Malware Analysis IV

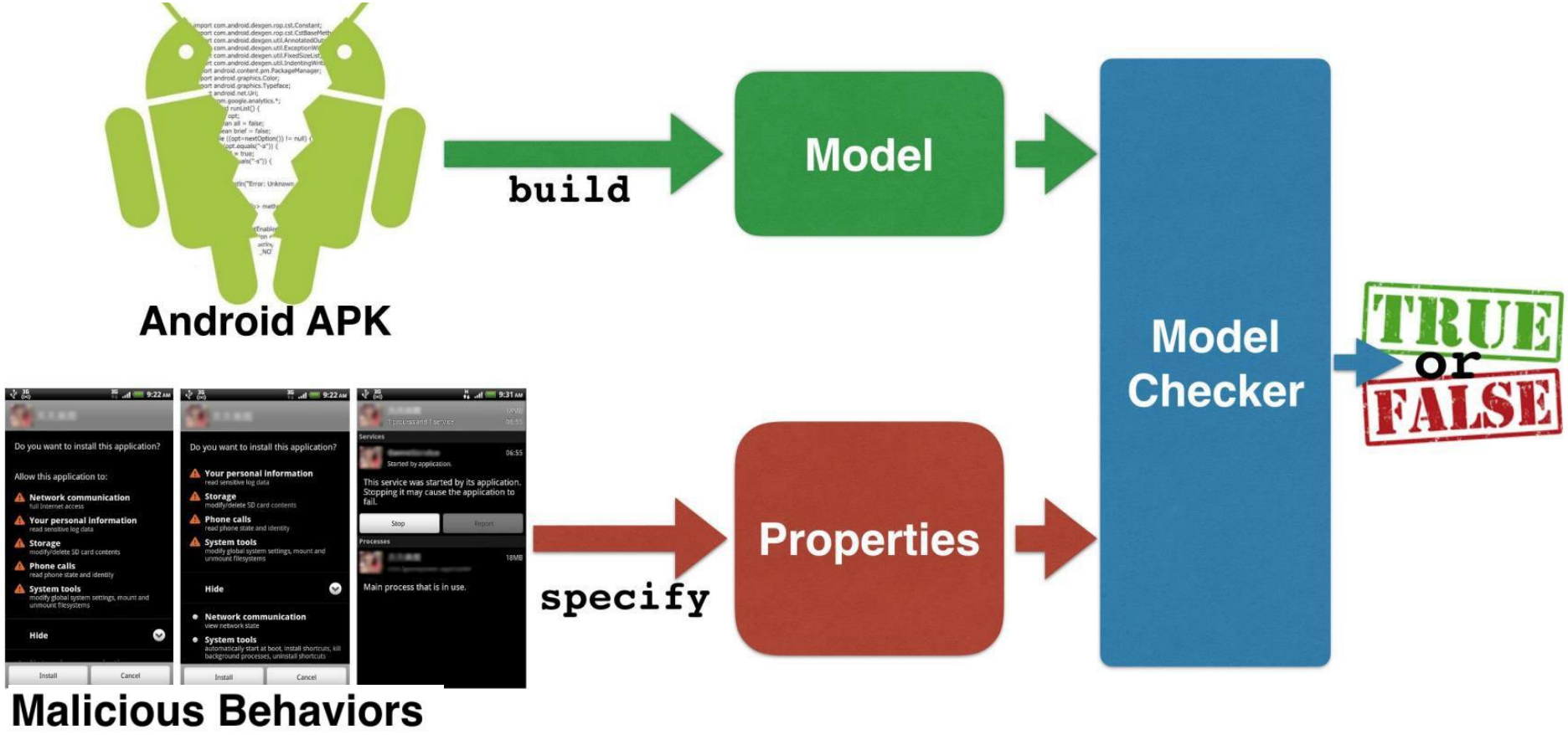
Francesco Mercaldo
University of Molise, IIT-CNR
francesco.mercaldo@unimol.it

Formal Methods for Secure Systems, University of Pisa

Finding Malicious Behaviour

- The manual ispection process
- Take a look to the Bytecode and the Java source code

A Formal Framework for Mobile Malicious Detection

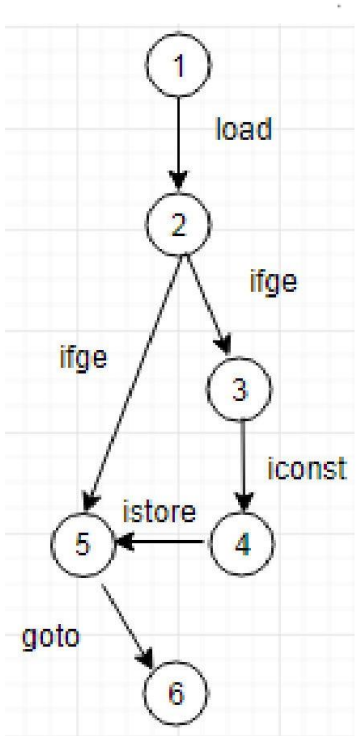


Example of bytecode instructions

`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

From bytecode to automaton

```
1  .....
2  1:   load x                    // load x on the stack
3  2:   ifge 5                    // jump if greater or equal
4  3: iconst_1                    // push 1 on the stack
5  4: istore_2                    // pop off the stack and store the value in a register
6  5: goto 6                      // jump
7  6: .....
```



From bytecode to automaton

The automaton describes mainly the behavior of method as a set of reachable states and actions (instructions) that trigger a change of state.

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.

- sequential instructions are modeled as a state with only one outgoing transition to the successive instruction
- conditional branch instruction are modeled as a state with two outgoing transitions depending on the result of the valuation of the condition
- 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

Calculus of Communicating Systems (CCS) can be used instead of generating directly the automaton.

Why Formal Methods?

- The checking process is supported by tools
- The possibility of using the diagnostic counterexamples
- Temporal logic can easily and correctly express the behaviour of a spyware application
- Formal verification allows evaluating all possible scenarios, the entire state space all at once

Model checking for malware analysis

- Model: control flow graph of the code (plus some additional information)
- Logic formula: malware pattern

Verification by model checking algorithms that the model satisfies the formula

The verification is automated, there is no need to construct the correctness proof

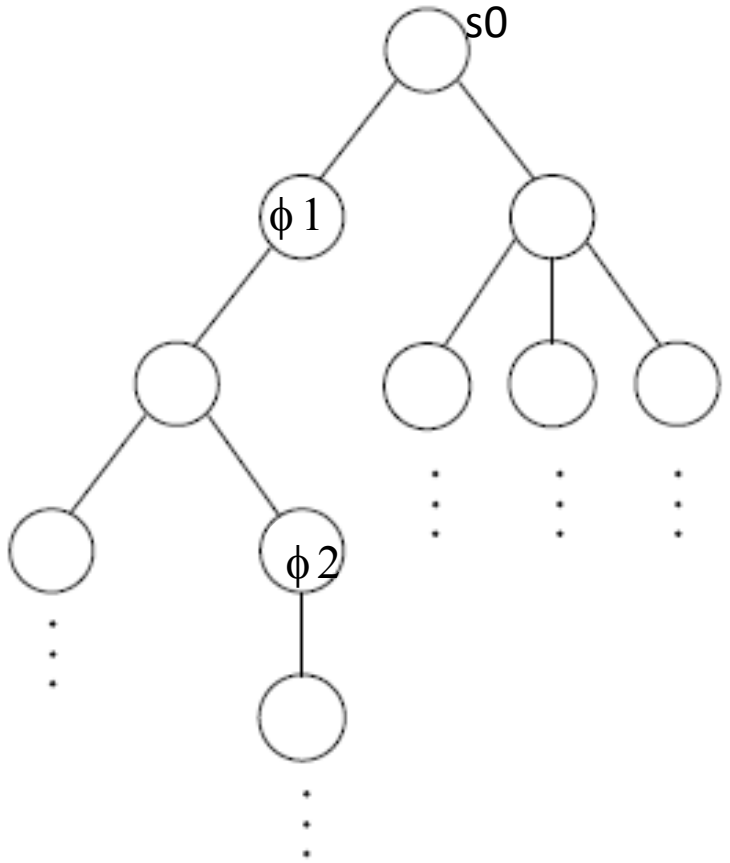
Many model checking tools exist, with different formalisms for the specification of the model and for the specification of the properties.

- NOME : LEILA
- Developed at Univ. of Molise, Campobasso /IIT-CNR, Pisa
- Malware detection using model checking in android app
- Based on the CWB model checker (CCS and mu-calculus)

Example: Sequences of bytecode instructions in the CFG of a method

ϕ1: ldc *Activation*

ϕ2: ldc *HomePage*



Property 1 (P1)

There exists a path such that: a state is reached that satisfies ϕ1 and successively

a state is reached that satisfies ϕ2

S0 |= P1 TRUE

Example: Sequences of bytecode instructions in the CFG of a method

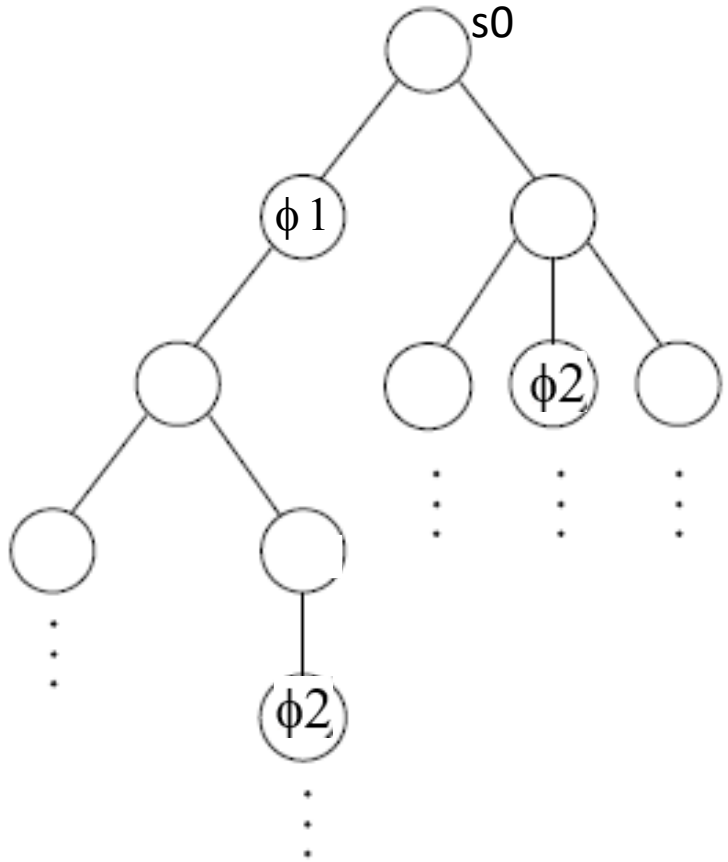
ϕ1: ldc *Activation*

ϕ2: ldc *HomePage*

Property 1 (P1)

There exists a path such that: a state is reached that satisfies ϕ1 and successively a state is reached that satisfies ϕ2

S0 |= P1 TRUE



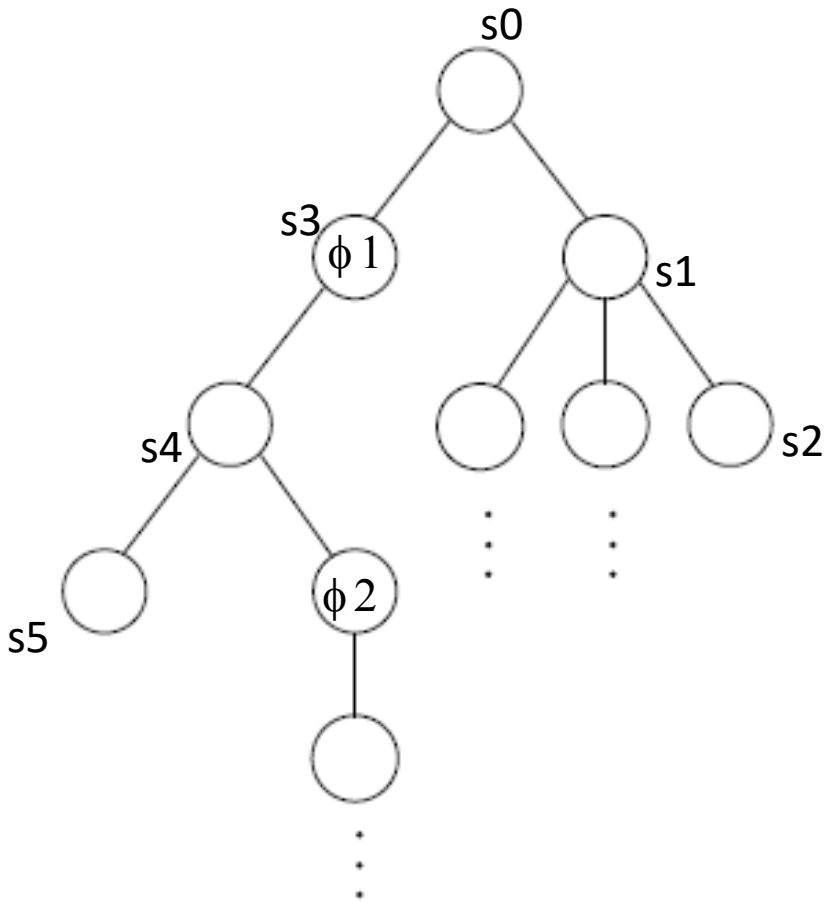
Example: counter-example facility

ϕ 1: ldc
Activation
ϕ 2: ldc
HomePage

Property 2 (P2)
For all paths: a state is reached that satisfies ϕ1 and successively a state is reached that satisfies ϕ2

S0 |= P2 FALSE

Counter-example: s0 s1 s2
Counter-example: s0 s3 s4 s5



Building a model for an app

The model of an app is a set of automata, one for each method.

After having constructed the automata for an app, we define the characteristic behaviour of the malware by means of the definition of the temporal logic properties.

This process tries to recognize specific and distinctive features of the malware 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 malicious behavior implementation at Bytecode level.
The properties are proved separately on each method.

An app is a malware if there exists at least one method that satisfies the properties.

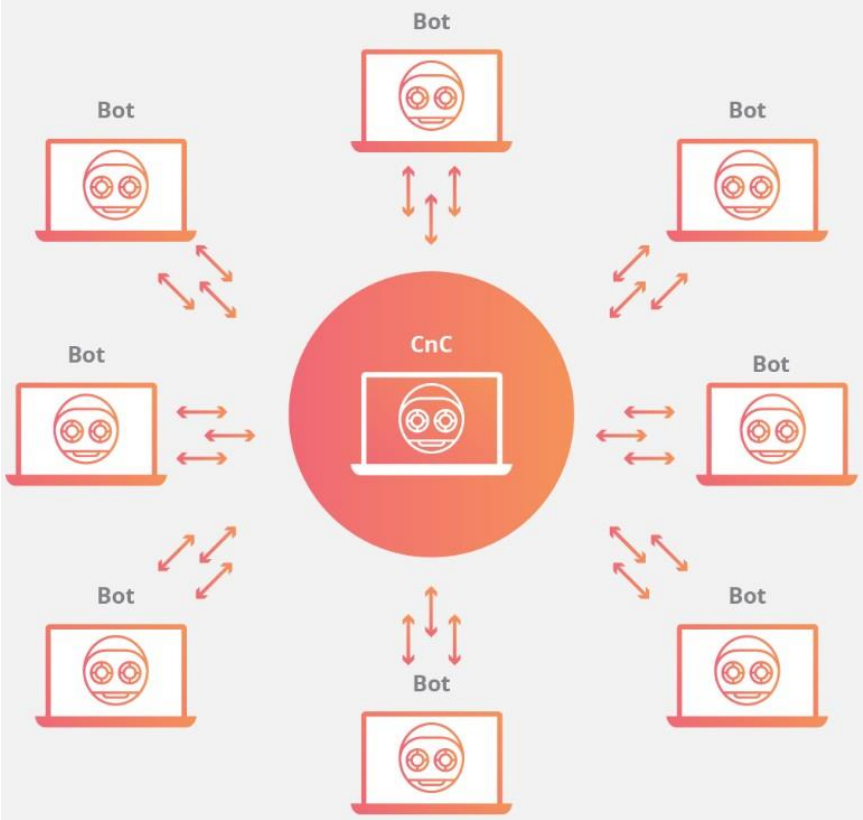
This formalisation assumes that the malware is local to a method. This is the more common case in practice, and we will see how to overcome this limitaton

Examples of (botnet) malware family payload

- The term *mobile botnet* refers to a group of compromised smartphones that are remotely controlled by botmaster using Command & Control (C&C) channels.
- 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.

Botnet main schema

A botnet is a network of hijacked devices infected with bot malware and remotely controlled by a hacker (C&C)
The C&C tells the malware what to do next and acts as a repository for captured information



TigerBot Botnet

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. The SMS is silently processed by the malware.

The attacker tells the malware what to do next and acts as a repository for capturing information.

TigerBot Botnet

Downloaded by the user as a legitimate app, TigerBot registers itself as a receiver with high priority of messages with android.provider.Telephony.SMS_RECEIVED

Requests numerous permissions such as:
READ_PHONE_STATE
RECORD_AUDIO
MODIFY_AUDIO_SETTINGS
READ_CONTACTS
WRITE_CONTACTS
SEND_SMS
RECEIVE_SMS
BLUETOOTH
BROADCAST_SMS
……

TigerBot Botnet

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

TigerBot Botnet

```
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");
    /* ... */
  }
```

The snipped of code shows a method of the malcious app. The code contains obfuscation techniques: name of the method a$a

The method reads the content of the variable paramMessage of type Message. Message is a HashMap where there are the key *packName*, *OpenGPS*, …..

The app read the value of these keys and if they are set to a given value, the corresponding malicious action is executed.

TigerBot Botnet

```
1  public a$a(a parama, Message paramMessage)
2    {
3      /* ... */
4      this.jdField_for = paramMessage.getData().getString("packName");
5      /* ... */
6      this.jdField_if.jdField_char = paramMessage.getData().getString("addrType");
7      this.jdField_if.jdField_case = paramMessage.getData().getBoolean("openGPS");
8      /* ... */
9      this.jdField_if.jdField_long = paramMessage.getData().getInt("timeOut");
10     this.jdField_if.jdField_goto = paramMessage.getData().getInt("priority");
11     this.jdField_if.jdField_void = paramMessage.getData().getBoolean("location_change_notify");
12     /* ... */
13   }
```

packName command: sends the name of the malware application on the device

addrType command: converts a coordinate to a String representation.
The type can be one of FORMAT DEGREE, FORMAT MINUTES, FORMAT SECONDS

openGPS command: activates the fine-grained localization (e.g., GPS sensors)

timeOut command: represents the frequency at which geolocalization listener receives updates

priority command: sets the priority related to the botnet command execution

location_change_notify command: sends a notification to the C&C server whether the device under analysis change location

TigerBot Botnet Payload Detection Temporal Logic Formula

```
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");
    /* ... */
  }
```

$$
\begin{aligned}
\varphi_{TIGERBOT} &= \mu X.\langle invokegetData\rangle\, \varphi_{1_1} \vee \langle -invokegetData\rangle\, X \\
\varphi_{1_1} &= \mu X.\langle pushpackName\rangle\, \varphi_{1_2} \vee \langle -pushpackName\rangle\, X \\
\varphi_{1_2} &= \mu X.\langle invokegetString\rangle\, \varphi_{1_3} \vee \langle -invokegetString\rangle\, X \\
\varphi_{1_3} &= \mu X.\langle invokegetData\rangle\, \varphi_{1_4} \vee \langle -invokegetData\rangle\, X \\
\varphi_{1_4} &= \mu X.\langle pushprodName\rangle\, \varphi_{1_5} \vee \langle -pushprodName\rangle\, X \\
\varphi_{1_5} &= \mu X.\langle invokegetString\rangle\, \varphi_{1_6} \vee \langle -invokegetString\rangle\, X \\
\varphi_{1_6} &= \mu X.\langle invokegetData\rangle\, \varphi_{1_7} \vee \langle -invokegetData\rangle\, X \\
\varphi_{1_7} &= \mu X.\langle pushcoorType\rangle\, \varphi_{1_8} \vee \langle -pushcoorType\rangle\, X \\
\varphi_{1_8} &= \mu X.\langle invokegetString\rangle\, \varphi_{1_9} \vee \langle -invokegetString\rangle\, X \\
\varphi_{1_9} &= \ldots
\end{aligned}
$$

The formula is verified if there exists at least one automaton that contains a path with the sequence of actions <invokegetData> <pushpackName> ……, possibly interleaved with other actions.

RootSmart Botnet

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.

RootSmart Botnet

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.

Whether RootSmart fails to obtain the root privilege, it will still attempt to install the downloaded apps.

RootSmart Botnet Payload Detection Temporal Logic Formula

```
{ /* ... */
  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)) { /* ... */ }
    /* ... */
}
```

The formula is verified if there exists at least one automaton that contains a path with the sequence of actions <pushactionhoststart> <invokeequals> ……, possibly interleaved with other actions.

$$
\begin{aligned}
\varphi_{ROOTSMART} &= \mu X.\langle pushactionhoststart\rangle\, \varphi_{2_1} \vee \langle -pushactionhoststart\rangle\, X \\
\varphi_{2_1} &= \mu X.\langle invokeequals\rangle\, \varphi_{2_2} \vee \langle -invokeequals\rangle\, X \\
\varphi_{2_2} &= \mu X.\langle ifeq\rangle\, \varphi_{2_3} \vee \langle -ifeq\rangle\, X \\
\varphi_{2_3} &= \mu X.\langle pushactionboot\rangle\, \varphi_{2_4} \vee \langle -pushactionboot\rangle\, X \\
\varphi_{2_4} &= \mu X.\langle invokeequals\rangle\, \varphi_{2_5} \vee \langle -invokeequals\rangle\, X \\
\varphi_{2_5} &= \mu X.\langle ifeq\rangle\, \varphi_{2_6} \vee \langle -ifeq\rangle\, X \\
\varphi_{2_6} &= \mu X.\langle pushactionshutdown\rangle\, \varphi_{2_7} \vee \langle -pushactionshutdown\rangle\, X \\
\varphi_{2_7} &= \mu X.\langle invokeequals\rangle\, \varphi_{2_8} \vee \langle -invokeequals\rangle\, X \\
\varphi_{2_8} &= \mu X.\langle ifeq\rangle\, \varphi_{2_9} \vee \langle -ifeq\rangle\, X \\
\varphi_{2_9} &= \ldots
\end{aligned}
$$

Data set

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

Results obtained

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

Legitimate dataset: 1000 Android goodware apps were downloaded from the Google's official app store

Botnet behaviour distributed in more methods

So far, the behaviour of a botnet is identified in a single method.

However, to identify cases in which the behaviour of the botnet is distributed in different methods (i.e., automata), the following approach could be applied.

If an invoke bytecode instruction is encountered, and the current method satisfies $\varphi 1j$ (for some j), we must also check if the invoked method is a model for the formula $\varphi 1k$, with k=j+1.

Conclusions

- On overview about malware taxonomy and classification
- We generated a X86 payload
- We discussed about Android security
- We use (real-world) tools for Android malware analysis for finding

interesting pattern

- We explained how formal methods can be useful for effective malware family detection
- In the future detection will be more and more difficult
  - because malware is increasing its obfuscation capabilities...

References

- Gerardo Canfora, Fabio Martinelli, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Corrado Aaron Visaggio: ***LEILA: Formal Tool for Identifying Mobile Malicious Behaviour***. IEEE Trans. Software Eng. 45(12): 1230-1252 (2019)

- Cinzia Bernardeschi, Francesco Mercaldo, Vittoria Nardone, Antonella Santone: ***Exploiting Model Checking for Mobile Botnet Detection***. KES 2019: 963-972

•Aniello Cimitile, Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Corrado Aaron Visaggio: ***Talos: no more ransomware victims with formal methods***. Int. J. Inf. Sec. 17(6): 719-738 (2018)

•Francesco Mercaldo, Vittoria Nardone, Antonella Santone, Corrado Aaron Visaggio: ***Download malware? no, thanks: how formal methods can block update attacks***. FormaliSE@ICSE 2016: 22-28