Dynamic analysis architecture

Emulator

Since virtualbox doesn't support ARM on x86 we most likely will use the goldfish one.
Here we need to document the ways common android malware is detecting the emulator (i.e. what are the emulation artifacts we need to take care of)

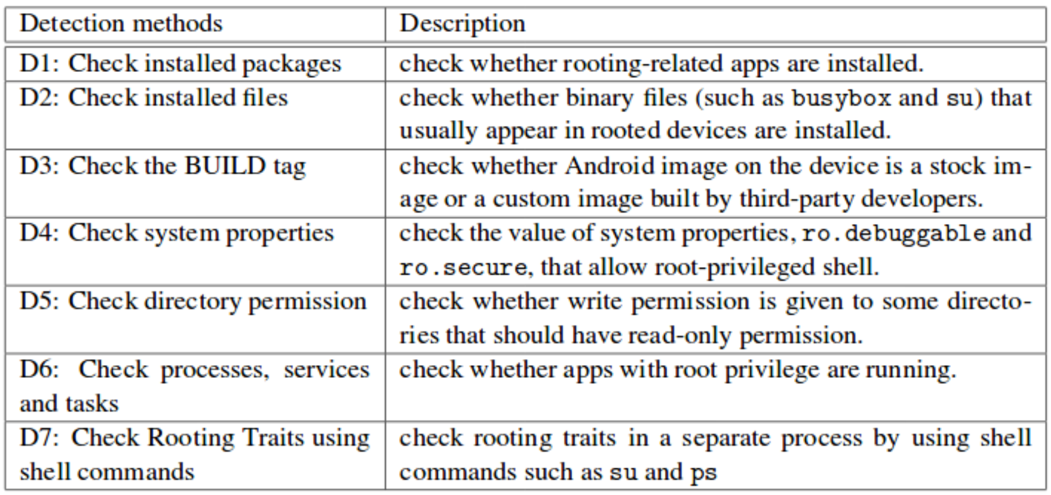
I(waqar) couldn't find any information about The "EmulatorAntiDetect" module contained in cuckoodroid, but I tested our emulator with that module enabled with an app that tries to detect emulator using some common methods and it can't detect our emulator. I will add more of the methods used here, currently I am looking for something goldfish specific but most of the information out there are very general and can only be verified by testing. We will probably need to make an app that employs all of our emulation detection methods and then we can test our EmulatorAntiDetection module against it. Below is the link to repo that I used to detect our emulator:
https://github.com/oguzhantopgul/Android-Emulator-Detection

Common Detection Methods and their anti-detection counterpart:

For introductory info and techniques Please refer to Section 2.2 , 2.4 of Jailbreak/Root Detection Evasion Study on iOS and Android .

Below are the rooting detection methods discussed in Android Rooting: Methods, Detection, and Evasion

Table 1: Rooting detection methods presented in [16]



Detection methods	Description
D1: Check installed packages	check whether rooting-related apps are installed.
D2: Check installed files	check whether binary files (such as `busybox` and `su`) that usually appear in rooted devices are installed.
D3: Check the BUILD tag	check whether Android image on the device is a stock image or a custom image built by third-party developers.
D4: Check system properties	check the value of system properties, `ro.debuggable` and `ro.secure`, that allow root-privileged shell.
D5: Check directory permission	check whether write permission is given to some directories that should have read-only permission.
D6: Check processes, services and tasks	check whether apps with root privilege are running.
D7: Check Rooting Traits using shell commands	check rooting traits in a separate process by using shell commands such as `su` and `ps`

These methods can be easily evaded by hooking the APIs using Xposed Framework.

Another paper An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks uses the techniques in Android Rooting: Methods, Detection, and Evasion with some addition to avoid dynamic analysis. Below is their approach:

1. It repackages the original APK
2. In repackaged APK the original dex file is moved to assets folder and a new stub dex file is added to apk which is contains all the code for detecting root and hooking
3. If the device is not rooted and no hooking is done then the stub DEX creates a class loader for the original dex file using `DexClassLoader()`
4. Then it replaces its class loader with the new class loader using makeApplication(). The original executable file invokes onCreate() method of its start component.

This process is illustrated in below diagram

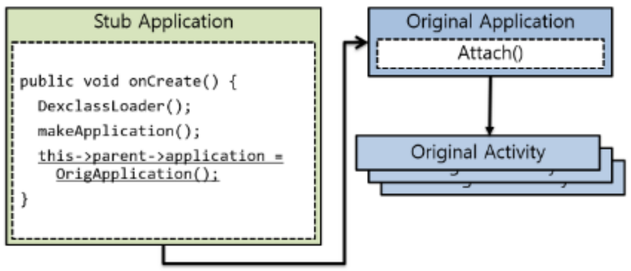
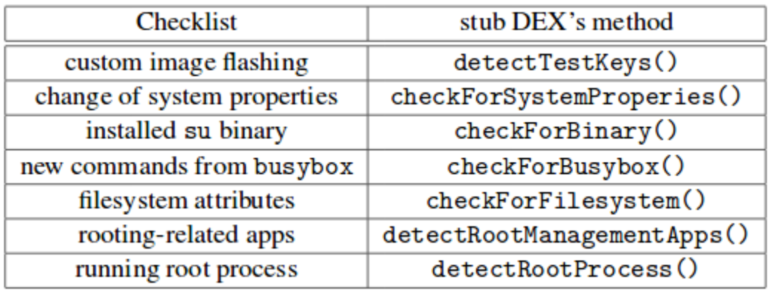


Figure 2: Altering execution flow using dynamic code loading

Table 2: Checklist and stub DEX's methods



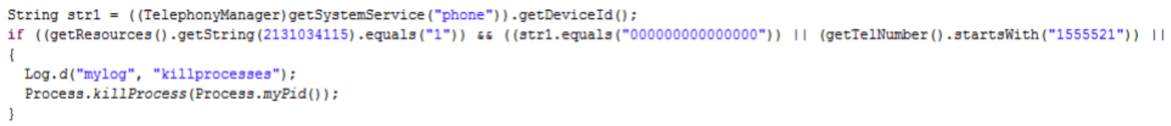
Checklist	stub DEX's method
custom image flashing	detectTestKeys()
change of system properties	checkForSystemProperies()
installed su binary	checkForBinary()
new commands from busybox	checkForBusybox()
filesystem attributes	checkForFilesystem()
rooting-related apps	detectRootManagementApps()
running root process	detectRootProcess()

Malware Detection techniques for mobile devices provides on some introductory information about Mobile Malware and some statistics from some sources which gives a general idea of mobile malware evasion techniques in use. I couldn't find any up-to date study discussing Mobile malware evasion techniques currently in use and we can probably do that once at some later time once our dynamic analysis system is up and running on platform.

A list of evasion techniques

Check how many of these are implemented in cuckoodroid and then implement the others so that we have the first 13 ones working.

1. Reading telephone number
2. Mobile country code
3. Build.Host property
4. Build.Fingerprints property
5. Reading device IMEI number

   ```
   String str1 = ((TelephonyManager)getSystemService("phone")).getDeviceId();
   if ((getResources().getString(2131034115).equals("1")) && ((str1.equals("000000000000000")) || (getTelNumber().startsWith("1555521")) ||
   {
     Log.d("mylog", "killprocesses");
     Process.killProcess(Process.myPid());
   }
   ```

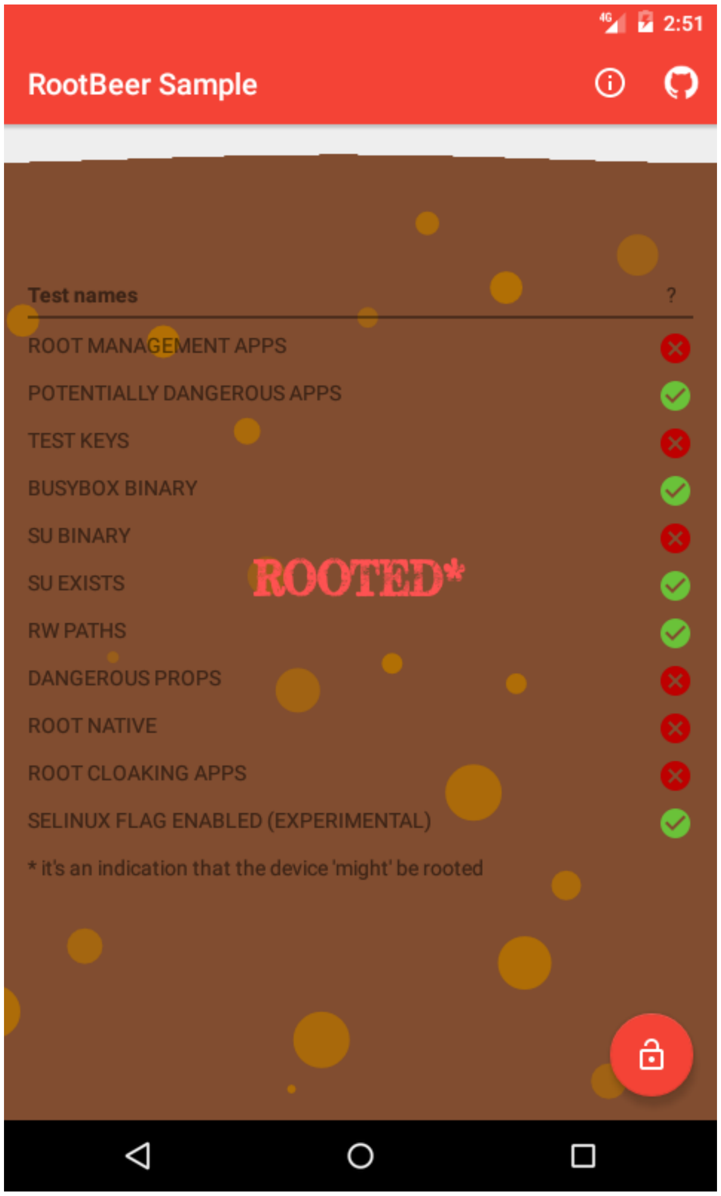
6. Network address e.g. 10.0.0.2/24 and routing table (netH)
   1. virtual router (address space: 10.0.2/24)
   2. Emulated network (IP address: 10.0.2.15)
7. Checking device uptime
8. Checking ping example.com
9. CPU serial number mostly in 0's
10. CPU frequency files absent
11. Number of sensors (SensorManager API)
12. Bluetooth not present
13. Battery level
14. Checking for Manufacturer related software which comes pre-installed
15. Checking for other usage related data e.g image, sms, calls, WLAN networks, browser history, installed applications e.g facebook etc.

16. Checking for Google internet services
17. Checking for marketplace application
18. Checking for carrier added applications
19. Delaying execution
20. Detecting lack or abundance of use input
21. Checking if Qemu can handle interrupt while executing blocks
22. isUserMonkey(), setRunAsMonkey(false) in Android API
23. Checking other network configurations
24. Using native code to detect Qemu based detection in CPU implementations like some bugs
25. CPU Performance
26. GPU performance (Low FPS, Large variation(May not work with new emulators because of improved integration))
27. Input from sensors is not spread enough
28. Camera detection
29. Checking for Superuser.apk
30. Checking for some specific applications with package names
31. **Checking for apps** which run only on rooted devices, e.g, busybox, executing su and id command,
32. **Checking for files** e.g, /system/bin/,/system/xbin/,/sbin/. Or searching them in $PATH (manually or by which "file")
33. **File permissions:** Some rooting tools make certain root folders readable (e.g.,/data) or writable (e.g., /etc, /system/xbin, /system, /proc,/vendor/bin) to any process on the device. The File.canRead and File.canWrite Java APIs, or access C API can be used to check whether such condition exists.
34. **Shell Command Execution:** Applications can use Runtime.exec Java API, ProcessBuilder Java class, or execve C API to execute a specific command in a separate process, and then examine the output of the shell command to detect rooting traits. Commonly employed shell commands are:
    1. **su:** If the su binary exists, this shell command will exit without error; otherwise, an IOException will be thrown to the calling application.
    2. **which su:** This check involves executing the Linux ”which” command with parameter “su” and verifying if the result is 0 (indicating the su binary was found). This is essentially the same as parsing the PATH variable, but requires less work for the caller.
    3. **id:** This check is based on executing the command ”id”, and verifying the UID to determine if it corresponds to root (uid=0 is root).
35. **System Mounted:** Normally the “/system” partition on an Android device is mounted “ro” (read only). Some rooting methods require this partition to be remounted “rw” (read-/write). There are mainly two variants of this check. The first simply runs the mount command and looks for a “rw” flag. The second attempts to create a file under “/system/” or “/data/”. If the file is successfully created, it implies the mount is “rw”.
36. **Ability to mount:** This method attempts to mount the “/system” partition with the command “mount -o remount,rw /system”, and then checks if the return code was 0.
37. **Check Processes/Services/Tasks:** This check consists of finding whether a specific application that requires root privileges is running on the device. The ActivityManager.getRunningAppProcesses method returns a list of currently running application processes. Similarly, getRunningServices or getRecentTasks APIs are also used by applications to retrieve a list of current running services or tasks.
38. **System properties:** If the value of system property ro.secure equals “0”, the adb shell will be running as root instead of shell user. System.getProperty Java API can be used by applications to examine this property value. In addition, examining ADB source code reveals that the adbd daemon is also running as root user if both “ro.debuggable” and “service.adb.root” properties are set to ”1”.
39. **hooking detection:** Detecting hooks using call stack

Emulator detectors or root checkers

Root Beer

Its a library for checking root access, can be easily used by any developer with not much experience.
Github Sample app
Below are the result of the sample app when I ran it on our emulator with EmulatorAntiDetect enabled:



Test names	?
ROOT MANAGEMENT APPS	✗
POTENTIALLY DANGEROUS APPS	✓
TEST KEYS	✗
BUSYBOX BINARY	✓
SU BINARY	✗
SU EXISTS	✓
RW PATHS	✓
DANGEROUS PROPS	✗
ROOT NATIVE	✗
ROOT CLOAKING APPS	✗
SELINUX FLAG ENABLED (EXPERIMENTAL)	✓

ROOTED*

* it's an indication that the device 'might' be rooted

To thwart test like these we can deploy Root Cloakers like RootCloask or make our own.

Emulator detection open source projects

A lot of them pretty old and not properly maintained.

- https://github.com/CalebFenton/AndroidEmulatorDetect
- https://github.com/oguzhantopgul/Android-Emulator-Detection
- https://github.com/strazzere/anti-emulator

HARVESTING method or Slicing method that overcome detection and other measures used to avoid analysis

Seems to be a lot more work and probably patented.

- http://www.bodden.de/pubs/ssme16harvesting.pdf

Here is another tool (static analysis)which is open source and the above tool is partially based on this one:

- https://github.com/SAAF-Developers/saaf
- https://www.ei.rub.de/media/emma/veroeffentlichungen/2013/04/16/slicing_droids-sac13.pdf

API call interception options:

up until now we identified these options for API call interception during dynamic analysis:

- Frida (appmon)
- xposed + inspackage
- xposed + droidmon

for each of them we need to know:

- how well is it maintained
- license
- what sort of fingerprint is it introducing inside the VM
- which android versions / architectures are supported
- what kind of API calls can we intercept (java calls, C calls, system calls)
- what is the impact on performance

If we go for frida: can you write some demo script to place some hooks at the various supported levels (java, C calls, system calls)?

How about native code?

Links

Rooting_detection_methods.png (48.6 KB) Waqar Rashid, 2017-12-15 12:36 PM
Stub_dex_working.png (61.1 KB) Waqar Rashid, 2017-12-15 01:00 PM
root_beer_result.png - Result of runnnig rootbeer root detection (53.8 KB) Waqar Rashid, 2017-12-15 02:55 PM
getdeviceid_emulator_detection.png (33.6 KB) Waqar Rashid, 2018-01-09 03:23 PM