Protection against Code Obfuscation Attacks based on control dependencies in Android Systems

Mariem Graa*†, Nora Cuppens-Boulahia*, Frédéric Cuppens*, and Ana Cavalli†
*Telecom-Bretagne, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
Emails: {mariem.benabdallah,nora.cuppens,frederic.cuppens}@telecom-bretagne.eu
†Telecom-SudParis, 9 Rue Charles Fourier, 91000 Evry - France
Emails: {mariem.graa,ana.cavalli}@it-sudparis.eu

***Abstract*—In Android systems, an attacker can obfuscate an application code to leak sensitive information. TaintDroid is an information flow tracking system that protects private data in smartphones. But, TainDroid cannot detect control flows. Thus, it can be circumvented by an obfuscated code attack based on control dependencies. In this paper, we present a collection of obfuscated code attacks on TaintDroid system. We propose a technical solution based on a hybrid approach that combines static and dynamic analysis. We formally specify our solution based on two propagation rules. Finally, we evaluate our approach and show that we can avoid the obfuscated code attacks based on control dependencies by using these propagation rules.**

***Keywords*-Android system; Code obfuscation attacks; Control dependencies; Leakage of sensitive information; Information flow tracking; Propagation rules**

I. Introduction

Mobile devices such as smartphones are increasingly used in our daily lives. To satisfy smartphones user's requirements, the development of smartphone applications have been growing at a high rate. AppStore [1] contains more than half-a-million applications, and Android Market [2] has just crossed the two hundred thousand marks. Apple's AppStore applications have been tested for attacks, while the Android Market applications are available to users without any code review. We can see 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 can be used by an attacker that obfuscates code exploiting control dependencies to compromise the confidentiality and integrity of the Android system and can leak private information without user authorization. Therefore, there is a need to provide adequate security mechanisms that resist to the code obfuscation attacks based on control dependencies in third-party applications. TaintDroid [3] implements a dynamic taint analysis mechanism to track information flow in real-time and to control the handling of private data in smartphones. It can only track the explicit flows but not the control flows. Thus, it is not able to detect code obfuscation attacks based on control dependencies. In a previous work [4], we have proposed an enhancement of the TaintDroid approach that propagates taint along control dependencies to track implicit flows in smartphones. In this paper, we show that our approach can resist to code obfuscation attacks based on control dependencies in the Android system. We use correct and complete taint propagation rules (see [5] for a formal proof). The rest of this paper is organized as follows: section 2 presents the TaintDroid approach. Section 3 presents some code obfuscation attacks based on control dependencies that TaintDroid cannot detect. Section 4 discusses the related work that can be used to detect code obfuscation attacks based on control dependencies. Section 5 describes how our approach can resist to this type of attacks. We provide our evaluation of our approach in section 6. The limitations of our work are discussed in section 7. Finally, section 8 concludes with an outline of future work.

II. TaintDroid

Third-party smartphone applications can leak sensitive data and compromise confidentiality of Android systems. TaintDroid [3] is an extension of the Android mobile-phone platform, implemented in the Dalvik virtual machine (see Figure 1).

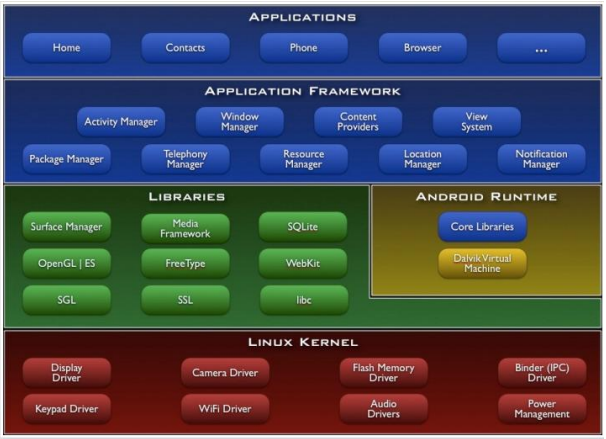


Figure 1. Android system architecture.TaintDroid is implemented in Dalvik VM (yellow)

It uses dynamic taint analysis to track explicit flows and to control the handling of private data on smartphones. It addresses different challenges specific to mobile phones like resource limitations. The TaintDroid process is summarized in Figure 2. First, it assigns taint to sensitive data (Device id, contacts, SMS/MMS). Then, TaintDroid tracks propagation of tainted data at the instruction level. Malicious application can interfere in the taint propagation level to untaint sensitive data which should be tainted and in taint sink level to leak these data. TaintDroid issues warning reports when the tainted data are leaked

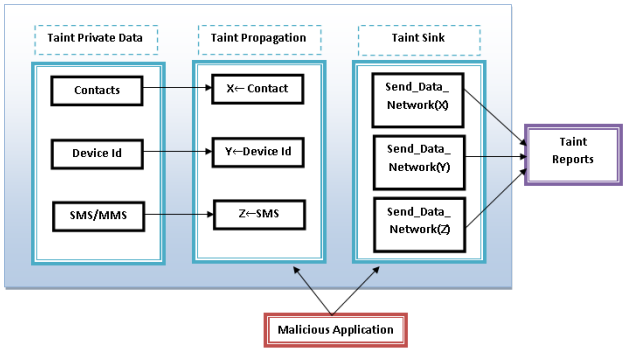


Figure 2. TaintDroid process

by malicious applications. This, can be detected when sensitive data are used in a taint sink (network interface). One limit of TaintDroid is that it cannot detect control flows. Thus, it cannot resist to code obfuscation attacks based on control dependencies. We now present different examples of code attacks based on control flows that TaintDroid cannot detect.

III. Code obfuscation attacks

Sarwar *et al.*[6] introduce the control dependence class of attacks against taint-based data leak protection. They evaluate experimentally the success rates for these attacks to circumvent taint tracking with TaintDroid. We present in this section examples of these obfuscated code attacks based on control dependencies that TaintDroid cannot detect. The taint is not propagated in the control flow statements. The attacker exploits untainted variable that should be tainted to leak private data.

Algorithm 1 Code obfuscation attacks 1

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  for each $s \in AsciiTable$ **do**
    if $(s == x)$ **then**
      $Y \leftarrow Y + s$
    end if
  end for
end for
$Send\_Network\_Data(Y)$

Algorithm 1 presents the first attack. The variable X contains the private data. The attacker obfuscates the code and tries to get each character of X by comparing it with symbols s in $AsciiTable$. He stored the right character founded in Y. At the end of the loop, the attacker succeeded to know the correct value of the $Private\_Data$ stored in Y. The variable Y is not tainted because TaintDroid does not propagate taint in the control flows. Thus, Y is leaked through the network connection.

Algorithm 2 presents the second attack. The attacker saves the private data in variable X. Then, he reads each character of X and converts it to integer. In the next loop, he tries to find the value of the integer by incrementing y. He converts the integer to character and concatenates all characters in Y to find the value of X. Thus, Y contains the $Private\_Data$ value but it is not tainted

Algorithm 2 Code obfuscation attacks 2

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $y \leftarrow 0$
  for $i = 0$ to n **do**
    $y \leftarrow y + 1$
  end for
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

because TaintDroid does not track control flow. Therefore, the attacker succeeds to leak the $Private\_Data$ value without any warning reports.

Algorithm 3 Code obfuscation attacks 3

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $y \leftarrow 0$
  while $y < n$ **do**
    $Try\{$
    $Throw\_New\_Exception()\}$
    $Catch(Exception\ e)\{$
    $Y \leftarrow Y + 1\}$
  end while
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

Algorithm 3 presents an obfuscated code attacks based on an exception. The variable n contains an integer value that corresponds to the conversion of a character in private data. The attacker raises an exception n times in the try bloc. He handles the thrown exception in the catch bloc by incrementing y to achieve the correct value of each character in $Private\_Data$. By concatenating the characters, Y contains the value of private data and Y is not tained because TaintDroid does not detect exceptions used in control flow. Thus, an attacker can successfully leak sensitive information by throwing exceptions to control flow. We present existing approaches that can be used to detect code obfuscation attacks based on control dependencies in the next section.

IV. Related Work

Obfuscation techniques are used in the Android platform to protect applications against reverse engineering [7]. In order to achieve this protection, the obfuscation methods transform the program code without changing its behavior. ProGuard [8] is applied to obfuscate program code and protect the Android application. In this paper, we study the obfuscation techniques used in malware context to evade detection of private data leakage in the android system.

Data Tainting is used to trace data propagation in a system. The principle of this mechanism is to "color" (tag) some of the data in a program and then spread the colors to other dependent objects. It is used for vulnerability detection, protection of sensitive data, and more recently, for analysis of binary malware. A vulnerability is detected when tainted data is used in a taint sink (network sink). Data tainting is implemented in interpreters [9],[10] to analyze sensitive data. It is used on dynamic analysis [11],[12],[13],[14] at binary level by instrumenting the code to trace and maintain information about the propagation. Thus, this mechanism suffers from a significant performance overhead that does not encourage their use in real-time applications.

Privacy issues on smartphones are a growing concern. Enck et al. [15] designed and implemented the Dalvik decompiler "ded", dedicated to retrieve and analyze the Java source of an Android Market application. The decompiler extraction occurs in three stages: retargeting, optimization, and decompilation. They identify class and method constants and variables in the retargeting phase. Then, they make bytecode optimization and decompile the retargeted .class files. Their analysis is based on automated tests and manual inspection. A slight current limitation of ded decompiler is that it requires the Java source code to be available. FLOWDROID [16] is a static taint analysis tool that automatically scans Android applications for privacy-sensitive data leaks. The static analysis approaches implemented in smartphones allow detecting data leaks but they cannot capture all runtime configurations and inputs, unlike dynamic analysis approaches.

TaintDroid [3] 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 resource limitations. AppFence [17] extends Taintdroid to implement policy enforcement. A significant limitation of these approaches is that they track only explicit flows and they cannot detect control flows. Thus, they cannot detect code obfuscation attacks based on control dependencies.

Cavallaro *et al.* [18] describe the 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.* show that it is necessary to reason about assignments that take place on the program branches. We implement the same idea in our taint propagation rules.

Some implementations exist in the literature to track control flows [19], [20], [21], [22]. They combine static and dynamic taint analysis techniques to correctly identify implicit flow of information and to detect a leak of sensitive information. DTA++ [21] presents an enhancement of dynamic taint analysis to track control flows. However DTA++ is evaluated only on benign applications and it is not tested on malicious programs in which an adversary uses implicit flows to obfuscate code. Furthermore, these approaches are not implemented in smartphones application and do not formally give a proof to resist to code obfuscation attacks based on control dependencies.

Fenton [23] proposed a Data Mark Machine, an abstract model, to handle control flows. Fenton gives a formal description of his model and a proof of its correctness in terms of information flow. Aries [24] 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 the program counter. The approach of Aries is based only on high and low security classes. Denning [25] enhances the run time mechanism defined by Fenton with a compile time mechanism to detect all control flows. The updating instructions are inserted whether the branch is taken or not to reflect the information flow. Denning and Denning [26] gave an informal argument for the soundness of their compile time mechanism. We draw our inspiration from the Denning approach, but we formally define a set of correct and complete taint propagation rules to avoid code obfuscation attacks. Graa et al. [4] propose an approach that combine dynamic taint analysis and static analysis to track control flows in embedded systems such as the Google Android operating system. But, this approach was not proven to resist to code obfuscation attacks based on control dependencies.

We were inspired by these prior works, but we combine static and dynamic analysis to avoid code obfuscation attacks based on control dependencies in the Android system. Precisely, we enhance the TaintDroid approach by propagating taint along control flow to taint all sensitive data. We describe our approach in more details in the following section.

V. Detecting obfuscated code attacks

The attacker exploits control dependencies to launch code obfuscation attacks because TaintDroid cannot propagate the taint tags through control flows. We aim to enhance the TaintDroid approach by tracking implicit flow in the Android system. To do so, we integrate a control flow module in the TaintDroid system. We use also a hybrid approach that combines static and dynamic analyses. TaintDroid does not taint assigned variables in control flow statements. So, we have an under-tainting problem. We formally specify two propagation rules to solve the under-tainting problem and to avoid the code obfuscation attack based on control dependencies. We present in the following our technical and formal approach.

A. Technical Approach Overview

In a previous work [4], we have proposed a technical approach that enhances TaintDroid by tracking control flow in the Android system. This approach combines the static and dynamic analyses. In a first step, we use static analysis to detect control dependencies. This is achieved by using the control flow graphs [27], [28] which will be analyzed to determine branches in the conditional structure. We assign a basic block to each control flow branch. Then,

we detect the flow of the condition-dependencies from blocks in the graph. Our approach allows us to handle not executed branches by detecting variable assignments in a basic block of the control flow graph. In a second step, we apply the dynamic analysis using information provided by the static analysis. The dynamic analysis allows tainting, in the conditional instruction, all variables which a value is assigned to. We create an array of context taints that includes all condition taints and we apply the propagation rules to correctly taint variables to which a value is assigned whether the branch is taken or not. We

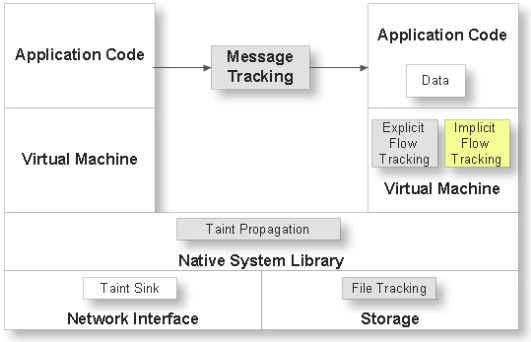


Figure 3. Modified architecture to handle implicit flow in TaintDroid system.

make a special exception handling to detect obfuscated code attacks based on an exception and to avoid leaking information. The catch block depends on the type of the exception object raised in the throw statement. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block. So, an edge is added in the CFG from the throw statement to the catch block to indicate that the throw statement will transfer control to the appropriate catch block. If an exception occurs, the current context taint and the exceptions taint are stored. The variables assigned in any of the catch blocks will be tainted depending on the exceptions taint. Each catch block has an entry in the context taint for this purpose.

To track control 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 created an array of variables to which a value is assigned to handle the branch that is not executed. Figure 3 presents the modified architecture to handle implicit flow in TaintDroid system.

In the following, we formally specify two propagation rules to avoid code obfuscation attacks based on control dependencies.

B. Formal Approach Overview

To launch a code obfuscation attack, an attacker exploits the under-tainting problem i.e. that some values should be marked as tainted, but are not. We formally specify the under-tainting problem and we present two propagation rules to solve it and to avoid the code obfuscation attack. We formally specify the under-tainting problem based on Denning's information flow model. However, we assign taints 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.

Definition. *Under-Tainting Problem*

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 $condition$ such that:

$$\begin{aligned} \textit{Is assigned}(x,y) \wedge Dependency(x, condition) \\ \wedge Tainted(condition) \wedge \neg Tainted(x) \end{aligned} \quad (1)$$

where:

- $\textit{Is assigned}(x,y)$ associates with x the value of y.

$$\textit{Is assigned}(x,y) \stackrel{def}{\equiv} \quad (x \leftarrow y)$$

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

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

Obfuscated code attack solution

To launch code obfuscation attacks, the attacker exploits untainted variables that should be tainted. We specify a set of formally rules that define the taint propagation and allow detecting the obfuscated code attacks based on control dependencies. By using these rules, all variables to which a value is assigned in the conditional branch are tainted whether the branch is taken or not. The taint of these variables reflects the dependency to a condition. We consider that $Context\_Taint$ is the taint of the $condition$.

- Rule 1: if the value of x is modified and x depends on the $condition$ and the branch is taken, we will apply the rule (2) to taint x.

where: The predicate $BranchTaken(br, conditionalst)$ specifies that branch br in the $conditionalstatement$ is executed. So, an explicit flow which contains x is executed. *IsModified (x, explicitflowst)* associates with x the result of an explicit flow statement.

$$IsModified(x, explicitflowst) \stackrel{def}{\equiv} (x \leftarrow explicitflowst)$$

- 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 rule (3) to taint x.

In a previous work [5], we gave a proof of the completeness of those rules. Also, we provided a correct and complete algorithm based on these rules that allows solving the under-tainting problem.

$$\frac{\textit{IsModified}(x,explicitflowst)\wedge Dependency(x,condition)\wedge BranchTaken(br,conditionalst)}{Taint(x)\leftarrow Context\_Taint\oplus Taint(explicitflowst)} \quad (2)$$

$$\frac{\textit{Is assigned}(x,y)\wedge Dependency(x,condition)\wedge\neg BranchTaken(br,conditionalstatement)}{Taint(x)\leftarrow Taint(x)\oplus Context\_Taint} \quad (3)$$

VI. Evaluation

To test the effectiveness of our approach, we have implemented the three obfuscated code attacks based on control dependencies presented in section III.

```
String X = contact_name;
String Y="";
char[] TabAsc;
 int k=0;
TabAsc = new char [96];

while (codeAsc < 0x80) {

  for (column = 0; column < 16; column++) {
     TabAsc[k] = codeAsc;
    codeAsc++;
    k++;
  }
  row++;
}

for (int i = 0;i< X.length(); i++)
{
    char x=X.charAt(i);

    for (int j=1;j<TabAsc.length;j++)
    {
         if (x==TabAsc[j])
         Y=Y+TabAsc[j];
    }

}

NetworkTransfer(Y);
```

Figure 4. Code obfuscation attack 1.

We have tested these attacks using a Nexus One mobile device running Android OS version 2.3 modified to track control flows. We use the Traceview tool to evaluate the performace of these attacks. We present both the inclusive and exclusive times. Exclusive time is the time spent in the method. Inclusive time is the time spent in the method plus the time spent in any called functions. We install the TaintDroidNotify application to enable notifications on the device when tainted data is leaked.

Let us consider the first obfuscated code attack (see Figure 4). The first loop is used to fill the table of ASCII characters. The attacker tries to get the private data (user contact name= 'Graa Mariem') by comparing it with symboles of Ascii table in the second loop. The taint of the user contact name is $((u4)0 \times 00000002)$.

The variable x is tainted because it belongs to the tainted character string X. Thus, the condition $(x == TabAsc[j])$ is tainted. Our system allows propagating the taint in the control flow. Using the first rule, Y is tainted and $Taint(Y) = Taint(x == TabAsc[j]) \oplus Taint(Y + TabAsc[j])$. We can show in the log file given in Figure 5(a) that Y is tainted with the same taint as the user contact name. A notification appears (see Figure 6(a)) reporting the leakage of Y that contains the value of private data. The execution of the first algorithm takes 88 ms as Inclusive CPU Time using Taintdroid modified to track control flows and 36ms in android not modified.

```
String X = Get_IMEI();
String Y="";
 for (int i = 0;i<X.length(); i++)
 {
     char x=X.charAt(i);
     int n =x;
     int y=0;
         for (int j=0;j<n;j++)
             {
                 y=y+1;
             }

         char c = (char) y;

         Y=Y+c;
 }

 NetworkTransfer(Y);
```

Figure 7. Code obfuscation attack 2.

The second obfuscated code attack is illustrated in Figure 7. The attacker tries to get a secret information X that is the IMEI of the smartphone. The taint of the IMEI is $((u4)0 \times 00000400)$. The variable x is tainted because it belongs to the character string X that is tainted. The result n of converting x to integer is tainted. Thus, the condition $(j = 0\ to\ n)$ is tainted. Using the first rule, y is tainted and $Taint(y) = Taint(j = 0\ to\ n) \oplus Taint(y + 1)$. In the first loop, the condition $x \in X$ is tainted. We apply the first rule, Y is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. This result is shown in the log file given in Figure 5(b). The leakage of the private data event is presented in the notification (see Figure 6(b)). The execution of the second algorithm takes 101 ms as Exclusive CPU Time using Taintdroid modified to track control flows and 20ms in unmodified android. The execution time in our approach is more important because it includes the time of the taint propagation in the control flow.

The third obfuscated code attack is illustrated in Figure 8. The attacker exploits exception to launch obfuscated code attacks and to leak sensitive data (phone number). The division by zero throws an ArithmeticException. This exception is tainted and its taint depends on the while condition $y < n$. Also, the while condition $(y < n)$ is tainted because the variable n that corresponds to the conversion of a character in $phone\_number$ is tainted. TaintDroid does not assign taint to exception. We define taint of exception($Taint\_Exception = ((u4)0 \times 00010000)$). Then, we propagate exception's taint in the

```
W/dalvikvm( 1209): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with tag
 0x2 data=[??
          Mariem Graa]
```

(a)

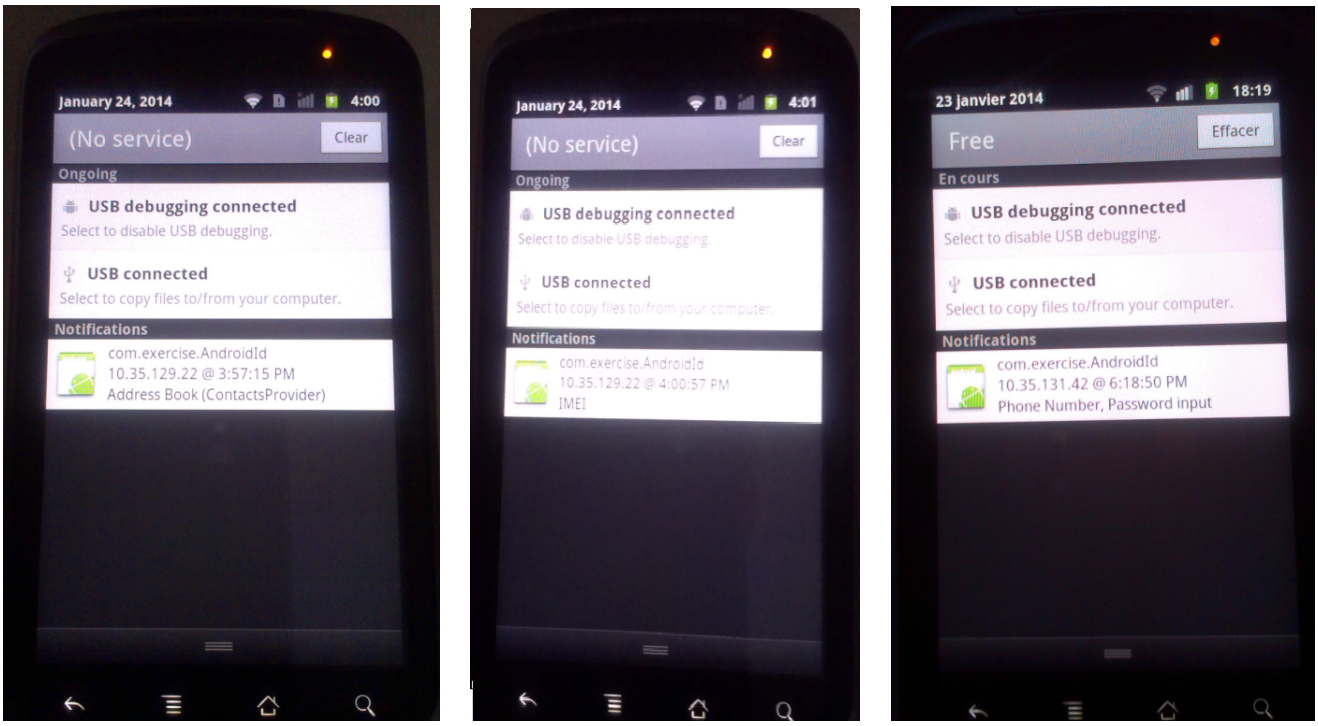
```
W/dalvikvm(  712): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with
 tag 0x400 data=[??354957033679070]
```

(b)

```
W/dalvikvm(  488): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with tag
 0x10008 data=[??
W/dalvikvm(  488): 3627890380]
```

(b)

Figure 5. Log files of Code obfuscation attacks



(a) (b) (c)

Figure 6. Notification reporting the leakage of sensitive data

catch block. We apply the first rule to taint y. We obtain $Taint(y) = Taint(exception) \oplus Taint(y+1)$. Finally, the string Y which contains the private data is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. In the log file given in Figure 5(c), we can show that the taint of Y is the combination of the taint of the exception($((u4)0 \times 00010000)$) and the taint of the phone number ($((u4)0 \times 00000008)$). A warning message appears indicated the leakage of sensitive information (see the notification in Figure 6(c)). The execution of the third algorithm takes 1385 ms as Inclusive CPU Time using Taintdroid modified to track control flows and 1437 ms in unmodified android. This difference is due to the taint propagation in the control flow.

VII. Discussion

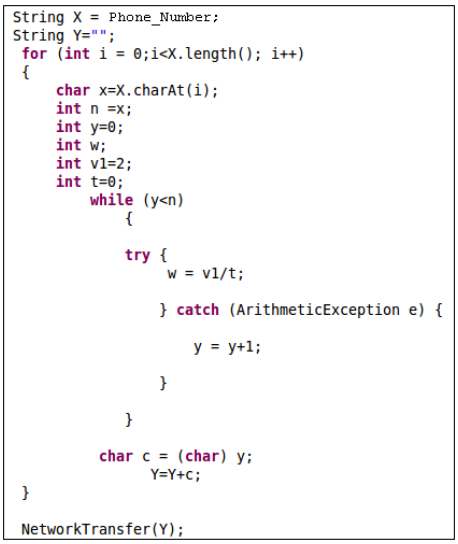
Side Channels

Our approach makes it possible to detect obfuscated code attacks applied in the control flow statement (if, loop, while, exception...) in order to leak sensitive information. But, it cannot detect all obfuscated code attacks.

Algorithm 4 Timing Attack

$X \leftarrow Private\_Data$
$n \leftarrow CharToInt(X)$
$StartTime \leftarrow ReadSystemTime()$
$Wait(n)$
$StopTime \leftarrow ReadSystemTime()$
$y \leftarrow (StopTime - StartTime)$
$Y \leftarrow Y + IntToChar(y)$
$Send\_Network\_Data(Y)$

The condition of the control flow statement includes a character of the private data. Most presented attacks need to be applied in a loop to leak one character at a time. A side channel attack is another category of attacks that can be used to obfuscate code and to leak private information.



```
String X = Phone_Number;
String Y="";
 for (int i = 0;i<X.length(); i++)
 {
     char x=X.charAt(i);
     int n =x;
     int y=0;
     int w;
     int v1=2;
     int t=0;
         while (y<n)
             {

             try {
                  w = v1/t;

                 } catch (ArithmeticException e) {

                     y = y+1;

                 }

             }

          char c = (char) y;
               Y=Y+c;
 }

 NetworkTransfer(Y);
```

Figure 8. Code obfuscation attacks 3.

It is based on information (timing information, power consumption,...) gained from a medium and used to easily extract the secret data. It is difficult to detect this category of attacks. Let us consider the timing attack presented in Algorithm 4. It is a side channel attack in which the attacker attempts to get private data by analyzing the difference in time readings before and after a waiting period. The sleep period duration is the value of the private variable.

Algorithm 5 Timing Attack 2

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $StartTime \leftarrow ReadSystemTime()$
  $Wait(n)$
  $StopTime \leftarrow ReadSystemTime()$
  $y \leftarrow (StopTime - StartTime)$
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

The difference in time y is not tainted because it does not depend on the tainted variables. It is assigned to Y that is leaked through the network connection. Our approach cannot directly detect this timing attack and the private information will be leaked without any warning report. However, this timing attack can be detected by tainting the system clock. Thus, the ReadSystemTime() function returns a tainted value. Therefore, the StartTime is tainted. Also, we propose to add rule that propagates the private data taint to the clock if Wait() function has a tainted parameter. Thus, the taint of StopTime includes the private data taint. Thus, the difference in time readings before and after a waiting period is tainted. Therefore, the attacker cannot get the value of private data using this timing attack. The same attack can be written differently (see Algorithm 5). We use a loop statement to get the private data. The loop condition is tainted and propagated in the loop block. Thus, we apply the first rule: Y is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + IntToChar(y))$. So, the private data cannot be leaked.

TaintDroid cannot track taint tags on Direct Buffer objects, because the data is stored in opaque native data structures. The side channel attack presented in Algorithm 6 exploits this limitation to leak private data. The memory buffer created is used to write a tainted variable at a specific address. Then, the attacker reads the content of the Direct Buffer specific address. The buffer contains private data but it is not tainted. Using our approach, we can avoid the leak of private data because Y will be tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + IntToChar(y))$.

Algorithm 6 DirectBuffer Attack

$X \leftarrow Private\_Data$
$D \leftarrow NewDirectBuffer()$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $DirectBufferWrite(n; D(0 \times 00))$
  $y \leftarrow DirectBufferRead(D; 0 \times 00)$
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

Note that our approach will not detect this side channel attack if the attack code is not included in a control statement. To detect this direct buffer attack, we need to refine our approach by adding a taint propagation rule that associates a private data taint to Direct Buffer objects at the execution of the DirectBufferWrite() function. This solution is similar to the one used in the Algorithm 4 to detect a side channel attack by tainting the clock.

False positives

In our approach, we taint all variables in the conditional branch. This, can lead to an over-tainting problem (false positives). The problem has been addressed in [21] and [29] but not solved though. Kang *et al.* [21] 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.* [29] 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.

Our approach can cause an over-tainting problem. 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 we consider that false negatives are more dangerous than false positives since false negatives can create security flaws. To balance (trade-off) between over-tainting and leakage of private information, we plan to apply expert rules that allow to reduce the over-tainting problem and protect private data. This represents a relevant extension of the approach presented in this paper.

VIII. Conclusion

In order to protect smartphones from obfuscated code attacks based on control dependencies, we have proposed a technical and formal approach that combines static and dynamic analysis. In this paper, we presented obfuscated attacks in control flow statements that exploit taint propagation to leak sensitive information. We formally specified two propagation rules to detect these attacks based on control dependencies. We showed how our approach can successfully avoid them. Thus, using our technique, malicious applications cannot bypass the Android system and get privacy sensitive information through obfuscated code attacks. Future work will be to characterize the performance of our enhanced android system that tracks control flows and compare it with the original android system. Also, we will test more complex conditional structures (nested control branches, switch,...) and other types of attacks based on control dependencies using our approach. Finally, we will plan to refine our apporach to reduce the number of false alarms.

References

[1] "Appstore," http://www.apple.com/iphone/apps-for-iphone/.

[2] "Android market," http://www.android.com/market/.

[3] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–6.

[4] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli, "Detecting control flow in smarphones: Combining static and dynamic analyses," in *Cyberspace Safety and Security*, pp. 33–47. Springer, 2012.

[5] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli, "Formal characterization of illegal control flow in android system," in *Signal Image Technology & Internet Systems*. IEEE, 2013.

[6] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices," in *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[7] Patrick Schulz, "Code protection in android," 2012.

[8] Eric Lafortune et al., "Proguard," 2006.

[9] L. Wall, T. Christiansen, and J. Orwant, *Programming perl*, O'Reilly Media, 2000.

[10] A. Hunt and D. Thomas, "Programming ruby: The pragmatic programmer's guide," *New York: Addison-Wesley Professional.*, vol. 2, 2000.

[11] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," Citeseer, 2005.

[12] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *ISCC'06. Proceedings. 11th IEEE Symposium on*. IEEE, 2006, pp. 749–754.

[13] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu, "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*. IEEE Computer Society, 2006, pp. 135–148.

[14] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007, pp. 116–127.

[15] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri, "A study of android application security.," in *USENIX security symposium*, 2011.

[16] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," 2014.

[17] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 639–652.

[18] Lorenzo Cavallaro, Prateek Saxena, and R Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 143–163. Springer, 2008.

[19] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song, "Dynamic spyware analysis," in *Usenix Annual Technical Conference*, 2007.

[20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," *Information Systems Security*, pp. 1–25, 2008.

[21] M.G. Kang, S. McCamant, P. Poosankam, and D. Song, "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.

[22] S.K. Nair, P.N.D. Simpson, B. Crispo, and A.S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 3–16, 2008.

[23] J.S. Fenton, "Memoryless subsystem," *Computer Journal*, vol. 17, no. 2, pp. 143–147, 1974.

[24] J. Brown and T.F. Knight Jr, "A minimal trusted computing base for dynamically ensuring secure information flow," *Project Aries TM-015 (November 2001)*, 2001.

[25] D.E.R. Denning, *Secure information flow in computer systems*, Ph.D. thesis, Purdue University, 1975.

[26] D.E. Denning and P.J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[27] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[28] Frances E Allen, "Control flow analysis," in *ACM Sigplan Notices*. ACM, 1970, vol. 5, pp. 1–19.

[29] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 13–24.