Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 159 (2019) 963–972

Procedia Computer Science

www.elsevier.com/locate/procedia

23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

Exploiting Model Checking for Mobile Botnet Detection

Cinzia Bernardeschi[a], Francesco Mercaldo[b,d,*], Vittoria Nardone[c], Antonella Santone[d,*]

[a]*Department of Information Engineering, University of Pisa, Pisa, Italy*
[b]*Institute for Informatics and Telematics, National Research Council of Italy (CNR), Pisa, Italy*
[c]*Department of Engineering, University of Sannio, Benevento, Italy*
[d]*Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy*

Abstract

Android malware is increasing from the point of view of the complexity and the harmful actions. As a matter fact, malware writers are developing sophisticated techniques to infect mobile devices very closed to their counterpart for personal computers. One of these threats is represented by the possibility to control the infected devices from the attacker i.e., the so-called botnet. In this paper a method able to identify botnet in Android environment through model checking is proposed. Starting from the malicious payload definition, the proposed method is able to detect and to localize the code related to the malicious botnet. We experiment real-world botnet based Android malware, obtaining encouraging results.

Keywords: malware, botnet, Android, model checking, formal methods, temporal logic, security

1. Introduction

In 2017, Kaspersky security specialists detected 5,730,916 mobile malicious installation packages, which is almost 1.5 times fewer than in the previous year, although more than in any other year before and almost twice as much as in 2015 [6,15]. Despite the decrease in the number of detected malicious installation packages, in 2017 security researchers registered a growing number of mobile malware attacks, i.e., 42.7 million vs. 40 million in 2016 [3,17].

This trend is symptomatic of the growing interest of malicious writers in mobile environment. Malicious writers increased their techniques in order to perpetrate more and more aggressive attacks. As a matter of fact, the first malicious sample in mobile environment was a proof-of-concept malware written in C++ using the Nokia SDK targeting the Symbian operating system. The malware itself was harmless, doing little more than displaying the message "Caribe" on the device's display every time it was turned on. It was not even released into the wild [22].

Since then, more and more techniques have been developed by malware writers in order to elude the current anti-malware technologies and to perform a plethora of harmful actions. Mobile malware nowadays exhibits a complexity

* Francesco Mercaldo, Antonella Santone
E-mail address: francesco.mercaldo@iit.cnr.it, antonella.santone@unimol.it

1877-0509
10.1016/j.procs.2019.09.263

near to their counterpart for computer platforms, for instance currently Android malware embeds botnet functionalities[1].

The term mobile botnet refers to a group of compromised smartphones that are remotely controlled by botmasters using Command & Control (C&C) channels [19,13]. While PC-based botnets, as common platforms for many Internet attacks, they become one of the most serious threats to Internet, mobile botnets targeted for smartphones are not as popular as their counterparts for several reasons including resource issues, limited battery power, and Internet access constraints [9,12].

Consequently, both the occurrence of practical mobile botnets and corresponding research on them are very limited. However, this trend could change with the recent surge in popularity and use of smartphones. Smartphones are now widely used by billions of end users due to their enhanced computing ability and efficient Internet access. Moreover, smartphones always store a large amount of sensitive personal data and are often used in online payment. The emergence of open-source smartphone platforms such as Android and third-party applications made available to the public also provides more opportunities for malware creators. Therefore, smartphones have become one of the most attractive target for malware writers and for these reasons detection of Android malware is becoming increasingly important.

In this paper, we present a method to identify botnet in Android environment based on static analysis of the code. The approach is based on modelchecking [8], an automatic technique for verifying finite state systems. This is accomplished by checking whether a structure, representing the system, satisfies a temporal logic formula describing the expected behavior. In particular, from the Java byte code we derive a set of finite state automata that collect approximate information about the run-time behavior of an app. Then, we formulate the botnet malicious behaviour using temporal logic formulae [7]. Finally, adopting a model checker, we are able to automatically check if the code is malicious and to identify where the botnet is located in the code. The properties are checked by using the CAAL [1] formal verification environment.

The main contributions of the paper are the following:

- formal description of botnet behaviours in logic;
- botnet identification in Android environment;
- malicious code localization at method-grain level;
- resilience with respect to obfuscation techniques.

The rest of the paper is organized as follows: Section 2 discusses the current literature related to mobile malware detection, Section 3 introduces the method, Section 4 describes the formulae representing the botnet behaviours, Section 5 illustrates the results of the experiment. Finally, conclusions and future works are given in Section 6.

2. Related Work

In this section we review the current state-of-the-art literature related to the identification of botnet in mobile environment.

Authors in [11] uncover the relationships between the majority of the analyzed botnet families and offer an insight into each malicious infrastructure. They analyse a dataset containing 1929 samples representing 14 Android botnet families. Basically through static and dynamic analysis, they extract and visualize all embedded URLs, including the obfuscated URLs.

Researchers in [10] propose a cloud-based Android botnet Detection System. Their method exploits dynamic analysis using a virtual environment and with cluster analisys. The toolchain for dynamic analysis is composed by *strace*, in order to extract system calls, *netflow*, to monitor network traffic, *logcat*, to gather application level functional call data, *sysdump*, to retrieve Android system configuration and battery usage related information and *wireshark/tcpdump*, with

[1] goo.gl/qWDsNf

`op`	pop two operands off the stack, perform the operation, and push the result onto the stack
`pop`	discard the top value from the stack
`push` k	push the constant k onto the stack
`load` x	push the value of the variable x onto the stack
`store` x	pop off the stack and store the value into variable x
`if` j	pop off the stack and jump to j if non-zero
`goto` j	jump to j
`jsr` j	at address p, jump to address j and push return address $p+1$ onto the operand stack
`ret` x	jump to the address stored in x

Fig. 1. Examples of bytecode instructions

the aim to capture traffic at device interface. The authors do not provide experiments results related to the effectiveness of the proposed solution.

Tansettanakorn and colleagues[21] develop ABIS (Android Botnet Identification System) to check Android applications whether they can possibly be malware or not. They consider static and dynamic analysis features: API calls, permission and network traffic. The detection module exploits supervised machine learning: as a matter of fact, they evaluate their system by using the Weka toolsuite with Random Forest classification algorithm obtaining a precision equal to 97.2% and a recall equal to 96.9%.

Authors in[4] propose a botnet detection *pull* style C&C channel by inspecting C&C traffic while bots communicate through the C&C server. Their proposed solution is related to the identification through HTTP packets, this is the reason why they build VPN between a mobile device and the intrusion detection system. The designed VPN provides a shared path for both 3/4G and WiFi, and is implemented in each mobile device under analysis.

As emerging from the state of the art literature, the method presented in this paper represents the first attempt to detect mobile botnet using formal methods.

3. The Method

Model checking[8] is an efficient verification technique to search and check a behavioral property in a code in order to locate design errors. When no errors are found, the code is considered free of errors or verified with respect to a given property. In practice, desirable properties are described with a temporal logic and the code is modeled as an automaton.

Our method consists in the construction of an automaton that mimics the behavior of an Android application. An Android application, the so-called .apk (i.e., Android Package) is a variant of the well-known .jar archive file. An .apk file typically contains the executable code for the Dalvik Virtual Machine (i.e., the .dex file), the re-source folder (i.e., images, icons and sounds) and the Manifest file. Our method acts at Bytecode level. We obtain Bytecode instructions starting from the .apk file.

Once obtained the Java Bytecode, an inference algorithm is used to obtain an automaton. Informally, the automaton describes mainly the behavior of an app as a set of reachable states and actions (instructions) that trigger a change of state. In fact, the states express the possible values of the program counter. Each transition describes the execution of a given instruction, the labels represent the *opcode*.

Basically, we divide the Java Bytecode instructions into three sets: *sequential instructions*, *conditional instructions* and *unconditional instructions*. Figure 1 shows examples of bytecode instructions.

Sequential instructions are modeled as a state with only one outgoing transition to the successive instruction, while conditional instruction are modeled as a state with two outgoing transitions depending on the result of the valuation of the condition. Finally, unconditional branch instructions are modeled with a state with an outgoing transition that jumps to the state representing the instruction target of the unconditional instruction.

For example, suppose that at the address i we have the $goto\ j$ instruction. The instruction i is translated with a state i that performs the action $goto\,j$ and then jumps to the state j encoding the instruction address j.

Each address of a Java Bytecode instructions is encoded in a state of the automaton. The transition encodes the instruction, i.e., the opcodes. In this way the automaton mimics the control-flow transitions from one instruction to it successor(s).

If w is a finite sequence, let $\sharp w$ denote the length of w, i.e., the number of elements of w. A bytecode is a sequence c of instructions, numbered starting from address 1; and $\forall i \in \{1,\cdots,\sharp c\}$, $c[i]$ is the instruction at address i.

Let c be the bytecode of the method mt. The *automaton* for mt is the pair $A_{mt} = (G,L)$, where G is a directed graph and L is a function that assigns labels to the edges of the graph. In particular,

- $G = (V,E)$ is a graph where $V = \{1,..,\sharp c\}$ is the set of nodes; and $E = \subseteq V \times V$ contains the edge (i,j) if and only if the instruction at address j can be immediately executed after that at address i.
- $L : E \to O$, where $O = \{pop, push, load, store, invoke, \cdots\}$ is the set of opcodes.

As an example, the automaton for the bytecode in Listing 1 is shown in Figure 2.

```
.....
1:   load x                    // load x on the stack
2:   ifge 5                    // jump if greater or equal
3:  iconst_1                   // push 1 on the stack
4:  istore_2                   // pop off the stack and store the value in a register
5:  goto 6                     // jump
6:  .....
```

Listing 1. An example

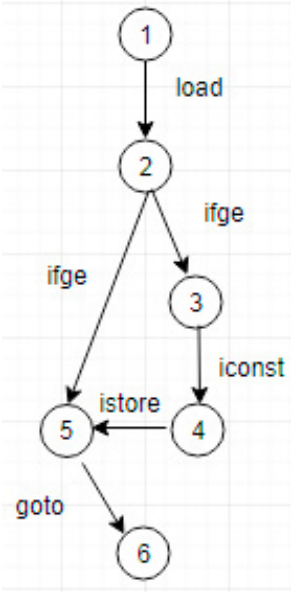

Fig. 2. Automaton related to Listing 1.

A similar approach can be found in [16], where Calculus of Communicating Systems (CCS) [18] has been used instead of generating directly the automaton.

After having constructed the automata for an Android app, we define the characteristic behaviour of a botnet by means of the definition of the temporal logic properties. This process tries to recognize specific and distinctive features of the botnet behaviour. Thus, this specific behaviour is written as a set of properties. To specify the properties, we manually inspected a few samples in order to find the botnet malicious behavior implementation at Bytecode level.

```
public a$a(a parama, Message paramMessage)
  {
    /* ... */
    this.jdField_for = paramMessage.getData().getString("packName");
    /* ... */
    this.jdField_if.jdField_char = paramMessage.getData().getString("addrType");
    this.jdField_if.jdField_case = paramMessage.getData().getBoolean("openGPS");
    /* ... */
    this.jdField_if.jdField_long = paramMessage.getData().getInt("timeOut");
    this.jdField_if.jdField_goto = paramMessage.getData().getInt("priority");
    this.jdField_if.jdField_void = paramMessage.getData().getBoolean("location_change_notify");
    /* ... */
  }
```

Listing 2. TigerBot.

The properties are proved separately on each method. An app is a botnet if there exists at least one method that satisfies the formulae. Let $\{A_{mt_1}, \cdots, A_{mt_n}\}$ be the set of automata of the app, whose methods are $mt_1, \cdots, mt_n$. Given a formula ϕ, the app satisfies the formula if one automaton is a model of the formula: $\exists i,\ 1 \leq i \leq n,$ and $A_{mt_i} \models \phi$.

This formalisation assumes that the botnet is local to a method. This is the more common case in practice, as we will see in Section 5. In the next section we give some hints to overcome this limitation.

4. Formulae representing the botnet behaviour

After the construction of the automata that models the app, the logic formulae characterizing a botnet behaviour need to be designed and defined. Thus, this specific behaviour is written as a set of properties. To specify the properties, the mu-calculus logic[20] is used, which is a branching temporal logic to express behavioural properties. Writing the correct rules can be a rather complex task. Actually, the formulae have been defined manually by inspecting very few samples.

We manually defined two rules, one related to the action for the TigerBot family and the other one related to the RootSmart one. Both of the rules are related to the C&C command of the botnet.

Listing 2 shows the Java code snippet related to the C&C action. The bytecode is generated using the *dex2jar* tool (`https://github.com/pxb1988/dex2jar`), to obtain the jar archive containing the class files from the Android executable. Once obtained the class files we use the BCEL library (`https://commons.apache.org/proper/commons-bcel/`) to extract the bytecode of the classes. The code analysed contains obfuscation techniques, as shown by the name of the method `a$a`.

The code implements the following C&C commands that the TigerBot malicious payload is able to receive:

- *packName* command sends the name of malware application on the device;
- *addrType* converts a coordinate to a String representation. The type may be one of FORMAT_DEGREES, FORMAT_MINUTES, or FORMAT_SECONDS.
- *openGPS* exhibits the ability to activate the fine-grain localization (i.e., the GPS sensor);
- *timeOut* represents the frequency at which the geolocalization listener receives updates;
- *priority* command sets the priority related to the botnet command execution;
- *location_change_notify* exhibits the ability to send a notification to the C&C server whether the device under analysis change location.

Listing 3 shows the code snippet related to the C&C command botnet for the RootSmart family. The code represents the following command exhibited by the RootSmart malicious payload:

- *action.host_start*: marks the malware as running;

```
{ /* ... */
  if ("action.host_start".equals(str)) { /* ... */ }
  if (!"action.shutdown".equals(str)) { /* ... */ }
  if (!"action.screen_off".equals(str)) { /* ... */ }
  if (!"action.install".equals(str)) { /* ... */ }
  if ("action.installed".equals(str)) { /* ... */ }
  if ("action.check_live".equals(str)) { /* ... */ }
  if ("action.download_shells".equals(str)) { /* ... */ }
  if ("action.exploid".equals(str)) { /* ... */ }
  if ("action.first_commit_localinfo".equals(str)) { /* ... */ }
  if ("action.second_commit_localinfo".equals(str)) { /* ... */ }
  if ("action.load_taskinfo".equals(str)) { /* ... */ }
  if ("action.download_apk".equals(str)) { /* ... */ }
    /* ... */
}
```

Listing 3. RootSmart.

- *action.shutdown*: sets the amount of time the backdoor has been running and if the phone has been exploited;
- *action.screen_off*: checks the power state, keeping the device on when needed;
- *action.install*: downloads and installs the exploit and root shell;
- *action.installed*: retrieves phone information and does the final setup of the backdoor;
- *action.check_live*: schedules an hourly self-check and detects OS version;
- *action.download_shells*: downloads the shell package from the remote server;
- *action.exploid*: unpacks shell file, runs the exploit, sets the result and does the cleanup;
- *action.first_commit_localinfo*: sets a flag with the local time of first connection to remote server;
- *action.second_commit_localinfo*: like above, with some other internal flag initializations;
- *action.load_taskinfo*: gets the IMEI of the device and retrieves information about the package;
- *action.download_apk*: downloads the requested apk and installs it (silently, whether the malware was able to obtain root privileges).

In Table 1 two fragments of formulae are shown. In particular, the formulae specify the patterns of the two malicious behaviours discussed above. The first formula, $\varphi_{TIGERBOT}$, is related to the TrigerBot malware family, while the second one, $\varphi_{ROOTSMART}$, is related to the RootSmart family. For lack of space, only a small part of them has been reported. In the formulae, φ_{1_9} (resp. φ_{2_9}) specifies the missing part of the malicious behaviour.

Formula $\varphi_{TIGERBOT}$ is verified if there exists at least one automaton that contains a path with the sequence of actions $\langle invokegetData \rangle$ $\langle pushpackName \rangle$..., possibly interleaved with other actions. The same reasoning applies to formula $\varphi_{ROOTSMART}$.

However, to identify cases in which the behaviour of the botnet is distributed in different methods (i.e., automata), an approach similar to that used in [2] for secure information flow checking could be applied. In this case, if an `invoke` bytecode instruction is encountered, and the current method satisfies φ_{1_j} (for some j), we must also check if the invoked method is a model for the formula $\varphi_{1_{j+1}}$.

5. Experiment

In this section we discuss the experiment we performed to evaluate the effectiveness of our approach in Android botnet payload identification, discriminating samples belonging to other Android botnet malware families and legitimate applications.

Table 1. The formulae for Commands Botnet payload detection

$$
\begin{array}{ll}
\varphi_{TIGERBOT} & = \mu X.\, \langle invokegetData \rangle\, \varphi_{1_1} \vee \langle -invokegetData \rangle\, X \\
\varphi_{1_1} & = \mu X.\, \langle pushpackName \rangle\, \varphi_{1_2} \vee \langle -pushpackName \rangle\, X \\
\varphi_{1_2} & = \mu X.\, \langle invokegetString \rangle\, \varphi_{1_3} \vee \langle -invokegetString \rangle\, X \\
\varphi_{1_3} & = \mu X.\, \langle invokegetData \rangle\, \varphi_{1_4} \vee \langle -invokegetData \rangle\, X \\
\varphi_{1_4} & = \mu X.\, \langle pushprodName \rangle\, \varphi_{1_5} \vee \langle -pushprodName \rangle\, X \\
\varphi_{1_5} & = \mu X.\, \langle invokegetString \rangle\, \varphi_{1_6} \vee \langle -invokegetString \rangle\, X \\
\varphi_{1_6} & = \mu X.\, \langle invokegetData \rangle\, \varphi_{1_7} \vee \langle -invokegetData \rangle\, X \\
\varphi_{1_7} & = \mu X.\, \langle pushcoorType \rangle\, \varphi_{1_8} \vee \langle -pushcoorType \rangle\, X \\
\varphi_{1_8} & = \mu X.\, \langle invokegetString \rangle\, \varphi_{1_9} \vee \langle -invokegetString \rangle\, X \\
\varphi_{1_9} & = \ldots \\
\varphi_{ROOTSMART} & = \mu X.\, \langle pushactionhoststart \rangle\, \varphi_{2_1} \vee \langle -pushactionhoststart \rangle\, X \\
\varphi_{2_1} & = \mu X.\, \langle invokeequals \rangle\, \varphi_{2_2} \vee \langle -invokeequals \rangle\, X \\
\varphi_{2_2} & = \mu X.\, \langle ifeq \rangle\, \varphi_{2_3} \vee \langle -ifeq \rangle\, X \\
\varphi_{2_3} & = \mu X.\, \langle pushactionboot \rangle\, \varphi_{2_4} \vee \langle -pushactionboot \rangle\, X \\
\varphi_{2_4} & = \mu X.\, \langle invokeequals \rangle\, \varphi_{2_5} \vee \langle -invokeequals \rangle\, X \\
\varphi_{2_5} & = \mu X.\, \langle ifeq \rangle\, \varphi_{2_6} \vee \langle -ifeq \rangle\, X \\
\varphi_{2_6} & = \mu X.\, \langle pushactionshutdown \rangle\, \varphi_{2_7} \vee \langle -pushactionshutdown \rangle\, X \\
\varphi_{2_7} & = \mu X.\, \langle invokeequals \rangle\, \varphi_{2_8} \vee \langle -invokeequals \rangle\, X \\
\varphi_{2_8} & = \mu X.\, \langle ifeq \rangle\, \varphi_{2_9} \vee \langle -ifeq \rangle\, X \\
\varphi_{2_9} & = \ldots
\end{array}
$$

5.1. Dataset

The real world samples examined in the experiment were gathered from the Android Botnet dataset, freely available for research purpose[2].

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[23]. The dataset includes samples spawning a period of 2010 (the first appearance of Android botnet) to 2014[12].

In this preliminary work we evaluate two of the most diffused botnet for Android environment: TigerBot and RootSmart.

TigerBot botnet differs from usual botnet-powered malware in that it is controlled via SMS rather than from a C&C server on the Internet (i.e., exploiting the HTTP protocol). Malicious payload belonging to this family are able to execute a range of commands including uploading the current location of the infected device, sending SMS messages, and even recording phone calls. It works by intercepting SMS messages sent to the phone and checking to see if they are commands for it to act. If they are, it executes the command and then prevents the message from being seen by the user.

Furthermore, TigerBot tries to hide itself from the user by not showing any icon on the home screen and by using legitimate sounding app names (for instance, System) or by copying names from trusted vendors like Google or Adobe. The main harmful actions performed are the following: (i) it records the sounds in the phone, including the phone calls and the surrounding sounds, (ii) it changes the network setting, (iii) it uploads the current GPS location, (iv) it captures and upload the image, (iv) it sends SMS to a particular number, (v) it reboots the device and (vi) it kills other running processes [3].

[2] http://www.unb.ca/cic/datasets/android-botnet.html

[3] goo.gl/GDaZha

The RootSmart malware hides in an Android app named com.google.android.smart, which has the same icon with the default Android system setting app. Once installed, it will register several system-wide receivers to wait for various events (for instance, new outgoing calls). When these system events occur, its malicious payload will automatically run in the background. Specifically, when started, RootSmart will connect to its C&C server with various information collected from the phone. Our analysis shows that the collected information includes the Android OS version number, the device IMEI number, as well as the package name. To impede reverse engineering, the malware does not directly include the C&C server URL in plaintext. Instead, it encrypts the C&C URL inside a raw resource file. The key used to decrypt this resource file is generated by providing a fixed seed number to the Java random number generator. After obtaining the root privilege, RootSmart will download additional (malicious) apps from its C&C server and install them onto infected devices[4]. Whether RootSmart fails to obtain the root privilege, it will still attempt to install the downloaded apps.

Table 2 shows the Android botnet families analysed in the dataset evaluate the effectiveness of the proposed method. In the Table, column ♯ is the number of samples available in the dataset, column **C&C** is the type of attack and column **Description** contains a brief description of the malicious payload characterizing the families.

Family	#	C&C	Description
TigerBot	96	SMS	phone-call recording ability
RootSmart	28	HTTP	root privilege escalation

Table 2. Samples in the dataset .

In order to build the legitimate dataset 1000 Android goodware apps were downloaded from the Google's official app store[5]. To obtain legitimate applications we crawled the Google Play market using an open-source crawler[6]. The obtained goodware dataset includes samples belonging to all the different categories available on the market.

The crawler is configured to equally download goodware applications from the different categories of apps available in the market. In order to ensure their trustworthiness, the mined applications have been checked through the VirusTotal service[7], a service able to run 60 different antimalware software (i.e., Symantec, Avast, Kasperky, McAfee, Panda, and others). The analysis confirmed that the crawled applications do not contain any malicious payload i.e., we consider as legitimate an application downloaded from the Google Play that results to be legitimate for all 60 considered antimalware.

5.2. Evaluation

In order to evaluate the proposed solution as mobile botnet identifier we consider the Precision (P), the Recall (R) and Accuracy (Acc) metrics.

The Precision is the fraction of relevant instances among the retrieved instances:

$$Precision = \frac{TP}{TP+FP}$$

The Recall is the fraction of relevant instances that have been retrieved over the total amount of relevant instances:

$$Recall = \frac{TP}{TP+FN}$$

The Accuracy is the fraction of the classifications that are correct and it is computed as the sum of true positives and negatives divided all the evaluated samples:

[4] goo.gl/4G6hbW
[5] https://play.google.com/store
[6] https://github.com/liato/android-market-api-py
[7] https://www.virustotal.com

$$Accuracy = \frac{TP+TN}{TP+FN+FP+TN}$$

where *TP* indicates the number of true positives, *TN* indicates the number of true negatives, *FN* indicates the number of false negatives and *FP* indicates the number of false positives.

Table 3 shows the results obtained using our method. In particular, it is organized as follow: the first column indicates the tested formula; the second, third and fourth columns indicate the families involved in the experiment and their corresponding number of samples. The last three column show the Precision, the Recall and the Accuracy metrics.

Family	# of TigerBot	# of RootSmart	# of Trusted	TP	FP	FN	TN	P	R	Acc
$\varphi_{TIGERBOT}$	96	28	1000	96	0	0	1028	**1**	**1**	**1**
$\varphi_{ROOTSMART}$	96	28	1000	28	0	0	1096	**1**	**1**	**1**

Table 3. Performance Evaluation

The results in Table 3 seem to be very promising: we obtain a Precision, a Recall and an Accuracy equal to 1. The proposed method does not performs misclassification errors, as a matter of fact we obtain an FP rate equal to 0.

6. Conclusion and Future Work

Mobile malware is increasing its techniques aimed to infect mobile devices and to infer more and more harmful actions. As a matter of fact, Android malware is exhibiting botnet functionalities that enable attackers to remotely control the infected devices.

In this paper we propose a model checking-based method able to detect whether mobile embedded payload contain botnet code. We perform preliminary experiments using a dataset composed of 1124 real-world applications, 1000 of which are legitimate apps and 124 are malware. The malware code belongs to two Android botnet-based families: TigerBot and RootSmart.

Results show high accuracy in botnet detection. In particular, our approach for malware detection enables a modular analysis of the code and it has the advantage of being fully automatic. As future work, we plan to apply the proposed method to the analysis of other widespread Android botnet families, and we intend to investigate whether machine learning techniques can be helpful in the automatic rules formulation of botnet patterns. Furthermore, we will investigate whether the proposed method is able to detect mobile botnet malicious payload morphed with widespread obfuscation techniques [14,5] currently employed by malware writers.

References

Andersen, J.R., Andersen, N., Enevoldsen, S., Hansen, M.M., Larsen, K.G., Olesen, S.R., Srba, J., 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, Springer. pp. 573–582.

Avvenuti, M., Bernardeschi, C., De Francesco, N., Masci, P., 2012. JCSI: A tool for checking secure information flow in java card applications. Journal of Systems and Software 85, 2479–2493. URL: https://doi.org/10.1016/j.jss.2012.05.061, doi:10.1016/j.jss.2012.05.061.

Canfora, G., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2018. Leila: formal tool for identifying mobile malicious behaviour. IEEE Transactions on Software Engineering .

Choi, B., Choi, S.K., Cho, K., 2013. Detection of mobile botnet using vpn, in: Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on, IEEE. pp. 142–148.

Cimitile, A., Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., 2017. Formal methods meet mobile code obfuscation identification of code reordering technique, in: 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), IEEE. pp. 263–268.

Cimitile, A., Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2018. Talos: no more ransomware victims with formal methods. International Journal of Information Security 17, 719–738.

Clarke, E., Emerson, E., Sistla, A., 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on proramming Languages and Systems 8, 244–263.

Clarke, E.M., Grumberg, O., Peled, D., 2001. Model checking. MIT Press.

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M., 2015. Android security: a survey of issues, malware penetration, and defenses. IEEE communications surveys & tutorials 17, 998–1022.

Jadhav, S., Dutia, S., Calangutkar, K., Oh, T., Kim, Y.H., Kim, J.N., 2015. Cloud-based android botnet malware detection system, in: Advanced Communication Technology (ICACT), 2015 17th International Conference on, IEEE. pp. 347–352.

Kadir, A.F.A., Stakhanova, N., Ghorbani, A.A., 2015. Android botnets: What urls are telling us, in: International Conference on Network and System Security, Springer. pp. 78–91.

Karim, A., Salleh, R.B., Shiraz, M., Shah, S.A.A., Awan, I., Anuar, N.B., 2014. Botnet detection techniques: review, future trends, and issues. Journal of Zhejiang University SCIENCE C 15, 943–983.

Letteri, I., Del Rosso, M., Caianiello, P., Cassioli, D., 2018. Performance of botnet detection by neural networks in software-defined networks, in: CEUR WORKSHOP PROCEEDINGS, CEUR-WS.

Martinelli, F., Mercaldo, F., Nardone, V., Santone, A., Sangaiah, A.K., Cimitile, A., 2018. Evaluating model checking for cyber threats code obfuscation identification. Journal of Parallel and Distributed Computing 119, 203–218.

Martinelli, F., Mercaldo, F., Saracino, A., 2017. Bridemaid: An hybrid tool for accurate detection of android malware, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ACM. pp. 899–901.

Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.A., 2016a. Download malware? no, thanks: how formal methods can block update attacks, in: Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2016, Austin, Texas, USA, May 15, 2016, ACM. pp. 22–28.

Mercaldo, F., Visaggio, C.A., Canfora, G., Cimitile, A., 2016b. Mobile malware detection in the real world, in: Proceedings of the 38th International Conference on Software Engineering Companion, ACM. pp. 744–746.

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

Pieterse, H., Olivier, M.S., 2012. Android botnets on the rise: Trends and characteristics, in: Information Security for South Africa (ISSA), 2012, IEEE. pp. 1–5.

Stirling, C., 1989. An introduction to modal and temporal logics for ccs, in: Yonezawa, A., Ito, T. (Eds.), Concurrency: Theory, Language, And Architecture, Springer. pp. 2–20.

Tansettanakorn, C., Thongprasit, S., Thamkongka, S., Visoottiviseth, V., 2016. Abis: a prototype of android botnet identification system, in: Student Project Conference (ICT-ISPC), 2016 Fifth ICT International, IEEE. pp. 1–5.

Zhang, Y., Zheng, J., Ma, M., 2008. Handbook of research on wireless security .

Zhou, Y., Jiang, X., 2012. Dissecting android malware: Characterization and evolution, in: Proceedings of 33rd IEEE Symposium on Security and Privacy (Oakland 2012).