Identifying Mobile Repackaged Applications through Formal Methods

Fabio Martinelli[1], Francesco Mercaldo[1], Vittoria Nardone[2], Antonella Santone[2], Corrado Aaron Visaggio[2]

[1]*Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy*

[2]*Department of Engineering, University of Sannio, Benevento, Italy*

{*fabio.martinelli, francesco.mercaldo*}*@iit.cnr.it*, {*vnardone, santone, visaggio*}*@unisannio.it*

Keywords: Security, malware, model checking, Android, testing.

Abstract: Smartphones and tablets are rapidly become indispensable in every day activities. Android has become the most popular operating system for mobile environments in the world. These devices, owing to the open nature of Android, are continuously exposed to attacks, mostly to data exfiltration and monetary fraud. There are many techniques to embed the bad code, i.e. the instructions able to perform a malicious behaviour, into a legitimate application: the most diffused one is the so-called repackaged, that consists of reverse engineer the application in order to embed the malicious code and then (re)distribute them in the official and/or third party markets. In this paper we propose a technique to localize malicious payload of GinMaster family, one of the most representative repackaged trojan in Android environment. We obtain encouraging results, achieving an accuracy equal to 0.9.

1 Introduction

In 2015, the volume of mobile malware continued to grow. From 2004 to 2013 security experts of SecurList detected nearly 200,000 samples of malicious mobile code. In 2014 there were 295,539 new programs, while the number was 884,774 in 2015. Each malware sample has several installation packages: in 2015, they detected 2,961,727 malicious mobile installation packages [SecureList, 2015].

Signature-based malware detection, which is the most common technique adopted by commercial antimalware for mobile, is often ineffective. Moreover it is costly, as the process for obtaining and classifying a malware signature is laborious and time-consuming.

In order to mitigate the malware trend in February 2011, Google introduced Bouncer [GoogleMobile, 2014] to screen submitted apps for detecting malicious behaviors, but this has not eliminated the problem, as it is discussed in [Oberheide and Miller, 2012].

The Fraunhofer Research Institution for Applied and Integrated Security has performed an evaluation of antivirus for Android [Fedler et al., 2014]: the conclusion is there are many techniques for evading the detection of most antivirus and for installing malicious payload.

The most employed installation technique is the so-called repackaging [Zhou and Jiang, 2012]: the attacker decompiles a trusted application to get the source code, then adds the malicious payload and recompiles the application with the payload to make it available on various market alternatives, and sometimes also on the official market. The user is often encouraged to download such malicious applications because they are free versions of trusted applications sold on the official market.

Scientific community in last years has proposed a lot of static [Canfora et al., 2016, Liang and Du, 2014, Yerima et al., 2013, Arp et al., 2014, Spreitzenbarth et al., 2013] and dynamic [Isohara et al., 2011, Tchakount and Dayang, 2013, Reina et al., 2013] techniques basically based on machine learning methods in order to solve the problem: the main limitation in this case is due to the false positive ratio.

Indeed existing solutions for protecting privacy and security on smartphones are still ineffective in many facets [Marforio et al., 2011], and many analysts warn that the malware families and their variants for Android are rapidly increasing. This scenario calls for new security models and tools to limit the spreading of malware for smartphones.

For these reasons, in this paper we evaluate the effectiveness of a model-checking approach to identify Android malware. We evaluate our method using *GinMaster* malware, one of the most widespread family in mobile malware landscape, with over 6,000 known variants belonging to this family.

The approach can be easily applied to other repackaged families: as reported in (citazione), the so-called repackaged malware contains the malicious payload at installation time, this is the reason why the payload is embedded into the application and the logic rules, able to verify the existance of the malicious payload, can be verified without run the sample.

The reason why we explore the effectiveness of our approach on GinMaster family is represented by the fact that this is one of the most widespread trojan malware family in mobile environment. The payload belonging to this family is embedded into legitimate applications using repackaging technique. GinMaster has gone through three significant generations since it was first found by researchers from North Carolina State University on 17 August 2011.

GinMaster is distributed in third-party app markets in China. The attackers injected GinMaster code into thousands of legitimate game, ringtone and picture applications. These applications have more chance to lure mobile users into installing the malicious payload.

The trojan contains a malicious service able to root specific devices in order to escalate privileges. It also has the ability to modify and delete contents in the SD card of device, steal confidential information and send it to a remote website, execute command-and-control services from the remote website, as well as download and install applications regardless of user interaction.

The approach can be easily applied to other repackaged families: as reported in [Zhou and Jiang, 2012], the so-called repackaged malware contains the malicious payload at installation time, this is the reason why the payload is embedded into the application and the logic rules, able to verify the existence of the malicious payload, can be verified without run the sample.

The salient characteristics of our methodology are:

- the use of formal methods;
- the inspection of Java Bytecode and not on the source code;
- the use of static analysis;
- the capture of malicious behaviours at a finer granularity.

In practice, from the Java Bytecode application files we generate CCS processes, which are successively used for checking properties expressing the most common behaviours exhibit by *GinMaster* family samples.

Performing automatic analysis on the Bytecode and not directly on the source code has several advantages:

- independence of the source programming language;
- identification of *GinMaster* without decompilation even when source code is lacking;
- ease of parsing a lower-level code;
- independence from obfuscation.

The paper proceeds as follows: Section 2 discusses related work; Section 3 describes and motivates our approach; Section 4 illustrates the results of experiments; finally, conclusions are drawn in the Section 5.

2 Related work

In this section we review the current literature related to malware identification with particular regards to malicious family identification.

A very thorough dissecting of GinMaster family is provided in [Yu, 2013]. The paper gives an overview of three generations of the GinMaster family, examines the core malicious functionality, tracks their evolution from source code, and presents notable techniques utilized by the specific variants. Basically, Ginmaster is able to set-up via mobile botnet malicious code hidden in the affected app. However, instead of directly taking advantage of these zombies devices to make profit from end-users, the malware controller employs a botnet to generate millions of installations and large volumes of advertising traffic to legitimate developers and advertising services.

In [Canfora et al., 2016] authors experimentally evaluate two techniques for detecting Android malware: the first one is based on Hidden Markov Model (HMM), while the second one exploits Structural Entropy. They demonstrate that these methods obtain a precision of 0.96 to discriminate a malware application. In addition they also analyse ransomware samples, obtaining a precision of 0.961 with LADTree classification algorithm using the Structural Entropy method, and a precision of 0.824 with J48 algorithm using the HMM one in ransomware identification.

Song et al. [Song et al., 2016] propose a framework to statically detect Android malware, consisting of four layers of filtering mechanisms: the message digest values, the combination of malicious permissions, the dangerous permissions, and the dangerous intention. As additional contribute, they propose a novel threat degree threshold model of dangerous permissions on malware detection. They experiment the method on real mobile devices, using 83 real mobile devices and achieving a 98.8% pass rate, where the versions of Android range from 2.3 to 5.1.

Neuhaus et al. [Neuhaus and Zimmermann, 2010] crawl the vulnerability reports in the Common Vulnerability and Exposures database by using topic models to find prevalent vulnerability types and new trends semi-automatically. They analyze 39393 unique reports until the end of 2009, with the aim of characterizing many vulnerability trends: SQL injection (PHP), buffer overflows, format strings, cross-site request forgery and so on.

Zhao et al. [Zhao et al., 2014] propose a LDA-based method to analyze the trends of network security which consist of three steps: collect data from web sites, extract topics from the collected data, and makes the curves of trends over time. They select 620 documents sorted by time and extract 10 topic from each document. Six interesting topics are discovered by LDA model, according to the autors: dns-ddos, vulnerability, mobile-malware, mac-malware, Browser malware and Java-vulnerability.

Formal methods have been applied for studying malware in some recent papers. In [Kinder et al., 2005] the authors introduce the specification language CTPL (Computation Tree Predicate Logic) confirming the malicious behavior of thirteen Windows malware variants using as dataset a set of worms dating from 2002 to 2004.

Song et al. present an approach to model Microsoft Windows XP binary programs as a PushDown System (PDS) [Song and Touili, 2001]. They evaluate 200 malware variants (generated by NGVCK and VCL32 engines) and 8 benign programs.

The tool PoMMaDe [Song and Touili, 2013] is able to detect 600 real malware, 200 malware generated by two malware generators (NGVCK and VCL32), and proves the reliability of benign programs: a Microsoft Windows binary program is modeled as a PDS which allows to track the stack of the program.

Song et al. model mobile applications using a PDS in order to discovery private data leaking working at Smali code level [Song and Touili, 2014]. Illegal flow of information in Java bytecode has been also studied in [Bernardeschi et al., 2004], using a static analysis approach.

Jacob and colleagues provide a basis for a malware model, founded on the Join-Calculus: they consider the system call sequences to build the model [Jacob et al., 2010].

Recently, the possibility to identify the malicious payload in Android malware using a model checking based approach has been explored in [Battista et al., 2016, Mercaldo et al., 2016a, Mercaldo et al., 2016b]. Starting from payload behavior definition they formulate logic rules and then test them by using a real world dataset composed by Ransomware, DroidKungFu, Opfake families and update attack samples.

As it emerges from the literature in the last years, formal methods have been applied to detect mobile malware, but at the best knowledge of the authors they have never been applied for identifying specifically the repackaged attack provided by GinMaster family on Android malware.

3 The Method

In this section a model checking-based approach for the detection of GinMaster apps is presented. While model checking [Barbuti et al., 2005] was originally developed to verify the correctness of systems against specifications, recently it has been highlighted in connection with a variety of disciplines such as biology [De Ruvo et al., 2015], clone detection [Santone, 2011], secure information flow [Barbuti et al., 2002], among others. In this paper we present the use of model checking in the security field. Fig. 1 describes all the phases of our approach.

During the first phase, we generate a formal model from the Java Bytecode of the .class files derived by the app under analysis. As formal specification language, we use Milner's Calculus of Communicating Systems (CCS) [Milner, 1989], one of the most well known process algebras. CCS contains basic operators to build finite processes, communication operators to express concurrency, and some notion of recursion to capture the infinite behaviour. Thus, the formal model is obtained by transforming each Java Bytecode instruction in CCS processes. More precisely, from the CCS we generate an automaton such that the nodes represent instruction addresses while the edges (labeled with opcodes) represent the control-flow transitions from one instruction to it successors(s).

In the second phase, we try to discover Android malware GinMaster apps. The behavior of the GinMaster family is encoded into a property φ expressed in a branching temporal logic: the mu-calculus logic [Stirling, 1989]. Temporal logics are logical formalisms for expressing properties such as liveness and safety properties. The syntax of the mu-calculus

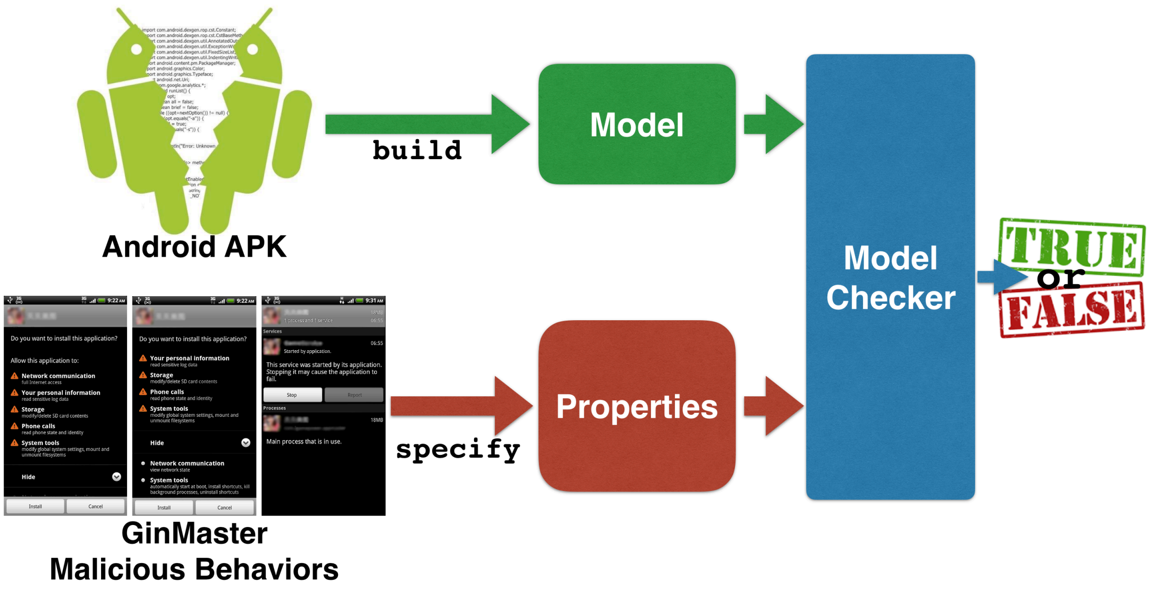


Figure 1: The Workflow of The Approach.

is the following, where K ranges over sets of actions and Z ranges over variables:

$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \wedge \phi \mid \phi \vee \phi \mid [K]\phi \mid$$
$$\langle K \rangle \phi \mid \nu Z.\phi \mid \mu Z.\phi$$

A fixpoint formula has the form $\mu Z.\phi$ (resp. $\nu Z.\phi$) where μZ (resp. νZ) *binds* free occurrences of Z in ϕ. An occurrence of Z is free if it is not within the scope of a binder μZ (resp. νZ). A formula is *closed* if it contains no free variables. $\mu Z.\phi$ is the least fixpoint of the recursive equation $Z = \phi$, while $\nu Z.\phi$ is the greatest one. From now on we consider only closed formulae.

The satisfaction of a formula ϕ by a state s of a transition system is defined as follows:

- each state satisfies $\mathtt{tt}$ and no state satisfies $\mathtt{ff}$;
- a state satisfies $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) if it satisfies ϕ_1 or (and) ϕ_2.
- $[K]\,\phi$ is satisfied by a state which, for every performance of an action in K, evolves to a state obeying ϕ.
- $\langle K \rangle\, \phi$ is satisfied by a state which can evolve to a state obeying ϕ by performing an action in K.

For the precise definition of the satisfaction of a closed formula φ by a state s (written $s \models \varphi$) the reader can refer to [Stirling, 1989].

For example, $\mu Y. \langle a \rangle\, \mathtt{tt} \wedge \langle -a, b \rangle\, Y$ means that "it is possible to perform the action a non preceded by the action b".

The CCS formal model, generated by the Java Bytecode during the first phase, is now used to prove the property φ: using the model checking we determine the detection of Ginmaster malware apps.

In Table 1 we show the GinMaster formulae to give the reader the flavour of the approach followed.

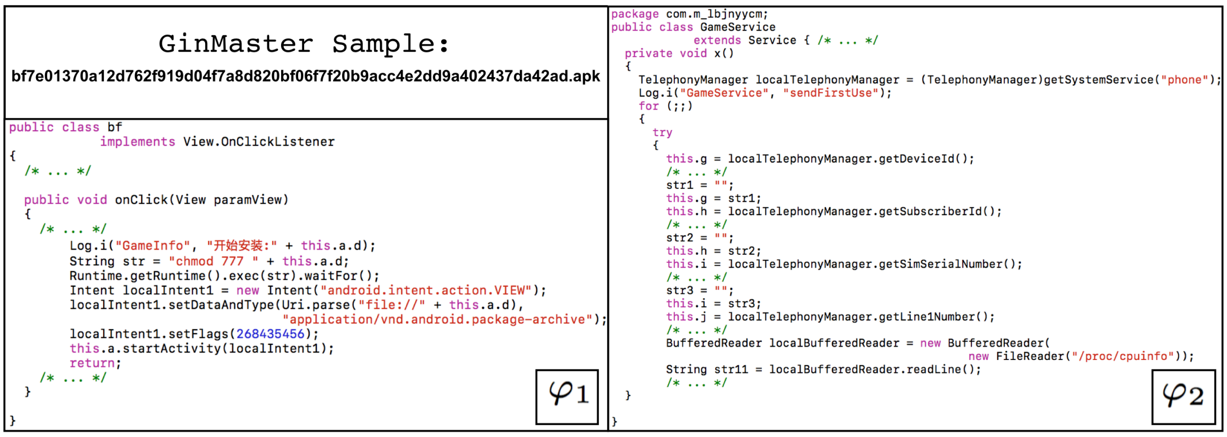
Table 1 shows the logic formulae to catch the GinMaster malicious payload: the first one, i.e., φ_1, is able to catch the root ability of the GinMaster malicious payload; the second one, i.e., φ_2, identifies the gather of user information, one of the most representative mobile trojan behavior, while φ_3 identifies the commands to communicate with the C&C server. Figure 2 shows two snippets of code belonging to a GinMaster sample. The reported snippets are related to the two malicious behaviours catch by the logic formulae φ_1 and φ_2.

More precisely, formula φ_1 is able to catch following actions:

- "*new javalangStringBuilder*": it represents mutable sequence of characters, typically used to build path where resources, i.e. files, are located. In the case of GinMaster the built string represents the path of the script able to root the device;
- "*pushchmod*": to successfully run the script the malware needs to set the admin privileges to the root script, this action is performed with the "chmod" command;
- "*invokeappend*": this represent a method of the StringBuilder class used to concatenate different String;
- "*invokeexec*": this instruction is able to run the

Table 1: The formulae for GinMaster payload detection

$$
\begin{aligned}
\varphi_1 &= \mu X.\, \langle new\, javalangStringBuilder \rangle\, \varphi_{1_1} \vee \langle - new\, javalangStringBuilder \rangle\, X \\
\varphi_{1_1} &= \mu X.\, \langle pushchmod \rangle\, \varphi_{1_2} \vee \langle - pushchmod \rangle\, X \\
\varphi_{1_2} &= \mu X.\, \langle invokeappend \rangle\, \varphi_{1_3} \vee \langle - invokeappend \rangle\, X \\
\varphi_{1_3} &= \mu X.\, \langle invokeexec \rangle\, \varphi_{1_4} \vee \langle - invokeexec \rangle\, X \\
\varphi_{1_4} &= \mu X.\, \langle invokewaitFor \rangle\, \varphi_{1_5} \vee \langle - invokewaitFor \rangle\, X \\
\varphi_{1_5} &= \mu X.\, \langle pushfile \rangle\, \mathtt{tt} \vee \langle - pushfile \rangle\, X \\
\\
\varphi_2 &= \mu X.\, \langle pushphone \rangle\, \varphi_{2_1} \vee \langle - pushphone \rangle\, X \\
\varphi_{2_1} &= \mu X.\, \langle invokegetSystemService \rangle\, \varphi_{2_2} \vee \langle - invokegetSystemService \rangle\, X \\
\varphi_{2_2} &= \mu X.\, \langle checkcastandroidtelephonyTelephonyManager \rangle\, \varphi_{2_3} \vee \\
&\quad \langle - checkcastandroidtelephonyTelephonyManager \rangle\, X \\
\varphi_{2_3} &= \mu X.\, \langle invokegetDeviceId \rangle\, \varphi_{2_4} \vee \langle - invokegetDeviceId \rangle\, X \\
\varphi_{2_4} &= \mu X.\, \langle invokegetSubscriberId \rangle\, \varphi_{2_5} \vee \langle - invokegetSubscriberId \rangle\, X \\
\varphi_{2_5} &= \mu X.\, \langle invokegetSimSerialNumber \rangle\, \varphi_{2_6} \vee \langle - invokegetSimSerialNumber \rangle\, X \\
\varphi_{2_6} &= \mu X.\, \langle invokegetLineUunoNumber \rangle\, \varphi_{2_7} \vee \langle - invokegetLineUunoNumber \rangle\, X \\
\varphi_{2_7} &= \mu X.\, \langle new\, javaioBufferedReader \rangle\, \mathtt{tt} \vee \langle - new\, javaioBufferedReader \rangle\, X \\
\\
\varphi_3 &= \mu X.\, \langle A,B,C,D,E,F,G,H,I,J,K,L,M,N \rangle\, \mathtt{tt} \vee \langle - A,B,C,D,E,F,G,H,I,J,K,L,M,N \rangle\, X
\end{aligned}
$$



GinMaster Sample:
bf7e01370a12d762f919d04f7a8d820bf06f7f20b9acc4e2dd9a402437da42ad.apk

```
public class bf
            implements View.OnClickListener
{
  /* ... */

  public void onClick(View paramView)
  {
    /* ... */
        Log.i("GameInfo", "开始安装:" + this.a.d);
        String str = "chmod 777 " + this.a.d;
        Runtime.getRuntime().exec(str).waitFor();
        Intent localIntent1 = new Intent("android.intent.action.VIEW");
        localIntent1.setDataAndType(Uri.parse("file://" + this.a.d),
                            "application/vnd.android.package-archive");
        localIntent1.setFlags(268435456);
        this.a.startActivity(localIntent1);
        return;
    /* ... */
  }

}
```

```
package com.m_lbjnyycm;
public class GameService
            extends Service { /* ... */
  private void x()
  {
    TelephonyManager localTelephonyManager = (TelephonyManager)getSystemService("phone");
    Log.i("GameService", "sendFirstUse");
    for (;;)
    {
      try
      {
        this.g = localTelephonyManager.getDeviceId();
        /* ... */
        str1 = "";
        this.g = str1;
        this.h = localTelephonyManager.getSubscriberId();
        /* ... */
        str2 = "";
        this.h = str2;
        this.i = localTelephonyManager.getSimSerialNumber();
        /* ... */
        str3 = "";
        this.i = str3;
        this.j = localTelephonyManager.getLine1Number();
        /* ... */
        BufferedReader localBufferedReader = new BufferedReader(
                                        new FileReader("/proc/cpuinfo"));
        String str11 = localBufferedReader.readLine();
        /* ... */
  }
}
```

Figure 2: Code Snippet Related to Logic Formulae φ_1 and φ_2.

specified command and arguments in a separate process with the specified environment and working directory. The "exec" method, belonging to "Runtime" class, represents the command able to run the script to root the phone that already obtained the admin privileges;

- "*invokewaitFor*": this instruction causes the current thread to wait, if necessary, until the process represented by this Process object has terminated. This method returns immediately if the subprocess has already terminated. If the subprocess has not yet terminated, the calling thread will be blocked until the subprocess exits. In this case the subprocess is represented by the execution of the script to root the device;
- "*pushphone*": it represents the retrieval of a file from an external source and the storage on the device; in this case the stored file in the device is the script the will be run in a separate thread.

Instead, formula φ_2 identifies the personal information gathering capability of the malicious payload performing following actions:

- "*invokegetSystemService*": it represents an interface to global information about the application environment. This is an abstract class whose implementation is provided by the Android system. It allows access to application-specific resources and classes, as well as up-calls for application-level operations such as launching activities, broadcasting and receiving intents;
- "*checkcastandroidtelephonyTelephonyManager*": it provides access to information about the tele-

phony services on the device. Applications can use the methods in this class to determine telephony services and states, as well as to access some types of subscriber information. Applications can also register a listener to receive notification of telephony state changes; "*invokegetDeviceId*": it is a method of the "TelephonyManager" class provided by Android environment and it returns the unique device ID, for instance, the IMEI for GSM and the MEID or ESN for CDMA phones; "*invokegetSubscriberId*": another method provided by "TelephonyManager" class Returns the unique subscriber ID, for example, the IMSI for a GSM phone;

- "*invokegetSimSerialNumber*": this method, belonging to "TelephonyManager" class, it returns the serial number of the SIM inserted into the device;
- "*invokegetLineUunoNumber*": this method returns the phone number string for line 1;
- "*newjavaioBufferedReader*": an object belonging to the BufferedReader class is able to read text from a character-input stream and buffer characters. In this case the buffer contains the personal information retrieved previously by the malicious payload.

The formula φ_3 identifies the Command and Control (C&C) list of instructions. Table 2 shows all the strings of commands and reports a brief description of them. In some sample these strings are encrypted and the algorithm used for decryption is shown in Figure 3. The decryption module (as Figure 3 shows) uses the XOR Byte to Byte with key 0x18 after decoding in Base 64. The first generation of GinMaster family exhibits the C&C instruction not encrypted, while in the second one the encryption of the commands is introduced. In order to evade detection by antimalware software, the second generation obfuscates class names and encrypts URLs as well as C&C instructions. It is impossible to catch this variant by detecting the class name or URLs. In both the generations, the malware has the capability of reporting package information relating to packages installed/uninstalled in the system, searching and listing package information from remote websites, and downloading additional applications to the device without the consent of the user.

The model checker accepts two inputs: the formal model of the app and the property expressing the malware characteristics of the GinMaster family. If the model checker returns true it means that we consider the app belonging to the GinMaster family, while if it returns false it means that the app can be either trusted of belong to another malware family. As formal verification environment in this paper we use the Concurrency Workbench of New Century (CWB-NC) [Cleaveland and Sims, 1996] which supports several different specification languages, among which CCS.

Since CWB-NC is no longer in active development, as future work we want to substitute CWB-NC with CALL (standing for Concurrency workbench developed at AALborg university) [Andersen et al., 2015], which supports CCS as input specification language (as CWB-NC), but uses a more efficient algorithm to perform model checking. Actually, there exist mature tools with modern designs like CADP [Garavel et al., 2013] with expressive input languages and efficient analysis methods. However, our aim is to develop an initial rapid research prototype to evaluate how our approach is able to identify GinMaster apps. For the same reason, logic properties are formulated manually, but as future work we plan to build rules automatically using, for instance, machine learning or clone detection.

An important feature of our approach is that an automatic dissection of an app can be achieved, with the advantage of the localization into the code of the instructions that implement the malicious behavior. To localize the payload, no manual inspection is needed. In fact, our approach is able to identify the exact position of the instructions characterizing the malicious behaviour, with a precision at method level.

This is a very novel result in the malware analysis. In addition, starting from the consideration that we analyze Java bytecode, our methodology is able to correctly identifies the malicious payload also when trivial obfuscation techniques (i.e., nop insertion, junk code, call reordering) are applied to Java source code [Mercaldo et al., 2016b].

4 Experiment

In this section we discuss the experiment we performed to evaluate the effectiveness of our approach in recognizing GinMaster payload, discriminating samples belonging to other Android malware families.

4.1 Dataset

The real world samples examined in the experiment were gathered from the Drebin project's dataset [Arp et al., 2014, Spreitzenbarth et al., 2013]: a very well known collection of malware used in many scientific works, which includes the most diffused Android

Table 2: The list of C&C commands.

ACTION	STRING	DESCRIPTION
A	”http://client.go360days.com/report/first_run.do”	Report the starting of GinMaster.
B	”http://client.go360days.com/request/tableclass.do”	show information stored in SQLite database
C	”http://client.go360days.com/request/config.do”	Change the frequency configuration for checking into the server.
D	”http://client.go360days.com/request/alert.do”	alert_last_id
E	”http://client.go360days.com/request/push.do”	soft_last_id
F	”http://client.go360days.com/report/return_config.do”	show configuration
G	”http://client.go360days.com/report/return_alert.do”	send alert
H	”http://client.go360days.com/report/return_push.do”	push file into the device
I	”http://client.go360days.com/report/install_list.do”	Report information when installing a list of packages.
J	”http://client.go360days.com/report/listener.do”	check the communication between server and device
K	”http://client.go360days.com/client.php?action=soft&soft_id=”	Get a link to a specified software.
L	”http://client.go360days.com/client.php?action=softlist&type=search&word=”	Search a list of software with a specified word.
M	”http://client.go360days.com/client.php?action=softlist”	Get the list of the available software
N	”http://client.go360days.com/client.php?action=list&list_id=9”	Get the software with the specified id

```java
package com.patRQsCtV;
public class bj {
    /* ... */
public static String b(String paramString)
  {
    if ((paramString != null) && (paramString.length() > 0))
    {
      byte[] arrayOfByte = paramString.getBytes();
      for (int k = 0; k < arrayOfByte.length; k++) {
        arrayOfByte[k] = ((byte)(0x18 ^ arrayOfByte[k]));
      }
      return new String(arrayOfByte);
    }
    return "";
  }
    /* ... */
}
```

Figure 3: Decryption algorithm used to decode the C&C list of commands.

families.

Malware dataset is also partitioned according to the *malware family*: each family contains samples which have in common several characteristics, like payload installation, the kind of attack and events that trigger malicious payload [Zhou and Jiang, 2012].

Table 3 shows the 10 malware families with the largest number of applications in our malware dataset with installation type, kind of attack and event activating malicious payload.

The malware was retrieved from the Drebin project [Arp et al., 2014, Spreitzenbarth et al., 2013] taking into account the top 10 most populous families.

We briefly describe the malicious payload action for the top 10 populous families in our dataset.

1. The samples of *FakeInstaller* family have the main payload in common but have different code implementations, and some of them also have an extra payload. FakeInstaller malware is server-side polymorphic, which means the server could provide different `.apk` files for the same URL request. There are variants of FakeInstaller that not only send SMS messages to premium rate numbers, but also include a backdoor to receive commands from a remote server. There is a large number of variants for this family, and it has distributed in hundreds of websites and alternative markets. The members of this family hide their malicious code inside repackaged version of popular applications. During the installation process the malware sends expensive SMS messages to premium services owned by the malware authors.
2. *DroidKungFu* installs a backdoor that allows attackers to access the smartphone when they want and use it as they please. They could even turn it into a bot. This malware encrypts two known root exploits, exploit and rage against the cage, to break out of the Android security container. When it runs, it decrypts these exploits and then contacts a remote server without the user knowing.
3. *Plankton* uses an available native functionality (i.e., class loading) to forward details like IMEI and browser history to a remote server. It is present in a wide number of versions as harmful adware that download unwanted advertisements and it changes the browser homepage or add unwanted bookmarks to it.
4. The *Opfake* samples make use of an algorithm that can change shape over time so to evade the antimalware. The Opfake malware demands payment for the application content through premium text messages. This family represents an example of polymorphic malware in Android environment: it is written with an algorithm that can change shape over time so to evade any detection by signature based antimalware.

Table 3: Families in Drebin dataset with details of the installation method (standalone, repackaging, update), the kind of attack (trojan, botnet), the events that trigger the malicious payload and a brief family description.

Family	Installation	Attack	Activation	Description
FakeInstaller	s	t,b		server-side polymorphic family
Plankton	s,u	t,b		it uses class loading to forward details
DroidKungFu	r	t	boot,batt,sys	it installs a backdoor
GinMaster	r	t	boot	malicious service to root devices
BaseBridge	r,u	t	boot,sms,net,batt	it sends information to a remote server
Adrd	r	t	net,call	it compromises personal data
Kmin	s	t	boot	it sends info to premium-rate numbers
Geinimi	r	t	boot,sms	first Android botnet
DroidDream	r	b	main	botnet, it gained root access
Opfake	r	t		first Android polymorphic malware

5. *GinMaster* family contains a malicious service with the ability to root devices to escalate privileges, steal confidential information and send to a remote website, as well as install applications without user interaction. It is also a trojan application and similarly to the DroidKungFu family the malware starts its malicious services as soon as it receives a BOOT_COMPLETED or USER_PRESENT intent. The malware can successfully avoid detection by mobile anti-virus software by using polymorphic techniques to hide malicious code, obfuscating class names for each infected object, and randomizing package names and self-signed certificates for applications.
6. *BaseBridge* malware sends information to a remote server running one ore more malicious services in background, like IMEI, IMSI and other files to premium-rate numbers. BaseBridge malware is able to obtain the permissions to use Internet and to kill the processes of antimalware application in background.
7. *Kmin* malware is similar to BaseBridge, but does not kill antimalware processes.
8. *Geinimi* is the first Android malware in the wild that displays botnet-like capabilities. Once the malware is installed, it has the potential to receive commands from a remote server that allows the owner of that server to control the phone. Geinimi makes use of a bytecode obfuscator. The malware belonging to this family is able to read, collect, delete SMS, send contact informations to a remote server, make phone call silently and also launch a web browser to a specific URL to start files download.
9. *Adrd* family is very close to Geinimi but with less server side commands, it also compromises personal data such as IMEI and IMSI of infected device. In addiction to Geinimi, this one is able to modify device settings.
10. *DroidDream* is another example of botnet, it gained root access to device to access unique identification information. This malware could also downloads additional malicious programs without the user's knowledge as well as open the phone up to control by hackers. The name derives from the fact that it was set up to run between the hours of 11pm and 8am when users were most likely to be sleeping and their phones less likely to be in use.

4.2 Evaluation

To estimate the detection performance of our methodology we compute the metrics of precision and recall, F-measure (Fm) and Accuracy (Acc), defined as follows:

$$PR = \frac{TP}{TP+FP}; \; RC = \frac{TP}{TP+FN};$$

$$Fm = \frac{2PR\,RC}{PR+RC}; \; Acc = \frac{TP+TN}{TP+FN+FP+TN}$$

where TP is the number of malware that was correctly identified in the GinMaster family (True Positives), TN is the number of malware correctly identified as not belonging to the GinMaster family (True Negatives), FP is the number of malware that was incorrectly identified in the GinMaster family (False Positives), and FN is the number of malware that was not identified as belonging to the GinMaster family (False Negatives).

Table 4 shows the results obtained using our method.

We consider in the column *GinMaster* the samples belonging to *GinMaster* family, while in the column *Malware* the malware samples belonging to others families considered in the study: the detail about the malicious payload of the family we considered is shown in Table 3. We demonstrate the effectiveness

Table 4: Performance Evaluation

GinMaster	Malware	TP	FP	FN	TN	PR	RC	Fm	Acc
100	761	81	2	19	759	0.98	0.81	**0.89**	**0.98**

of our approach evaluating 100 malware belonging to *GinMaster* family and 761 malware belonging to other families.

Results in Table 4 seems to be very promising: we obtain an Accuracy equal to 0.9. Concerning the GinMaster results, we are not able to identify the malicious payloads of just 2 samples on 100. It is worth noting the above values are also due to the fact that the dataset is unbalanced, i.e., 100 malware belonging to *GinMaster* family and 761.

5 Conclusion and Future Work

The most common way to inject malicious payload in Android environment is represented by the repackaging attack, that basically consists to distribute legitimate well-known applications with the malicious behaviour in order to lure users. In this paper we propose an approach, based on formal methods, able to catch the malicious payload related to GinMaster family, one of the most populous repackaged trojan embed in legitimate Android applications. GinMaster family is able to root Android devices in order to execute shell scripts with admin privileges, in addition it is able to send personal user information to the attacker using C&C server. We identified a set of rules specific to GinMaster payload behaviour and we evaluate the effectiveness of our approach using a dataset of real-world malware, obtaining an accuracy equal to 0.9. As future work, we plan to test our approach on mobile malware belonging to other families that exhibit trojan behaviour to evaluate the rule set on families with similar payload.

Acknowledgements

This work has been partially supported by H2020 EU-funded projects NeCS and C3ISP and EIT-Digital Project HII.

REFERENCES

Andersen, J. R., Andersen, N., Enevoldsen, S., Hansen, M. M., Larsen, K. G., Olesen, S. R., Srba, J., and Wortmann, J. K. (2015). CAAL: concurrency workbench, aalborg edition. In *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, pages 573–582.

Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., and Rieck, K. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of 21th Annual Network and Distributed System Security Symposium (NDSS)*. IEEE.

Barbuti, R., De Francesco, N., Santone, A., and Tesei, L. (2002). A notion of non-interference for timed automata. *Fundamenta Informaticae*, 51(1-2):1–11. cited By 6.

Barbuti, R., Francesco, N. D., Santone, A., and Vaglini, G. (2005). Reduced models for efficient CCS verification. *Formal Methods in System Design*, 26(3):319–350.

Battista, P., Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016). Identification of android malware families with model checking. In *International Conference on Information Systems Security and Privacy*. SCITEPRESS.

Bernardeschi, C., De Francesco, N., Lettieri, G., and Martini, L. (2004). Checking secure information flow in java bytecode by code transformation and standard bytecode verification. *Software - Practice and Experience*, 34(13):1225–1255.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2016). An hmm and structural entropy based detector for android malware: An empirical study. *Computers & Security*, 61:1–18.

Cleaveland, R. and Sims, S. (1996). The ncsu concurrency workbench. In *CAV*. Springer.

De Ruvo, G., Nardone, V., Santone, A., Ceccarelli, M., and Cerulo, L. (2015). Infer gene regulatory networks from time series data with probabilistic model checking. pages 26–32. cited By 7.

Fedler, R., Schütte, J., and Kulicke, M. (2014). On the effectiveness of malware protection on android: An evaluation of android antivirus apps, http://www.aisec.fraunhofer.de/.

Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2013). CADP 2011: a toolbox for the construction and analysis of distributed processes. *STTT*, 15(2):89–107.

GoogleMobile (2014). http://googlemobile.blogspot.it/2012/02/android-and-security.html.

Isohara, T., Takemori, K., and Kubota, A. (2011). Kernel-based behavior analysis for android malware detection. In *Proceedings of Seventh International Conference on Computational Intelligence and Security, pp. 1011-1015*.

Jacob, G., Filiol, E., and Debar, H. (2010). Formalization of viruses and malware through process algebras. In *International Conference on Availability, Reliability and Security (ARES 2010)*. IEEE.

Kinder, J., Katzenbeisser, S., Schallhart, C., and Veith, H. (2005). Detecting malicious code by model checking. Springer.

Liang, S. and Du, X. (2014). Permission-combination-based scheme for android mobile malware detection. In *International Conference on Communications*, pages 2301–2306.

Marforio, C., Aurelien, F., and Srdjan, C. (2011). Application collusion attack on the permission-based security model and its implications for modern smartphone systems, ftp://ftp.inf.ethz.ch/doc/tech-reports/7xx/724.pdf.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016a). Download malware? No, thanks. How formal methods can block update attacks. In *Formal Methods in Software Engineering (FormaliSE), 2016 IEEE/ACM 4th FME Workshop on*. IEEE.

Mercaldo, F., Nardone, V., Santone, A., and Visaggio, C. A. (2016b). Ransomware steals your phone. formal methods rescue it. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 212–221. Springer.

Milner, R. (1989). *Communication and concurrency*. PHI Series in computer science. Prentice Hall.

Neuhaus, S. and Zimmermann, T. (2010). Security trend analysis with cve topic models. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 111–120. IEEE.

Oberheide, J. and Miller, C. (2012). Dissecting the android bouncer. In *SummerCon, https://jon.oberheide.org/files/summercon12-bouncer.pdf*.

Reina, A., Fattori, A., and Cavallaro, L. (2013). A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of EuroSec*.

Santone, A. (2011). Clone detection through process algebras and java bytecode. pages 73–74. cited By 10.

SecureList (2015). https://securelist.com/analysis/kaspersky-security-bulletin/73839/mobile-malware-evolution-2015/.

Song, F. and Touili, T. (2001). Efficient malware detection using model-checking. Springer.

Song, F. and Touili, T. (2013). Pommade: Pushdown model-checking for malware detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM.

Song, F. and Touili, T. (2014). Model-checking for android malware detection. Springer.

Song, J., Han, C., Wang, K., Zhao, J., Ranjan, R., and Wang, L. (2016). An integrated static detection and analysis framework for android. *Pervasive and Mobile Computing*.

Spreitzenbarth, M., Echtler, F., Schreck, T., Freling, F. C., and Hoffmann, J. (2013). Mobilesandbox: Looking deeper into android applications. In *28th International ACM Symposium on Applied Computing (SAC)*. ACM.

Stirling, C. (1989). An introduction to modal and temporal logics for ccs. In *Concurrency: Theory, Language, And Architecture*, pages 2–20.

Tchakount, F. and Dayang, P. (2013). System calls analysis of malwares on android. In *International Journal of Science and Tecnology (IJST) Volume, 2 No. 9*.

Yerima, S. Y., Sezer, S., McWilliams, G., and Muttik, I. (2013). A new android malware detection approach using bayesian classification. In *International Conference on Advanced Information Networking and Applications*, pages 121–128.

Yu, R. (2013). Ginmaster: a case study in android malware. In *Virus bulletin conference*, pages 92–104.

Zhao, Y.-B., Liu, S.-M., and Guo, S.-Q. (2014). Extraction and prediction of hot topics in network security. In *Computer Science and Network Security, 2014 International Conference on*, pages 347–353.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE.