the Dependency_Algorithm, we show that Ferrante et al. prove that there does not exist another value of L (either A’s parent or A itself) to consider.

Proof: Let us assume that X is the parent of A in the post-dominator tree. So, X is not B because B is not an ancestor of A in the post-dominator tree (by construction of S). Ferrante et al. perform a proof reductio ad absurdum to demonstrate that X post-dominates B, and suppose it does not. Thus, there would be a path from B to STOP that does not contain X. But, by adding edge (A, B) to this path, a path from A to STOP does not pass through X (since, by construction, X is not B). This contradicts the fact that X post-dominates A. Thus, X post-dominates B and it must be an ancestor of B in the post-dominator tree. If X, immediate post-dominator of A, post-dominates B; then the least common ancestor of A and B in the post-dominator tree must be either X or A itself.

As only two values of L exist, there does not exist another case to compute the control dependency. The Case 2 captures loop dependency and all other dependencies are determined according to Case 1. Thus, Dependency_Algorithm is complete.

We proved the completeness of the two propagation rules in section V-A thus we can conclude that Taint_Assigned_Variable is complete. Therefore, we can conclude the completeness of the Taint_Algorithm.

E. Time complexity of the algorithm

The Dependency_Algorithm performs with a time of at most $O(N^2)$ where N is the number of nodes in the control flow graph. Linear time algorithm to calculate control dependencies have been proposed in [15] but no proof of correctness of this algorithm was given. For each (X, Y) examined in the Dependency_List, setting context taint and tainting variables can be done in constant time $O(N)$. Thus, the Taint_Algorithm requires linear time using algorithm defined in [15] and at most $O(N^2)$ using Ferrante et al. algorithm.

VI. RELATED WORK

Privacy issues on smartphones are a growing concern. Several works [16], [17], [18], [19], [20] have been proposed to control access to private data in mobile operating systems. These access control approaches do not track the flow of information and cannot prevent leakage of sensitive data. A number of researches have been proposed to prevent private information leakage by untrusted android applications. [21], [22], [23] substitute private data by fake information in the data flow. This can cause a problem and disrupt execution of applications. Enck et al. [3] implement dynamic taint analysis to track explicit flows on smartphones. AppFence [23] extends Taintdroid to implement enforcement policies. One limit of these approaches is that they cannot detect control flows because they use dynamic taint analysis. The static analysis approaches implemented in smartphones [24], [25], [26] allow detecting data leaks but they cannot capture all runtime configuration and input. We were inspired by these prior works, but we combine static and dynamic analysis to prevent sensitive information leakage by untrusted android applications. Some implementations exist in the literature to solve the under-tainting problem. BitBlaze [27] presents a novel fusion of static and dynamic taint analysis techniques to track implicit and explicit flow. DTA++ [28], 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. Trishul [29] correctly identifies implicit flow of information to detect a leak of sensitive information. Egele et al. [30] associate a taint label with the program counter to enhance dynamic taint-tracking technique. They identify unknown components like spyware and provide comprehensive reports on their behavior. Furthermore, these approaches do not formally give a proof to solve the under-tainting problem and are not implemented in smartphones application. Fenton [31] proposed a Data Mark Machine, an abstract model, to handle control flows. Fenton associates a security class to information and defines an interaction matrix to manipulate data in the system. He gives a formal description of his model and a proof of its correctness in terms of information flow. The Data Mark Machine is based on a runtime mechanism that does not take into account the implicit flow when the branch is not executed. This can cause an under-tainting problem. To solve this problem, Aries [32] 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 program counter. The Aries approach is based only on high and low security classes. Denning [33] enhances the run time mechanism used by Fenton with a compile time mechanism to solve the under-tainting problem. Denning inserts updating instructions whether the branch is taken or not to reflect the information flow. Denning and Denning [34] gave an informal argument for the soundness of their compile time mechanism. Graa et al. [4] propose an approach based on dynamic taint analysis that propagates taint along control dependencies using static analysis to track implicit flows in embedded systems such as the Google Android operating system. They define a set of formal propagation rules to solve the under-tainting problem but they do not prove the
completness of these rules. We draw our inspiration from the Denning approach, but we define formally a set of taint propagation rules to solve the under-tainting problem and we improve the approach of Graa et al. by proving the correctness and completeness of these rules.

VII. DISCUSSION

Cavallaro et al. [35] describe evasion techniques that can easily defeat dynamic information flow analysis. These evasion attacks can use control dependencies. They demonstrate that a malware writer can propagate an arbitrarily large amount of information through control dependencies. Cavallaro et al. see that it is necessary to reason about assignments that take place on the unexecuted program branches. We implement the same idea in our taint propagation rules. Unfortunately, this will lead to an over-tainting problem (false positives). The problem has been addressed in [28] and [36] but not solved though. Kang et al. [28] used a diagnosis technique to select branches that could be responsible for under-tainting and propagated taint only along these branches in order to reduce over-tainting. However a smaller amount of over tainting occurs even with DTA++, as we can see by comparing the "Optimal" and "DTA++" results in the evaluation. Bao et al. [36] define the concept of strict control dependencies (SCDs) and introduce its semantics. They use a static analysis to identify predicate branches that give rise to SCDs. They do not consider all control dependencies to reduce the number of false positives. Their implementation gives similar results as DTA++ in many cases, but is based on the syntax of a comparison expression. Contrariwise, DTA++ uses a more general and precise semantic-level condition, implemented using symbolic execution.

In our approach, we taint all variables in the conditional branch. This causes an over-tainting. But it provides more security because all confidential data are tainted. So, the sensitive information cannot be leaked. We are interested in solving the under tainting because the false negatives are much more dangerous than the false positives since the false negatives can lead to a false sense of security. We can try to reduce the over-tainting problem by considering expert rules.

VIII. CONCLUSION

Smartphones are extensively used. However, the use of untrusted android applications can provoke the leakage of private data by exploiting the under-tainting problem. In this paper, we propose a formal approach to detect control flow and to solve the under-tainting problem in android system. We formally specify the under-tainting problem. As a solution, we provide an algorithm based on a set of formally defined rules that describe the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm. In the future works, we will evaluate our approach in terms of overhead and false alarms. We will also test our approach and show that it resists to code obfuscation attacks based on control dependencies.

REFERENCES


