Taxonomy of Evasive controls in Android

1 Environment verification

Environment checks aim to detect the reliability of the environment in which apps are installed.

1.1 Root detection

Android has been designed to work in a way that the end user does not need to use the root account, so its usage is turned off by default. A method known as *rooting* allows an end user to get super-user access to an Android smartphone. With super-user permissions, it is possible to alter system settings, access private areas in the primary memory, and install specialized apps. Therefore, some anomalous executables that need root permissions to run correctly may indicate a counterfeit environment, such as a sandbox. Of all the possible controls, the most common are verifying if the `su` or `busybox` executables are present in the file system and checking if well-known paths that are usually read-only have the write permission. Obtaining root permissions also allows the end user to use a debugger or use dynamic analysis tools.

1.2 Debugging detection

A debugger introduces changes to the memory space of the target app's processes and may impact the execution time of certain code snippets. Thus, anti-debugging techniques can detect (looking for specific artifacts) or prevent the app from being debugged.

1.3 Hook detection

Anti-hooking controls aim to detect dynamic binary instrumentation tools (e.g., Xposed and Frida) that can hook and tamper with the execution flow of an app. The simplest way to detect them is to scan package names, files, or binaries to look for well-known frameworks' resources. However, each dynamic analysis framework works differently and may require specific detection techniques. For instance, Xposed is an Android app that applies modules directly to the Android OS ROM and requires root privileges. At the same time, Frida injects a JavaScript engine into the instrumented process.

1.4 Emulator detection

Anti-emulation techniques check whether an app is running on an actual device. For example, an emulated environment may not provide the same hardware functionalities of an actual phone, such as sensors (i.e., gyroscope, accelerometer), or some particular artifacts may or may not be present (e.g., different files or files' content, Android system properties). For instance, the Android emulator is built on top of QEMU. Some of them may not have the full support with the Google Play Services (e.g., Genymotion) or changed some system property (e.g., `ro.build.tag`).

1.5 Memory integrity verification

This type of evasive control aims to verify the integrity of the app's memory space against memory patches applied at runtime. For instance, hooks to C/C++ code can be installed by overwriting function pointers in memory or patching parts of the function code itself (e.g., inline hooking by modifying the function prologue). Thus, an app can check the integrity of the respective memory regions to detect any alteration.

1.6 App-level virtualization detection

Android virtualization is a recent technique that enables an app (*container*) to create a virtual environment in which other apps (*plugins*) can run while fully preserving their functionalities. The container app acts like a proxy, intercepting each request going towards the plugin app to the Android OS and vice versa to fool the OS that such a request comes from the container. Thus, anti-virtualization techniques aim to detect such virtual environment (i.e., container app) in which the app is executing. For instance, an app can verify its UID, the number of running processes, and the object instance of the Android API clients.

1.7 Network artifact detection

This type of evasive control aims to inspect the network interfaces to detect artifacts, such as unreal interface names or adb connected over the network. Moreover, several procedures aim to analyze the app's network traffic to understand its behavior. Thus, evasive apps may also check for VPNs or proxies.

2 APK tampering verification

Anti-tampering techniques detect any modification on the original app during its execution. If modifications are detected, the app can take evasive actions, such as turning off certain features or terminating its execution.

2.1 Signature checking

Each Android app is distributed and installed as an Android PacKage (APK) file. In a nutshell, an APK file is a ZIP archive containing all the necessary files to run the first execution of the app, i.e., compiled code, resources (e.g., images for the user interface), and a *Manifest* file. To ensure the APK integrity, it is signed with the developer's private key and contains the corresponding public certificate of the developer. During the installation process, the Android OS verifies the integrity of the APK and its resources. It is worth noticing that this mechanism does not provide any authentication guarantee, as the developer certificate does not need to be issued by any trusted certificate authority. Thus, this control checks if the certificate is the expected one.

2.2 Code integrity

It checks whether some code or resource has been tampered with by computing its signature at runtime (w.r.t. pre-computed and hardcoded valued).

2.3 Installer verification

Since API level 5, the Android Package Manager also stores information on which 'installer' app (e.g., Google Play Store or Samsung Store) was used to start installing a target app. It checks the package name of the installer app to verify whether the app has been installed from the expected app store. Tampered apps are more likely to be distributed on unofficial app stores that differ from the original. Moreover, an APK can also be downloaded directly from a website, and thus in this particular case, the installer app can be a browser or a file manager.

3 High-level verification

3.1 SafetyNet attestation & Integrity API

SafetyNet is a platform security service offered by Google that provides a set of APIs to help protect apps against security threats, such as device tampering and potentially harmful apps. From a technical standpoint, every service is related to a different API. For instance, to verify the integrity of a device, an app leverages the Attestation API by invoking the `attest` method of the SafetyNet client. Contrary, to check if malicious apps are installed on the device, an app invokes the `listHarmfulApps` API.

Starting January 2023, the SafetyNet attestation is deprecated and replaced by the Play Integrity API. It is an enhanced security mechanism that verifies the app's integrity to defend against tampering and redistribution of your app and the environment in which it is running. Moreover, it consolidates multiple integrity offerings (including the ones offered by SafetyNet) under a single API.

3.2 Interaction with a human being

Sophisticated Android malware sandboxes attempt to prevent sandbox detection by patching runtime properties, but they neglect other aspects, such as simulating real user behavior. In particular, user-related artifacts (e.g., number of photos and songs, list of contacts) can be abused to distinguish an actual device from a sandbox environment.

4 Direct and Indirect techniques

This further orthogonal subdivision based on the type of data that an evasive technique implementation verifies is crucial. *Direct* evasive techniques (DET) retrieve specific data which can be directly used in evasive controls. In contrast, the data returned using *indirect* evasive techniques (IET) need further processing to find the information necessary for detecting an analysis environment.

We clarify this concept with an example. Magisk is a famous open-source software for customizing Android. It needs root access and is installed as a regular Android app. To verify if this app is installed, a developer can interact with the `getPackageInfo` or the `getInstalledApplications` methods of the `PackageManager` (having previously correctly specified permissions in the Manifest file). The former accepts the package name of the target app, while the second does not take any argument as input and returns a list of *all* apps installed for the current user. Thus, if a sample invokes the `getPackageInfo` method with the `com.topjohnwu.magisk` argument, there is no doubt that it is checking for the presence of Magisk; this is a DET. On the other hand, if a sample retrieves the app list invoking `getInstalledApplications`, it can look for the Magisk package name in several stealth ways (e.g., hash comparison). Hence, this is an IET. Therefore, if an analysis system detects a DET, it will always be a true positive, while the presence of an IET can also be a false positive.

Each implementation has a unique identifier, consisting of the concatenations of three strings (the macro technique, the goal, and the type of control) with the symbol "-". For instance, the ROOT-SU-FILE denotes a *root detection* (macro technique) evasive control, which aims to verify the presence of the `su` (goal) binary *file* (type of control).

Table 1 recaps the categorization of Android evasive controls, highlighting possible DET and IET implementations.

	Technique	Description	Goal TAG	Goal Description	Impl. TAG	Implementation Description	Example
Environment Verification	ROOT	Root detection	APPS	Detect if 'root' apps are installed on the device	APP_INFO†	Query the package manager with a specific 'root' package name	PackageManager. getPackageInfo("magisk")
					INST_APPS§	Retrieve the list of all installed apps on the device	PackageManager. getInstalledPackages()
					STORAGE†	Try to access to the external storage of other 'root' apps	
					CMD†	Execute command to find/execute the 'root' apps	popen("magisk")
			SU	Check the presence of the 'su' binary	FILE†	Access to well known super-user paths	open("/system/bin/su")
					CMD†	Execute command to find/execute the 'su' binary	popen("which su")
			BUSYBOX	Check the presence of the 'busybox' binary	FILE†	Access to well known busybox paths	
					CMD†	Execute command to find/execute the 'busybox' binary	
			PROPS	Check root-related Android system properties	PROP†	Retrieve a specific system property	
					CMD†	Retrieve a specific system property through command line	popen ("getprop ro.build.tags")
					FOREACH§	Retrieve all Android system property	popen("getprop")
			RO_PATHS	Check if some paths that should be only readable is also writable	FILE§	Check the property of the 'read-only' paths	faccess("/system/bin")
					CMD§	Check the mounted partitions	popen("mount")
	DEBUG	Debugging detection	PROPS	Check debug-related Android system properies	PROP†	Retrieve a specific system property	ro.debuggable
					CMD†	Retrieve a specific system property through command line	
					FOREACH§	Retrieve all Android system property	
			MANIFEST	Check if the app was built with the debugabble flag enable	APP_INFO§	Query the package manager to retrieve the app metadata with the GET_ATTRIBUTIONS flag	
					INST_APPS§	Retrieve the info related to the GET_ATTRIBUTIONS flag for all installed apps	
			IS_CON	Check if a debugger is connected	API†	android.os.Debug. isDebuggerConnected()	
			JDWP	Check the presence of the JDWP debugger	FILE§	Parse the 'comm' files under the '/proc/self/*' directory	open ("proc/self/comm")
					CMD§		popen ("cat /proc/self/comm")
			PORT	Check if the default debugger port is already taken	FILE§	Parse the /proc/net/tcp file	open("proc/net/tcp")

				CMD§		popen ("cat /proc/net/tcp")
				SOCKET†	Connect with a socket to the default debugger port	socket ('127.0.0.1', 23946)
		TRACERPID	Check for the presence of a tracer PID	FILE§	Parse the status file of the current process or the tasks of the process	
				CMD§		
		TIME	Time checks	API§	Check the elapsed time in Java or C/C++ code	SystemClock. elapsedRealtime()
				FILE§	Parse the '/proc/utime' to retrive the elapsed time	
				CMD§		
	HOOK	Hook detection	APPS	Detect if 'hook' apps are installed on the device	APP_INFO†	Query the package manager with a specific 'hook' package name	PackageManager. getPackageInfo("xposed")
					INST_APPS§	Retrieve the list of all installed apps on the device	
					STORAGE†	Try to access to the external storage of other 'hook' apps	
					CMD†	Execute command to find/execute the 'hook' apps	popen ("find . -name *xposed*")
			PROC_ART	Detect artifacts in the process memory mapping	MAPS§	Parse the '/proc/self/maps' file	
					NF§	Inspect the memory mapping of the process through C/C++ functions	
					CMD§		popen("ps")
					API§	Check high-level process info by quering the Android APIs	Application. getProcessName()
			STACKTRACE	Retrieve and parse the stacktrace to detect call graph artifacts	API§		Thread. getStackTrace()
			FRIDA	Frida-specific controls (not contained in previous checks)	FILE§	Check the content of the '/proc/self/fs/*' files	
					CMD§		
					SOCKET†	Connect to well known ip/port of the frameworks	socket ("127.0.0.1", 27042)
			CLASS	Detect if the target app retrives well known framework classes	API†		de.robv.android.*
	EMU	Emulator detection	PROPS	Check emulator-related Android system properties	PROP†	Access to a specific system property	
					CMD†	Retrieve a specific system property through command line	
					FOREACH§	Retrieve all Android system property	
			ADB	Check if the adb is emulated	FILE§	Check the content of the '/proc/net/tcp' file	
					CMD§		

			SENSOR	Check if sensors' values are not related to real behavior	API[§]	Register a listener to collect sensors' data	gyroscope
			SYSTEM	Check System related properties (e.g., Device ID, Subscriber ID)	API[†]		TelephonyManager. getLine1Number()
					STATS[§]	Get (file-)system stats	ustat
					LOGCAT[§]	Exploit *logcat* to retrieve system information	
			KNOWN_EMU	Check artifact of well known emulators' artifacts	FILE[†]	Check the presence of the file	access ("/dev/socket/genyd")
					CMD[†]		popen ("ls /init.vbox86.rc")
			QEMU	Check QEMU artifacts	FILE[†]	Check the presence of a file	open ("/dev/qemu_pipe")
					PROP[†]	Access to QEMU-specific system properties	
					PROC[§]	Check hardware informations	/proc/cpuinfo
					CMD[§]		
			BEHAVIOR	Check (emulated) device features	BATTERY[§]	Check the battery status through API or broadcast receivers	New receiver for: BATTERY_CHANGED
					USER_PROF[§]		
					HARDWARE[§]	Check hardware-related discrepancy. For instance, graphical low video frame rate	
	MEMTMP	Memory integrity verification	MEM2DISK	Check the content of a file w.r.t. the one loaded in memory	FILE[§]	Open libraries that are already mapped into the memory of the process	
			PLT	Check for PLT modifications	—	It is not possible to detect	
			INLINE	Check for inline breakpoints or trampolines	—	It is not possible to detect	
	VIRT	App-level virtualization detection	FAKE_COMP	Check components' name w.r.t. the ones in the Manifest	API[§]	Retrieve the list of registered/running components of the app	
					APP_INFO[§]	Query the package manager to retrieve the metadata of the app with one of the following flags: GET_ACTIVITIES, GET_ATTRIBUTIONS, GET_RECEIVERS, GET_SERVICES, GET_PROVIDERS, GET_SHARED_LIBRARY_FILES	
					INST_APPS[§]	Retrieve the info for all installed apps with one of the following flags: GET_ACTIVITIES, GET_ATTRIBUTIONS, GET_RECEIVERS, GET_SERVICES, GET_PROVIDERS, GET_SHARED_LIBRARY_FILES	

		APP	Check if the app is really installed on the device	APP_INFO[§]	Query the package manager to retrieve the metadata of the app itself	
				INST_APPS[§]	Retrieve the info for all installed apps	
				NATIVE[§]	Retrieve the info of native libraries	
		PROC	Check the app's processes	CMD[§]		popen("ps")
				API[§]	Retrieve the list of running components of the app and their metadata	Application. getProcessName()
		UND_PERMS	Check if the app has or check for permissions that are not declared in the Manifest file	API[†]		Context. checkPermission (<no_manifest_perm>)
				APP_INFO[§]	Query the package manager to retrieve the metadata of the app with the GET_PERMISSIONS, or GET_URI_PERMISSIONS_PATTERNS flag	
				INST_APPS[§]	Retrieve the info for all installed apps with GET_PERMISSIONS, or GET_URI_PERMISSIONS_PATTERNS flag	
		APP_DIR	Check the installation path and other app's private folders	APP_INFO[§]	Query the package manager to retrieve the metadata of the app with the GET_METADATA flag	
				INST_APPS[§]	Retrieve the info for all installed apps with the GET_METADATA flag	
	NET	Network detection	ADB	Check if the adb is emulated	SOCKET[†]	Connect to the default adb port	socket ("127.0.0.1", 5555)
			VPN	Check the presence of a VPN	API[†]		NetworkCapabilities. hasTransport()
					APP_INFO[†]	Query the package manager to retrieve the metadata of well known VPN apps	
					INST_APPS[§]	Retrieve the info for all installed apps	
					STORAGE[†]	Try to access to the external storage of other VPN apps	
					CMD[†]	Execute command to find/execute the VPN apps	
			INTERFACE	Check the metadata of the network interfaces	NF[§]		getifaddrs()
					FILE[§]	Check the content of '/proc/sys/net/ipv(4—6)' and '/sys/class/net'	
					CMD[§]	Command line commnds	popen("ifconfig")
					API[§]		ConnectivityManager. getNetworkInfo()

			FAKEIP	Check if the current IP is not real	CMD[§]		/system/bin/netcfg
			KNOWNIP	Check if the current IP is well known	SOCKET[†]		maxmind.com
			ADB	Check if the standard adb port is already used	SOCKET[†]		maxmind.com
			SSL_PINNING	Check if the app uses ssl pinning	API[§]	The app uses the SSL Context to perform a network request	
			LISTENER	Monitor changes for network	API[§]	Register a new listener though the Android API	ConnectivityManager. listenForNetwork()
					BR[§]	Register a broadcast receiver at runtime for: 'CONNECTIVITY_CHANGE', 'WIFI_STATE_CHANGED', and 'AIRPLANE_MODE'	
APK Tampering Verification	SIGNATURE	Signature check	ZIP	Read the certificate files in the APK	FILE[§]	Access to the APK	ZipFile ("/path/to/base.apk")
					CMD[§]		popen ("unzip base.apk")
			APP		APP_INFO[§]	Query the package manager to retrieve the metadata of the app with GET_SIGNATURES, or GET_SIGNING_CERTIFICATES flags	
					INST_APPS[§]	Retrieve the info for all installed apps with the GET_SIGNATURES, or GET_SIGNING_CERTIFICATES flags	
	INSTALL	Installer Verification	SOURCE		API[†]		PackageManager. getInstallSourceInfo()
High-level Verification	GOOGLE	SafetyNet & Integrity API	SN	SafetyNet checks	BINDER[†]	Check the binder methods for the SafetyNet requests	
			IA	Integrity API	API[†]		
	HUMAN	Interaction with a human being	—		—	It is not possible to detect	

Table 1: List of evasive techniques.
†: Direct Evasive Technique (DET)
§: Indirect Evasive Technique (IET)