Article

DroidFDR: Automatic Classification of Android Malware Using Model Checking

Zhi Yang [1,2], Fan Chao [1,2,*], Xingyuan Chen [1,2,3,*], Shuyuan Jin [4], Lei Sun [1,2] and Xuehui Du [1,2]

1 Zhengzhou Information Science and Technology Institute, PLA Information Engineering University, Zhengzhou 450001, China; zynoah@163.com (Z.Y.); sun_lei21@126.com (L.S.); duxuehui0411@126.com (X.D.)
2 Henan Province Key Laboratory of Information Security, Information Engineering University, Zhengzhou 450001, China
3 State Key Laboratory of Cryptology, Beijing 100084, China
4 School of Computer Science and Engineering, SUN YAT-SEN University, Guangzhou 510006, China; jinshuyuan@mail.sysu.edu.cn
* Correspondence: chaofan0411@163.com (F.C.); chxy302@vip.sina.com (X.C.)

Abstract: Android faces an increasing threat of malware attacks. The few existing formal detection methods have drawbacks such as complex code modeling, incomplete and inaccurate expression of family properties, and excessive manual participation. To this end, this paper proposes a formal detection method, called DroidFDR, for Android malware classification based on communicating sequential processes (CSP). In this method, the APK file of an application is converted to an easy-to-analyze representation, namely Jimple, in order to model the code behavior with CSP. The process describing the behavior of a sample is inputted to an FDR model checker to be simplified and verified against a process that is automatically abstracted from the malware to express the property of a family. The sample is classified by detecting whether it has the typical behavior of any family property. DroidFDR can capture the behavioral characteristics of malicious code such as control flow, data flow, procedure calls, and API calls. The experimental results show that the automated method can characterize the behavior patterns of applications from the structure level, with a high family classification accuracy of 99.06% in comparison with another formal detection method.

Keywords: Android; malware detection; communicating sequential processes; formal method; model checking

check for updates

Citation: Yang, Z.; Chao, F.; Chen, X.; Jin, S.; Sun, L.; Du, X. DroidFDR: Automatic Classification of Android Malware Using Model Checking. *Electronics* **2022**, *11*, 1798. https://doi.org/10.3390/electronics11111798

Academic Editor: Jemal Abawajy

Received: 6 April 2022
Accepted: 3 June 2022
Published: 6 June 2022

1. Introduction

The Android operating system, one of the most popular mobile platforms today, accounts for more than half of the mobile platform market share. Despite this success, Android devices continue to face significant security risks. The open ecosystem of Android is convenient for application programming; however, it increases the number of vulnerabilities that can be exploited by malware. To conduct unlawful activities covertly without the user's knowledge, malware from different families can be infiltrated into the market through various channels. Once installed, users may face various security threats such as malicious payment, privacy theft, remote control, malicious spread, expense consumption, system destruction, fraud and deceptive practices, and rogue behavior.

Android malware detection is a necessity. Common anti-virus software mainly uses a fingerprint identification method, which is quick and simple [1]. Whether the software to be analyzed is malicious is determined by searching and matching the unique characteristics of known malicious code, such as classes, methods, and data. However, code obfuscation attacks cannot be effectively prevented this way, because the fingerprints of malicious code variants will be unrecognized because of the changes due to obfuscation.

Many existing studies have focused on machine learning-based detection methods [2–22], which usually have good performance. In this approach, the features of known malicious code, such as application components, intent filters, permissions, and API calls, are first extracted and then used to train a detection model, which is finally used to classify the test samples. A disadvantage of machine learning is that it mainly extracts statistical features, which cannot accurately capture the behavioral and structural characteristics of software, such as function call relationship and control flow logic. These characteristics are important for judgment in some cases. For example, some malware can steal user text messages by intercepting and sending them; however, texting applications also have the function of sending and receiving SMS. Without a behavioral correlation analysis it is difficult to distinguish between benign and malicious applications by only collecting statistical characteristics of whether to send and receive text messages. Another major shortcoming of machine learning is its lack of interpretability. Although we can confirm whether a program is malicious, we may not know its essential characteristics or working principle; this makes it difficult to conduct subsequent evaluation and analysis.

The information flow analysis [23–34] is another type of mainstream method whereby malicious behaviors, such as privacy theft, hijacking, and deception, are detected by calculating the reachability of information flow to search for potential leaks and penetration paths. Although this method analyzes the behavior of information flow, the tracking overhead is considerable. It is also prone to path explosion and incorrect judgment, e.g., certain risky paths will not appear in practice because of the various control parameters and conditions. The root cause is that the method typically does not consider or simplify the influence of control flow when analyzing the reachability of the data flow.

Compared with the methods mentioned above, formal methods [35–41], are mathematically based languages, techniques and tools for detection. They can model complex program structures and capture the overall behavioral characteristics of malicious code. There are few formal detection methods for Android malware, of which LEILA [41] is an excellent one that can capture the call structure of important function call relation. However, we consider LEILA to have some drawbacks, including complex code modeling, incomplete and inaccurate expression of family properties, and excessive manual participation. We make a more comprehensive abstraction of malware behavior with reference to LEILA.

On the basis of communicating sequential processes (CSP) [42], which are well-known process algebra, we propose an effective method for the formal detection of Android malware, called DroidFDR. It converts an APK file into Jimple, an intermediate representation in Soot [43], for easy analysis, and then formally models the code into a CSP process to express the behavior pattern of the application. An automated algorithm for family property extraction is conceived to abstract family-specific property from malware of the same family. After the process that expresses the behavior of a sample to be tested is simplified, it is inputted to the FDR tool [44] for model checking against the process that abstracts the property of a family. Based on whether the sample has the property of any malware family, the appropriate classification is made.

To summarize, the main contributions of this paper are:

1. Being first to model the Jimple code with CSP, whose few statements and simple forms facilitate a quick and accurate formal modeling;
2. Capturing the characteristics of malicious code such as control flow, data flow, procedure calls, and API calls, which makes it possible for a more comprehensive behavior analysis with a higher accuracy;
3. An automatic property extraction algorithm for malware family, which significantly improves the efficiency and avoids the possibility of human error;
4. Explaining the family behavior to reveal its working mechanism, thus supporting the evaluation and analysis of malicious code.

The remainder of this paper is organized as follows. Section 2 presents discussions in the context of existing studies. Section 3 briefly reports the preliminaries on DroidFDR. Based on the system design given in Section 4, Section 5 introduces the modeling of different Jimple statements into CSP processes. Section 6 presents a novel algorithm to automatically extract the properties of malware families. Section 7 shows the principle of model checking in FDR. We present the results of the experiment and some analysis in Section 8. Finally, Section 9 concludes the paper.

2. Related Work

The rapid increase in the number of malware has promoted research interest in developing security tools. Malicious behavior around Android mainly includes malicious payment, privacy theft, remote control, malicious spread, expense consumption, system destruction, fraud and deceptive practices, rogue behavior, etc. Although the Android framework has some in-built security designs, such as sandbox isolation and permission mechanisms, these are insufficient to prevent malware, which are increasingly hidden and have sophisticated means of penetrating devices. Studies in the field of Android security have focused on malware detection, mainly analyzing the characteristics of malicious behavior from aspects such as inter-component communication (ICC) mechanisms, reflection mechanisms, callback mechanisms, and human–computer interaction.

With the development of artificial intelligence technologies, machine learning-based malware detection is gradually being explored [2], with the support vector machine (SVM) being one of the typical models used for classification [3–7]. The mostly used features for this type of detection include application components, intent filters, permissions, API calls, process calls, source codes, and annotations [8], etc. Deep learning-based detection has been rapidly promoted because of the deep connection mining between features [9]. Detection systems based on deep learning can be classified in terms of the type of features being extracted: dynamic features [10–12], static features [13–16], and mixed features [17–20]. In these studies, deep networks, such as deep neural network (DNN), convolutional neural network (CNN), and deep belief network (DBN), have seen a wide range of applications. To solve the model aging problem, SDAC [21] has evolved effectively by evaluating new APIs' contributions to malware detection according to the contributions of existing APIs. Machine learning methods exhibit a good detection performance and are accurate as well. However, most of the proposed ML-based methods are black-box. Given an app, they only output a family identity or none to which the app belongs through complex computing that can hardly be understood by humans. Interpreting what a classification model has learned should be as important as the detection accuracy, since an explainable detection can help researchers to inspect patterns in malware and gain a deeper insight into its functionality. There are few studies in interpretable Android malware classification. Drebin [3], XMal [13], and MAMBA [22], the state-of-the-art interpretable solutions, can identify features contributing to the detection of malware and provide the semantics of each feature. Moreover, XMal produces a semantic order of selected key features as a behavioral explanation, while the order rule is made manually; something which is labor-intensive and possibly prone to error. To explain the malicious behavior, MAMBA collects rich information and knowledge about attack behaviors from the open-source intelligence (OSINT); something which also requires a lot of manual support and maintenance.

Information flow analysis (IFA), which is an important method used to analyze software, cannot detect malicious behavior by itself. It needs to be combined with security policies to discover possible attacks such as theft, deception, and injection. TaintDroid [23] is a dynamic system-wide information flow tracking tool that can simultaneously track multiple sources of sensitive data. TaintDroid achieves high efficiency by integrating four granularities of taint propagation, namely variable-level, message level, method level, and file level. FlowDroid [24] performs a static taint analysis on an application. FlowDroid adequately models Android-specific challenges, like the application lifecycle or callback methods, which helps reduce missed leaks or false positives. FlowDroid performs taint analysis within single components of Android applications. Amandroid [25] and IccTA [26] perform an inter-component communication taint analysis to detect ICC-based privacy leaks. Bianchi et al. [27] adopted a backward data-flow analysis to track the information flow of GUI-related APIs and found dangerous system calls that were aimed at security risks surrounding touchscreen interfaces such as click hijacking, and phishing attacks with a view overlay replacement. InputScope [28] uses a static taint analysis to mark user input and monitor its propagation to identify user input validations that can expose hidden secrets such as backdoors and blacklists. Path tracking [29], taint value graph analysis [30], and hybrid analysis [31] are proposed to quickly detect malicious privacy leakage. IFA itself does not provide an automatic judgment of whether the revealed information flow path is malicious, and what kind of information disclosure is malicious needs to be specified by experts. Some machine learning-based detection methods apply information flow as a feature for malware classification. Appcontext [32] can extract the source/sink [33] of information flow to draw a feature vector and can distinguish between normal and malicious behaviors with the SVM after using FlowDroid to track and analyze Android programs. On this basis, Shen et al. [34] refined the feature vector of the information flow path into multiple API call flows to learn to discover malicious privacy leakage during the classification, while ICCDetector [5] was used to extract the ICC features of the malicious code for classification. Although the combination of ML and IFA can realize automatic classification, it still faces difficulties in explanation, that is, they cannot explain which information flows are decisive for the result. The weights of information flow features and the relationship between information flow features needs to be further studied.

Formal methods have been widely used in hardware circuit verification and security protocol analysis. In addition, researchers have explored the role of formal methods in the detection of Android malware. However, this type of work is relatively limited; studies have mainly modeled the programs formally for model checking [35–41]. Song et al. [35] leveraged a PushDown system to model the Smali code of Android applications, and expressed malicious behavior through a computational tree logic or a linear temporal logic, only to discover private data leakage in malware. Bai et al. [36] built a general framework, namely DROIDPF, to verify Android applications against security properties. Since DROIDPF focuses on common security vulnerabilities in applications, a state machine model corresponding to each vulnerability needs to be established manually. Mercaldo et al. published a series of research results around the detection and classification of malware families. They detected families related to ransomware, update attacks, HummingBad, and repackaging, separately in [37–40], based on which a general classification method for malware families was proposed in [41]. The problems addressed in these studies were solved through a consistent procedure: modeling of the Java bytecode based on CCS [45], defining the selective mu-calculus formulae [46] to encode the temporal logic properties of malware families [47], and obtaining the family classification through model checking [48]. However, this method has problems such as complex code modeling, incomplete and inaccurate expression of family properties, such as data flow, and excessive manual participation. In particular, each interpretable malware behavior expressed in logic formulae is pre-defined manually before use to verify whether a program is malicious, which can be a rather complex task even for security experts. We continue this line of formal methodology and address the challenges to that which LEILA declared. In DroidFDR, the model-checking process is automated by formal tools through mathematical derivation. The comparison of the related work above is shown in Table 1.

Table 1. Comparison with related work.

Category	Examples	Features	Interpretability	Family Classification
ML-based detection	[3]	Components, intent filters, permissions, API calls	Only provide the top features that contribute to the classification as a behavioral explanation	Y
	[13]	Permissions, API calls	Produce the semantic order of selected key features, while the order rule is made manually	Y
	[22]	API calls	Select explanation template from OSINT, which also requires a lot of manual support	Y
	[5–7,10–12,14,15,20,21]	One or more of the features of components, intent filters, intent, API calls	Not supported	Y
	[8,16–19]	One or more of the features of components, intent filters, permissions, API/function calls, annotations	Not supported	N
Information flow analysis	[23–31]	data flows	Not supported	N
	[4,32,34]	data flows	Not supported	Y
Formal methods	[41,47,48]	API calls, control flows function call	Output fine-grained behavior descriptions, but express family properties manually	Y
	Ours	API calls, control flows, function call, and data flows	Output fine-grained behavior descriptions, and extract family properties from samples automatically	Y

3. Preliminaries

3.1. CSP

CSP is a process algebra used for specifying systems as a set of parallel state machines that can sometimes synchronize on events. We briefly review the CSP herein, borrowing heavily from Hoare's book [42].

The basic operator is the prefix operator. If x is an event, and P is a process, then $(x \rightarrow P)$ represents a process that engages in event x and then behaves like process P. For any CSP process P, we can have a trace of events that P may accept.

The next important operator is "choice", denoted by "$\mid$". If x and y are distinct events, then $(x \rightarrow P \mid y \rightarrow Q)$ denotes a process that accepts x and then behaves like P or accepts y and then behaves like Q.

The next class of operators relates to parallelism. The notation $P \underset{A}{||} Q$ represents P running in parallel with Q, synchronizing on events in A. This means that a stream of incoming events can be arbitrarily assigned to either P or Q, assuming that these events are not in A. However, for events in A, both P and Q must accept them synchronously.

A variation of parallel composition is interleaving, denoted by $P|||Q$. In interleaving, P and Q never synchronize; they operate independently. $P|||Q$ is therefore equivalent to $P \underset{\{\}}{||} Q$, which means that P and Q run in parallel and synchronize on the empty set.

Processes that run in parallel can communicate with one another over channels. A typical channel c can carry various values v, denoted by $c.v$. This represents a sending process that accepts the event $c!v$ while a receiving process accepts the event $c?x$ (where x is thus far unbound) and sets x to v. Communication on a channel is possible only when the sender and receiver processes are in their respective states simultaneously. If one process is in the suitable state and the other is not, the ready process waits until its partner becomes ready. If the channel has a compound name such as $i.c$, its values are respectively denoted by $i.c.v$. Channel names are prefix-free, so this is never ambiguous.

The next important CSP feature is concealment. For a process P and a set of symbols C, the process $P \backslash C$ is P with symbols in C concealed. The events in C become internal transitions that cannot be observed by other processes through synchronization or channel communication.

In addition, CSP provides some useful predefined processes such as *STOP* which accepts no events.

The operational behavior of CSP processes can be considered a labeled state transition system. Figure 1 shows the operational semantics of CSP when the process P can be transferred from one state to another through the execution of event a, where the contents above and below the horizontal line are hypothesis and conclusion, respectively, and the bracket on the right represents an additional condition.

$$\text{Prefix} \quad \frac{}{a \rightarrow P \xrightarrow{a} P} \qquad \text{Concurrency} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q'}$$

$$\text{Choice} \quad \frac{}{a \rightarrow P \mid b \rightarrow Q \xrightarrow{a} P} \qquad \text{interleaving} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{b} Q'}{P \parallel Q \xrightarrow{a} P' \parallel Q} \; [a \neq b]$$

Figure 1. Operational semantics of CSP.

3.2. Soot

Soot is a Java bytecode optimization framework developed by the Sable research group from McGill University. Its original objective was to help analysts gain a better understanding and faster execution of Java programs. So far, Soot has played an important role in the field of program analyses such as class files parsing, points-to analysis, null-pointer analysis, data-flow analysis, call graph construction, and control-flow graph extraction. A new custom analysis can also be built on the basis of the Soot framework.

To better serve different analyses, Soot provides four intermediate representations with different levels of abstraction for Java bytecode: Baf, Jimple, Shimple, and Grimp. Jimple is the main intermediate representation of Soot, which is a typed, stackless three-address representation suitable for most analyses and optimization. Jimple will eliminate redundant parts of the original code, e.g., variables or assignments never used. The key to convert the original code into Jimple is the linearization of expressions and the naming of variables, because each Jimple statement only refers to at most three variables or constants. There are more than 200 types of statements in Java bytecode, whereas Jimple has only 15 statements; therefore, converting code into Jimple can make an analysis more convenient and efficient.

3.3. Jimple

As an example, Figure 2 shows the source code of a Java program and its Jimple representation obtained by Soot conversion. As shown in Figure 2a, the method example belongs to the class Test, and it calls the method bar of the class *Foo* for calculation. Figure 2b presents the Jimple code generated by Soot for this method. From a formal viewpoint, Jimple can be considered a language that combines Java source code and Java bytecode. In terms of the declaration and assignment of variables, the statement-based structure that belongs to Java source code can be identified, while Jimple is similar to Java bytecode in terms of the control flow and method invocation.

```
[1]   public int example(int x) {
[2]       Foo f = new Foo();
[3]       int y = f.bar(x) + 1;
[4]       if(y < 0) {
[5]           System.out.println("hello");
[6]           return y * y;
          }
[7]       else
[8]           return y * 5 + 16;
      }
```

(**a**) Java source code

```
[1]  public int example(int) {
[2]      Test r0;
[3]      int i0, $i1, i2, $i3, $i4, $i5;
[4]      Foo $r1, r2;
[5]      java.io.PrintStream $r3;

[6]      r0 := @this: Test;
[7]      i0 := @parameter0: int;

[8]      $r1 = new Foo;
[9]      specialinvoke $r1.<Foo: void <init>()>();
[10]     r2 = $r1;
[11]     $i1 = virtualinvoke r2.<Foo: int bar(int)>(i0);
[12]     i2 = $i1 + 1;

[13]     if i2 >= 0 goto label0;
[14]     $r3 = java.lang.System.out;
[15]     $r3.println("hello");
[16]     $i3 = i2 * i2;
[17]     return $i3;

[18]     label0:
[19]         $i4 = i2 * 5;
[20]         $i5 = $i4 + 16;
[21]         return $i5;
     }
```

(**b**) Jimple representation

Figure 2. Different representations of an example program: (**a**) Java source code; (**b**) Jimple representation.

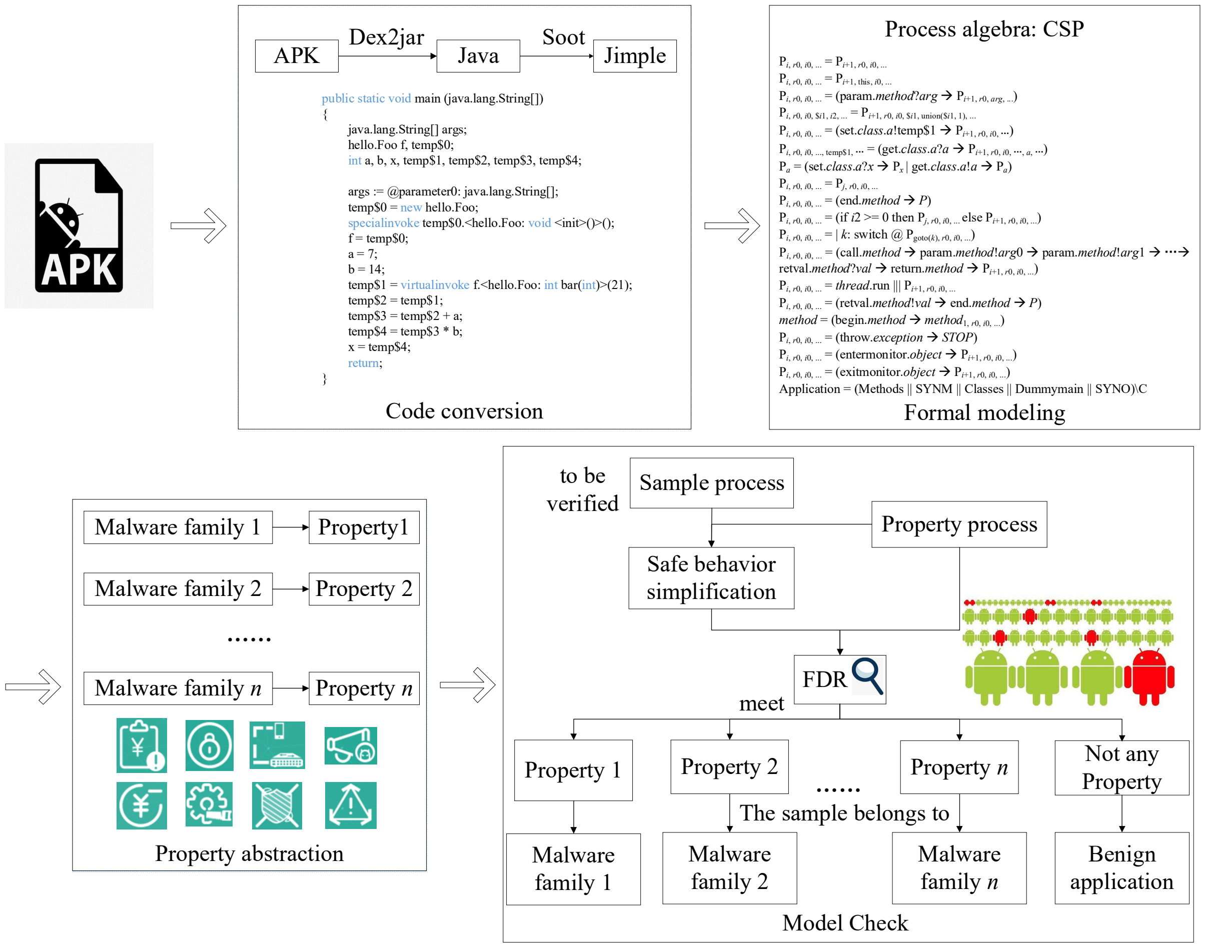
Unlike the Java source code, the method header (Line 1) in Jimple only displays the types of parameters, without the names. The Jimple variables strictly follow the definition-use pattern in that the definitions of all the variables to be used below are given at the beginning of the method body (Lines 2–5), and no more definition statements will appear thereafter. Subsequently, the reference this of the current class and the parameters of the method are assigned to the corresponding variables, whose types are explicitly declared (Lines 6–7). Notably, all the classes and types involved, whether defined or used, appear in the form of a complete class hierarchy to which they belong. For example, the complete type of $\$r3$ is java.io.PrintStream (Line 5). Thus, the types of variables are clear; this is particularly important for program analyses.

Jimple is a three-address intermediate representation, i.e., at most three variables or constants can appear in a statement. However, Java statements are concise, and there is no limit to the number of variables in one statement, which may have many hidden stack positions unnamed. This generates a large number of intermediate variables when a Java code is converted to Jimple.

This process is managed by the linearization of expressions. Taking Java statement "return y * 5 + 16" (Line 8) as an example, we have three corresponding Jimple statements, namely "$\$i4 = i2 * 5$," "$\$i5 = \$i4 + 16$," and "return $\$i5$" (Lines 19–21), which clearly illustrate the complete calculation process by introducing the intermediate variables $\$i4$ and $\$i5$ to ensure the three-address form of Jimple statements.

4. System Design

To conduct a formal detection of Android malware based on CSP, Android applications need to be gradually transformed into CSP processes, and an abstract expression of the family properties should be provided, so that the classification results of the applications can be achieved through model checking. The system design of DroidFDR is divided into four main steps, as shown in Figure 3.



```
public static void main (java.lang.String[])
{
    java.lang.String[] args;
    hello.Foo f, temp$0;
    int a, b, x, temp$1, temp$2, temp$3, temp$4;

    args := @parameter0: java.lang.String[];
    temp$0 = new hello.Foo;
    specialinvoke temp$0.<hello.Foo: void <init>()>();
    f = temp$0;
    a = 7;
    b = 14;
    temp$1 = virtualinvoke f.<hello.Foo: int bar(int)>(21);
    temp$2 = temp$1;
    temp$3 = temp$2 + a;
    temp$4 = temp$3 * b;
    x = temp$4;
    return;
}
```

$$P_{i,\,r0,\,i0,\,...} = P_{i+1,\,r0,\,i0,\,...}$$
$$P_{i,\,r0,\,i0,\,...} = P_{i+1,\,\mathrm{this},\,i0,\,...}$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{param}.\mathit{method}?\mathit{arg} \rightarrow P_{i+1,\,r0,\,arg,\,...})$$
$$P_{i,\,r0,\,i0,\,\$i1,\,i2,\,...} = P_{i+1,\,r0,\,i0,\,\$i1,\,\mathrm{union}(\$i1,\,1),\,...}$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{set}.\mathit{class}.\mathit{a}!\mathrm{temp}\$1 \rightarrow P_{i+1,\,r0,\,i0,\,...})$$
$$P_{i,\,r0,\,i0,\,...,\,\mathrm{temp}\$1,\,...} = (\mathrm{get}.\mathit{class}.\mathit{a}?\mathit{a} \rightarrow P_{i+1,\,r0,\,i0,\,...,\,a,\,...})$$
$$P_{a} = (\mathrm{set}.\mathit{class}.\mathit{a}?\mathit{x} \rightarrow P_{x} \mid \mathrm{get}.\mathit{class}.\mathit{a}!\mathit{a} \rightarrow P_{a})$$
$$P_{i,\,r0,\,i0,\,...} = P_{j,\,r0,\,i0,\,...}$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{end}.\mathit{method} \rightarrow \mathit{P})$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{if}\ i2 >= 0\ \mathrm{then}\ P_{j,\,r0,\,i0,\,...}\ \mathrm{else}\ P_{i+1,\,r0,\,i0,\,...})$$
$$P_{i,\,r0,\,i0,\,...} = \mid k\!: \mathrm{switch}\ @\ P_{\mathrm{goto}(k),\,r0,\,i0,\,...})$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{call}.\mathit{method} \rightarrow \mathrm{param}.\mathit{method}!\mathit{arg}0 \rightarrow \mathrm{param}.\mathit{method}!\mathit{arg}1 \rightarrow \cdots \rightarrow \mathrm{retval}.\mathit{method}?\mathit{val} \rightarrow \mathrm{return}.\mathit{method} \rightarrow P_{i+1,\,r0,\,i0,\,...})$$
$$P_{i,\,r0,\,i0,\,...} = \mathit{thread}.\mathrm{run} \mid\mid\mid P_{i+1,\,r0,\,i0,\,...}$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{retval}.\mathit{method}!\mathit{val} \rightarrow \mathrm{end}.\mathit{method} \rightarrow \mathit{P})$$
$$\mathit{method} = (\mathrm{begin}.\mathit{method} \rightarrow \mathit{method}_{1,\,r0,\,i0,\,...})$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{throw}.\mathit{exception} \rightarrow \mathit{STOP})$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{entermonitor}.\mathit{object} \rightarrow P_{i+1,\,r0,\,i0,\,...})$$
$$P_{i,\,r0,\,i0,\,...} = (\mathrm{exitmonitor}.\mathit{object} \rightarrow P_{i+1,\,r0,\,i0,\,...})$$
$$\mathrm{Application} = (\mathrm{Methods} \parallel \mathrm{SYNM} \parallel \mathrm{Classes} \parallel \mathrm{Dummymain} \parallel \mathrm{SYNO})\backslash C$$

Figure 3. System design of DroidFDR.

4.1. Code Conversion

Android applications are usually distributed and circulated in the form of APK files in the market, so relevant analysis tools are necessary. Moreover, extracting the information in the code quickly and effectively is important because the statement types in Java source code and Java bytecode are very complex. Based on the above considerations, APK files can be decompiled into class files in a compressed form (.jar) by dex2jar, (note: https://sourceforge.net/projects/dex2jar/, accessed on 17 February 2021) and then converted to Jimple by Soot. Jimple has several advantages, such as having few statement types, being clearly typed, and having three-address forms, which are beneficial to subsequent formal modeling.

4.2. Formal Modeling

By corresponding each Jimple statement to a CSP process, we can formally model Android applications. Jimple statements can be broadly divided into five categories: core statements, statements for intraprocedural control flow, statements for interprocedural control flow, monitor statements, and other statements. Since there are only 15 statements in Jimple and the three-address form makes the semantics very clear, it would be easier to express Jimple as CSP processes. A process paradigm of each statement is built on the basis of its characteristics, which is believed to be a good representation of code behavior. A complete modeling of Android applications is proposed on this basis. More details can be found in Section 5.

4.3. Property Abstraction

Each malicious sample involved has a family label, and we can extract common malicious behavior patterns from homologous samples as the property of the family. After illustrating the relationship between different behaviors in the composition of property, we propose an automated algorithm for extracting the CSP process that abstracts the property of each family, whose predictive information is given by observing the malicious code and referring to related technical reports. Compared with the original process of modeling the entire sample behavior, the expression of the property process is more concise, as it involves only the relevant malicious events. More details can be found in Section 6.

4.4. Model Checking

Finally, for Android malware detection, we use the FDR model checker, which is a formal tool applied to CSP. The CSP process modeling the sample behavior is considered the one to be verified, and it is simplified with only the events appearing in the property process retained. Based on the traces model, the sample process is simplified and tested against various property processes, and the result indicates the malware family to which the sample belongs. Since benign applications do not have a corresponding property process, a process that does not conform to any property is considered benign. More details can be found in Section 7.

5. Formal Modeling with CSP

If a piece of Jimple code is considered an entire system, each statement contained is an integral part of it and can be modeled as a CSP process. The process name of a statement consists of the complete method name including parameter types and subscripts. The first subscript denotes the number of the current statement in its method, followed by non-static local variables of the method, non-static member variables of its class, and that of class objects created in the method to better indicate the change in the variable values. For example, for the method example shown in Figure 2b, the process name corresponding to the i-th statement is Test_example (int)$_{i,\, r0,\, i0,\, \dots}$, where the value of i is no greater than the total number of Jimple statements in this method (start counting after definition statements). We specifically explain how to convert different types of Jimple statements into CSP processes below, where all example processes are named $P_{i,\, r0,\, i0,\, \dots}$ for convenience.

5.1. Core Statements

5.1.1. NopStmt

NopStmt is a no-operation statement, and the corresponding CSP process

$$P_{i,r0,i0,\dots} = P_{i+1,r0,i0,\dots}$$

indicates that no event has occurred, and that the variables values remained unchanged. The execution continues directly based on the process behavior of the next statement.

5.1.2. IdentityStmt

IdentityStmt is a statement that assigns parameter values or the *this* reference to variables. If it is related to the parameters, for example, the CSP process of "$i0$:= @parameter0: int" is

$$P_{i,r0,i0,\dots} = (\mathrm{param}.\mathit{method}?\mathit{arg} \rightarrow P_{i+1,\, r0,\, arg,\, \dots})$$

Otherwise, for example, the CSP process of "$r0$:= @this: Test" is

$$P_{i,r0,i0,\dots} = P_{i+1,\mathrm{this},i0,\dots}$$

where param.*method*?*arg* is the event where the current method receives the argument *arg*. The particularity of parameter assignment is that the arguments in the calling statement need to be passed onto the called method. In the case of multiple parameters, the corresponding input value can be easily found because the parameter assignments are performed in order.

5.1.3. AssignStmt

AssignStmt is a general assignment statement of variables, for example, the CSP process of "*i*2 = $*i*1 + 1" is

$$P_{i,r0,i0,\$i1,i2,\ldots} = P_{i+1,r0,i0,\$i1,\,\text{union}(\$i1,\,1),\ldots}$$

which is different from IdentityStmt in that the process mainly models the relationship where the variable *i*2 on the left side of the equation is infected by the variable $*i*1 and the constant 1 on the right side. The type of variable *i*2 is abstracted into a set to express the propagation of data flow, so *i*2 will be replaced by union($*i*1, 1) in the subscripts of the *i* + 1-th process here.

In addition, when AssignStmt involves static member variables of a class or static variables of member methods in the class, it is necessary to transfer data between processes. If it is to set their values, for example, the CSP process of "*c*.<Class: int *a*> = temp$1" is

$$P_{i,r0,i0,\ldots} = (\text{set}.\mathit{class}.a!\text{temp\$1} \rightarrow P_{i+1,r0,i0,\ldots})$$

Otherwise, for example, the CSP process of "temp$1 = *c*.<Class: int *a*>" is

$$P_{i,r0,i0,\ldots,\text{temp\$1},\ldots} = (\text{get}.\mathit{class}.a?a \rightarrow P_{i+1,r0,i0,\ldots,a,\ldots})$$

where set.*class*.*a*!temp$1 is the event where AssignStmt outputs temp$1 as an assignment to *class*.*a*, and get.*class*.*a*?*a* is the event where AssignStmt receives the value of *class*.*a*.

A process should be modeled for each class to manage the values of all static member variables and static variables of the member methods in it. If the class has one such variable *a*, the corresponding CSP process is

$$P_a = (\text{set}.\mathit{class}.a?x \rightarrow P_x \mid \text{get}.\mathit{class}.a!a \rightarrow P_a)$$

where set.*class*.*a*?*x* is the event where the current class receives *x* as an assignment to *a*, and get.*class*.*a*!*a* is the event where *class* outputs the value of *a*.

5.2. Statements for Intraprocedural Control Flow

5.2.1. IfStmt

IfStmt is a conditional judgment statement and is usually followed by GotoStmt as a complete sentence instead of appearing alone.

5.2.2. GotoStmt

GotoStmt is a jump statement, which can appear alone like "goto label0," or be combined with IfStmt to indicate the location of the processing statements when the judgment condition is satisfied, such as "if *i*2 >= 0 goto label0." In Jimple, the target of different jumps is identified by *label*, which is sequentially numbered starting from label0.

The CSP process corresponding to the former case is

$$P_{i,r0,i0,\ldots} = P_{j,r0,i0,\ldots}$$

where *j* is the statement number of the jump target. To obtain the value of *j*, we need to search for the label (in the form of "label0:") in the subsequent range of the current method. The process does not execute any actual events, but simply jumps to the process where the label is located.

The CSP process corresponding to the latter case is

$$P_{i,r0,i0,\ldots} = (\text{if } i2 >= 0 \text{ then } P_{j,r0,i0,\ldots} \text{ else } P_{i+1,r0,i0,\ldots})$$

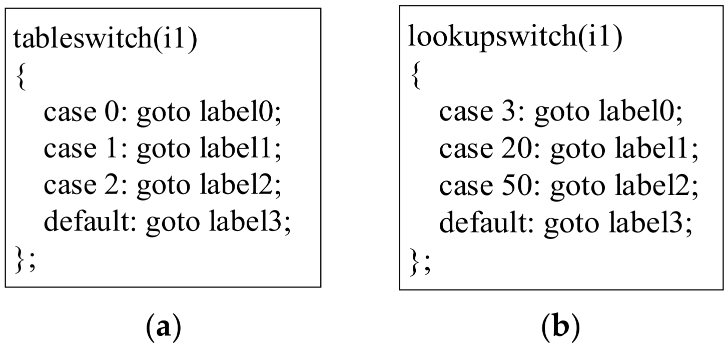
The process provides two options in that if the condition given by IfStmt is established, the execution continues based on the statement process at the corresponding label; otherwise, the subsequent process will be executed in sequence.

5.2.3. TableSwitchStmt

TableSwitchStmt is a table switch statement, translated from the instruction *tableswitch* in Java bytecode. It is applied to scenarios where the distribution of the conditional values in the switch branches is relatively concentrated, as shown in Figure 4a. The CSP process corresponding to TableSwitchStmt is

$$P_{i,r0,i0,\ldots} = \mid k\text{: switch @ } P_{\text{goto}(k),r0,i0,\ldots})$$

where k is the case value of the branch, and goto(k) is the start number of the statements handling the case. The process indicates that the execution jumps to the corresponding label position after choosing the appropriate case branch.



```
tableswitch(i1)
{
   case 0: goto label0;
   case 1: goto label1;
   case 2: goto label2;
   default: goto label3;
};
```

(**a**)

```
lookupswitch(i1)
{
   case 3: goto label0;
   case 20: goto label1;
   case 50: goto label2;
   default: goto label3;
};
```

(**b**)

Figure 4. Examples of two switch Jimple statements: (**a**) TableSwitchStmt; (**b**) LookupSwitchStmt.

5.2.4. LookupSwitchStmt

LookupSwitchStmt is a lookup switch statement, translated from the instruction *lookupswitch* in Java bytecode. It is applied to scenarios where the distribution of the condition values in the switch branches is relatively sparse, as shown in Figure 4b. Although LookupSwitchStmt accesses the jump table by index whereas TableSwitchStmt by key-value matching, the two statements can share the same CSP.

5.3. Statements for Interprocedural Control Flow

5.3.1. InvokeStmt

InvokeStmt is a calling statement, such as "$\$i1$ = virtualinvoke $r2$.<Foo: int bar(int)>($i0$)." If the called method has both parameters and a return value, the corresponding CSP process of InvokeStmt is

$$P_{i,\,r0,\,i0,\ldots} = (\text{call}.\mathit{method} \rightarrow \text{param}.\mathit{method}!\mathit{arg}0 \rightarrow \text{param}.\mathit{method}!\mathit{arg}1 \rightarrow \ldots \rightarrow \text{retval}.\mathit{method}?\mathit{val} \rightarrow \text{return}.\mathit{method} \rightarrow P_{i+1,r0,i0,\ldots})$$

where param.*method*!*arg*0, param.*method*!*arg*1 … are events, where pass arguments to *method* in turn, retval.*method*?*val* is the event where receives *val* from *method*, and call.*method* and return.*method* are the events representing the start and finish of the calling of *method*, respectively. When the called method has no parameters or return value, the corresponding events will be removed from the process. For the called method, its argument reception is done in IdentityStmt.

In particular, the creation of a new thread follows the form of method calls; however, it need not wait for the callee to complete the execution as in general calls. Therefore, its corresponding CSP process is

$$P_{i,r0,i0,\ldots} = \mathit{thread}.\mathrm{run} \mid\mid\mid P_{i+1,r0,i0,\ldots}$$

where *thread*.run refers to the process of the *run* method in the created thread class *thread*.

5.3.2. ReturnStmt

ReturnStmt is a return statement with a return value, such as "return $*i*3." Its corresponding CSP process is

$$P_{i,r0,i0,\ldots} = (\mathrm{retval}.\mathit{method}!\mathit{val} \rightarrow \mathrm{end}.\mathit{method} \rightarrow P)$$

where retval.*method*!*val* is the event that the current method returns the variable *val*. The process indicates that the execution of the method ends after returning the required result and returns to the beginning of the method for the next call.

5.3.3. ReturnVoidStmt

ReturnVoidStmt is the return statement "return" with no return value, the corresponding CSP process of which is

$$P_{i,r0,i0,\ldots} = (\mathrm{end}.\mathit{method} \rightarrow P)$$

where *method* refers to the current method. The process ends this execution without any value returned and goes back to the beginning of the method.

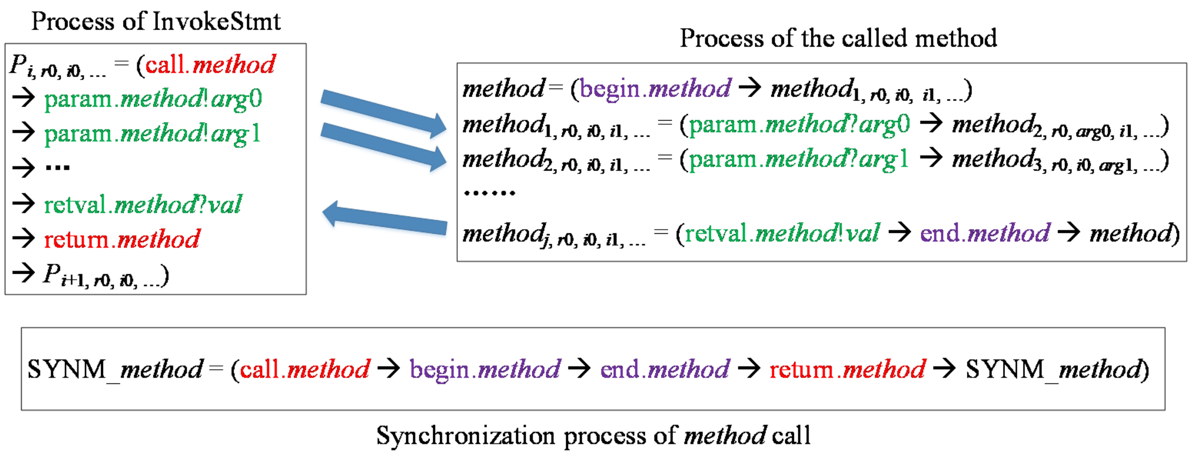
To ensure that any method can only be called by at most one method at a time, the SYNM process is necessary to describe the call synchronization mechanism of the system, composed of the control process of each call synchronization event, SYNM_*method*,

$$\mathrm{SYNM\_}\mathit{method} = (\mathrm{call}.\mathit{method} \rightarrow \mathrm{begin}.\mathit{method} \rightarrow \mathrm{end}.\mathit{method} \rightarrow \mathrm{return}.\mathit{method} \rightarrow \mathrm{SYNM\_}\mathit{method})$$

$$\mathrm{SYNM} = (\mathrm{SYNM\_}\mathit{method}_1 \mid\mid\mid \mathrm{SYN\_}\mathit{method}_2 \mid\mid\mid \ldots)$$

where *method* refers to the called method, and $method_1$, $method_2$ ... are different called methods in the system.

Figure 5 shows the synchronization between InvokeStmt and the called method with two parameters and a return value. We set the prompt events *begin* and *end* for the beginning and end of each method execution, respectively.



Process of InvokeStmt
P_{i, r0, i0, ...} = (call.method
→ param.method!arg0
→ param.method!arg1
→ ...
→ retval.method?val
→ return.method
→ P_{i+1, r0, i0, ...})
Process of the called method
method = (begin.method → method_{1, r0, i0, i1, ...})
method_{1, r0, i0, i1, ...} = (param.method?arg0 → method_{2, r0, arg0, i1, ...})
method_{2, r0, i0, i1, ...} = (param.method?arg1 → method_{3, r0, i0, arg1, ...})
......
method_{j, r0, i0, i1, ...} = (retval.method!val → end.method → method)
SYNM_method = (call.method → begin.method → end.method → return.method → SYNM_method)
Synchronization process of method call

Figure 5. Synchronization between InvokeStmt and the called method.

Since the first statement of a method is uncertain, *begin* is additionally used as the prefix of the first process, and *end* is placed in the expression of ReturnStmt or ReturnVoidStmt because the return statement usually marks the completion of a method. Therefore, the CSP of the method is

$$method = (\text{begin}.method \rightarrow method_{1,\text{r0,i0},\ldots})$$

while in InvokeStmt, we set the *call* and *return* events as the start and completion of this call, respectively. The synchronization set of SYNM contains the *call*, *begin*, *end*, and *return* events, and the control will return to the caller's subsequent statement after executing the called method.

5.4. Other Statements

5.4.1. ThrowStmt

ThrowStmt is an exception throwing statement, such as "throw *r*4." Its corresponding CSP process is

$$P_{i,r0,i0,\ldots} = (\text{throw}.exception \rightarrow STOP)$$

where *exception* refers to the type of variable being thrown, and throw.*exception* is the event that throws the exception. This process indicates that the execution of the current method stops after an exception is thrown. The specific type of exception should be confirmed on the basis of the variable in the statement, which can be found in the definition statements at the beginning of the method.

5.4.2. RetStmt

RetStmt is a return statement of standard request, translated from the instruction *ret* in Java bytecode. It realizes the jump and return of the *finally* module in Java source code with the instruction *jsr*. However, starting from Java SE 7, Java compilers after JDK 1.4.2 no longer use these two instructions to process the module, so RetStmt is not considered.

5.4.3. MonitorStmt

EnterMonitorStmt and ExitMonitorStmt are monitor statements translated from Java bytecode instructions monitorenter and monitorexit, respectively, providing explicit support for synchronization. Since each object has a corresponding monitor, the two statements appear in pairs to play the role of acquiring and releasing the monitor lock separately, ensuring that at most one thread can execute one synchronization event at any time.

The corresponding CSP process of EnterMonitorStmt is

$$P_{i,r0,i0,\ldots} = (\text{entermonitor}.object \rightarrow P_{i+1,r0,i0,\ldots})$$

and that of ExitMonitorStmt is

$$P_{i,r0,i0,\ldots} = (\text{exitmonitor}.object \rightarrow P_{i+1,r0,i0,\ldots})$$

where *object* refers to the monitored object, and entermonitor.*object* and exitmonitor.*object* are events that enter and exit the monitor of the object, respectively.

To ensure the exclusiveness of the synchronized objects, the SYNO process is necessary to describe the object synchronization mechanism of the system; this process comprises the control process of each synchronization event, SYNO_*object*,

$$\text{SYNO\_}object = (\text{entermonitor}.object \rightarrow \text{exitmonitor}.object \rightarrow \text{SYNO\_}object)$$

$$\text{SYNO} = (\text{SYNO\_}object1 \mid\mid\mid \text{SYNO\_}object2 \mid\mid\mid \ldots)$$

where *object*1, *object*2 ... are different monitored objects in the system.

5.5. Modeling of Application

From the perspective of the entire program, the system consists of the following processes, compounded concurrently.

1. The processes of all methods in the program, denoted by method_1, method_2 . . . , are responsible for the execution of the respective method.
2. The synchronization process of method calls, SYNM, is responsible for the exclusive call to methods.
3. The processes of all classes in the program, denoted by class_1, class_2 . . . , are responsible for the management of static member variables and static variables of the member methods in the respective class.
4. The process used to simulate the main method of the program, denoted by Dummymain, is responsible for modeling the application entries. Since an Android application may have multiple entry points, we refer to FlowDroid to customize a dummy main method for each application, and the processes of the other methods will be executed only when they are called.
5. The synchronization process of the objects, SYNO, is responsible for the exclusive access to monitored objects.

Therefore, the CSP process corresponding to the application is

$$\text{Methods} = (\text{method}_1 \mid\mid\mid \text{method}_2 \mid\mid\mid \ldots)$$

$$\text{Classes} = (\text{class}_1 \mid\mid\mid \text{class}_2 \mid\mid\mid \ldots)$$

$$\text{Application} = (\text{Methods} \mid\mid \text{SYNM} \mid\mid \text{Classes} \mid\mid \text{Dummymain} \mid\mid \text{SYNO}) \backslash C$$

$$C = \{\text{call, return, begin, end, param, retval, set, get, entermonitor, exitmonitor}\}$$

6. Abstraction of Family Property

Common behavioral characteristics of malware belonging to the same family are abstracted as family-specific properties. Unlike previous studies where the full participation of experts was required, DroidFDR adopts an automated extraction algorithm for family properties. The CSP process that expresses the property of a family can be automatically extracted by observing malicious code and referring to relevant technical reports to provide the predictive information required by the algorithm.

6.1. Composition Structure of Behavior in Property

Here, the property is defined as the behavior pattern of a malware family, which can determine the dependency relationship between malicious behaviors, and it is the foundation of our classification. The property can be formally modeled in CSP with various structures, such as sequence, selection, parallelism, iteration, and conditional control, thus bringing a good description ability. We briefly summarize the behavioral composition of the properties, along with some examples:

6.1.1. Sequential Structure

The sequential structure expresses the order of execution between behaviors, usually emphasizing that the former behavior must occur before the latter behavior. For example, malware intended to send personal information of a user to a remote server must obtain the data of interest through relevant system APIs first. The process

$$S = (a \rightarrow b \rightarrow S_1)$$

corresponds to the diagram shown in Figure 6a, which indicates that the property must perform the two behaviors a and b in order, and then continue to execute the subsequent events based on the process behavior of S_1.

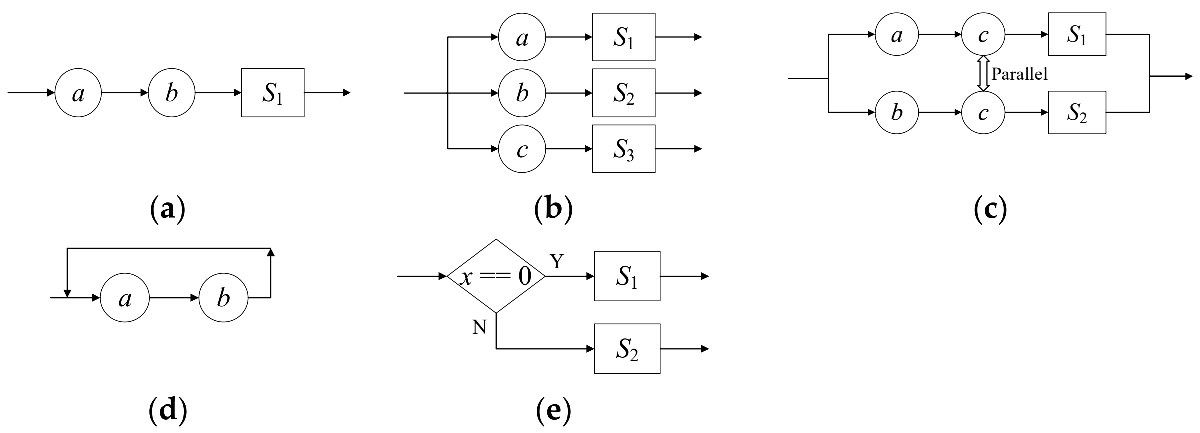


Figure 6. Structural diagram: (**a**) sequential structure; (**b**) selective structure; (**c**) parallel structure; (**d**) iterative structure; (**e**) conditional control structure.

6.1.2. Selective Structure

The selective structure presents the options of process branches, typically used to perform multiple similar behaviors that will trigger different subsequent actions. For example, malware may choose to collect the geographic location of a device through GPS or a network. The process

$$S = (a \to S_1 \mid b \to S_2 \mid c \to S_3)$$

corresponds to the diagram shown in Figure 6b, which indicates that the property can choose a, b, or c as the initial behavior, and then continue to execute the respective subsequent processes S_1, S_2, or S_3.

6.1.3. Parallel Structure

The parallel structure expresses the concurrency between processes, usually emphasizing that the two branches can run simultaneously with some interactive behaviors. For example, malware can start two different malicious activities when informed of system boot. The process

$$S = (a \to c \to S_1 \mid\mid b \to c \to S_2)$$

corresponds to the diagram shown in Figure 6c, which indicates that when involving the behavior c, the two branch processes will be executed synchronously, but alternately for the remaining time.

6.1.4. Iterative Structure

The iterative structure expresses the repeated execution of behavior sequence, typically used to show the persistence of malicious behavior. For example, useless SMS could be repeatedly sent in the background to consume user tariffs. The process

$$S = (a \to b \to S)$$

corresponds to the diagram shown in Figure 6d, which indicates that the property is composed of two behaviors: a should be executed followed by b, and this procedure is repeated.

6.1.5. Conditional Control Structure

The conditional control structure describes the relationship between judgment conditions and subsequent processes, usually emphasizing the control of conditions on process behavior. For example, malware will start some illegal activities once the screen-off duration is detected for a long enough duration. The process

$$S = (\text{if } x == 0 \text{ then } S_1 \text{ else } S_2)$$

corresponds to the diagram shown in Figure 6e, which indicates that the property is closely related to whether the variable x is equal to 0, and S_1 is executed when the condition is satisfied, whereas S_2 is executed when it is not.

6.2. Algorithm for Extracting Family Properties

In previous research, a property specific to each family was usually determined by analysts through observation of malicious samples; this process requires considerable human involvement. To improve the efficiency, an automated algorithm for extracting family properties is proposed for DroidFDR. The sensitive API list and sensitive string information are believed to be the most important components in the extraction of family properties. First, the algorithm needs to define a list of sensitive APIs closely related to malicious behaviors in advance for all malware families, and then determine a family-specific list of sensitive strings based on the actual situation of each family. The corresponding sensitive events can be given by combining the information from two aspects, thereafter determining the processes constituted by these events in the samples, and perfecting the possible arrangement of sensitive events through a series of rules. Algorithm 1 shows the pseudocode of our property extraction algorithm.

For a specific malware family, the algorithm first simplifies the CSP process of test samples belonging to the family through the function *Simplify*, in a way that retains the family-related sensitive events and removes other events unrelated to malicious behavior. The loop statement indicates that the process abstracting the family property can be affected by all the inputted sample processes. These contain, as comprehensively as possible, structures comprising sensitive events that the family may present through the conditional judgment statements.

7. Model Checking in FDR

DroidFDR utilizes a model checker, namely the failures-divergence refinement (FDR), to conduct a formal analysis of whether an Android application belongs to a malware family. FDR is a refinement detection tool, applied to formal models built in CSP. FDR2 is a new version of the tool, with an improved flexibility and scalability over previous versions. Although model checking can automatically verify with a limited state space, FDR2 employs some state-space compression algorithms, thus significantly reducing the size of the state space to be explored during the detection, and effectively alleviating the problem of state-space explosion.

Algorithm 1 Extracting the property of a malware family *f*

```
Input:
$Samp_{f1}$, $Samp_{f2}$, ... $Samp_{fn}$ ——The processes that express the behavior of test samples in the malware family $f$
$SE_f$ ——The sensitive event set of malware family $f$
Output:
$Spec_f$ ——The process that abstracts the property of malware family $f$

Procedure Family_PropertyExtraction
begin
$P_1 \leftarrow Simplify(Samp_{f1}, SE_f)$
$Spec_f := P_1$
for($i = 1, i < n+1, i{+}{+}$) do
begin
$P_i \leftarrow Simplify(Samp_{fi}, SE_f)$
if($traces(Spec_f) \subseteq traces(P_i)$) $Spec_f := P_i$
else if($traces(P_i) \subseteq traces(Spec_f)$) continue
else $Spec_f := Spec_{f|}\, P_i$
end
end
```

The theory of CSP is classically based on mathematical models unrelated to the language itself. These models are based on observable behaviors of processes, rather than attempting to capture a full operational picture of how the process progresses. FDR provides three alternative models: traces model, stable failures model, and failures/divergences model. Depending on these semantic models, equivalence relations can be defined for systems described in CSP in several ways, which can be used for proving different properties. In the context of Android malware detection, we only focus on security, i.e., whether the sample process has the property of any malware family. To meet this demand, the traces model is sufficient. In the traces model, a process is represented by a set of finite sequences of communications it can perform. The set of (finite) traces of process P is given by traces(P). A process Q is a trace equivalence of another, P, if traces(Q) = traces(P).

The heart of the FDR is refinement, which is a relationship that reflects that a process has at least some specific behaviors that can be satisfied by another process. Two processes are necessary in each FDR test, called P and Q here. In the traces model, Q is a traces refinement of P, which is written as $P \subseteq_T \mathrm{Q}$, if traces(Q) $\subseteq$ traces(P).

Specific to the detection of Android malware, P represents the abstracted property of each family, whereas Q represents the modeled code behavior of a sample. In one test, only a sample process and a property process can participate simultaneously. Under the premise that the sample process only retains the events contained in the property process, it is determined whether the former is a traces refinement of the latter. If FDR judges that the test passes, it means that the sample has the property of the family, so it can be classified into the family. Otherwise, it means that the sample does not belong to the family, and the sample process is continued to be tested against other property processes. A sample should be classified as a benign application if its process cannot become a traces refinement of any property process.

8. Experimental Analysis

8.1. Dataset

The malware used in our experiment was obtained from

- Drebin dataset [3], covering multiple international Android app markets, websites, malware forums, and security blogs, which contains all malware from the Android Malware Genome Project [49],
- AndroZoo (note: https://androzoo.uni.lu/, accessed on 14 May 2022), a growing collection of Android Applications collected from several sources, whose malware has been analyzed by tens of different AntiVirus products, and
- VirusShare (note: https://virusshare.com/, accessed on 15 May 2022), a continuously updated malware database for share.

The malware in the dataset has a clear family label, making it possible to evaluate the effectiveness of Android malware detection and family classification methods.

The benign applications used in our experiment come from the top apps in various categories of the Huawei app market, downloaded during the August–December period of 2019. To exclude potentially bad applications, they were inspected using VirusTotal (note: https://www.virustotal.com/, accessed on 11 March 2021). VirusTotal is a website that provides an analysis service for suspicious files for free, with more than 70 anti-virus scanning engines involved in the detection of uploaded files. In the case of no timeout, an application is considered benign only when all engines show no warnings.

8.2. Experimental Results

To better reflect the detection effect of DroidFDR, we compared it with the same type of formal detection method for Android. The comparison subject was the research tool LEILA [41], proposed by Canfora et al. in 2019. LEILA uses a formal method similar to DroidFDR in that it employs model checking to perform the family classification of Android applications. The difference is that the process algebra in LEILA for code modeling is Milner's calculus of communicating systems (CCS) [45], the properties of the malware families are expressed by temporal logic formulae, and CAAL [50] is used as the verification environment.

Table 2 gives a comparison between LEILA and DroidFDR, including the classification results of malicious samples against four families, and the experimental evaluation here is largely influenced by that of LEILA. When characterizing the property of each family, LEILA randomly selects five samples for manual inspection, whereas DroidFDR automatically extracts the property process from five samples. Each time a model checking is performed against a family,

- Fifty samples of the family that are not involved in the property extraction process,
- Three-hundred malicious samples belonging to other families, and
- Fifty benign applications

are randomly chosen to be verified.

Table 2. Comparison of family classification results between LEILA and DroidFDR.

Malware Family	Precision (%)		Recall (%)		F1 Score (%)		Accuracy (%)		AUC	
	LEILA	DroidFDR	L	D	L	D	L	D	L	D
Opfake	92.00	94.12	92.00	96.00	92.00	95.05	98.00	98.75	0.9543	0.9757
GingerMaster	95.92	95.92	94.00	94.00	94.95	94.95	98.75	98.75	0.9671	0.9671
FakeInstaller	84.21	96.15	96.00	100.00	89.72	98.04	97.25	99.50	0.9671	0.9971
Plankton	100.00	96.08	100.00	98.00	100.00	97.03	100.00	99.25	1.00	0.9871
Average	93.03	95.57	95.50	97.00	94.12	96.27	98.50	99.06	0.9721	0.9818

The performance measures for the classification were the precision, recall, F1 score, accuracy, and AUC [51]. The precision, recall, and accuracy represent the proportions of correct predictions in all samples classified as positive, in all positive samples, and in all samples, respectively. Since the precision and recall are a pair of contradictory metrics, the F1 score is a measure of the comprehensive performance because it is the harmonic average of the precision and recall. In addition, AUC is a good indicator of classification performance.

The family classification results show that, although the two methods have respective advantages in different indicators of different families, the detection performance of DroidFDR is slightly better than that of LEILA on average. Considering that formal methods generally have a high detection accuracy, this is sufficient to confirm that DroidFDR has advantages over similar type of methods. Notably, under the same number of test samples in each family, the classification effect of different families is relatively close in DroidFDR, whereas the detection results of LEILA have a certain degree of fluctuation. In particular, the precision of the FakeInstaller family is low, probably because of the uncontrollable influence of manual property extraction, resulting in misclassifications of many samples that do not belong to the family.

Because of the small number of families considered in the experiment with LEILA, we selected additional families with more than 50 samples from the Android Malware Genome project and 300 benign applications for another experiment, the proportion and results of which are listed in Tables 3 and 4. A 10-fold cross-validation [51] was used for the evaluation. On the premise that the data are evenly distributed, the malicious and benign applications were divided into 10 subsets each. In each round of the experiment, a malicious subset that had not been selected was chosen, the samples of which were inputted to the family property extraction algorithm, and the remaining subsets were kept for testing. Evidently, DroidFDR exhibits a good detection performance.

Table 3. The proportion of malicious and benign samples in each experiment.

Malware Family	# Family Sample	# Other Malware	# Benign Applications	Total Number
AnserverBot	187	706	300	1193
BaseBridge	122	771	300	1193
DroidKungFu3	309	584	300	1193
DroidKungFu4	96	797	300	1193
Geinimi	69	824	300	1193
KMin	52	841	300	1193
Pjapps	58	835	300	1193
Adrd	91	802	300	1193
DroidDream	81	812	300	1193
FakeDoc	132	761	300	1193
Dowgin	100	793	300	1193
Wooboo	100	793	300	1193

Table 4. Classification results of large families with DroidFDR.

Malware Family	Sample Number	Precision (%)	Recall (%)	F1 Score (%)	Accuracy (%)	AUC
AnserverBot	187	97.30	96.26	96.77	98.91	0.9783
BaseBridge	122	95.87	95.08	95.47	99.08	0.9731
DroidKungFu3	309	97.72	97.09	97.40	98.66	0.9815
DroidKungFu4	96	96.81	94.79	95.79	99.33	0.9726
Geinimi	69	95.65	95.65	95.65	99.50	0.9769
KMin	52	94.32	96.15	95.24	99.58	0.9795
Pjapps	58	91.52	93.10	92.31	99.25	0.9633
Adrd	91	96.74	97.80	97.27	99.58	0.9876
DroidDream	81	95.18	97.53	96.34	99.50	0.9859
FakeDoc	132	99.23	98.48	98.85	99.66	0.9882
Dowgin	100	97.06	99.00	98.02	99.66	0.9936
Wooboo	100	96.04	97.00	96.52	99.41	0.9832

The effectiveness of the FDR tool for the formal detection of Android malware depends on the appropriate description of the code and the behavioral consistency within the malware family. From a theoretical basis, DroidFDR has the following advantages over LEILA:

- LEILA directly models Java bytecode, whereas DroidFDR converts APK files into an intermediate representation, i.e., Jimple, with fewer types of statements and simpler forms, which is conducive to fast and accurate formal modeling.
- LEILA focuses on capturing sensitive API calls when defining family properties, whereas DroidFDR also considers sensitive string information and sensitive data flow to make the expression of the properties more comprehensive, thus yielding a better detection performance.
- The temporal logic formulae for the property are defined by manually inspecting a few samples from each family in LEILA, whereas DroidFDR includes an automatic property extraction algorithm, which significantly improves the efficiency and avoids the possibility of human error.

Most current studies in the field of Android malware detection have applied machine learning technology to statistical pattern analyses. In comparison, the structural pattern analysis with the help of formal languages has a relatively higher detection performance while ensuring the accuracy of code modeling and family property abstraction; moreover, it can capture the behavioral structure of applications with interpretability, which is not possible for machine learning methods. The former requires a large, comprehensive, and accurate extraction of the features from the code, which is difficult to research, because it largely determines the learning effect. The latter only requires formulating modeling rules for the code, making it easy to fully express the behavior; however, there is the problem of a longer detection time. In short, formal detection methods, such as DroidFDR, have a very broad application prospect in specific scenarios with less time limit.

8.3. Performance Analysis

An experiment was performed on a laptop equipped with a 6-core 1.10 GHz Intel Core i7-10710U CPU and 16 GB of memory. Based on the system design of DroidFDR mentioned in Section 4, the runtime overhead was mainly divided into four parts: code conversion, formal modeling, property abstraction, and model checking. First, the APK file of the Android application should be converted into Java class files and Jimple intermediate representation in turn using tools dex2jar and Soot. Subsequently, each Jimple statement is modeled as a CSP process under the rules formulated in advance, and all the statements in the program are integrated into a system process. The family-specific sensitive event set and sample processes are inputted to the property extraction algorithm to obtain the abstract process of the family property. Notably, this step need not be repeated more than once for a family. Finally, for any Android application, verifying its corresponding process against a property process via model checking by the FDR can determine whether it belongs to the family. Table 5 lists the average time spent in each phase of the comparison experiment, where the second and third columns represent the processing time of a sample, the fourth column represents the time required to extract a property from five samples, and the last column represents the time required for model checking.

Table 5. Classification results of large families with DroidFDR.

Phase	Code Conversion	Formal Modeling	Property Abstraction	Model Checking
Time (s)	10.94	5.97	1505.32	268.23

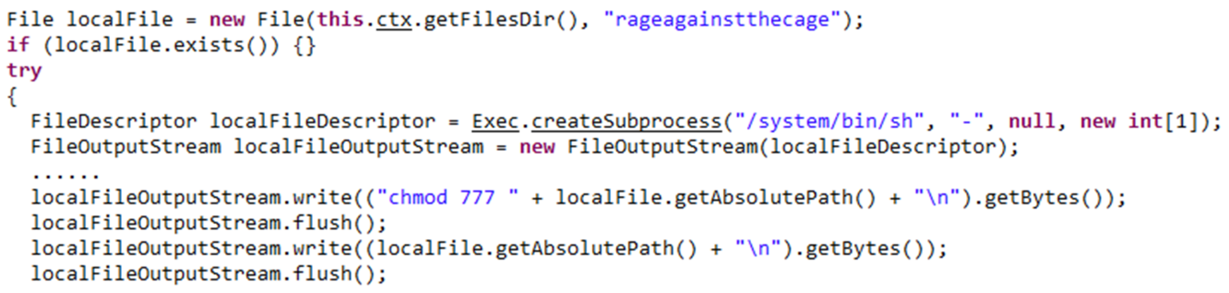
Like other formal methods, the analysis process of DroidFDR is time-consuming, particularly for property abstraction and model checking. However, DroidFDR is intended for the anti-malware analysis of applications to be released to the market. In a commercial application environment, DroidFDR can be used as an analysis module for malware behavior in the security center. If equipped with a high-performance server, DroidFDR can be made more efficient, and the required time can be limited to an acceptable range.

8.4. Analysis of Typical Malicious Behaviors

By applying the property extraction algorithm to the malicious samples in our dataset, we can obtain the formal expression of each family property to better understand the behavior patterns when performing malicious activities. Some behavior patterns are very common in malware family properties. Although there are various implementations to achieve an objective, samples from the same family will be encoded in the same way, suggesting the same formal expression. Based on the experimental results, we briefly elaborate on the patterns of several typical malicious behaviors to provide a reference for further research on the security of Android applications.

8.4.1. Root Privilege Escalation

Rooting is a basic operation that several malware carry out before performing actual malicious activities, because it can escalate the privileges of Android applications. However, the Android system does not officially provide a legal implementation of this function, so the purpose of improper privilege escalation can be achieved only by exploiting various root vulnerabilities. The "Rageagainstthecage" vulnerability is very common, and DroidDream is one of the malware families that exploit it. Figure 7 shows the code related to root privilege escalation in the sample of DroidDream, indicating how the family implements this operation.



```
File localFile = new File(this.ctx.getFilesDir(), "rageagainstthecage");
if (localFile.exists()) {}
try
{
  FileDescriptor localFileDescriptor = Exec.createSubprocess("/system/bin/sh", "-", null, new int[1]);
  FileOutputStream localFileOutputStream = new FileOutputStream(localFileDescriptor);
  ......
  localFileOutputStream.write(("chmod 777 " + localFile.getAbsolutePath() + "\n").getBytes());
  localFileOutputStream.flush();
  localFileOutputStream.write((localFile.getAbsolutePath() + "\n").getBytes());
  localFileOutputStream.flush();
```

Figure 7. Implementation code for root privilege escalation in the sample of DroidDream.

There are three key steps in the code.

Determine whether the file "rageagainstthecage" exists, which is a vulnerability that needs to be exploited to obtain root privileges.

If the file exists, the Exec.createSubprocess method is used to run a shell terminal command. System files are placed in the/system directory, and system files related to root privileges are placed in/system/bin/sh.

"chmod" is the prefix of the command for modifying file access permissions, and "chmod 777 directory" can make the file permissions under the path become readable, writable, and executable.

Based on the above description, the part of the property related to root privilege escalation in DroidDream can be expressed as a behavior model comprising sensitive events in the form of a conditional judgment:

java_io_File_init(File, String)!"rageagainstthecage"→android_os_Exec_createSubprocess (String, String, String, int[])!"/system/bin/sh"→channel?*str*→if member(*str*, "chmod 777") then java_io_FileOutputStream_write(int)!*str*

where *str* refers to the parameter of the *write* method, and channel?*str* indicates its source. The value set of *str* should contain "chmod 777" after AssignStmt, so the member(*str*, "chmod 777") provided by FDR is required to test whether it is satisfied. Notably, the *channel* here does not exist in the actual code modeling, so it should be hidden from the property process during model checking.

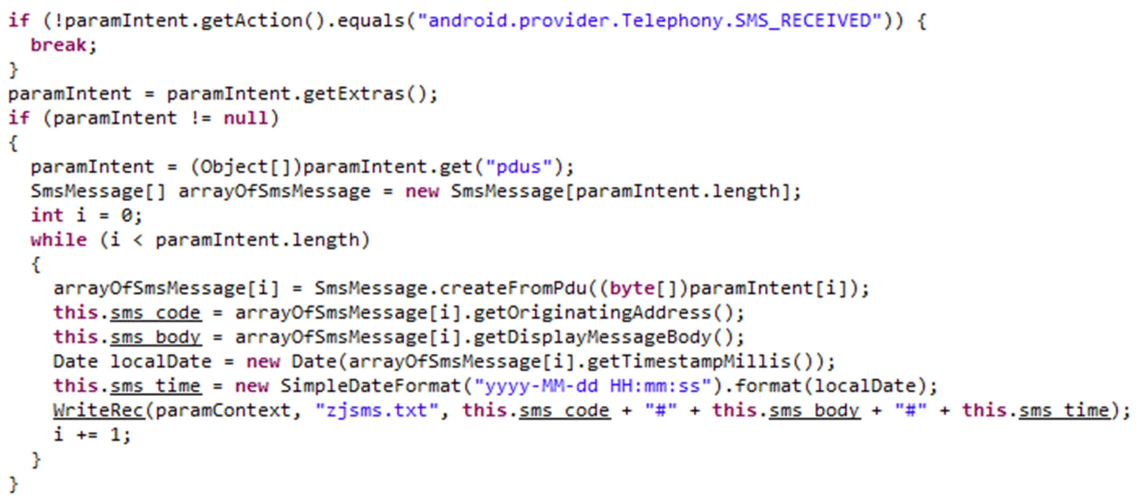
8.4.2. Access to Sensitive Information

A privacy theft is a very common malicious activity, because malware authors can collect device information and user privacy to conduct underground criminal transactions for illegal reasons. Malware families are usually interested in the following three types of sensitive information: information related to device and SIM card, such as International Mobile Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI), and Integrate Circuit Card Identification (ICCID) information; the geographic location of the user; and the call records and the content of text messages. The code related to obtaining the information in incoming SMS in the sample of GoldDream is shown in Figure 8, indicating the sensitive content secretly recorded by the family when receiving a new text message.

There are three key steps in the code.

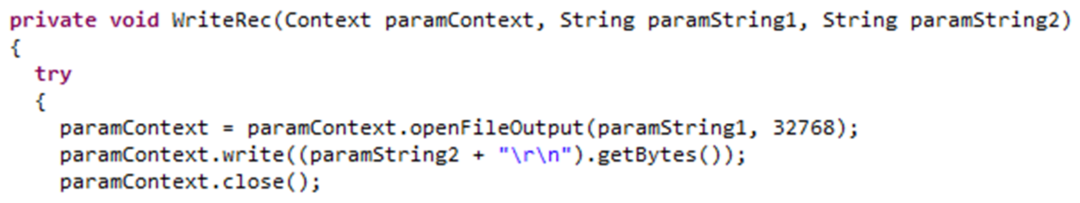
The system event android.provider.Telephony.SMS_RECEIVED is monitored, and the corresponding broadcast will be triggered when the Android device receives a new text message.

Since the SMS received by Android devices is in the PDU format, it is necessary to extract some sensitive information such as the sender's number, body of the message, and reception time.



```
if (!paramIntent.getAction().equals("android.provider.Telephony.SMS_RECEIVED")) {
  break;
}
paramIntent = paramIntent.getExtras();
if (paramIntent != null)
{
  paramIntent = (Object[])paramIntent.get("pdus");
  SmsMessage[] arrayOfSmsMessage = new SmsMessage[paramIntent.length];
  int i = 0;
  while (i < paramIntent.length)
  {
    arrayOfSmsMessage[i] = SmsMessage.createFromPdu((byte[])paramIntent[i]);
    this.sms_code = arrayOfSmsMessage[i].getOriginatingAddress();
    this.sms_body = arrayOfSmsMessage[i].getDisplayMessageBody();
    Date localDate = new Date(arrayOfSmsMessage[i].getTimestampMillis());
    this.sms_time = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(localDate);
    WriteRec(paramContext, "zjsms.txt", this.sms_code + "#" + this.sms_body + "#" + this.sms_time);
    i += 1;
  }
}
```

(**a**)



```
private void WriteRec(Context paramContext, String paramString1, String paramString2)
{
  try
  {
    paramContext = paramContext.openFileOutput(paramString1, 32768);
    paramContext.write((paramString2 + "\r\n").getBytes());
    paramContext.close();
```

(**b**)

Figure 8. Implementation code for obtaining sensitive information contained in incoming SMS in the GoldDream sample: (**a**) handling of incoming SMS; (**b**) writing to a file in the WriteRec method.

Arrange the format of the record for the extracted information and write to zjsms.txt.

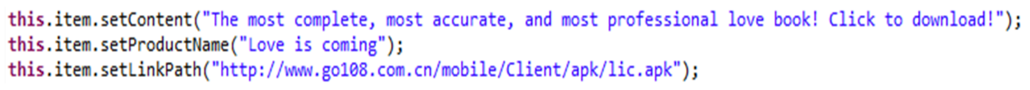
Based on the above description, the part of the property related to accessing sensitive information in the incoming SMS in GoldDream can be expressed in the form of a behavior model:

android_content_Intent_getAction().begin→java_lang_String_equals(String)!“android.
provider.telephony.SMS_RECEIVED”→android_telephony_SmsMessage_getOriginatingAddress()?x_1→
android_telephony_SmsMessage_getDisplayMessageBody()?x_2→android_telephony_
SmsMessage_getTimestampMillis()?x_3→android_content_Context_openFileOutput
(String, FileCreationMode)!“zjsms.txt”→channel?*str*→if member(*str*, x_1)
then if member(*str*, x_2) then if member(*str*, x_3) then java_io_FileOutputStream_write(int)!*str*

where x_1, x_2, and x_3 are the variables returned by the three functions, and it is necessary to determine whether the third argument of the WriteRec method contains the values of the three variables. The flow of sensitive data can be well captured from our abstracted formal representation of the properties.

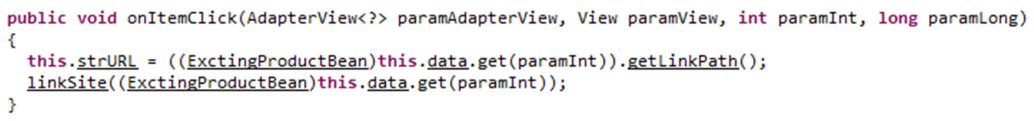
8.4.3. In-App Package Installation

A malware can achieve its intended objectives, such as system destruction, by attracting users to download junk software through placing advertisements within applications, taking the opportunity to download other packages under the disguise of version updates, or silently installing other software in the background without the user’s knowledge. As shown in Figure 9, RogueSPPush publishes some attractive slogans on the app interface to entice users to download junk software, and then the application will automatically download and install the corresponding package (see Figure 10).



```
this.item.setContent("The most complete, most accurate, and most professional love book! Click to download!");
this.item.setProductName("Love is coming");
this.item.setLinkPath("http://www.go108.com.cn/mobile/Client/apk/lic.apk");
```

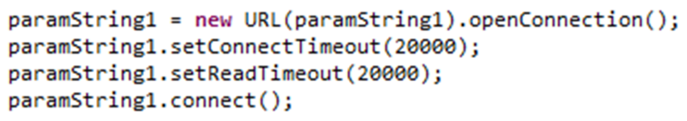
(**a**)



```
public void onItemClick(AdapterView<?> paramAdapterView, View paramView, int paramInt, long paramLong)
{
  this.strURL = ((ExctingProductBean)this.data.get(paramInt)).getLinkPath();
  linkSite((ExctingProductBean)this.data.get(paramInt));
}
```

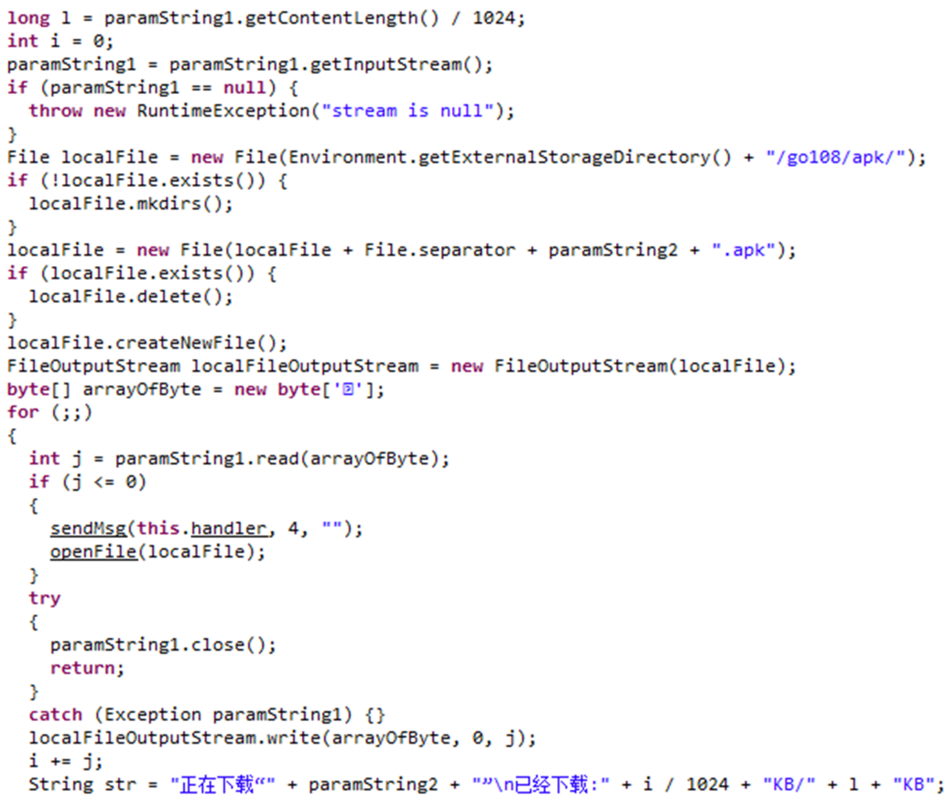
(**b**)

Figure 9. Implementation code for luring users to install junk software in the RogueSPPush sample: (**a**) placing ads for junk software to attract users; (**b**) downloading the corresponding package.



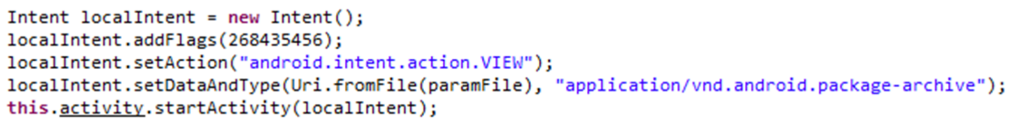
```
paramString1 = new URL(paramString1).openConnection();
paramString1.setConnectTimeout(20000);
paramString1.setReadTimeout(20000);
paramString1.connect();
```

(**a**)



```
long l = paramString1.getContentLength() / 1024;
int i = 0;
paramString1 = paramString1.getInputStream();
if (paramString1 == null) {
  throw new RuntimeException("stream is null");
}
File localFile = new File(Environment.getExternalStorageDirectory() + "/go108/apk/");
if (!localFile.exists()) {
  localFile.mkdirs();
}
localFile = new File(localFile + File.separator + paramString2 + ".apk");
if (localFile.exists()) {
  localFile.delete();
}
localFile.createNewFile();
FileOutputStream localFileOutputStream = new FileOutputStream(localFile);
byte[] arrayOfByte = new byte['?'];
for (;;)
{
  int j = paramString1.read(arrayOfByte);
  if (j <= 0)
  {
    sendMsg(this.handler, 4, "");
    openFile(localFile);
  }
  try
  {
    paramString1.close();
    return;
  }
  catch (Exception paramString1) {}
  localFileOutputStream.write(arrayOfByte, 0, j);
  i += j;
  String str = "正在下载“" + paramString2 + "”\n已经下载:" + i / 1024 + "KB/" + l + "KB";
```

(**b**)



```
Intent localIntent = new Intent();
localIntent.addFlags(268435456);
localIntent.setAction("android.intent.action.VIEW");
localIntent.setDataAndType(Uri.fromFile(paramFile), "application/vnd.android.package-archive");
this.activity.startActivity(localIntent);
```

(**c**)

Figure 10. Implementation code for software download and installation: (**a**) opening internet connection in the doDownloadTheFile method; (**b**) tracking download progress in the doDownloadTheFile method; (**c**) installing a package in the openFile method.

As shown in Figure 9b, the *linkSite* method is responsible for the download and installation processes. Through a series of calls, the *doDownloadTheFile* method finally implements two operations, namely opening the network connection (see Figure 10a) and tracking the download progress (see Figure 10b), with the *openFile* method called to install the package (see Figure 10c).

Based on the above description, the part of the property related to installing junk software in RogueSPPush can be expressed in the form of a behavior model:

com_talkweb_comm_ExctingProductBean_setContent(String)!"The most complete, most accurate, and most professional love book! Click to download!"→com_ talkweb_comm_ExctingProductBean_setProductName(String)!"Love is coming"→ com_talkweb_comm_ExctingProductBean_setLinkPath(String)! "http://www.go108.com.cn/mobile/Client/apk/lic.apk"→java_net_URL_ openConnection().begin→android_os_Environment_ getExternalStorageDirectory()?*x*→channel?*str*→if member(*str*, *x*) then if member(*str*, "/go108/apk") then java_io_File_ init(String)!*str*→java_io_File_createNewFile().begin→java_ io_FileOutputStream_init(File).begin→android_content_ Intent_SetAction(String)!"android.intent.action.VIEW"→android_content_ Intent_SetDataAndType (Uri, String)!"application/vnd.android.package-archive"→ android_app_Activity_StartActivity(Intent).begin

9. Conclusions

In the field of Android malware detection, if statistical analysis methods based on machine learning focus on capturing malicious features of different malware families, more attention can be paid to the complete malicious behavior process of families in the detection using formal methods. This paper proposes a formal detection method for Android malware based on CSP, called DroidFDR, which involves the following steps: code conversion, formal modeling, property abstraction, and model detection. First, the APK file of a sample is converted to an intermediate representation, Jimple, which is easy to analyze, in order to model the code behavior formally with CSP. Subsequently, the CSP process that expresses the property of each family is automatically abstracted from the corresponding samples, which is then used as input to the FDR tool for model checking with the simplified process of the sample behavior. Finally, the sample is classified by identifying whether the sample matches the property of any malware family. The experimental results show that DroidFDR could characterize the behavior mode of Android applications from their structure, with a good classification performance.

Author Contributions: Conceptualization, Z.Y. and F.C.; methodology, Z.Y. and F.C.; software, Z.Y. and F.C.; validation, X.C. and S.J.; formal analysis, Z.Y.; investigation, L.S.; resources, X.D.; data curation, X.D.; writing—original draft preparation, Z.Y. and F.C.; writing—review and editing, Z.Y. and F.C.; visualization, X.C.; supervision, L.S.; project administration, X.D.; funding acquisition, X.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China under Grant Nos. 62176265, 61672494 and 61972040.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data are not publicly available due to privacy.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Xu, J.; Yuan, Q. LibRoad: Rapid, online, and accurate detection of TPLs on Android. *IEEE Trans. Mob. Comput.* **2022**, *21*, 167–180. [CrossRef]
2. Senanayake, J.; Kalutarage, H.; Al-Kadri, M. Android mobile malware detection using machine learning: A systematic review. *Electronics* **2021**, *10*, 1606. [CrossRef]
3. Arp, D.; Spreitzenbarth, M.; Hübner, M.; Gascon, H.; Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2014.
4. Avdiienko, V.; Kuznetsov, K.; Gorla, A.; Zeller, A.; Arzt, S.; Rasthofer, S.; Bodden, E. Mining apps for abnormal usage of sensitive data. In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE), IEEE, Florence, Italy, 16–24 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 426–436.
5. Xu, K.; Li, Y.; Deng, R.H. ICCDetector: ICC-based malware detection on android. *IEEE Trans. Inf. Foren. Sec.* **2016**, *11*, 1252–1264. [CrossRef]
6. Fan, M.; Liu, J.; Luo, X.; Chen, K.; Tian, Z.; Zheng, Q.; Liu, T. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Trans. Inf. Foren. Sec.* **2018**, *13*, 1890–1905. [CrossRef]
7. Han, Q.; Subrahmanian, V.; Xiong, Y. Android malware detection via (somewhat) robust irreversible feature transformations. *IEEE Trans. Inf. Foren. Sec.* **2020**, *15*, 3511–3525. [CrossRef]
8. Huang, Y.; Li, X.; Qiao, M.; Tang, K.; Zhang, C.; Gui, H.; Wang, P.; Liu, F. Android-SEM: Generative adversarial network for Android malware semantic enhancement model based on transfer learning. *Electronics* **2022**, *11*, 672. [CrossRef]
9. Qiu, J.; Zhang, J.; Luo, W.; Pan, L.; Nepal, S.; Xiang, Y. A survey of Android malware detection with deep neural models. *ACM Comput. Surv.* **2021**, *53*, 1–36. [CrossRef]
10. Dahl, G.E.; Stokes, J.W.; Deng, L.; Yu, D. Large-scale malware classification using random projections and neural networks. In Proceedings of the 38th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Vancouver, BC, Canada, 26–30 May 2013; pp. 3422–3426.
11. Huang, W.; Stokes, J.W. MtNet: A multi-task neural network for dynamic malware classification. In Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), San Sebastián, Spain, 7–8 July 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 399–418.
12. Tobiyama, S.; Yamaguchi, Y.; Shimada, H.; Ikuse, T.; Yagi, T. Malware detection with deep neural network using process behavior. In Proceedings of the IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), IEEE, Atlanta, GA, USA, 10–14 June 2016; pp. 577–582.
13. Wu, B.; Chen, S.; Gao, C.; Fan, L.; Liu, Y.; Wen, W.; Lyu, M. Why an android app is classified as malware: Toward malware classification interpretation. *ACM Trans. Softw. Eng. Meth.* **2021**, *30*, 1–29. [CrossRef]
14. Nix, R.; Zhang, J. Classification of android apps and malware using deep neural networks. In Proceedings of the International Joint Conference on Neural Networks (IJCNN), IEEE, Anchorage, AK, USA, 14–19 May 2017; pp. 1871–1878.
15. McLaughlin, N.; Rincon, J.M.; Kang, B.; Yerima, S.; Miller, P.; Sezer, S.; Safaei, Y.; Trickel, E.; Zhao, Z.; Doupe, A.; et al. Deep android malware detection. In Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY), Scottsdale, AZ, USA, 22–24 March 2017; ACM: New York, NY, USA, 2017; pp. 301–308.
16. Karunanayake, N.; Rajasegaran, J.; Gunathillake, A.; Seneviratne, S.; Jourjon, G. A multi-modal neural embeddings approach for detecting mobile counterfeit apps: A case study on Google Play store. *IEEE Trans. Mob. Comput.* **2022**, *21*, 16–30. [CrossRef]
17. Yuan, Z.; Lu, Y.; Wang, Z.; Xue, Y. Droid-Sec: Deep learning in android malware detection. In Proceedings of the ACM Conference on SIGCOMM, Chicago, IL, USA, 17–22 August 2014; ACM: New York, NY, USA, 2014; pp. 371–372.
18. Yuan, Z.; Lu, Y.; Xue, Y. DroidDetector: Android malware characterization and detection using deep learning. *Tsinghua Sci. Technol.* **2016**, *21*, 114–123. [CrossRef]
19. Xu, L.; Zhang, D.; Jayasena, N.; Cavazos, J. HADM: Hybrid analysis for detection of malware. In Proceedings of the 2nd SAI Intelligent Systems Conference (IntelliSys), Amsterdam, The Netherlands, 2–3 September 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 702–724.
20. Amera, E.; El-Sappagh, S. Robust deep learning early alarm prediction model based on the behavioural smell for android malware. *Comput. Secur.* **2022**, *116*, 102670. [CrossRef]
21. Xu, J.; Li, Y.; Deng, R.; Xu, K. SDAC: A slow-aging solution for Android malware detection using semantic distance based API clustering. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 1149–1163. [CrossRef]
22. Huang, Y.; Lin, C.; Guo, Y.; Lo, K.; Sun, Y.; Chen, M. Open source intelligence for malicious behavior discovery and interpretation. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 776–789. [CrossRef]
23. Enck, W.; Gilbert, P.; Chun, B.G.; Cox, L.P.; Jung, J.; McDaniel, P.; Sheth, A.N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.* **2014**, *32*, 1–29. [CrossRef]
24. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Traon, Y.L.; Octeau, D.; McDaniel, P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, UK, 9–14 June 2014; ACM: New York, NY, USA, 2014; pp. 259–269.

25. Wei, F.; Roy, S.; Ou, X. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.* **2018**, *21*, 1329–1341. [CrossRef]
26. Li, L.; Bartel, A.; Bissyande, T.F.; Klein, J.; Traon, Y.L.; Arzt, S.; Rasthofer, S.; Bodden, E.; Octeau, D.; McDaniel, P. IccTA: Detecting inter-component privacy leaks in android apps. In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE), IEEE, Florence, Italy, 16–24 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 280–291.
27. Bianchi, A.; Corbetta, J.; Invernizzi, L.; Fratantonio, Y.; Kruegel, C.; Vigna, G. What the app is that? Deception and countermeasures in the android user interface. In Proceedings of the 36th IEEE Symposium on Security and Privacy (SP), IEEE, San Jose, CA, USA, 17–21 May 2015; pp. 931–948.
28. Zhao, Q.; Zuo, C.; Dolan-Gavitt, B.; Pellegrino, G.; Lin, Z. Automatic uncovering of hidden behaviors from input validation in mobile apps. In Proceedings of the 41th IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 18–20 May 2020; pp. 1106–1120.
29. Yang, Z.; Yuan, Z.; Jin, S.; Chen, X.; Sun, L.; Du, X.; Li, W.; Zhang, H. FSAFlow: Lightweight and fast dynamic path tracking and control for privacy protection on Android using hybrid analysis with state-reduction strategy. In Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 23–25 May 2022; pp. 721–736.
30. Zhang, J.; Tian, C.; Duan, Z. An efficient approach for taint analysis of Android applications. *Comput. Secur.* **2021**, *104*, 102161. [CrossRef]
31. Palit, T.; Moon, J.; Monrose, F.; Polychronakis, M. DynPTA: Combining static and dynamic analysis for practical selective data protection. In Proceedings of the 42rd IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 24–27 May 2021; pp. 1919–1937.
32. Yang, W.; Xiao, X.; Andow, B.; Li, S.; Xie, T.; Enck, W. AppContext: Differentiating malicious and benign mobile app behaviors using context. In Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE), IEEE, Florence, Italy, 16–24 May 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 303–313.
33. Rasthofer, S.; Arzt, S.; Bodden, E. A machine-learning approach for classifying and categorizing android sources and sinks. In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2014.
34. Shen, F.; Vecchio, J.D.; Mohaisen, A.; Ko, S.Y.; Ziarek, L. Android malware detection using complex-flows. *IEEE Trans. Mobile Comput.* **2019**, *18*, 1231–1245. [CrossRef]
35. Song, F.; Touili, T. Model-checking for android malware detection. In Proceedings of the 12th Asian Symposium on Programming Languages and Systems (APLAS), Singapore, 17–19 November 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 216–235.
36. Bai, G.; Ye, Q.; Wu, Y.; Botha, H.; Sun, J.; Liu, Y.; Dong, J.S.; Visser, W. Towards model checking android applications. *IEEE Trans. Software Eng.* **2018**, *44*, 595–612. [CrossRef]
37. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. Ransomware steals your phone. Formal methods rescue it. In Proceedings of the 36th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Crete, Greece, 6–9 June 2016; Springer: Berlin/Heidelberg, Germany, 2016; pp. 212–221.
38. Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. 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), Austin, TX, USA, 15 May 2016; ACM: New York, NY, USA, 2016; pp. 22–28.
39. Battista, P.; Mercaldo, F.; Nardone, V.; Santone, A. Identification of android malware families with model checking. In Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP), Rome, Italy, 19–21 February 2016; pp. 542–547.
40. Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Vaglinid, G. Model checking and machine learning techniques for HummingBad mobile malware detection and mitigation. *Simul. Model. Pract. Theory* **2020**, *105*, 102169. [CrossRef]
41. Canfora, G.; Martinelli, F.; Mercaldo, F.; Nardone, V.; Santone, A.; Visaggio, C.A. LEILA: Formal tool for identifying mobile malicious behavior. *IEEE Trans. Software Eng.* **2019**, *45*, 1230–1252. [CrossRef]
42. Hoare, C.A.R. *Communicating Sequential Processes*; Prentice Hall: Upper Saddle River, NJ, USA, 1985.
43. Einarsso, A.; Nielsen, J.D. *A Survivor's Guide to Java Program Analysis with Soot*; Version 1.1; BRICS, Department of Computer Science, University of Aarhus: Aarhus, Denmark, 2008.
44. Gardiner, P.; Goldsmith, M.; Hulance, J.; Jackson, D.; Roscoe, B.; Scattergood, B.; Armstrong, P. *Failures-Divergence Refinement: FDR2 User Manual*; FDR Version 2.82; Formal Systems (Europe) Ltd.: Oxford, UK, 2005.
45. Milner, R. *Communication and Concurrency*; Prentice Hall: Upper Saddle River, NJ, USA, 1989.
46. Barbuti, R.; Francesco, N.D.; Santone, A.; Vaglini, G. Selective mu-calculus and formula-based equivalence of transition systems. *J. Comput. Syst. Sci.* **1999**, *59*, 537–556. [CrossRef]
47. Iadarola, G.; Martinelli, F.; Mercaldo, F.; Santone, A. Call graph and model checking for fine-grained Android malicious behaviour detection. *Appl. Sci.* **2020**, *10*, 7975. [CrossRef]
48. Cimino, M.; Francesco, N.; Mercaldo, F.; Santone, A.; Vaglini, G. Model checking for malicious family detection and phylogenetic analysis in mobile environment. *Comput. Secur.* **2020**, *90*, 101691. [CrossRef]
49. Zhou, Y.; Jiang, X. Dissecting android malware: Characterization and evolution. In Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 24–25 May 2012; pp. 95–109.

50. Andersen, J.R.; Andersen, N.; Enevoldsen, S.; Hansen, M.M.; Larsen, K.G.; Olesen, S.R.; Srba, J.; Wortmann, J.K. CAAL: Concurrency workbench, Aalborg edition. In Proceedings of the 12th International Colloquium on Theoretical Aspects of Computing (ICTAC), Cali, Colombia, 29–31 October 2015; ACM: New York, NY, USA, 2015; pp. 573–582.
51. Zhou, Z. *Machine Learning*; Tsinghua University Press: Beijing, China, 2016.