Identification of Android Malware Families with Model Checking

Pasquale Battista, Francesco Mercaldo, Vittoria Nardone,
Antonella Santone and Corrado Aaron Visaggio
Department of Engineering, University of Sannio, Benevento, Italy

Keywords: Malware, Android, Security, Formal Methods, Process Algebras.

Abstract: Android malware is increasing more and more in complexity. Current signature based antimalware mechanisms are not able to detect zero-day attacks, also trivial code transformations may evade detection. Malware writers usually add functionality to existing malware or merge different pieces of malware code: this is the reason why Android malware is grouped into families, i.e., every family has in common the malicious behavior. In this paper we present a model checking based approach in detecting Android malware families by means of analysing and verifying the Java Bytecode that is produced when the source code is compiled. A preliminary investigation has been also conducted to assess the validity of the proposed approach.

1 INTRODUCTION

Malware, as well as, any other software evolves. Evidence exists that the majority of newly detected malware are tweaked variants of well-known malware (Bailey et al., 2009; Hu et al., 2009; Jang et al., 2011).

As a matter of fact, attackers use to modify existing malware, by adding new behaviors or merging together parts of different existing malware's codes. Existing malware can be embedded in apparently benign programs (usually popular apps) with repackaging (Zhou and Jiang, 2012): malware authors locate and download popular apps, disassemble them, enclose malicious payloads, re-assemble and then submit the new apps to official and/or alternative Android markets. This scenario leads to group malware in families, where a family defines a set of behaviors common to all its members. Identifying the family a malware belongs to is of primary importance as it helps to discover new malware families (Khoo and Lio, 2011; Ma et al., 2006), create models of provenance and lineage (Dumitras and Neamtiu, 2011), and generate phylogeny models (Karim et al., 2005). Recognizing a malware family is at the basis of a variety of security tasks, from malware characterization to threat detection and cyber-attack prevention. In malware triage (Bailey et al., 2009; Hu et al., 2009; Jang et al., 2011), lineage can be used by malware analysts to understand trends over time and make informed decisions about the dissection strategies to dissect the malware samples. This is particularly important since the order in which the variants of a malware are captured does not necessarily mirror its evolution. In software security, lineage can help to find vulnerabilities in software when the source code is not available. For example, if we know that a vulnerability is present in an earlier version of an application, then it may also reside in applications that are derived from it.

Although literature provides several proposals to detect Android malware (Canfora et al., 2013; Arp et al., 2014), the proposed techniques are not able to isolate the payload responsible for malicious action, and this impedes the recognisance of the family.

Moreover, in mobile malware landscape, malware is becoming aggressive and hundreds of families are spread at a very fast pace (Zhou and Jiang, 2012): simple forms of polymorphic attacks (i.e., malware that mutates at each infection) targeting Android platform have already been seen[1]. An example of polymorphic behaviour is represented by *Opfake* family. The authors demonstrated that by using simple code transformations (Canfora et al., 2015) to existing malware that is well recognized by malware detectors turns it in a version that is anymore recognized by the most malware detectors.

DroidKungFu is a widespread malware family. Its payload is able to install a backdoor that allows attackers to access the smartphone when they want and use the device as they please. Since DroidKungFu contains root exploits, this family represents one of

[1] http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications

Battista, P., Mercaldo, F., Nardone, V., Santone, A. and Visaggio, C.
Identification of Android Malware Families with Model Checking.
DOI: 10.5220/0005809205420547
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 542-547
ISBN: 978-989-758-167-0

the most serious threats to mobile users[2].

Starting from these considerations, it urges to study new techniques which are able to effectively recognize the family a malware belongs to.

In this paper we investigate whether model checking could detect payloads properly and resist against common obfuscation used by attackers to generate malware variants belonging to same family.

Thus, we pose the following research questions:

- *RQ1: is our method able to correctly identify the malware family?*
- *RQ2: is our method able to correctly identify morphed versions of known malware?*

The paper proceeds as follows: comparisons with related work are made in Section 2. Section 3 is a review of the basic concepts of formal methods, while Section 4 describes our methodology. In Section 5 the experimental results we obtained are reported and discussed; and, finally, conclusions are drawn in the last section.

2 RELATED WORK

In this section, coherently with the research questions we stated in the introduction, we review related literature about malware detection with particular emphasis on studies using formal methods. As our method performs a static analysis, we discuss related works that do not require to run applications, i.e. static ones.

Authors in (Kinder et al., 2005) introduce the specification language CTPL (Computation Tree Predicate Logic) which extends the well-known logic CTL, and describes an efficient model checking algorithm. They confirm the malicious behavior of thirteen Windows malware variants using as dataset a set of worms dating from the years 2002-2004.

Song et al. (Song and Touili, 2001) present an approach to model Microsoft Windows XP binary programs as a PushDown System (PDS). 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. (Song and Touili, 2014) model mobile applications using a PDS in order to discovery private data leaking. They identify information leak working at Smali code level.

Jacob and colleagues (Jacob et al., 2010) provide a basis for a malware model, founded on the Join-Calculus: the process-based model supports the fundamental notion of self-replication but also interactions, concurrency and non-termination to cover evolved malware. They consider the system call sequences to build the model.

As emerges from this discussion and at the best knowledge of the authors, the payload identification in Android environment proposed in this paper was never used in any of the works on mobile malware detection in literature.

3 PRELIMINARIES ON FORMAL METHODS

In this section we introduce the basic concepts of formal methods. For applying formal methods, we need:
1. A Precise Notation for Defining Systems: Specification is the process of describing a system. We assume that the system behaviour is represented as an automaton. It basically consists of a set of nodes together with a set of labelled edges between these nodes. A node represents a system state, while a labelled edge represents a transition from one system state to the next. That is, if the automaton contains an edge $s \xrightarrow{a} s'$, then the system can evolve from state s into state s' by the execution of action a. One state is selected to be the root state (initial state). However, for the purpose of mathematical reasoning it is often convenient to represent the automaton algebraically in the form of processes. For this aim, we use Milner's Calculus of Communicating Systems (CCS) (Milner, 1989), one 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 infinite behaviour. The syntax of *processes* is the following:

$$p ::= nil \mid \alpha.p \mid p+p \mid \ p|p \ \mid p\backslash L \mid p[f] \mid x$$

where α ranges over a finite set of actions $\mathcal{A} = \{\tau, a, \overline{a}, b, \overline{b}, ...\}$. Input actions are labeled with "non-barred" names, i.e., a, while output actions are "barred", i.e., $\overline{a}$. The action $\tau \in \mathcal{A}$ is called *internal action*. The set L ranges over sets of *visible actions* ($\mathcal{A} - \{\tau\}$), f ranges over functions from actions to actions, while x ranges over a set of *constant* names: each constant x is defined by a constant definition $x \stackrel{\text{def}}{=} p$.

We give the semantics for CCS by induction over the structure of processes.

[2]https://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/

- The process *nil* can perform no actions.
- The process $\alpha.p$ can perform the action α and thereby become the process p.
- The process $p+q$ can behave either as p or as q.
- The operator $|$ expresses parallel composition: if the process p can perform α and become p', then $p|q$ can perform α and become $p'|q$, and similarly for q. Furthermore, if p can perform a visible action l and become p', and q can perform $\bar{l}$ and become q', then $p|q$ can perform τ and become $p'|q'$.
- The operator $\backslash$ expresses the restriction of actions. If p can perform α and become p', then $p\backslash L$ can perform α to become $p'\backslash L$ only if $\alpha, \overline{\alpha} \notin L$.
- The operator $[f]$ expresses the relabeling of actions. If p can perform α and become p', then $p[f]$ can perform $f(\alpha)$ and become $p'[f]$.
- Each relabeling function f has the property that $f(\tau) = \tau$.
- A constant x behaves as p if $x \stackrel{\text{def}}{=} p$.

The operational semantics of a process p is a labelled transition system, i.e., an automaton whose states correspond to processes (the initial state corresponds to p) and whose transitions are labelled by actions in $\mathcal{A}$.

2. A Precise Notation for Defining Properties: This need can be solved using a temporal logic. Temporal logics present constructs allowing to state in a formal way that, for instance, all scenarios will respect some property at every step, or that some particular event will eventually happen, and so on. A model checker then accepts two inputs, a system described, for example, in process-algebraic notations and a temporal formula, and returns "true" if the system satisfies the formula and "false" otherwise. In this paper we use the logic *selective mu-calculus* (Barbuti et al., 1999; Santone and Vaglini,). It was defined with the goal of reducing the number of states of the transition systems in such a way that the reduction is driven by the formulae to be checked, and in particular by the syntactic structure of the formulae. The selective mu-calculus is a variant of the mu-calculus (Stirling, 1989), and differs from it in the definition of the modal operators. The syntax of the selective mu-calculus is the following, where K and R range over sets of actions, while Z ranges over a set of variables:

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

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

- each state satisfies `tt` and no state satisfies `ff`;
- a state satisfies $\phi_1 \vee \phi_2$ ($\phi_1 \wedge \phi_2$) if it satisfies ϕ_1 or (and) ϕ_2.
- $[K]_R \phi$ and $\langle K \rangle_R \phi$ are the selective modal operators. $[K]_R \phi$ is satisfied by a state which, for every performance of a sequence of actions not belonging to $R \cup K$, followed by an action in K, evolves in a state obeying ϕ. $\langle K \rangle_R \phi$ is satisfied by a state which can evolve to a state obeying ϕ by performing a sequence of actions not belonging to $R \cup K$ followed by an action in K.

One of the most popular environments for verifying concurrent systems is the Concurrency Workbench of New Century (CWB-NC) (Cleaveland and Sims, 1996), which supports several different specification languages, among which CCS. In the CWB-NC the verification of temporal logic formulae is based on model checking (Clarke et al., 2001).

4 THE METHODOLOGY

In this section we present our methodology for the detection of Android malware families using model checking. It is based on two main steps:

Step 1: Java Bytecode-to-CCS Transform Operator

The first step generates a CCS specification from the Java Bytecode of the .class files derived by the analysed apps. This is obtained by defining a Java Bytecode-to-CCS transform operator $\mathcal{T}$. The function $\mathcal{T}$ directly applies to the Java Bytecode and translates it into CCS process specifications. The function $\mathcal{T}$ is defined for each instruction of the Java Bytecode.

In the following, a Java Bytecode program P is a sequence c of instructions, numbered starting from address 0; $\forall i \in \{0, \ldots, \sharp c\}$, and $c[i]$ is the instruction at address i, where $\sharp c$ denotes the length of c. All Java Bytecode instructions have been translated in CCS; below we will show only a few, just to give the reader the flavor of the approach followed.

Instruction: $c[i] =$ `goto j`

$$\mathcal{T}(i) \quad = \quad x_i \stackrel{\text{def}}{=} \texttt{gotoj}.x_j$$

The instruction $c[i] =$ `goto j` is translated into a CCS process x_i that performs the action `gotoj` and then jumps to the instruction `j`, corresponding to the CCS process x_j.

Instruction: $c[i] =$ `tstore x`

$$\mathcal{T}(i) \quad = \quad x_i \stackrel{\text{def}}{=} \texttt{store}.x_{i+1}$$

Each `tstore x` instruction is translated, regardless of the type `t` and of the name of the variable `x`, as `store` followed by the constant process x_{i+1} representing the CCS translation of the successive instruction.

Step 2: Expressing Android Malware Families into Temporal Logic

The second step aims at discovering android malware families, expressed in temporal logic. The CCS processes obtained in the first step are used to prove properties: using model checking we determine the detection of malware families. Codes described as CCS processes are first mapped to labelled transition systems and the CWB-NC is used. Different properties have been defined characterizing the behaviour of the families.

Table 1 elicits the malicious behaviours for the analysed families and the resulting translation into logic rules. Logic rules model the malicious behavior in order to find it in the model.

The distinctive features of this methodology are: (i) the use of formal methods; (ii) the detection on Java Bytecode and not on the source code; (iii) the detection of malicious payloads; (iv) the use of static analysis; (v) the capture of malicious payloads at a finer granularity.

In practice, from the Java Bytecode we derive CCS processes, which are successively used for checking properties expressing the major characteristics of a malware family. Moreover, our methodology exploits the Bytecode representation of the analysed apps. Performing Android malware families detection on the Bytecode and not directly on the source code has several advantages: (i) independence of the source programming language; (ii) detection of malware families without decompilation even when source code is lacking; (iii) ease of parsing a lower-level code; (iv) independence from obfuscation.

5 RESULTS AND DISCUSSION

The malware samples used in the evaluation were collected from Drebin project (Arp et al., 2014; Spreitzenbarth et al., 2013). Each malware sample is labelled according to the *malware family*: each family contains samples which have in common the same payload.

In the following preliminary study we consider the *DroidKungFu* and the *Opfake* families, 100 samples for each family. Furthermore, we develop a framework able to inject several obfuscation levels in Android applications: (i) changing package name; (ii) identifier renaming; (iii) data encoding; (iv) call indirections; (v) code reordering; (vi) junk code insertion. The reader can refer to (Canfora et al., 2015) for further details. We produce the morphed version of the 200 applications: the full dataset is composed by 400 different applications. To highlight the effectiveness of the proposed solution, we submitted the dataset to the top 5 ranked mobile antimalware from AVTEST[3], an independent Security Institute for IT.

Table 2 shows the results obtained with *DroidKungFu* and *Opfake* families and with morphed version (*DroidKungFuMorph* and *OpfakeMorph*).

We consider only samples identified in the right family (column *ident* in Table 2). We also report the samples detected as malicious but not identified in the right family and the samples not recognized as malware (column *unident.* in Table 2). According to the research questions, the problem of identifying malicious payload should be a further research direction in malware analysis. Due to the novelty of the problem, antimalware are not still specialized in family identification. For these reasons some antimalware are unskilled to detect families. To better understand this lack see Table 2, in particular the *unident.* column.

Another problem is that current antimalware are not able to detect malware when the signature mutates: their performance decrease dramatically with morphed samples. In order to try to circumvent the above problems we introduce our methodology and we discuss preliminary results.

5.1 Empirical Evaluation Procedure

To estimate the performance detection 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 are correctly identified in the right family (True Positives), TN is the number of malware correctly identified as not belonging to the family (True Negatives), FP is the number of malware that are incorrectly identified in the right family (False Positives), and FN is the number of malware that were not identified as belonging to the right family (False Negatives).

[3]https://www.av-test.org/en/antivirus/mobile-devices/

Table 1: Families Description and Corresponding Logic Rules.

DroidKungFu	**Rule (selective mu-calculus formulae)**
device rooting *IMEI* *OS type* *device ID* *network type* *C&C server*	$\varphi = \varphi_1 \vee \varphi_2 \vee \varphi_3$ where: $\varphi_1 = \langle pushphone \rangle_\emptyset \langle invokegetSystemService \rangle_\emptyset$ $\langle checkcastandroidtelephonyTelephonyManager \rangle_\emptyset$ $\langle invokegetDeviceId \rangle_\emptyset \, \mathtt{tt}$ $\varphi_2 = \langle pushIMEI \rangle_\emptyset \langle load \rangle_\emptyset \langle invokeinit \rangle_\emptyset \langle invokeadd \rangle_\emptyset \, \mathtt{tt}$ $\varphi_3 = \langle pushchmod \rangle_\emptyset \langle invokeinit \rangle_\emptyset \langle store \rangle_\emptyset \langle load \rangle_\emptyset \, \mathtt{tt}$
Opfake	**Rule (selective mu-calculus formulae)**
SMS sending *SMS monitoring* *download file* *phonebook*	$\psi = \psi_1 \vee \psi_2 \vee \psi_3$ where: $\psi_1 = \langle load \rangle_\emptyset \langle invokesendTextMessage \rangle_\emptyset \, \mathtt{tt}$ $\psi_2 = \langle push \rangle_\emptyset \langle anewarray \rangle_\emptyset \langle invokegetMethod \rangle_\emptyset \, \mathtt{tt}$ $\psi_3 = \langle pushsendTextMessage \rangle_\emptyset \langle load \rangle_\emptyset \langle invokegetMethod \rangle_\emptyset \, \mathtt{tt}$

Table 2: Antimalware Evaluation for DroidKungFu, Opfake DroidKungFuMorph and Opfake Morph Families.

AntiMalware	DroidKungFu		Opfake		DroidKungFuMorph		OpfakeMorph	
	ident.	unident.	ident.	unident.	ident.	unident.	ident.	unident.
AhnLab	2	98	66	34	0	100	44	56
Alibaba	0	100	0	100	0	100	0	100
Antiy	**93**	7	**96**	4	38	62	49	51
Avast	89	11	0	100	24	76	0	100
AVG	4	96	0	100	0	100	0	100
Our Method	87	13	73	27	**89**	11	**73**	27

Table 3: Preliminary Performance Evaluation.

	TP	**FP**	**FN**	**TN**	**PR**	**RC**	**Fm**	**Acc**
DroidKungFu	87	6	13	94	0.93	0.87	0.90	0.91
Opfake	73	8	27	92	0.90	0.73	0.80	0.83
DroidKungFuMorph	89	1	11	99	0.98	0.89	0.93	0.94
OpfakeMorph	73	8	27	92	0.90	0.73	0.80	0.83

5.2 Preliminary Evaluation

We have implemented a prototype tool and we have conducted experiments for a proof of concept of our methodology. Table 3 shows the results obtained using our prototype tool: we obtain an accuracy ranging from 0.83 to 0.94.

RQ1 response: Results in Table 2 show that our method is promising to identify malware payload. We obtain, when comparing not morphed malware, performance quite in line with top 5 mobile antimalware. Instead, the gap between our approach and the signature-based detection is broader in the morphed sample evaluation.

RQ2 response: We outperform the top 5 current signature-based approach in detecting morphed samples as shown in Table 2. Instead, when evaluating not-morphed samples, Antiy achieves better results.

It should be underlined that the method we propose is robust: Table 3 shows that Accuracy and F-Measure values are not affected from code obfuscation. It is worth noting that Accuracy and F-Measure increase in detecting DroidKungFu morphed samples than not-morphed ones, while they are the same in evaluating Opfake and OpfakeMorphed samples: this is the reason why our method is transparent respect to obfuscation, differently from the antimalware that dramatically decrease when evaluating morphed samples.

6 CONCLUDING REMARKS AND FUTURE WORK

Since previous works in mobile malware detection focus on the research in discriminating a malware application from a trusted one, in this paper we propose an approach to localize the malicious behaviour at a finer grain, i.e., at payload level.

We use model checking in order to test our model against two of most diffused malware family in Android environment: the *DroidKungFu* and the *Opfake* families. We test in addition the robustness of our solution generating morphed malware and testing it using the model. Results seem to be promising: we identify malicious payloads with a very high accuracy value and with a reasonable time. This implies that our methodology is efficient and scalable.

As future work we are going to extend our preliminary evaluation to other widespread families. Furthermore, we plan to track the phylogenesis of malware to characterize the payload family tree and to foresee the possible payload evolution.

ACKNOWLEDGEMENTS

The Authors thank Domenico Martino for helping in the implementation of the prototype tool used to test the methodology.

REFERENCES

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.

Bailey, U., Comparetti, P., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*. IEEE.

Barbuti, R., Francesco, N. D., Santone, A., and Vaglini, G. (1999). Selective mu-calculus and formula-based equivalence of transition systems. Elsevier.

Canfora, G., Di Sorbo, A., Mercaldo, F., and Visaggio, C. (2015). Obfuscation techniques against signature-based detection: a case study. In *Proceedings of Workshop on Mobile System Technologies*. IEEE.

Canfora, G., Mercaldo, F., and Visaggio, C. A. (2013). A classifier of malicious android applications. In *Proceedings of the 2nd International Workshop on Security of Mobile Applications, in conjunction with the International Conference on Availability, Reliability and Security*. IEEE.

Clarke, E. M., Grumberg, O., and Peled, D. (2001). *Model checking*. MIT Press.

Cleaveland, R. and Sims, S. (1996). The ncsu concurrency workbench. In Alur, R. and Henzinger, T. A., editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*. Springer.

Dumitras, T. and Neamtiu, I. (2011). Experimental challenges in cyber security: A story of provenance and lineage for malware. ACM.

Hu, X., Chiueh, T., Shin, K., Kruegel, C., and Kirda, E. (2009). Large-scale malware indexing using function call graphs. In *ACM Conference on Computer and Communications Security*. ACM.

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.

Jang, J., Brumley, D., and Venkataraman, S. (2011). Bitshred: feature hashing malware for scalable triage and semantic analysis. In *ACM Conference on Computer and Communications Security*. ACM.

Karim, M. E., Walenstein, A., Lakhotia, A., and Parida, L. (2005). Malware phylogeny generation using permutations of code. Springer.

Khoo, W. and Lio, P. (2011). Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. In *SysSec Workshop*. Springer.

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

Ma, J., Dunagan, J., Wang, H. J., Savage, S., and Voelker, G. M. (2006). Finding diversity in remote code injection exploits. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM.

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

Santone, A. and Vaglini, G. Abstract reduction in directed model checking CCS processes. Springer.

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.

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 Yonezawa, A. and Ito, T., editors, *Concurrency: Theory, Language, And Architecture*, LNCS, pages 2–20. Springer.

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