Data Leakage in Java applets with Exception Mechanism

Cinzia Bernardeschi[1], Paolo Masci[2], and Antonella Santone[3]

[1] Department of Information Engineering, University of Pisa, Italy
cinzia.bernardeschi@unipi.it
[2] Department de Informatica, Universidade do Minho, Braga, Portugal
paolo.masci@inesctec.pt
[3] Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy
antonella.santone@unimol.it

Abstract

It is becoming more and more important to study methods for protecting sensitive data in computer and communication systems from unauthorized access, use, modification, destruction or deletion. Sensitive data include intellectual properties, payment information, personal files, personal credit card and other information depending on the business and the industry. Therefore, data leakage is considered an emerging security threat to organizations and companies. In this paper we present a static analysis method for information flow analysis in Java bytecode with exceptions. Exceptions are special events that break the normal execution flow. They can be used as a device to leak high security data since exception throwing can be accurately driven. The proposed analysis is capable of tracing information flow caused by exceptions by identifying instruction handler protected instructions as virtual control instructions. A malicious Java applet that clones the user secret PIN through exceptions is shown.

Keywords: Information Flow, Data Leakage, Java Exceptions.

1 Introduction and Related Work

Data leakage is defined as the accidental or unintentional distribution of private or sensitive data to unauthorized entity [17]. Sensitive data include intellectual properties, payment information, personal files, personal credit card and other information depending on the business and the industry. Facing the challenges of data leakage requires complex and novel solutions.

Java applets downloaded from the Internet are handled by the Java Security Manager that assigns access privileges to their code and provides a customizable sandbox in which the Java bytecode runs. Nevertheless, the access-control mechanism does not protect data confidentiality, since accessed data could be unauthorizedly revealed by malicious applets to other applets that are not allowed to access the data. On the other hand, the control of data propagation entails studying the flow of information in the program [7]. For example, if data, besides having access rights, are also characterized by security levels reflecting their privacy level, to check data leakage, we need to verify that data with high security level are not transferred into lower security levels structures, since they could be read by low security observers.

This paper is an extension of the works [3], and checks data leakage in Java applets caused by exceptions by using the same approach. Exceptions are special events that break the normal execution flow. They can be used as a device to leak high security data since exception throwing can be accurately driven. The method is based on information flow analysis and abstract interpretation of the bytecode. Information flow analysis is considered one of the main techniques for studying security in computer systems. In particular, information flow properties are a particular class of security properties which aim at controlling the way information may flow among different entities. Assuming the security constraints to be known, the approach maps data confidentiality within a multi-level security model, where

private data are mapped to high security level and public data are mapped to low security level. Programs are then executed on security levels instead of real data. Programs are analyzed on a per-method basis with an iterative data-flow approach, and reaches a fix-point when a structure, called the *security context*, agrees for all methods. The *security context* contains information about the security levels by which each method has been verified and it is at the basis of the assume-guarantee principle used by the static analysis. The analysis is capable of tracing information flow caused by exceptions by identifying instruction handler protected instructions as virtual control instructions. Since the bytecode is unstructured, the region of influence of each protected instruction is computed by analysing the control flow graph of each method. A malicious Java applet that clones the user secret PIN through the exception mechanism has been considered as a proof of concept of our methodology. Although our study is a preliminary work, the results seem promising and a performance evaluation, larger discussions and greater experimentations are still in progress.

Several approaches have been developed for data leakage using information flow analysis. In [8] a dynamic mechanism for securing script executions by tracking information flow in JavaScript and its APIs is presented. In [10] JOANA is presented, which comes as an add-on for the Eclipse IDE. It is object, flow and context-sensitive as well and is used for analyzing Java bytecode for noninterference and information flow control. It can also detect leaks that result from the interleaving of different threads. However, JOANA requires annotations to be present in the code during development. In [11] the authors use Program Dependence Graphs (PDGs) for representing the flows of information found in Java bytecode and use a custom PDG query language to allow users to express their own application-specific security policies, without interfering with the development of the program. Many applications of information flow analysis have been proposed also for Android Applications. SCanDal [13], a sound and automatic static analyzer, exploits abstract interpretation to detect privacy leaks in Android apps. IccTA [14] uses a static taint analysis technique to find privacy leaks, i.e., paths from sensitive data, called *sources*, to statements sending the data outside the application or device, called *sinks*. A path may be within a single component or cross multiple components. IccTA relies on Epicc [16] and FlowDroid [1] to find data leaks between components of Android applications. They can both detect intra- and inter-component leaks within a single application or between multiple applications.

Our approach is based on abstract interpretation of the operational semantics, therefore the analysis can be fully automated. Moreover, the approach is modular, and it has been applied for checking leakages between packages in Java cards real applications [2], and to analyse data secure flow in Autosar security annotated models for automotive applications [6].

2 Exception handling in the Java language

Exceptions are special events used for signaling errors during the execution of a program. The rising of an exception is referred as *throwing*. Every time an exception is thrown, the Java runtime system breaks the standard execution flow of the program and calls the handler that catches and manages the exception. The correct handler for a given exception type can be found searching backwards through the call stack of the method: if no appropriate handler is found the program terminates.

Exceptions can be explicitly thrown by a program with the *throw* statement or they can automatically be thrown by the Java Virtual Machine whenever it detects an abnormal execution condition of a program (i.e., a Java programming language constraint violation) [9]. The exception type is specified by passing an object to the Java runtime system: it is an instance of the *throwable* class and it contains information about the program state and the kind of error occurred.

In the high level Java programming language exceptions are described with *try-catch* blocks: *try* blocks contain protected instructions, i.e., instructions whose exceptions are handled by the method, while *catch* blocks contain the body of implemented handlers. Instructions can be protected by more

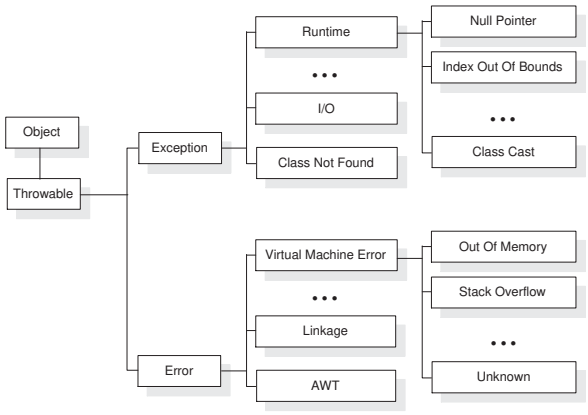

Figure 1: Java Exception Hierarchy (incomplete)

than one handler since any *try* block can be followed by many *catch* blocks, one for each possible thrown exception.

From a low-level point of view the bytecode contains the body of implemented handlers and an exception table which contains their descriptors. For each handler, the descriptors specify the type of the handled exception and the fragment of protected code. The handlers code is usually not directly targeted by any instruction of the method: the standard Java compiler follows this behavior.

2.1 Exception Types

In the Java platform the *Throwable* class is the superclass of all errors and exceptions (see Figure 1).

Java exceptions can be grouped into two categories: *checked* and *unchecked*. *Checked* exceptions represent error conditions that can be foreseen by the programmers. Code fragments containing this kind of exceptions must declare them and must have an appropriate handler: the Java compiler enforces these constraints. On the other hand, the *unchecked* exceptions are linked to conditions that should never happen during the execution of well-written programs (i.e., the access of array elements out of the array bounds) or to severe Virtual Machine errors (i.e., out of memory or stack overflow). They can be handled by exception handlers although it is not required and there is no compile-time checking. The Java programming language allows programmers to declare user-defined exceptions, of both *checked* and *unchecked* type.

3 The method

Data leakage is directly connected to the violation of the secure information flow in programs. Given a program P, in which each variable is assigned a security level, P has a secure information flow if, when P terminates, the value of each variable with low security level does not depend on the initial value of variables with high level. Let us suppose that variable `x` has a security level higher than the one assigned to variable `y`: examples of violation of secure information flow in high level languages are (*i*) `y=x` and (*ii*) `if (x==0) {y=1;} else {y=0;}`. In the first case there is an `explicit` information flow from `x` to `y`, while in the second case there is an `implicit` information flow. In both cases the final value of `y` reveals information on the value of the higher security variable `x`.

The proposed method checks for the secure information flow property in the Java platform by statically analysing the applet bytecode before its real execution within a multi-level security model. The

security model assigns to any applet input/ouput channel a security level. The files containing private information will be characterized by high security levels, while the public files will be characterized by low security levels. We split files into two disjoint sets of high and low security files. The secure information flow property determines whether highly secure (private) data are kept secret.

Definition 1. *Given a security policy and a terminating application, the application satisfies the secure information flow property if the contents of every low security output file do not depend on the contents of the high security input files.*

This property is also called *non-interference* in the literature. The analysis is based on the abstract interpretation approach proposed in [3], that has been proved to capture all the illegal flows of data. A notion of non-interference for real-time systems specified by timed automata has been also introduced in [4].

We trace data propagation in a Java applet by checking its methods, one by one, with a bytecode data-flow analysis on the domain of security levels. The information flow inside a method is propagated by inferring the security level of each Virtual Machine register (stack locations and local variables). The analysis directly catches the *explicit* information flow by observing the security level of instruction operands during the abstract execution of the code. The *implicit* information flow, instead, is propagated and caught by computing control regions [12]: each bytecode instruction is being verified under a *security environment* that takes into account the security level of all the control conditions an instruction is affected by. Data propagation caused by any method invocation and the access to common data structures in the heap is studied by executing each method inside a *security context*; the security context sets the security levels of object fields in the heap and the security level of input/output and return parameters of each method. The secure information flow analysis corresponds to an iterative verification of all methods within a common security context: it stops when a fix-point is reached.

Example of secure information flow violation are shown below. The bytecode fragment in the Listing 1 shows an explicit flow from `x` to `y`. In the bytecode, `x` is the register 1, while `y` is the register 2, respectively.

```
.........
4: iload_1    // load x on the stack
5: istore_2   // store the top of
              // the stack onto y
.......
```

Listing 1: Explicit flow

```
.....
6:  iload_1    // load x on the stack
7:  ifne 15    // jump if not equal
10: iconst_1   // load 1 on the stack
11: istore_2   // store the top of
               // the stack onto y
12: goto 17
15: iconst_0   // load 0 on the stack
16: istore_2   // store the top of
               // the stack onto y
.....
```

Listing 2: Implicit flow

The bytecode in Listing 2 shows an *implicit* flow from `x` to `y`. The final value of `y` is equal to 0 or 1 depending on the value of `x`.

We use the control flow graph and the notion of control region for modeling implicit information flows. Given a control instruction *i*, the control region of *i* is the set of instructions that can be executed conditionally according to *i*. For example, the control region of an `if` is the set of instructions in the two branches of the `if`, until the point at which the two branches join. In the analysis, the level of the implicit flow affecting an instruction is represented by the *security environment*.

Let us consider the control flow graph of the bytecode in Listing 2, shown in Figure 2 (b). Since the condition of the `if` at address 7 is *H* (`x` is a high variable), and the control region of 7 is $\{10, 11, 12, 15, 16\}$, both `iconst` instructions are executed in a high security environment. Therefore

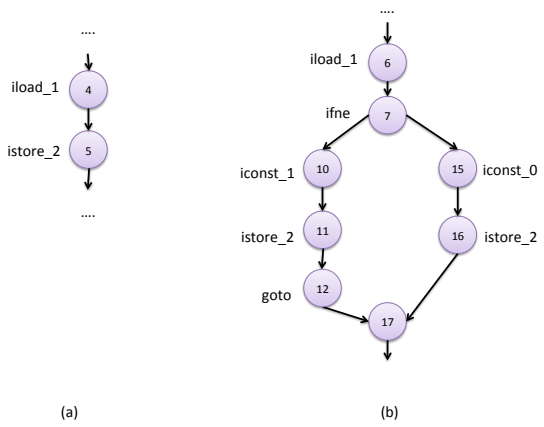

Figure 2: Control flow graph: (a) explicit flow, (b) implicit flow

a high value is pushed on the stack, and the high value is stored into the low security variable `y`.

An example of security context is the following, where *A* is a class with data member `f1` and method member `mt1`; *B* a class with data member `f2` and method member `mt2` and `mt3`. In particular, for each method, the security context maintains the level of parameters, return and the security environment.

```
SECURITY CONTEXT
classfields, arrays
A.f1 = L                    B.f2 = H
methods of class A          methods of class B
mt1(L, H)L; L               mt2(L)L; L
                            mt3(L)L; L
```

4 Information leakage with the exception mechanism

The Java code of a simple example with exceptions is shown in Listing 3 and the corresponding Java bytecode is shown in Listing 4.

```
1
2  import java.io.IOException;
3  public class Hello {
4    public static void main(String[] args) {
5    }
6
7    public void stampa(String s){
8          System.out.println("Outside");
9
10     try { System.out.println("Inside try");}
11     catch (ArithmeticException e) {System.out.println("Print catch");}
12
13    System.out.println("Print ...");
14    }
15 }
```

Listing 3: Simple Java Example

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1 //Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: return

  public void stampa(java.lang.String);
    Code:
       0: getstatic #2 //Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc       #3 //String Outside
       5: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: getstatic     #2 //Field java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #5 //String Inside try
      13: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
      16: goto          28
      19: astore_2
      20: getstatic     #2 //Field java/lang/System.out:Ljava/io/PrintStream;
      23: ldc           #7 //String Print catch
      25: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
      28: getstatic     #2 //Field java/lang/System.out:Ljava/io/PrintStream;
      31: ldc           #8 //String Print ...
      33: invokevirtual #4 //Method java/io/PrintStream.println:(Ljava/lang/String;)V
      36: return

    Exception table:
       from    to  target type
           8    16    19   Class java/lang/ArithmeticException
}
```

Listing 4: Simple Java Bytdecode Example

From the exception table, as shown in the bottom of Listing 4, we derive that instructions from `8` to `16` are executed in a protected way. Moreover, `19` is the first instruction of the exception handler. Instruction `8` is protected and it may throw an exception, that can be captured by an exception handler (executing the code at `19`).

Assume that a protected instruction is a control instruction that throws an exception depending on an high condition. The handler of the exception must be executed under a security environment that is high. The execution of the exception handler that captures the exception may reveal information on the value of the high condition, thus causing a leakage of information.

From this point of view, the handlers body can be thought as particular extensions of the methods code. An *extended control flow graph* can be defined as the graph obtained from the method graph augmented with edges starting from protected instructions to the first instruction of the protecting handlers. Figure 3 shows the extended control flow graph of the bytecode shown in Listing 4.

```
import java.io.IOException;

public class PinCloner {
  public static void main(String[] args) {
  }

  public void PinCloner(){
        try {
            int p;
```

```
10              for (...) { // reads a char from PIN_FILE and store it in p;
11              .....
12              if (p == 0) { throw  new ArithmeticException(); }
13              else { throw  new NullPointerException();}
14              }
15        }
16      catch (ArithmeticException e) { // write 0 in Clone_File
17                                      }
18      catch (NullPointerException e) {// write 1 in Clone_File
19                                      }
20          .....
21      }
22  }
```

Listing 5: PinCloner Java code

Under this point of view, the bytecode instructions act like virtual control instructions if they are contained in a section protected by handlers. This is due to the protected instructions run-time behavior: they can jump to their standard successors or to the starting of their exception handlers.

Notice that the exception handlers of a method can be exhaustively identified by inspecting the handler descriptor table of the method; moreover their code must be abstractly executed after all protected instructions since any execution flow has to be considered.

The region of influence of the protected instructions is computed on the *extended control flow graph*, highlighted by a box in Figure 3. Since the throw instruction in a method is similar to a return statement, the exceptions can be used as a vehicle to deliver information (on occurred errors) between methods. As well as for the return value of methods, the security context must contain identifiers for all the possible thrown exceptions of each method.

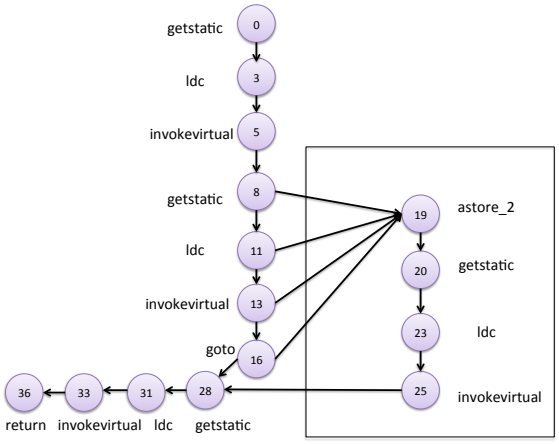

Figure 3: Extended control flow graph of the bytecode in 4

5 The *PINCloner* malicious applet

Let us consider the *PINcloner* applet, where `Pin_file` and `Clone_file` are the input and the output files, respectively. `Pin_file` is a private file containing a secret PIN (a sequence of 0/1 characters, for simplicity). Let us suppose that the *PINcloner* application can read from the private file. The applet clones the user secret PIN with the exception mechanism. After every character has read, it will be written in a the public file by the handler of the exception.

In particular, the *PINCloner* application clones the characters of the *Pin_file* by throwing different kind of exceptions depending on the value read (*NullPointerException* and *ArithmeticException*). The exceptions are thrown directly with a *throw* statement. A fragment of the source code is shown in Listing 5 and the bytecode in Listing 6.

```
Compiled from "PinCloner.java"
public class PinCloner {
  public PinCloner();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: return

  public void PinCloner();
    Code:
       0: iconst_1
       1: istore_1
       2: iload_1
       3: ifne          14
       6: new           #2 // class java/lang/ArithmeticException
       9: dup
      10: invokespecial #3 // Method java/lang/ArithmeticException."<init>":()V
      13: athrow
      14: new           #4 // class java/lang/NullPointerException
      17: dup
      18: invokespecial #5 // Method java/lang/NullPointerException."<init>":()V
      21: athrow
      22: ......          // write 1 in Clone_File
      31: goto          43
      34: ...             // write 1 in Clone_File
      43: return

    Exception table:
       from    to  target type
           0    22    22   Class java/lang/ArithmeticException
           0    22    34   Class java/lang/NullPointerException
}
```

Listing 6: PinCloner Java Bytecode

The Exception table, at the bottom of Listing 6, shows that the instructions from `0` to `22` are protected by two exception handles starting at instruction `22` and instruction `34`, respectively.

Instruction `3` is an `if` with four successors: the natural successors, plus the two entry points of the exception handlers. The control region of `3` includes the instructions of the exception handlers, and consequently these instructions are executed in a security environment given by the condition of the `ifne`. Since the condition depends on the 0/1 value of PIN character read from a high security file, the

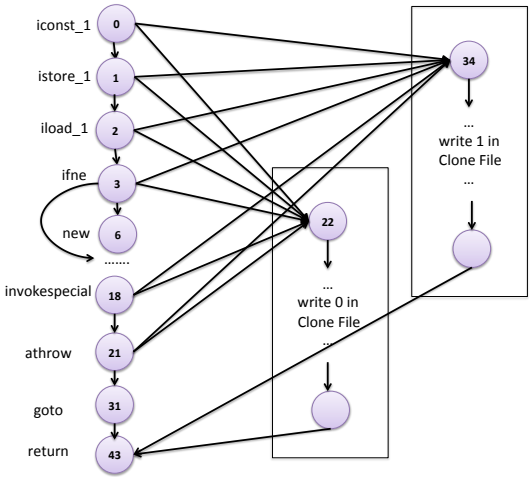

Figure 4: Extended control flow graph of the bytecode in 6

implicit flow is high. The handler of the exception, write such value into the low security `Clone_file` file. The application violates the secure information flow because high security data are written on a public file and our methodology successfully detects this data leakage.

6 Conclusion and Future Work

In this paper a static information flow based method for data leakage detection has been presented. The method applies to Java bytecode with exceptions. Exception are special events that break the normal execution flow. They can be used as a device to leak high security data since exception throwing can be accurately driven. The proposed analysis is capable of tracing information flow caused by exceptions by identifying instruction handler protected instructions as virtual control instructions. A malicious Java program that clones the user secret PIN through exception has been analysed, with encouraging results. The approach scales up because methods are analysed one at a time until the fixpoint is reached. The main drawback of the approach is that not all secure programs are accepted because of the approximation of the abstract interpretation approach. For example, when the same method is invoked at different points, it is verified under the highest security level for parameters and returns. This causes the detection of false illegal flows. As future work, we plan to extend our methodology for Android applications. The idea is to extract the class files of an Android application, using, for example, dex2jar[1] tool in order to convert the `dex` (i.e., the Dalvik Executable) file into `jar` (i.e., Java Archive) file. Using the dex2jar tool, we obtain the Android application in a compressed format. To extract the classes from the `jar` file, we can use the command: `jar -xvf` provided by the Java Development Kit. The outputs of this package are the classes relative to the the Android application and our methodology is directly applicable [15, 5].

[1] https://sourceforge.net/projects/dex2jar/

References

[1] 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. *ACM SIGPLAN Notices*, 49(6):259–269, 2014.

[2] Marco Avvenuti, Cinzia Bernardeschi, Nicoletta De Francesco, and Paolo Masci. Jcsi: A tool for checking secure information flow in java card applications. *Journal of Systems and Software*, 85(11):2479–2493, 2012.

[3] Roberto Barbuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Analyzing information flow properties in assembly code by abstract interpretation. *The Computer Journal*, 47(1):25–45, 2004.

[4] Roberto Barbuti, Nicoletta De Francesco, Antonella Santone, and Luca Tesei. A notion of non-interference for timed automata. *Fundam. Inform.*, 51(1-2):1–11, 2002.

[5] Pasquale Battista, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Identification of android malware families with model checking. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy, ICISSP 2016, Rome, Italy, February 19-21, 2016.*, pages 542–547. SciTePress, 2016.

[6] Cinzia Bernardeschi, Marco Di Natale, Gianluca Dini, and Maurizio Palmieri. Verifying data secure flow in autosar models by static analysis. In *1st International Workshop on FORmal methods for Security Engineering*, pages 704–713, 2017.

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

[8] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. Secure information flow verification with mutable dependent types. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 6. ACM, 2017.

[9] J Gosling, B Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[10] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[11] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 291–302. ACM, 2015.

[12] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time.

[13] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[14] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 280–291. IEEE Press, 2015.

[15] Francesco Mercaldo, Vittoria Nardone, Antonella Santone, and Corrado Aaron Visaggio. Download malware? no, thanks: how formal methods can block update attacks. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2016, Austin, Texas, USA, May 15, 2016*, pages 22–28. ACM, 2016.

[16] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[17] Asaf Shabtai, Yuval Elovici, and Lior Rokach. *A Survey of Data Leakage Detection and Prevention Solutions*. Springer Publishing Company, Incorporated, 2012.